UNIX® AND LINUX® SYSTEM ADMINISTRATION HANDBOOK

FIFTH EDITION



EVI NEMETH • GARTH SNYDER • TRENT R. HEIN
BEN WHALEY • DAN MACKIN
with James Garnett, Fabrizio Branca, and Adrian Mouat

UNIX® AND LINUX® SYSTEM ADMINISTRATION HANDBOOK

FIFTH EDITION

Evi Nemeth
Garth Snyder
Trent R. Hein
Ben Whaley
Dan Mackin

with James Garnett, Fabrizio Branca, and Adrian Mouat

Addison-Wesley

Boston • Columbus • Indianapolis • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

Ubuntu is a registered trademark of Canonical Limited, and is used with permission.

Debian is a registered trademark of Software in the Public Interest Incorporated.

CentOS is a registered trademark of Red Hat Inc., and is used with permission.

FreeBSD is a registered trademark of The FreeBSD Foundation, and is used with permission.

The Linux Tux logo was created by Larry Ewing, lewing@isc.tamu.edu.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Visit us on the web: informit.com

Library of Congress Control Number: 2017945559

Copyright © 2018 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit www.pearsoned.com/permissions/.

ISBN-13: 978-0-13-427755-4
ISBN-10: 0-13-427755-4

1 17

Table of Contents

TRIBUTE TO EVI
PREFACE
FOREWORD
ACKNOWLEDGMENTS

SECTION ONE: BASIC ADMINISTRATION

CHAPTER 1: WHERE TO START

- Essential duties of a system administrator
  - Controlling access
  - Adding hardware
  - Automating tasks
  - Overseeing backups
  - Installing and upgrading software
  - Monitoring
  - Troubleshooting
  - Maintaining local documentation
  - Vigilantly monitoring security
  - Tuning performance
  - Developing site policies
  - Working with vendors
  - Fire fighting
- Suggested background
- Linux distributions
- Example systems used in this book
  - Example Linux distributions

- Example UNIX distribution
- Notation and typographical conventions
- Units
- Man pages and other on-line documentation
  - Organization of the man pages
  - **man**: read man pages
  - Storage of man pages
- Other authoritative documentation
  - System-specific guides
  - Package-specific documentation
  - Books
  - RFC publications
- Other sources of information
  - Keeping current
  - HowTos and reference sites
  - Conferences
- Ways to find and install software
  - Determining if software is already installed
  - Adding new software
  - Building software from source code
  - Installing from a web script
- Where to host
- Specialization and adjacent disciplines
  - DevOps
  - Site reliability engineers
  - Security operations engineers
  - Network administrators
  - Database administrators
  - Network operations center (NOC) engineers
  - Data center technicians
  - Architects
- Recommended reading
  - System administration and DevOps
  - Essential tools

CHAPTER 2: BOOTING AND SYSTEM MANAGEMENT DAEMONS

- Boot process overview
- System firmware
  - BIOS vs. UEFI
  - Legacy BIOS

- UEFI
- Boot loaders
- GRUB: the GRand Unified Boot loader
  - GRUB configuration
  - The GRUB command line
  - Linux kernel options
- The FreeBSD boot process
  - The BIOS path: **boot0**
  - The UEFI path
  - **loader** configuration
  - **loader** commands
- System management daemons
  - Responsibilities of **init**
  - Implementations of **init**
  - Traditional **init**
  - **systemd** vs. the world
  - **init**s judged and assigned their proper punishments
- **systemd** in detail
  - Units and unit files
  - **systemctl**: manage **systemd**
  - Unit statuses
  - Targets
  - Dependencies among units
  - Execution order
  - A more complex unit file example
  - Local services and customizations
  - Service and startup control caveats
  - **systemd** logging
- FreeBSD **init** and startup scripts
- Reboot and shutdown procedures
  - Shutting down physical systems
  - Shutting down cloud systems
- Stratagems for a nonbooting system
  - Single-user mode
  - Single-user mode on FreeBSD
  - Single-user mode with GRUB
  - Recovery of cloud systems

CHAPTER 3: ACCESS CONTROL AND ROOTLY POWERS

- Standard UNIX access control

- Filesystem access control
- Process ownership
- The root account
- Setuid and setgid execution

Management of the root account

- Root account login
- **su**: substitute user identity
- **sudo**: limited **su**
- Disabling the root account
- System accounts other than root

Extensions to the standard access control model

- Drawbacks of the standard model
- PAM: Pluggable Authentication Modules
- Kerberos: network cryptographic authentication
- Filesystem access control lists
- Linux capabilities
- Linux namespaces

Modern access control

- Separate ecosystems
- Mandatory access control
- Role-based access control
- SELinux: Security-Enhanced Linux
- AppArmor

Recommended reading

CHAPTER 4: PROCESS CONTROL

Components of a process

- PID: process ID number
- PPID: parent PID
- UID and EUID: real and effective user ID
- GID and EGID: real and effective group ID
- Niceness
- Control terminal

The life cycle of a process

- Signals
- **kill**: send signals
- Process and thread states

ps: monitor processes

Interactive monitoring with **top**

nice and **renice**: influence scheduling priority

The **/proc** filesystem
strace and **truss**: trace signals and system calls
Runaway processes
Periodic processes
cron: schedule commands
systemd timers
Common uses for scheduled tasks

CHAPTER 5: THE FILESYSTEM

Pathnames
Filesystem mounting and unmounting
Organization of the file tree
File types
Regular files
Directories
Hard links
Character and block device files
Local domain sockets
Named pipes
Symbolic links
File attributes
The permission bits
The setuid and setgid bits
The sticky bit
ls: list and inspect files
chmod: change permissions
chown and **chgrp**: change ownership and group
umask: assign default permissions
Linux bonus flags
Access control lists
A cautionary note
ACL types
Implementation of ACLs
Linux ACL support
FreeBSD ACL support
POSIX ACLs
NFSv4 ACLs

CHAPTER 6: SOFTWARE INSTALLATION AND MANAGEMENT

Operating system installation

- Installing from the network
- Setting up PXE
- Using kickstart, the automated installer for Red Hat and CentOS
- Automating installation for Debian and Ubuntu
- Netbooting with Cobbler, the open source Linux provisioning server
- Automating FreeBSD installation

Managing packages

Linux package management systems

- **rpm**: manage RPM packages
- **dpkg**: manage **.deb** packages

High-level Linux package management systems

- Package repositories
- RHN: the Red Hat Network
- APT: the Advanced Package Tool
- Repository configuration
- An example **/etc/apt/sources.list** file
- Creation of a local repository mirror
- APT automation
- **yum**: release management for RPM

FreeBSD software management

- The base system
- **pkg**: the FreeBSD package manager
- The ports collection

Software localization and configuration

- Organizing your localization
- Structuring updates
- Limiting the field of play
- Testing

Recommended reading

CHAPTER 7: SCRIPTING AND THE SHELL

Scripting philosophy

- Write microscripts
- Learn a few tools well
- Automate all the things
- Don’t optimize prematurely
- Pick the right scripting language
- Follow best practices

Shell basics

- Command editing

- Pipes and redirection
- Variables and quoting
- Environment variables
- Common filter commands

sh scripting
- Execution
- From commands to scripts
- Input and output
- Spaces in filenames
- Command-line arguments and functions
- Control flow
- Loops
- Arithmetic

Regular expressions
- The matching process
- Literal characters
- Special characters
- Example regular expressions
- Captures
- Greediness, laziness, and catastrophic backtracking

Python programming
- The passion of Python 3
- Python 2 or Python 3?
- Python quick start
- Objects, strings, numbers, lists, dictionaries, tuples, and files
- Input validation example
- Loops

Ruby programming
- Installation
- Ruby quick start
- Blocks
- Symbols and option hashes
- Regular expressions in Ruby
- Ruby as a filter

Library and environment management for Python and Ruby
- Finding and installing packages
- Creating reproducible environments
- Multiple environments

Revision control with Git
- A simple Git example
- Git caveats

- Social coding with Git
- Recommended reading
  - Shells and shell scripting
  - Regular expressions
  - Python
  - Ruby

CHAPTER 8: USER MANAGEMENT

- Account mechanics
- The **/etc/passwd** file
  - Login name
  - Encrypted password
  - UID (user ID) number
  - Default GID (group ID) number
  - GECOS field
  - Home directory
  - Login shell
- The Linux **/etc/shadow** file
- FreeBSD's **/etc/master.passwd** and **/etc/login.conf** files
  - The **/etc/master.passwd** file
  - The **/etc/login.conf** file
- The **/etc/group** file
- Manual steps for adding users
  - Editing the **passwd** and **group** files
  - Setting a password
  - Creating the home directory and installing startup files
  - Setting home directory permissions and ownerships
  - Configuring roles and administrative privileges
  - Finishing up
- Scripts for adding users: **useradd**, **adduser**, and **newusers**
  - **useradd** on Linux
  - **adduser** on Debian and Ubuntu
  - **adduser** on FreeBSD
  - **newusers** on Linux: adding in bulk
- Safe removal of a user's account and files
- User login lockout
- Risk reduction with PAM
- Centralized account management
  - LDAP and Active Directory
  - Application-level single sign-on systems

Identity management systems

Chapter 9: Cloud Computing

The cloud in context
Cloud platform choices
Public, private, and hybrid clouds
Amazon Web Services
Google Cloud Platform
DigitalOcean
Cloud service fundamentals
Access to the cloud
Regions and availability zones
Virtual private servers
Networking
Storage
Identity and authorization
Automation
Serverless functions
Clouds: VPS quick start by platform
Amazon Web Services
Google Cloud Platform
DigitalOcean
Cost control
Recommended Reading

Chapter 10: Logging

Log locations
Files not to manage
How to view logs in the **systemd** journal
The **systemd** journal
Configuring the **systemd** journal
Adding more filtering options for **journalctl**
Coexisting with syslog
Syslog
Reading syslog messages
Rsyslog architecture
Rsyslog versions
Rsyslog configuration
Config file examples
Syslog message security

- Syslog configuration debugging
- Kernel and boot-time logging
- Management and rotation of log files
  - **logrotate**: cross-platform log management
  - **newsyslog**: log management on FreeBSD
- Management of logs at scale
  - The ELK stack
  - Graylog
  - Logging as a service
- Logging policies

Chapter 11: Drivers and the Kernel

- Kernel chores for system administrators
- Kernel version numbering
  - Linux kernel versions
  - FreeBSD kernel versions
- Devices and their drivers
  - Device files and device numbers
  - Challenges of device file management
  - Manual creation of device files
  - Modern device file management
  - Linux device management
  - FreeBSD device management
- Linux kernel configuration
  - Tuning Linux kernel parameters
  - Building a custom kernel
  - Adding a Linux device driver
- FreeBSD kernel configuration
  - Tuning FreeBSD kernel parameters
  - Building a FreeBSD kernel
- Loadable kernel modules
  - Loadable kernel modules in Linux
  - Loadable kernel modules in FreeBSD
- Booting
  - Linux boot messages
  - FreeBSD boot messages
- Booting alternate kernels in the cloud
- Kernel errors
  - Linux kernel errors

FreeBSD kernel panics
Recommended reading

CHAPTER 12: PRINTING

CUPS printing
Interfaces to the printing system
The print queue
Multiple printers and queues
Printer instances
Network printer browsing
Filters
CUPS server administration
Network print server setup
Printer autoconfiguration
Network printer configuration
Printer configuration examples
Service shutoff
Other configuration tasks
Troubleshooting tips
Print daemon restart
Log files
Direct printing connections
Network printing problems
Recommended reading

SECTION TWO: NETWORKING

CHAPTER 13: TCP/IP NETWORKING

TCP/IP and its relationship to the Internet
Who runs the Internet?
Network standards and documentation
Networking basics
IPv4 and IPv6
Packets and encapsulation
Ethernet framing
Maximum transfer unit
Packet addressing
Hardware (MAC) addressing

- IP addressing
- Hostname “addressing”
- Ports
- Address types

IP addresses: the gory details

- IPv4 address classes
- IPv4 subnetting
- Tricks and tools for subnet arithmetic
- CIDR: Classless Inter-Domain Routing
- Address allocation
- Private addresses and network address translation (NAT)
- IPv6 addressing

Routing

- Routing tables
- ICMP redirects

IPv4 ARP and IPv6 neighbor discovery

DHCP: the Dynamic Host Configuration Protocol

- DHCP software
- DHCP behavior
- ISC’s DHCP software

Security issues

- IP forwarding
- ICMP redirects
- Source routing
- Broadcast pings and other directed broadcasts
- IP spoofing
- Host-based firewalls
- Virtual private networks

Basic network configuration

- Hostname and IP address assignment
- Network interface and IP configuration
- Routing configuration
- DNS configuration
- System-specific network configuration

Linux networking

- NetworkManager
- **ip**: manually configure a network
- Debian and Ubuntu network configuration
- Red Hat and CentOS network configuration
- Linux network hardware options
- Linux TCP/IP options

- Security-related kernel variables
- FreeBSD networking
  - **ifconfig**: configure network interfaces
  - FreeBSD network hardware configuration
  - FreeBSD boot-time network configuration
  - FreeBSD TCP/IP configuration
- Network troubleshooting
  - **ping**: check to see if a host is alive
  - **traceroute**: trace IP packets
  - Packet sniffers
- Network monitoring
  - SmokePing: gather ping statistics over time
  - iPerf: track network performance
  - Cacti: collect and graph data
- Firewalls and NAT
  - Linux **iptables**: rules, chains, and tables
  - IPFilter for UNIX systems
- Cloud networking
  - AWS's virtual private cloud (VPC)
  - Google Cloud Platform networking
  - DigitalOcean networking
- Recommended reading
  - History
  - Classics and bibles
  - Protocols

CHAPTER 14: PHYSICAL NETWORKING

- Ethernet: the Swiss Army knife of networking
  - Ethernet signaling
  - Ethernet topology
  - Unshielded twisted-pair cabling
  - Optical fiber
  - Ethernet connection and expansion
  - Autonegotiation
  - Power over Ethernet
  - Jumbo frames
- Wireless: Ethernet for nomads
  - Wireless standards
  - Wireless client access
  - Wireless infrastructure and WAPs

- Wireless security
- SDN: software-defined networking
- Network testing and debugging
- Building wiring
  - UTP cabling options
  - Connections to offices
  - Wiring standards
- Network design issues
  - Network architecture vs. building architecture
  - Expansion
  - Congestion
  - Maintenance and documentation
- Management issues
- Recommended vendors
  - Cables and connectors
  - Test equipment
  - Routers/switches
- Recommended reading

CHAPTER 15: IP ROUTING

- Packet forwarding: a closer look
- Routing daemons and routing protocols
  - Distance-vector protocols
  - Link-state protocols
  - Cost metrics
  - Interior and exterior protocols
- Protocols on parade
  - RIP and RIPng: Routing Information Protocol
  - OSPF: Open Shortest Path First
  - EIGRP: Enhanced Interior Gateway Routing Protocol
  - BGP: Border Gateway Protocol
- Routing protocol multicast coordination
- Routing strategy selection criteria
- Routing daemons
  - **routed**: obsolete RIP implementation
  - Quagga: mainstream routing daemon
  - XORP: router in a box
- Cisco routers
- Recommended reading

Chapter 16: DNS: The Domain Name System

- DNS architecture
  - Queries and responses
  - DNS service providers
- DNS for lookups
  - **resolv.conf**: client resolver configuration
  - **nsswitch.conf**: who do I ask for a name?
- The DNS namespace
  - Registering a domain name
  - Creating your own subdomains
- How DNS works
  - Name servers
  - Authoritative and caching-only servers
  - Recursive and nonrecursive servers
  - Resource records
  - Delegation
  - Caching and efficiency
  - Multiple answers and round robin DNS load balancing
  - Debugging with query tools
- The DNS database
  - Parser commands in zone files
  - Resource records
  - The SOA record
  - NS records
  - A records
  - AAAA records
  - PTR records
  - MX records
  - CNAME records
  - SRV records
  - TXT records
  - SPF, DKIM, and DMARC records
  - DNSSEC records
- The BIND software
  - Components of BIND
  - Configuration files
  - The include statement
  - The options statement
  - The acl statement
  - The (TSIG) key statement

- The server statement
- The masters statement
- The logging statement
- The statistics-channels statement
- The zone statement
- The controls statement for **rndc**

Split DNS and the view statement

BIND configuration examples

- The localhost zone
- A small security company

Zone file updating

- Zone transfers
- Dynamic updates

DNS security issues

- Access control lists in BIND, revisited
- Open resolvers
- Running in a **chroot**ed jail
- Secure server-to-server communication with TSIG and TKEY
- Setting up TSIG for BIND
- DNSSEC
- DNSSEC policy
- DNSSEC resource records
- Turning on DNSSEC
- Key pair generation
- Zone signing
- The DNSSEC chain of trust
- DNSSEC key rollover
- DNSSEC tools
- Debugging DNSSEC

BIND debugging

- Logging in BIND
- Name server control with **rndc**
- Command-line querying for lame delegations

Recommended reading

- Books and other documentation
- On-line resources
- The RFCs

CHAPTER 17: SINGLE SIGN-ON

Core SSO elements

LDAP: “lightweight” directory services
Uses for LDAP
The structure of LDAP data
OpenLDAP: the traditional open source LDAP server
389 Directory Server: alternative open source LDAP server
LDAP Querying
Conversion of **passwd** and **group** files to LDAP
Using directory services for login
Kerberos
sssd: the System Security Services Daemon
nsswitch.conf: the name service switch
PAM: cooking spray or authentication wonder?
Alternative approaches
NIS: the Network Information Service
rsync: transfer files securely
Recommended reading

CHAPTER 18: ELECTRONIC MAIL

Mail system architecture
User agents
Submission agents
Transport agents
Local delivery agents
Message stores
Access agents
Anatomy of a mail message
The SMTP protocol
You had me at EHLO
SMTP error codes
SMTP authentication
Spam and malware
Forgeries
SPF and Sender ID
DKIM
Message privacy and encryption
Mail aliases
Getting aliases from files
Mailing to files
Mailing to programs
Building the hashed alias database

Email configuration
sendmail
- The switch file
- Starting **sendmail**
- Mail queues
- **sendmail** configuration
- The **m4** preprocessor
- The **sendmail** configuration pieces
- A configuration file built from a sample **.mc** file
- Configuration primitives
- Tables and databases
- Generic macros and features
- Client configuration
- **m4** configuration options
- Spam-related features in **sendmail**
- Security and **sendmail**
- **sendmail** testing and debugging

Exim
- Exim installation
- Exim startup
- Exim utilities
- Exim configuration language
- Exim configuration file
- Global options
- Access control lists (ACLs)
- Content scanning at ACL time
- Authenticators
- Routers
- Transports
- Retry configuration
- Rewriting configuration
- Local scan function
- Logging
- Debugging

Postfix
- Postfix architecture
- Security
- Postfix commands and documentation
- Postfix configuration
- Virtual domains
- Access control
- Debugging

Recommended reading
sendmail references
Exim references
Postfix references
RFCs

CHAPTER 19: WEB HOSTING

HTTP: the Hypertext Transfer Protocol
Uniform Resource Locators (URLs)
Structure of an HTTP transaction
curl: HTTP from the command line
TCP connection reuse
HTTP over TLS
Virtual hosts
Web software basics
Web servers and HTTP proxy software
Load balancers
Caches
Content delivery networks
Languages of the web
Application programming interfaces (APIs)
Web hosting in the cloud
Build versus buy
Platform-as-a-Service
Static content hosting
Serverless web applications
Apache **httpd**
httpd in use
httpd configuration logistics
Virtual host configuration
Logging
NGINX
Installing and running NGINX
Configuring NGINX
Configuring TLS for NGINX
Load balancing with NGINX
HAProxy
Health checks
Server statistics
Sticky sessions

TLS termination
Recommended reading

SECTION THREE: STORAGE

CHAPTER 20: STORAGE

I just want to add a disk!
Linux recipe
FreeBSD recipe
Storage hardware
Hard disks
Solid state disks
Hybrid drives
Advanced Format and 4KiB blocks
Storage hardware interfaces
The SATA interface
The PCI Express interface
The SAS interface
USB
Attachment and low-level management of drives
Installation verification at the hardware level
Disk device files
Ephemeral device names
Formatting and bad block management
ATA secure erase
hdparm and **camcontrol**: set disk and interface parameters
Hard disk monitoring with SMART
The software side of storage: peeling the onion
Elements of a storage system
The Linux device mapper
Disk partitioning
Traditional partitioning
MBR partitioning
GPT: GUID partition tables
Linux partitioning
FreeBSD partitioning
Logical volume management
Linux logical volume management

FreeBSD logical volume management
RAID: redundant arrays of inexpensive disks
Software vs. hardware RAID
RAID levels
Disk failure recovery
Drawbacks of RAID 5
mdadm: Linux software RAID
Filesystems
Traditional filesystems: UFS, ext4, and XFS
Filesystem terminology
Filesystem polymorphism
Filesystem formatting
fsck: check and repair filesystems
Filesystem mounting
Setup for automatic mounting
USB drive mounting
Swapping recommendations
Next-generation filesystems: ZFS and Btrfs
Copy-on-write
Error detection
Performance
ZFS: all your storage problems solved
ZFS on Linux
ZFS architecture
Example: disk addition
Filesystems and properties
Property inheritance
One filesystem per user
Snapshots and clones
Raw volumes
Storage pool management
Btrfs: “ZFS lite” for Linux
Btrfs vs. ZFS
Setup and storage conversion
Volumes and subvolumes
Volume snapshots
Shallow copies
Data backup strategy
Recommended reading

CHAPTER 21: THE NETWORK FILE SYSTEM

- Meet network file services
  - The competition
  - Issues of state
  - Performance concerns
  - Security
- The NFS approach
  - Protocol versions and history
  - Remote procedure calls
  - Transport protocols
  - State
  - Filesystem exports
  - File locking
  - Security concerns
  - Identity mapping in version 4
  - Root access and the nobody account
  - Performance considerations in version 4
- Server-side NFS
  - Linux **exports**
  - FreeBSD **exports**
  - **nfsd**: serve files
- Client-side NFS
  - Mounting remote filesystems at boot time
  - Restricting exports to privileged ports
- Identity mapping for NFS version 4
- **nfsstat**: dump NFS statistics
- Dedicated NFS file servers
- Automatic mounting
  - Indirect maps
  - Direct maps
  - Master maps
  - Executable maps
  - Automount visibility
  - Replicated filesystems and **automount**
  - Automatic automounts (V3; all but Linux)
  - Specifics for Linux
- Recommended reading

CHAPTER 22: SMB

- Samba: SMB server for UNIX
- Installing and configuring Samba

File sharing with local authentication
File sharing with accounts authenticated by Active Directory
Configuring shares
Mounting SMB file shares
Browsing SMB file shares
Ensuring Samba security
Debugging Samba
Querying Samba's state with **smbstatus**
Configuring Samba logging
Managing character sets
Recommended reading

SECTION FOUR: OPERATIONS

CHAPTER 23: CONFIGURATION MANAGEMENT

Configuration management in a nutshell
Dangers of configuration management
Elements of configuration management
Operations and parameters
Variables
Facts
Change handlers
Bindings
Bundles and bundle repositories
Environments
Client inventory and registration
Popular CM systems compared
Terminology
Business models
Architectural options
Language options
Dependency management options
General comments on Chef
General comments on Puppet
General comments on Ansible and Salt
YAML: a rant
Introduction to Ansible
Ansible example

- Client setup
- Client groups
- Variable assignments
- Dynamic and computed client groups
- Task lists
- state parameters
- Iteration
- Interaction with Jinja
- Template rendering
- Bindings: plays and playbooks
- Roles
- Recommendations for structuring the configuration base
- Ansible access options

Introduction to Salt

- Minion setup
- Variable value binding for minions
- Minion matching
- Salt states
- Salt and Jinja
- State IDs and dependencies
- State and execution functions
- Parameters and names
- State binding to minions
- Highstates
- Salt formulas
- Environments
- Documentation roadmap

Ansible and Salt compared

- Deployment flexibility and scalability
- Built-in modules and extensibility
- Security
- Miscellaneous

Best practices

Recommended reading

CHAPTER 24: VIRTUALIZATION

Virtual vernacular

- Hypervisors
- Live migration
- Virtual machine images
- Containerization

- Virtualization with Linux
  - Xen
  - Xen guest installation
  - KVM
  - KVM guest installation
- FreeBSD bhyve
- VMware
- VirtualBox
- Packer
- Vagrant
- Recommended reading

CHAPTER 25: CONTAINERS

- Background and core concepts
  - Kernel support
  - Images
  - Networking
- Docker: the open source container engine
  - Basic architecture
  - Installation
  - Client setup
  - The container experience
  - Volumes
  - Data volume containers
  - Docker networks
  - Storage drivers
  - **dockerd** option editing
  - Image building
  - Registries
- Containers in practice
  - Logging
  - Security advice
  - Debugging and troubleshooting
- Container clustering and management
  - A synopsis of container management software
  - Kubernetes
  - Mesos and Marathon
  - Docker Swarm
  - AWS EC2 Container Service
- Recommended reading

Chapter 26: Continuous Integration and Delivery

- CI/CD essentials
  - Principles and practices
  - Environments
  - Feature flags
- Pipelines
  - The build process
  - Testing
  - Deployment
  - Zero-downtime deployment techniques
- Jenkins: the open source automation server
  - Basic Jenkins concepts
  - Distributed builds
  - Pipeline as code
- CI/CD in practice
  - UlsahGo, a trivial web application
  - Unit testing UlsahGo
  - Taking first steps with the Jenkins Pipeline
  - Building a DigitalOcean image
  - Provisioning a single system for testing
  - Testing the droplet
  - Deploying UlsahGo to a pair of droplets and a load balancer
  - Concluding the demonstration pipeline
- Containers and CI/CD
  - Containers as a build environment
  - Container images as build artifacts
- Recommended reading

Chapter 27: Security

- Elements of security
- How security is compromised
  - Social engineering
  - Software vulnerabilities
  - Distributed denial-of-service attacks (DDoS)
  - Insider abuse
  - Network, system, or application configuration errors
- Basic security measures
  - Software updates
  - Unnecessary services
  - Remote event logging

- Backups
- Viruses and worms
- Root kits
- Packet filtering
- Passwords and multifactor authentication
- Vigilance
- Application penetration testing

Passwords and user accounts

- Password changes
- Password vaults and password escrow
- Password aging
- Group logins and shared logins
- User shells
- Rootly entries

Security power tools

- Nmap: network port scanner
- Nessus: next-generation network scanner
- Metasploit: penetration testing software
- Lynis: on-box security auditing
- John the Ripper: finder of insecure passwords
- Bro: the programmable network intrusion detection system
- Snort: the popular network intrusion detection system
- OSSEC: host-based intrusion detection
- Fail2Ban: brute-force attack response system

Cryptography primer

- Symmetric key cryptography
- Public key cryptography
- Public key infrastructure
- Transport Layer Security
- Cryptographic hash functions
- Random number generation
- Cryptographic software selection
- The **openssl** command
- PGP: Pretty Good Privacy
- Kerberos: a unified approach to network security

SSH, the Secure SHell

- OpenSSH essentials
- The **ssh** client
- Public key authentication
- The **ssh-agent**
- Host aliases in **~/.ssh/config**

- Connection multiplexing
- Port forwarding
- **sshd**: the OpenSSH server
- Host key verification with SSHFP
- File transfers
- Alternatives for secure logins

Firewalls

- Packet-filtering firewalls
- Filtering of services
- Stateful inspection firewalls
- Firewalls: safe?

Virtual private networks (VPNs)

- IPsec tunnels
- All I need is a VPN, right?

Certifications and standards

- Certifications
- Security standards

Sources of security information

- SecurityFocus.com, the BugTraq mailing list, and the OSS mailing list
- Schneier on Security
- The Verizon Data Breach Investigations Report
- The SANS Institute
- Distribution-specific security resources
- Other mailing lists and web sites

When your site has been attacked

Recommended reading

CHAPTER 28: MONITORING

An overview of monitoring

- Instrumentation
- Data types
- Intake and processing
- Notifications
- Dashboards and UIs

The monitoring culture

The monitoring platforms

- Open source real-time platforms
- Open source time-series platforms
- Open source charting platforms
- Commercial monitoring platforms

Hosted monitoring platforms
Data collection
StatsD: generic data submission protocol
Data harvesting from command output
Network monitoring
Systems monitoring
Commands for systems monitoring
collectd: generalized system data harvester
sysdig and **dtrace**: execution tracers
Application monitoring
Log monitoring
Supervisor + Munin: a simple option for limited domains
Commercial application monitoring tools
Security monitoring
System integrity verification
Intrusion detection monitoring
SNMP: the Simple Network Management Protocol
SNMP organization
SNMP protocol operations
Net-SNMP: tools for servers
Tips and tricks for monitoring
Recommended reading

CHAPTER 29: PERFORMANCE ANALYSIS

Performance tuning philosophy
Ways to improve performance
Factors that affect performance
Stolen CPU cycles
Analysis of performance problems
System performance checkup
Taking stock of your equipment
Gathering performance data
Analyzing CPU usage
Understanding how the system manages memory
Analyzing memory usage
Analyzing disk I/O
fio: testing storage subsystem performance
sar: collecting and reporting statistics over time
Choosing a Linux I/O scheduler

perf: profiling Linux systems in detail
Help! My server just got really slow!
Recommended reading

CHAPTER 30: DATA CENTER BASICS

Racks
Power
Rack power requirements
kVA vs. kW
Energy efficiency
Metering
Cost
Remote control
Cooling and environment
Cooling load estimation
Hot aisles and cold aisles
Humidity
Environmental monitoring
Data center reliability tiers
Data center security
Location
Perimeter
Facility access
Rack access
Tools
Recommended reading

CHAPTER 31: METHODOLOGY, POLICY, AND POLITICS

The grand unified theory: DevOps
DevOps is CLAMS
System administration in a DevOps world
Ticketing and task management systems
Common functions of ticketing systems
Ticket ownership
User acceptance of ticketing systems
Sample ticketing systems
Ticket dispatching
Local documentation maintenance
Infrastructure as code

- Documentation standards
- Environment separation
- Disaster management
  - Risk assessment
  - Recovery planning
  - Staffing for a disaster
  - Security incidents
- IT policies and procedures
  - The difference between policies and procedures
  - Policy best practices
  - Procedures
- Service level agreements
  - Scope and descriptions of services
  - Queue prioritization policies
  - Conformance measurements
- Compliance: regulations and standards
- Legal issues
  - Privacy
  - Policy enforcement
  - Control = liability
  - Software licenses
- Organizations, conferences, and other resources
- Recommended reading

A Brief History of System Administration

Colophon

About the Contributors

About the Authors

Index

Tribute to Evi

Every field has an avatar who defines and embodies that space. For system administration, that person is Evi Nemeth.

This is the 5th edition of a book that Evi led as an author for almost three decades. Although Evi wasn't able to physically join us in writing this edition, she's with us in spirit and, in some cases, in the form of text and examples that have endured. We've gone to great efforts to maintain Evi's extraordinary style, candor, technical depth, and attention to detail.

An accomplished mathematician and cryptographer, Evi's professional days were spent (most recently) as a computer science professor at the University of Colorado at Boulder. How system administration came into being, and Evi's involvement in it, is detailed in the last chapter of this book, *A Brief History of System Administration*.

Throughout her career, Evi looked forward to retiring and sailing the world. In 2001, she did exactly that: she bought a sailboat *(Wonderland)* and set off on an adventure. Across the years, Evi kept us entertained with stories of amazing islands, cool new people, and other sailing escapades. We produced two editions of this book with Evi anchoring as close as possible to shoreline establishments so that she could camp on their Wi-Fi networks and upload chapter drafts.

Never one to decline an intriguing venture, Evi signed on in June 2013 as crew for the historic schooner *Nina* for a sail across the Tasman Sea. The *Nina* disappeared shortly thereafter in a bad storm, and we haven't heard from Evi since. She was living her dream.

Evi taught us much more than system administration. Even in her 70s, she ran circles around all of us. She was always the best at building a network, configuring a server, debugging a kernel, splitting wood, frying chicken, baking a quiche, or quaffing an occasional glass of wine. With Evi by your side, anything was achievable.

It's impossible to encapsulate all of Evi's wisdom here, but these tenets have stuck with us:

- Be conservative in what you send and liberal in what you receive. (This tenet is also known as Postel's Law, named in honor of Jon Postel, who served as Editor of the RFC series from 1969 until his death in 1998.)
- Be liberal in who you hire, but fire early.
- Don't use weasel words.
- Undergraduates are the secret superpower.
- You can never use too much red ink.
- You don't really understand something until you've implemented it.
- It's always time for sushi.
- Be willing to try something twice.
- Always use **sudo**.

We're sure some readers will write in to ask what, exactly, some of the guidance above really means. We've left that as an exercise for the reader, as Evi would have. You can hear her behind you now, saying "Try it yourself. See how it works."

Smooth sailing, Evi. We miss you.

Preface

Modern technologists are masters at the art of searching Google for answers. If another system administrator has already encountered (and possibly solved) a problem, chances are you can find their write-up on the Internet. We applaud and encourage this open sharing of ideas and solutions.

If great information is already available on the Internet, why write another edition of this book? Here's how this book helps system administrators grow:

- We offer philosophy, guidance, and context for applying technology appropriately. As with the blind men and the elephant, it's important to understand any given problem space from a variety of angles. Valuable perspectives include background on adjacent disciplines such as security, compliance, DevOps, cloud computing, and software development life cycles.
- We take a hands-on approach. Our purpose is to summarize our collective perspective on system administration and to recommend approaches that stand the test of time. This book contains numerous war stories and a wealth of pragmatic advice.
- This is not a book about how to run UNIX or Linux at home, in your garage, or on your smartphone. Instead, we describe the management of production environments such as businesses, government offices, and universities. These environments have requirements that are different from (and far outstrip) those of a typical hobbyist.
- We teach you how to be a professional. Effective system administration requires both technical and "soft" skills. It also requires a sense of humor.

THE ORGANIZATION OF THIS BOOK

This book is divided into four large chunks: Basic Administration, Networking, Storage, and Operations.

Basic Administration presents a broad overview of UNIX and Linux from a system administrator's perspective. The chapters in this section cover most of the facts and techniques needed to run a stand-alone system.

The Networking section describes the protocols used on UNIX systems and the techniques used to set up, extend, and maintain networks and Internet-facing servers. High-level network software is also covered here. Among the featured topics are the Domain Name System, electronic mail, single sign-on, and web hosting.

The Storage section tackles the challenges of storing and managing data. This section also covers subsystems that allow file sharing on a network, such as the Network File System and the Windows-friendly SMB protocol.

The Operations section addresses the key topics that a system administrator faces on a daily basis when managing production environments. These topics include monitoring, security, performance, interactions with developers, and the politics of running a system administration group.

Our contributors

We're delighted to welcome James Garnett, Fabrizio Branca, and Adrian Mouat as contributing authors for this edition. These contributors' deep knowledge of a variety of areas has greatly enriched the content of this book.

Contact information

Please send suggestions, comments, and bug reports to ulsah@book.admin.com. We do answer mail, but please be patient; it is sometimes a few days before one of us is able to respond. Because of the volume of email that this alias receives, we regret that we are unable to answer technical questions.

To view a copy of our current bug list and other late-breaking information, visit our web site, admin.com.

We hope you enjoy this book, and we wish you the best of luck with your adventures in system administration!

Garth Snyder
Trent R. Hein
Ben Whaley
Dan Mackin
July 2017

Foreword

In 1942, Winston Churchill described an early battle of WWII: “this is not the end—it is not even the beginning of the end—but it is, perhaps, the end of the beginning.” I was reminded of these words when I was approached to write this Foreword for the fifth edition of *UNIX and Linux System Administration Handbook*. The loss at sea of Evi Nemeth has been a great sadness for the UNIX community, but I’m pleased to see her legacy endure in the form of this book and in her many contributions to the field of system administration.

The way the world got its Internet was, originally, through UNIX. A remarkable departure from the complex and proprietary operating systems of its day, UNIX was minimalistic, tools-driven, portable, and widely used by people who wanted to share their work with others. What we today call open source software was already pervasive—but nameless—in the early days of UNIX and the Internet. Open source was just how the technical and academic communities did things, because the benefits so obviously outweighed the costs.

Detailed histories of UNIX, Linux, and the Internet have been lovingly presented elsewhere. I bring up these high-level touchpoints only to remind us all that the modern world owes much to open source software and to the Internet, and that the original foundation for this bounty was UNIX.

As early UNIX and Internet companies fought to hire the most brilliant people and to deliver the most innovative features, software portability was often sacrificed. Eventually, system administrators had to know a little bit about a lot of things because no two UNIX-style operating systems (then, or now) were entirely alike. As a working UNIX system administrator in the mid-1980s and later, I had to know not just shell scripting and Sendmail configuration but also kernel device drivers. It was also important to know how to fix a filesystem with an octal debugger. Fun times!

Out of that era came the first edition of this book and all the editions that followed it. In the parlance of the times, we called the authors “Evi and crew” or perhaps “Evi and her kids.” Because of my work on Cron and BIND, Evi spent a week or two with me (and my family, and my workplace) every time an edition of this book was in progress to make sure she was saying

enough, saying nothing wrong, and hopefully, saying something unique and useful about each of those programs. Frankly, being around Evi was exhausting, especially when she was curious about something, or on a deadline, or in my case, both. That having been said, I miss Evi terribly and I treasure every memory and every photograph of her.

In the decades of this book's multiple editions, much has changed. It has been fascinating to watch this book evolve along with UNIX itself. Every new edition omitted some technologies that were no longer interesting or relevant to make room for new topics that were just becoming important to UNIX administrators, or that the authors thought soon would be.

It's hard to believe that we ever spent dozens of kilowatts of power on truck-sized computers whose capabilities are now dwarfed by an Android smartphone. It's equally hard to believe that we used to run hundreds or thousands of individual server and desktop computers with now-antiquated technologies like **rdist**. In those years, various editions of this book helped people like me (and like Evi herself) cope with heterogeneous and sometimes proprietary computers that were each *real* rather than virtualized, and which each had to be *maintained* rather than being reinstalled (or in Docker, rebuilt) every time something needed patching or upgrading.

We adapt, or we exit. The "Evi kids" who carry on Evi's legacy have adapted, and they are back in this fifth edition to tell you what you need to know about how modern UNIX and Linux computers work and how you can make them work the way you want them to. Evi's loss marks the end of an era, but it's also sobering to consider how many aspects of system administration have passed into history alongside her. I know dozens of smart and successful technologists who will never dress cables in the back of an equipment rack, hear the tone of a modem, or see an RS-232 cable. This edition is for those whose systems live in the cloud or in virtualized data centers; those whose administrative work largely takes the form of automation and configuration source code; those who collaborate closely with developers, network engineers, compliance officers, and all the other worker bees who inhabit the modern hive.

You hold in your hand the latest, best edition of a book whose birth and evolution have precisely tracked the birth and evolution of the UNIX and Internet community. Evi would be extremely proud of her kids, both because of this book, and because of who they have each turned out to be. I am proud to know them.

Paul Vixie
La Honda, California
June 2017

Acknowledgments

Many people contributed to this project, bestowing everything from technical reviews and constructive suggestions to overall moral support.

The following individuals deserve special thanks for hanging in there with us: Jason Carolan, Randy Else, Steve Gaede, Asif Khan, Sam Leathers, Ned McClain, Beth McElroy, Paul Nelson, Tim O'Reilly, Madhuri Peri, Dave Roth, Peter Sankauskas, Deepak Singh, and Paul Vixie.

Our editor at Pearson, Mark Taub, deserves huge thanks for his wisdom, patient support, and gentle author herding throughout the production of this book. It's safe to say this edition would not have come to fruition without him.

Mary Lou Nohr has been our relentless behind-the-scenes copy editor for over 20 years. When we started work on this edition, Mary Lou was headed for well-deserved retirement. After a lot of begging and guilt-throwing, she agreed to join us for an encore. (Both Mary Lou Nohr and Evi Nemeth appear on the cover. Can you find them?)

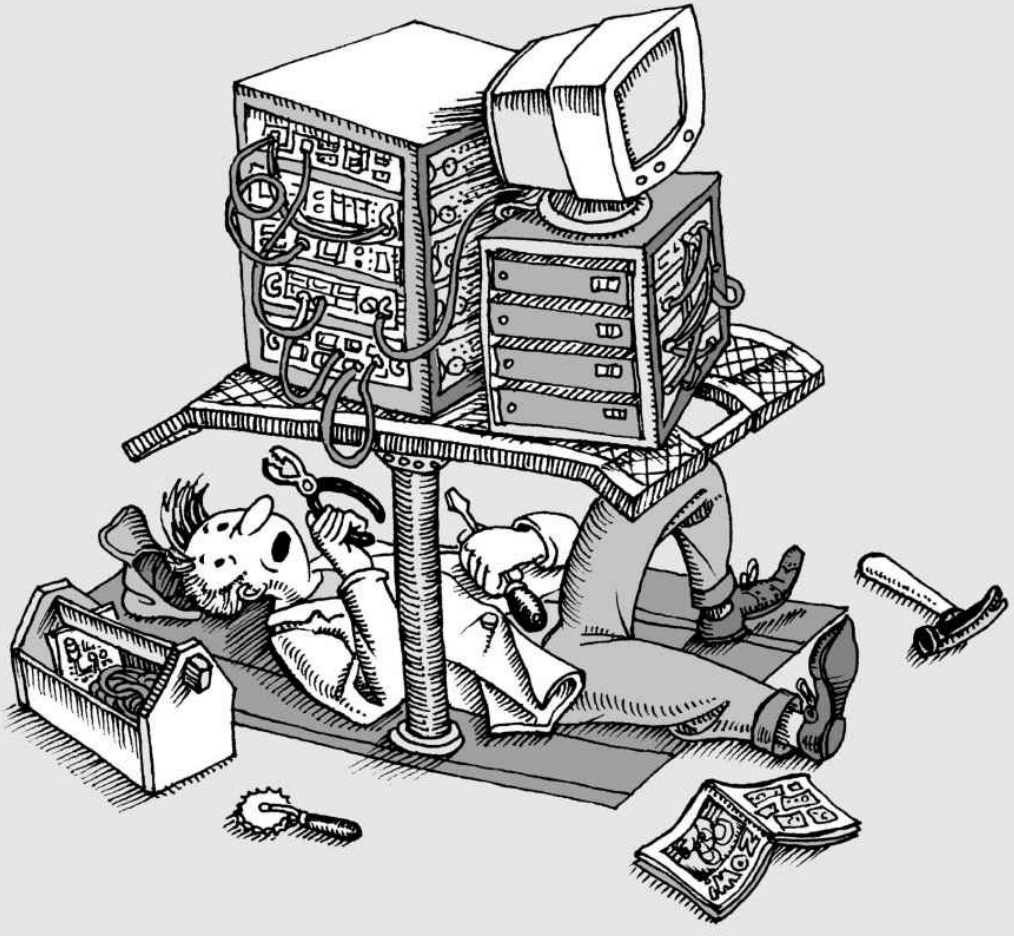
We've had a fantastic team of technical reviewers. Three dedicated souls reviewed the entire book: Jonathan Corbet, Pat Parseghian, and Jennine Townsend. We greatly appreciate their tenacity and tactfulness.

This edition's awesome cartoons and cover were conceived and executed by Lisa Haney. Her portfolio is on-line at lisahaney.com.

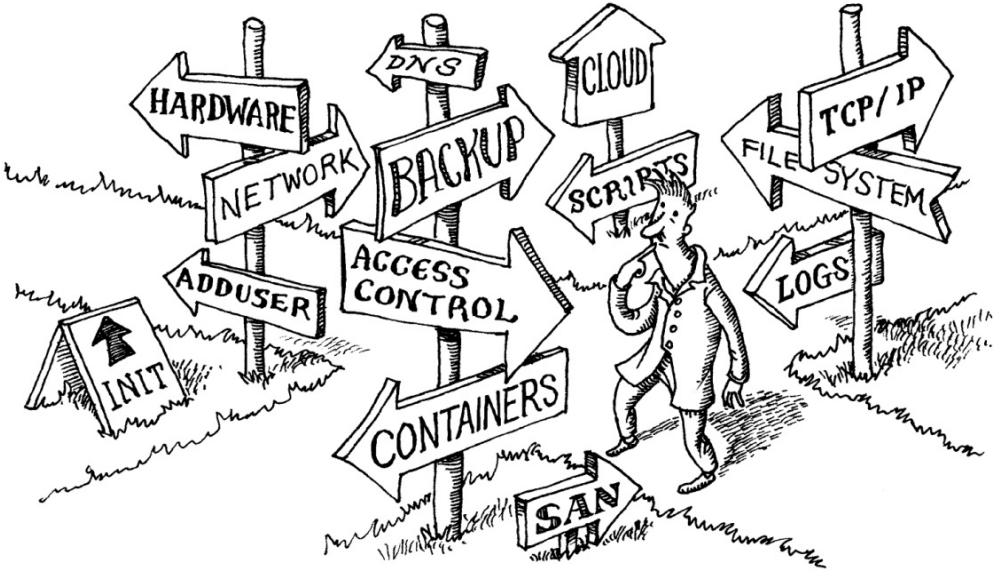
Last but not least, special thanks to Laszlo Nemeth for his willingness to support the continuation of this series.

SECTION ONE

BASIC ADMINISTRATION



1 Where to Start



We've designed this book to occupy a specific niche in the vast ecosystem of man pages, blogs, magazines, books, and other reference materials that address the needs of UNIX and Linux system administrators.

First, it's an orientation guide. It reviews the major administrative systems, identifies the different pieces of each, and explains how they work together. In the many cases where you must choose among various implementations of a concept, we describe the advantages and drawbacks of the most popular options.

Second, it's a quick-reference handbook that summarizes what you need to know to perform common tasks on a variety of common UNIX and Linux systems. For example, the **ps** command, which shows the status of running processes, supports more than 80 command-line options on Linux systems. But a few combinations of options satisfy the majority of a system administrator's needs; we summarize them on this page.

Finally, this book focuses on the administration of enterprise servers and networks. That is, *serious*, *professional* system administration. It's easy to set up a single system; harder to keep a distributed, cloud-based platform running smoothly in the face of viral popularity, network partitions, and targeted attacks. We describe techniques and rules of thumb that help you recover systems from adversity, and we help you choose solutions that scale as your empire grows in size, complexity, and heterogeneity.

We don’t claim to do all of this with perfect objectivity, but we think we’ve made our biases fairly clear throughout the text. One of the interesting things about system administration is that reasonable people can have dramatically different notions of what constitutes the most appropriate solution. We offer our subjective opinions to you as raw data. Decide for yourself how much to accept and how much of our comments apply to your environment.

1.1 Essential duties of a system administrator

The sections below summarize some of the main tasks that administrators are expected to perform. These duties need not necessarily be carried out by a single person, and at many sites the work is distributed among the members of a team. However, at least one person should understand all the components and ensure that every task is performed correctly.

Controlling access

See Chapters 8, 17, and 23 for information about user account provisioning.

The system administrator creates accounts for new users, removes the accounts of inactive users, and handles all the account-related issues that come up in between (e.g., forgotten passwords and lost key pairs). The process of actually adding and removing accounts is typically automated by a configuration management system or centralized directory service.

Adding hardware

Administrators who work with physical hardware (as opposed to cloud or hosted systems) must install it and configure it to be recognized by the operating system. Hardware support chores might range from the simple task of adding a network interface card to configuring a specialized external storage array.

Automating tasks

See Chapter 7, Scripting and the Shell, for information about scripting and automation.

Using tools to automate repetitive and time-consuming tasks increases your efficiency, reduces the likelihood of errors caused by humans, and improves your ability to respond rapidly to changing requirements. Administrators strive to reduce the amount of manual labor needed to keep systems functioning smoothly. Familiarity with scripting languages and automation tools is a large part of the job.

Overseeing backups

See this page for some tips on performing backups.

Backing up data and restoring it successfully when required are important administrative tasks. Although backups are time consuming and boring, the frequency of real-world disasters is simply too high to allow the job to be disregarded.

Operating systems and some individual software packages provide well-established tools and techniques to facilitate backups. Backups must be executed on a regular schedule and restores must be tested periodically to ensure that they are functioning correctly.

Installing and upgrading software

> *See Chapter 6 for information about software management.*

Software must be selected, installed, and configured, often on a variety of operating systems. As patches and security updates are released, they must be tested, reviewed, and incorporated into the local environment without endangering the stability of production systems.

> *See Chapter 26 for information about software deployment and continuous delivery.*

The term “software delivery” refers to the process of releasing updated versions of software—especially software developed in-house—to downstream users. “Continuous delivery” takes this process to the next level by automatically releasing software to users at a regular cadence as it is developed. Administrators help implement robust delivery processes that meet the requirements of the enterprise.

Monitoring

See Chapter 28 for information about monitoring.

Working around a problem is usually faster than taking the time to document and report it, and users internal to an organization often follow the path of least resistance. External users are more likely to voice their complaints publicly than to open a support inquiry. Administrators can help to prevent both of these outcomes by detecting problems and fixing them before public failures occur.

Some monitoring tasks include ensuring that web services respond quickly and correctly, collecting and analyzing log files, and keeping tabs on the availability of server resources such as disk space. All of these are excellent opportunities for automation, and a slew of open source and commercial monitoring systems can help sysadmins with these tasks.

Troubleshooting

See this page for an introduction to network troubleshooting.

Networked systems fail in unexpected and sometimes spectacular fashion. It's the administrator's job to play mechanic by diagnosing problems and calling in subject-matter experts as needed. Finding the source of a problem is often more challenging than resolving it.

Maintaining local documentation

See this page for suggestions regarding documentation.

Administrators choose vendors, write scripts, deploy software, and make many other decisions that may not be immediately obvious or intuitive to others. Thorough and accurate documentation is a blessing for team members who would otherwise need to reverse-engineer a system to resolve problems in the middle of the night. A lovingly crafted network diagram is more useful than many paragraphs of text when describing a design.

Vigilantly monitoring security

See Chapter 27 for more information about security.

Administrators are the first line of defense for protecting network-attached systems. The administrator must implement a security policy and set up procedures to prevent systems from being breached. This responsibility might include only a few basic checks for unauthorized access, or it might involve an elaborate network of traps and auditing programs, depending on the context. System administrators are cautious by nature and are often the primary champions of security across a technical organization.

Tuning performance

See Chapter 29 for more information about performance.

UNIX and Linux are general purpose operating systems that are well suited to almost any conceivable computing task. Administrators can tailor systems for optimal performance in accord with the needs of users, the available infrastructure, and the services the systems provide. When a server is performing poorly, it is the administrator's job to investigate its operation and identify areas that need improvement.

Developing site policies

See the sections starting on this page for information about local policy-making.

For legal and compliance reasons, most sites need policies that govern the acceptable use of computer systems, the management and retention of data, the privacy and security of networks and systems, and other areas of regulatory interest. System administrators often help organizations develop sensible policies that meet the letter and intent of the law and yet still promote progress and productivity.

Working with vendors

Most sites rely on third parties to provide a variety of ancillary services and products related to their computing infrastructure. These providers might include software developers, cloud infrastructure providers, hosted software-as-a-service (SaaS) shops, help-desk support staff, consultants, contractors, security experts, and platform or infrastructure vendors. Administrators may be tasked with selecting vendors, assisting with contract negotiations, and implementing solutions once the paperwork has been completed.

Fire fighting

Although helping other people with their various problems is rarely included in a system administrator's job description, these tasks claim a measurable portion of most administrators' workdays. System administrators are bombarded with problems ranging from "It worked yesterday and now it doesn't! What did you change?" to "I spilled coffee on my keyboard! Should I pour water on it to wash it out?"

In most cases, your response to these issues affects your perceived value as an administrator far more than does any actual technical skill you might possess. You can either howl at the injustice of it all, or you can delight in the fact that a single well-handled trouble ticket scores more brownie points than five hours of midnight debugging. Your choice!

1.2 SUGGESTED BACKGROUND

We assume in this book that you have a certain amount of Linux or UNIX experience. In particular, you should have a general concept of how the system looks and feels from a user's perspective since we do not review that material. Several good books can get you up to speed; see *Recommended reading*.

We love well-designed graphical interfaces. Unfortunately, GUI tools for system administration on UNIX and Linux remain rudimentary in comparison with the richness of the underlying software. In the real world, administrators must be comfortable using the command line.

For text editing, we strongly recommend learning **vi** (now seen more commonly in its enhanced form, **vim**), which is standard on all systems. It is simple, powerful, and efficient. Mastering **vim** is perhaps the single best productivity enhancement available to administrators. Use the **vimtutor** command for an excellent, interactive introduction.

Alternatively, GNU's **nano** is a simple and low-impact "starter editor" that has on-screen prompts. Use it discreetly; professional administrators may be visibly distressed if they witness a peer running **nano**.

See Chapter 7 for an introduction to scripting.

Although administrators are not usually considered software developers, industry trends are blurring the lines between these functions. Capable administrators are usually polyglot programmers who don't mind picking up a new language when the need arises.

For new scripting projects, we recommend Bash (aka **bash**, aka **sh**), Ruby, or Python. Bash is the default command shell on most UNIX and Linux systems. It is primitive as a programming language, but it serves well as the duct tape in an administrative tool box. Python is a clever language with a highly readable syntax, a large developer community, and libraries that facilitate many common tasks. Ruby developers describe the language as "a joy to work with" and "beautiful to behold." Ruby and Python are similar in many ways, and we've found them to be equally functional for administration. The choice between them is mostly a matter of personal preference.

We also suggest that you learn **expect**, which is not a programming language so much as a front end for driving interactive programs. It's an efficient glue technology that can replace some complex scripting and is easy to learn.

Chapter 7, *Scripting and the Shell*, summarizes the most important things to know about scripting for Bash, Python, and Ruby. It also reviews regular expressions (text matching patterns) and some shell idioms that are useful for sysadmins.

1.3 Linux distributions

A Linux distribution comprises the Linux kernel, which is the core of the operating system, and packages that make up all the commands you can run on the system. All distributions share the same kernel lineage, but the format, type, and number of packages differ quite a bit. Distributions also vary in their focus, support, and popularity. There continue to be hundreds of independent Linux distributions, but our sense is that distributions derived from the Debian and Red Hat lineages will predominate in production environments in the years ahead.

By and large, the differences among Linux distributions are not cosmically significant. In fact, it is something of a mystery why so many different distributions exist, each claiming “easy installation” and “a massive software library” as its distinguishing features. It’s hard to avoid the conclusion that people just like to make new Linux distributions.

Most major distributions include a relatively painless installation procedure, a desktop environment, and some form of package management. You can try them out easily by starting up a cloud instance or a local virtual machine.

See Chapter 25, Containers, for more information about Docker and containers.

Much of the insecurity of general-purpose operating systems derives from their complexity. Virtually all leading distributions are cluttered with scores of unused software packages; security vulnerabilities and administrative anguish often come along for the ride. In response, a relatively new breed of minimalist distributions has been gaining traction. CoreOS is leading the charge against the status quo and prefers to run all software in containers. Alpine Linux is a lightweight distribution that is used as the basis of many public Docker images. Given this reductionist trend, we expect the footprint of Linux to shrink over the coming years.

By adopting a distribution, you are making an investment in a particular vendor’s way of doing things. Instead of looking only at the features of the installed software, it’s wise to consider how your organization and that vendor are going to work with each other. Some important questions to ask are:

- Is this distribution going to be around in five years?
- Is this distribution going to stay on top of the latest security patches?
- Does this distribution have an active community and sufficient documentation?
- If I have problems, will the vendor talk to me, and how much will that cost?

Table 1.1 lists some of the most popular mainstream distributions.

Table 1.1: Most popular general-purpose Linux distributions

Distribution	Web site	Comments
Arch	archlinux.org	For those who fear not the command line
CentOS	centos.org	Free analog of Red Hat Enterprise
CoreOS	coreos.com	Containers, containers everywhere
Debian	debian.org	Free as in freedom, most GNUish distro
Fedora	fedoraproject.org	Test bed for Red Hat Linux
Kali	kali.org	For penetration testers
Linux Mint	linuxmint.com	Ubuntu-based, desktop-friendly
openSUSE	opensuse.org	Free analog of SUSE Linux Enterprise
openWRT	openwrt.org	Linux for routers and embedded devices
Oracle Linux	oracle.com	Oracle-supported version of RHEL
RancherOS	rancher.com	20MiB, everything in containers
Red Hat Enterprise	redhat.com	Reliable, slow-changing, commercial
Slackware	slackware.com	Grizzled, long-surviving distro
SUSE Linux Enterprise	suse.com	Strong in Europe, multilingual
Ubuntu	ubuntu.com	Cleaned-up version of Debian

The most viable distributions are not necessarily the most corporate. For example, we expect Debian Linux (OK, OK, Debian GNU/Linux!) to remain viable for a long time despite the fact that Debian is not a company, doesn't sell anything, and offers no enterprise-level support. Debian benefits from a committed group of contributors and from the enormous popularity of the Ubuntu distribution, which is based on it.

A comprehensive list of distributions, including many non-English distributions, can be found at lwn.net/Distributions or distrowatch.com.

1.4 EXAMPLE SYSTEMS USED IN THIS BOOK

We have chosen three popular Linux distributions and one UNIX variant as our primary examples for this book: Debian GNU/Linux, Ubuntu Linux, Red Hat Enterprise Linux (and its dopplegänger CentOS), and FreeBSD. These systems are representative of the overall marketplace and account collectively for a substantial portion of installations in use at large sites today.

Information in this book generally applies to all of our example systems unless a specific attribution is given. Details particular to one system are marked with a logo:


Debian GNU/Linux 9.0 “Stretch”


Ubuntu® 17.04 “Zesty Zapus”



Red Hat® Enterprise Linux® 7.1 and CentOS® 7.1


FreeBSD® 11.0

Most of these marks belong to the vendors that release the corresponding software and are used with the kind permission of their respective owners. However, the vendors have not reviewed or endorsed the contents of this book.

We repeatedly attempted and failed to obtain permission from Red Hat to use their famous red fedora logo, so you’re stuck with yet another technical acronym. At least this one is in the margins.

The paragraphs below provide a bit more detail about each of the example systems.

Example Linux distributions


Information that's specific to Linux but not to any particular distribution is marked with the Tux penguin logo shown at left.


Debian (pronounced *deb-ian*, named after the late founder Ian Murdock and his wife Debra), is one of the oldest and most well-regarded distributions. It is a noncommercial project with more than a thousand contributors worldwide. Debian maintains an ideological commitment to community development and open access, so there's never any question about which parts of the distribution are free or redistributable.

Debian defines three releases that are maintained simultaneously: stable, targeting production servers; unstable, with current packages that may have bugs and security vulnerabilities; and testing, which is somewhere in between.


Ubuntu is based on Debian and maintains Debian's commitment to free and open source software. The business behind Ubuntu is Canonical Ltd., founded by entrepreneur Mark Shuttleworth.

Canonical offers a variety of editions of Ubuntu targeting the cloud, the desktop, and bare metal. There are even releases intended for phones and tablets. Ubuntu version numbers derive from the year and month of release, so version 16.10 is from October, 2016. Each release also has an alliterative code name such as Vivid Vervet or Wily Werewolf.

Two versions of Ubuntu are released annually: one in April and one in October. The April releases in even-numbered years are long-term support (LTS) editions that promise five years of maintenance updates. These are the releases recommended for production use.


Red Hat has been a dominant force in the Linux world for more than two decades, and its distributions are widely used in North America and beyond. By the numbers, Red Hat, Inc., is the most successful open source software company in the world.

Red Hat Enterprise Linux, often shortened to RHEL, targets production environments at large enterprises that require support and consulting services to keep their systems running smoothly. Somewhat paradoxically, RHEL is open source but requires a license. If you're not willing to pay for the license, you're not going to be running Red Hat.

Red Hat also sponsors Fedora, a community-based distribution that serves as an incubator for bleeding-edge software not considered stable enough for RHEL. Fedora is used as the initial test bed for software and configurations that later find their way to RHEL.


CentOS is virtually identical to Red Hat Enterprise Linux, but free of charge. The CentOS Project (centos.org) is owned by Red Hat and employs its lead developers. However, they operate separately from the Red Hat Enterprise Linux team. The CentOS distribution lacks Red Hat's branding and a few proprietary tools, but is in other respects equivalent.

CentOS is an excellent choice for sites that want to deploy a production-oriented distribution without paying tithes to Red Hat. A hybrid approach is also feasible: front-line servers can run Red Hat Enterprise Linux and avail themselves of Red Hat's excellent support, even as nonproduction systems run CentOS. This arrangement covers the important bases in terms of risk and support while also minimizing cost and administrative complexity.

CentOS aspires to full binary and bug-for-bug compatibility with Red Hat Enterprise Linux. Rather than repeating "Red Hat and CentOS" ad nauseam, we generally mention only one or the other in this book. The text applies equally to Red Hat and CentOS unless we note otherwise.

Other popular distributions are also Red Hat descendants. Oracle sells a rebranded and customized version of CentOS to customers of its enterprise database software. Amazon Linux, available to Amazon Web Services users, was initially derived from CentOS and still shares many of its conventions.

Most administrators will encounter a Red Hat-like system at some point in their careers, and familiarity with its nuances is helpful even if it isn't the system of choice at your site.

Example UNIX distribution

The popularity of UNIX has been waning for some time, and most of the stalwart UNIX distributions (e.g., Solaris, HP-UX, and AIX) are no longer in common use. The open source descendants of BSD are exceptions to this trend and continue to enjoy a cult following, particularly among operating system experts, free software evangelists, and security-minded administrators. In other words, some of the world's foremost operating system authorities rely on the various BSD distributions. Apple's macOS has a BSD heritage.



FreeBSD, first released in late 1993, is the most widely used of the BSD derivatives. It commands a 70% market share among BSD variants according to some usage statistics. Users include major Internet companies such as WhatsApp, Google, and Netflix.

Unlike Linux, FreeBSD is a complete operating system, not just a kernel. Both the kernel and userland software are licensed under the permissive BSD License, a fact that encourages development by and additions from the business community.

1.5 Notation and typographical conventions

In this book, filenames, commands, and literal arguments to commands are shown in boldface. Placeholders (e.g., command arguments that should not be taken literally) are in italics. For example, in the command

cp *file directory*

you're supposed to replace *file* and *directory* with the names of an actual file and an actual directory.

Excerpts from configuration files and terminal sessions are shown in a code font. Sometimes, we annotate sessions with the **bash** comment character # and italic text. For example:

```
$ grep Bob /pub/phonelist     # Look up Bob's phone number
Bob Knowles 555-2834
Bob Smith 555-2311
```

We use `$` to denote the shell prompt for a normal, unprivileged user, and `#` for the root user. When a command is specific to a distribution or family of distributions, we prefix the prompt with the distribution name. For example:

```
$ sudo su - root          # Become root
# passwd                  # Change root's password
debian# dpkg -l           # List installed packages on Debian and Ubuntu
```

This convention is aligned with the one used by standard UNIX and Linux shells.

Outside of these specific cases, we have tried to keep special fonts and formatting conventions to a minimum as long as we could do so without compromising intelligibility. For example, we often talk about entities such as the daemon group with no special formatting at all.

We use the same conventions as the manual pages for command syntax:

- Anything between square brackets ("[" and "]") is optional.
- Anything followed by an ellipsis ("…") can be repeated.
- Curly braces ("{" and "}") mean that you should select one of the items separated by vertical bars ("|").

For example, the specification

bork [**-x**] { **on** | **off** } *filename* **...**

would match any of the following commands:

```
bork on /etc/passwd
bork -x off /etc/passwd /etc/smartd.conf
bork off /usr/lib/tmac
```

We use shell-style globbing characters for pattern matching:

- A star (*) matches zero or more characters.
- A question mark (?) matches one character.
- A tilde or “twiddle” (~) means the home directory of the current user.
- ~*user* means the home directory of *user*.

For example, we might refer to the startup script directories **/etc/rc0.d**, **/etc/rc1.d**, and so on with the shorthand pattern **/etc/rc*.d**.

Text within quotation marks often has a precise technical meaning. In these cases, we ignore the normal rules of U.S. English and put punctuation outside the quotes so that there can be no confusion about what’s included and what’s not.

1.6 Units

Metric prefixes such as kilo-, mega-, and giga- are defined as powers of 10; one megabuck is $1,000,000. However, computer types have long poached these prefixes and used them to refer to powers of 2. For example, one “megabyte” of memory is really 2^{20} or 1,048,576 bytes. The stolen units have even made their way into formal standards such as the JEDEC Solid State Technology Association’s Standard 100B.01, which recognizes the prefixes as denoting powers of 2 (albeit with some misgivings).

In an attempt to restore clarity, the International Electrotechnical Commission has defined a set of numeric prefixes (kibi-, mebi-, gibi-, and so on, abbreviated Ki, Mi, and Gi) based explicitly on powers of 2. Those units are always unambiguous, but they are just starting to be widely used. The original kilo-series prefixes are still used in both senses.

Context helps with decoding. RAM is always denominated in powers of 2, but network bandwidth is always a power of 10. Storage space is usually quoted in power-of-10 units, but block and page sizes are in fact powers of 2.

In this book, we use IEC units for powers of 2, metric units for powers of 10, and metric units for rough values and cases in which the exact basis is unclear, undocumented, or impossible to determine. In command output and in excerpts from configuration files, or where the delineation is not important, we leave the original values and unit designators. We abbreviate bit as b and byte as B. Table 1.2 shows some examples.

Table 1.2: Unit decoding examples

Example	Meaning
1kB file	A file that contains 1,000 bytes
4KiB SSD pages	SSD pages that contain 4,096 bytes
8KB of memory	Not used in this book; see note below
100MB file size limit	Nominally 10^8 bytes; in context, ambiguous
100MB disk partition	Nominally 10^8 bytes; in context, probably 99,999,744 bytes[a]
1GiB of RAM	1,073,741,824 bytes of memory
1 Gb/s Ethernet	A network that transmits 1,000,000,000 bits per second
6TB hard disk	A hard disk that stores about 6,000,000,000,000 bytes

a. That is, 10^8 rounded down to the nearest whole multiple of the disk’s 512-byte block size

The abbreviation K, as in “8KB of RAM!”, is not part of any standard. It’s a computerese adaptation of the metric abbreviation k, for kilo-, and originally meant 1,024 as opposed to 1,000. But since the abbreviations for the larger metric prefixes are already upper case, the analogy doesn’t scale. Later, people became confused about the distinction and started using K for factors of 1,000, too.

Most of the world doesn’t consider this to be an important matter and, like the use of imperial units in the United States, metric prefixes are likely to be misused for the foreseeable future. Ubuntu maintains a helpful units policy, though we suspect it has not been widely adopted even at Canonical; see wiki.ubuntu.com/UnitsPolicy for some additional details.

1.7 MAN PAGES AND OTHER ON-LINE DOCUMENTATION

The manual pages, usually called “man pages” because they are read with the **man** command, constitute the traditional “on-line” documentation. (Of course, these days all documentation is on-line in some form or another.) Program-specific man pages come along for the ride when you install new software packages. Even in the age of Google, we continue to consult man pages as an authoritative resource because they are accessible from the command line, typically include complete details on a program’s options, and show helpful examples and related commands.

Man pages are concise descriptions of individual commands, drivers, file formats, or library routines. They do not address more general topics such as “How do I install a new device?” or “Why is this system so damn slow?”

Organization of the man pages

FreeBSD and Linux divide the man pages into sections. Table 1.3 shows the basic schema. Other UNIX variants sometimes define the sections slightly differently.

Table 1.3: Sections of the man pages

Section	Contents
1	User-level commands and applications
2	System calls and kernel error codes
3	Library calls
4	Device drivers and network protocols
5	Standard file formats
6	Games and demonstrations
7	Miscellaneous files and documents
8	System administration commands
9	Obscure kernel specs and interfaces

The exact structure of the sections isn't important for most topics because **man** finds the appropriate page wherever it is stored. Just be aware of the section definitions when a topic with the same name appears in multiple sections. For example, **passwd** is both a command and a configuration file, so it has entries in both section 1 and section 5.

man: read man pages

man *title* formats a specific manual page and sends it to your terminal through **more**, **less**, or whatever program is specified in your PAGER environment variable. *title* is usually a command, device, filename, or name of a library routine. The sections of the manual are searched in roughly numeric order, although sections that describe commands (sections 1 and 8) are usually searched first.

See this page to learn about environment variables.

The form **man** *section title* gets you a man page from a particular section. Thus, on most systems, **man sync** gets you the man page for the **sync** command, and **man 2 sync** gets you the man page for the **sync** system call.

man -k *keyword* or **apropos** *keyword* prints a list of man pages that have *keyword* in their one-line synopses. For example:

```
$ man -k translate
objcopy (1)         - copy and translate object files
dcgettext (3)       - translate message
tr (1)              - translate or delete characters
snmptranslate (1)   - translate SNMP OID values into useful information
tr (1p)             - translate characters
...
```

The keywords database can become outdated. If you add additional man pages to your system, you may need to rebuild this file with **makewhatis** (Red Hat and FreeBSD) or **mandb** (Ubuntu).

Storage of man pages

nroff input for man pages (i.e., the man page source code) is stored in directories under **/usr/share/man** and compressed with **gzip** to save space. The **man** command knows how to decompress them on the fly.

man maintains a cache of formatted pages in **/var/cache/man** or **/usr/share/man** if the appropriate directories are writable; however, this is a security risk. Most systems preformat the man pages once at installation time (see **catman**) or not at all.

The **man** command can search several man page repositories to find the manual pages you request. On Linux systems, you can find out the current default search path with the **manpath** command. This path (from Ubuntu) is typical:

```
ubuntu$ manpath
/usr/local/man:/usr/local/share/man:/usr/share/man
```

If necessary, you can set your MANPATH environment variable to override the default path:

```
$ export MANPATH=/home/share/localman:/usr/share/man
```

Some systems let you set a custom system-wide default search path for man pages, which can be useful if you need to maintain a parallel tree of man pages such as those generated by OpenPKG. To distribute local documentation in the form of man pages, however, it is simpler to use your system's standard packaging mechanism and to put man pages in the standard man directories. See Chapter 6, *Software Installation and Management*, for more details.

1.8 Other authoritative documentation

Man pages are just a small part of the official documentation. Most of the rest, unfortunately, is scattered about on the web.

System-specific guides

Major vendors have their own dedicated documentation projects. Many continue to produce useful book-length manuals, including administration and installation guides. These are generally available on-line and as downloadable PDF files. Table 1.4 shows where to look.

Table 1.4: Where to find OS vendors' proprietary documentation

OS	URL	Comments
Debian	debian.org/doc	Admin handbook lags behind the current version
Ubuntu	help.ubuntu.com	User oriented, see "server guide" for LTS releases
RHEL	redhat.com/docs	Comprehensive docs for administrators
CentOS	wiki.centos.org	Includes tips, HowTos, and FAQs
FreeBSD	freebsd.org/docs.html	See the *FreeBSD Handbook* for sysadmin info

Although this documentation is helpful, it's not the sort of thing you keep next to your bed for light evening reading (though some vendors' versions would make useful sleep aids). We generally Google for answers before turning to vendor docs.

Package-specific documentation

Most of the important software packages in the UNIX and Linux world are maintained by individuals or by third parties such as the Internet Systems Consortium and the Apache Software Foundation. These groups write their own documentation. The quality runs the gamut from embarrassing to spectacular, but jewels such as *Pro Git* from git-scm.com/book make the hunt worthwhile.

Supplemental documents include white papers (technical reports), design rationales, and book- or pamphlet-length treatments of particular topics. These supplemental materials are not limited to describing just one command, so they can adopt a tutorial or procedural approach. Many pieces of software have both a man page and a long-form article. For example, the man page for **vim** tells you about the command-line arguments that **vim** understands, but you have to turn to an in-depth treatment to learn how to actually edit a file.

Most software projects have user and developer mailing lists and IRC channels. This is the first place to visit if you have questions about a specific configuration issue or if you encounter a bug.

Books

The O'Reilly books are favorites in the technology industry. The business began with *UNIX in a Nutshell* and now includes a separate volume on just about every important UNIX and Linux subsystem and command. O'Reilly also publishes books on network protocols, programming languages, Microsoft Windows, and other non-UNIX tech topics. All the books are reasonably priced, timely, and focused.

Many readers turn to O'Reilly's Safari Books Online, a subscription service that offers unlimited electronic access to books, videos, and other learning resources. Content from many publishers is included—not just O'Reilly—and you can choose from an immense library of material.

RFC publications

Request for Comments documents describe the protocols and procedures used on the Internet. Most of these are relatively detailed and technical, but some are written as overviews. The phrase “reference implementation” applied to software usually translates to “implemented by a trusted source according to the RFC specification.”

RFCs are absolutely authoritative, and many are quite useful for system administrators. See this page for a more complete description of these documents. We refer to various RFCs throughout this book.

1.9 Other sources of information

The sources discussed in the previous section are peer reviewed and written by authoritative sources, but they're hardly the last word in UNIX and Linux administration. Countless blogs, discussion forums, and news feeds are available on the Internet.

It should go without saying, but Google is a system administrator's best friend. Unless you're looking up the details of a specific command or file format, Google or an equivalent search engine should be the first resource you consult for any sysadmin question. Make it a habit; if nothing else, you'll avoid the delay and humiliation of having your questions in an on-line forum answered with a link to Google (or worse yet, a link to Google through lmgtfy.com). *When stuck, search the web.*

Keeping current

Operating systems and the tools and techniques that support them change rapidly. Read the sites in Table 1.5 with your morning coffee to keep abreast of industry trends.

Table 1.5: Resources for keeping up to date

Web site	Description
darkreading.com	Security news, trends, and discussion
devopsreactions.tumblr.com	Sysadmin humor in animated GIF form
linux.com	A Linux Foundation site; forum, good for new users
linuxfoundation.org	Nonprofit fostering OSS, employer of Linus Torvalds
lwn.net	High-quality, timely articles on Linux and OSS
lxer.com	Linux news aggregator
securityfocus.com	Vulnerability reports and security-related mailing lists
@SwiftOnSecurity	Infosec opinion from Taylor Swift (parody account)
@nixcraft	Tweets about UNIX and Linux administration
everythingsysadmin.com	Blog of Thomas Limoncelli, respected sysadmin [a]
sysadvent.blogspot.com	Advent for sysadmins with articles each December
oreilly.com/topics	Learning resources from O'Reilly on many topics
schneier.com	Blog of Bruce Schneier, privacy and security expert

a. See also Tom's collection of April Fools' Day RFCs at rfc-humor.com

Social media are also useful. Twitter and reddit in particular have strong, engaged communities with a lot to offer, though the signal-to-noise ratio can sometimes be quite bad. On reddit, join the sysadmin, linux, linuxadmin, and netsec subreddits.

HowTos and reference sites

The sites listed in Table 1.6 contain guides, tutorials, and articles about how to accomplish specific tasks on UNIX and Linux.

Table 1.6: Task-specific forums and reference sites

Web site	Description
wiki.archlinux.org	Articles and guides for Arch Linux; many are more general
askubuntu.com	Q&A for Ubuntu users and developers
digitalocean.com	Tutorials on many OSS, development, and sysadmin topics [a]
kernel.org	Official Linux kernel site
serverfault.com	Collaboratively edited database of sysadmin questions [b]
serversforhackers.com	High-quality videos, forums, and articles on administration

a. See digitalocean.com/community/tutorials
b. Also see the sister site stackoverflow.com, which is dedicated to programming but useful for sysadmins

Stack Overflow and Server Fault, both listed in Table 1.6 (and both members of the Stack Exchange group of sites), warrant a closer look. If you're having a problem, chances are that somebody else has already seen it and asked for help on one of these sites. The reputation-based Q&A format used by the Stack Exchange sites has proved well suited to the kinds of problems that sysadmins and programmers encounter. It's worth creating an account and joining this large community.

Conferences

Industry conferences are a great way to network with other professionals, keep tabs on technology trends, take training classes, gain certifications, and learn about the latest services and products. The number of conferences pertinent to administration has exploded in recent years. Table 1.7 highlights some of the most prominent ones.

Table 1.7: Conferences relevant to system administrators

Conference	Location	When	Description
LISA	Varies	Q4	Large Installation System Administration
Monitorama	Portland	June	Monitoring tools and techniques
OSCON	Varies (US/EU)	Q2 or Q3	Long-running O'Reilly OSS conference
SCALE	Pasadena	Jan	Southern California Linux Expo
DefCon	Las Vegas	July	Oldest and largest hacker convention
Velocity	Global	Varies	O'Reilly conference on web operations
BSDCan	Ottawa	May/June	Everything BSD from novices to gurus
re:Invent	Las Vegas	Q4	AWS cloud computing conference
VMWorld	Varies (US/EU)	Q3 or Q4	Virtualization and cloud computing
LinuxCon	Global	Varies	The future of Linux
RSA	San Francisco	Q1 or Q2	Enterprise cryptography and infosec
DevOpsDays	Global	Varies	A range of topics on bridging the gap between development and ops teams
QCon	Global	Varies	A conference for software developers

Meetups (meetup.com) are another way to network and engage with like-minded people. Most urban areas in the United States and around the world have a Linux user group or DevOps meetup that sponsors speakers, discussions, and hack days.

1.10 Ways to find and install software

Chapter 6, *Software Installation and Management*, addresses software provisioning in detail. But for the impatient, here's a quick primer on how to find out what's installed on your system and how to obtain and install new software.

Modern operating systems divide their contents into packages that can be installed independently of one another. The default installation includes a range of starter packages that you can expand and contract according to your needs. When adding software, don your security hat and remember that additional software creates additional attack surface. Only install what's necessary.

Add-on software is often provided in the form of precompiled packages as well, although the degree to which this is a mainstream approach varies widely among systems. Most software is developed by independent groups that release the software in the form of source code. Package repositories then pick up the source code, compile it appropriately for the conventions in use on the systems they serve, and package the resulting binaries. It's usually easier to install a system-specific binary package than to fetch and compile the original source code. However, packagers are sometimes a release or two behind the current version.

The fact that two systems use the same package format doesn't necessarily mean that packages for the two systems are interchangeable. Red Hat and SUSE both use RPM, for example, but their filesystem layouts are somewhat different. It's best to use packages designed for your particular system if they are available.

Our example systems provide excellent package management systems that include tools for accessing and searching hosted software repositories. Distributors aggressively maintain these repositories on behalf of the community, to facilitate patching and software updates. Life is good.

When the packaged format is insufficient, administrators must install software the old-fashioned way: by downloading a **tar** archive of the source code and manually configuring, compiling, and installing it. Depending on the software and the operating system, this process can range from trivial to nightmarish.

In this book, we generally assume that optional software is already installed rather than torturing you with boilerplate instructions for installing every package. If there's a potential for confusion, we sometimes mention the exact names of the packages needed to complete a particular project. For the most part, however, we don't repeat installation instructions since they tend to be similar from one package to the next.

Determining if software is already installed

For a variety of reasons, it can be a bit tricky to determine which package contains the software you actually need. Rather than starting at the package level, it's easier to use the shell's **which** command to find out if a relevant binary is already in your search path. For example, the following command reveals that the GNU C compiler has already been installed on this machine:

```
ubuntu$ which gcc
/usr/bin/gcc
```

If **which** can't find the command you're looking for, try **whereis**; it searches a broader range of system directories and is independent of your shell's search path.

Another alternative is the incredibly useful **locate** command, which consults a precompiled index of the filesystem to locate filenames that match a particular pattern.

FreeBSD includes **locate** as part of the base system. In Linux, the current implementation of **locate** is in the **mlocate** package. On Red Hat and CentOS, install the **mlocate** package with **yum**; see this page.

locate can find any type of file; it is not specific to commands or packages. For example, if you weren't sure where to find the **signal.h** include file, you could try

```
freebsd$ locate signal.h
/usr/include/machine/signal.h
/usr/include/signal.h
/usr/include/sys/signal.h
...
```

locate's database is updated periodically by the **updatedb** command (in FreeBSD, **locate.updatedb**), which runs periodically out of **cron**. Therefore, the results of a **locate** don't always reflect recent changes to the filesystem.

See Chapter 6 for more information about package management.

If you know the name of the package you're looking for, you can also use your system's packaging utilities to check directly for the package's presence. For example, on a Red Hat system, the following command checks for the presence (and installed version) of the Python interpreter:

```
redhat$ rpm -q python
python-2.7.5-18.el7_1.1.x86_64
```

You can also find out which package a particular file belongs to:

```
redhat$ rpm -qf /etc/httpd
httpd-2.4.6-31.el7.centos.x86_64

freebsd$ pkg which /usr/local/sbin/httpd
/usr/local/sbin/httpd was installed by package apache24-2.4.12

ubuntu$ dpkg-query -S /etc/apache2
apache2: /etc/apache2
```

Adding new software

If you do need to install additional software, you first need to determine the canonical name of the relevant software package. For example, you'd need to translate "I want to install **locate**" to **"**I need to install the **mlocate** package," or translate "I need **named**" to "I have to install BIND." A variety of system-specific indexes on the web can help with this, but Google is usually just as effective. For example, a search for "locate command" takes you directly to several relevant discussions.

The following examples show the installation of the **tcpdump** command on each of our example systems. **tcpdump** is a packet capture tool that lets you view the raw packets being sent to and from the system on the network.

 Debian and Ubuntu use APT, the Debian Advanced Package Tool:

```
ubuntu# sudo apt-get install tcpdump
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following NEW packages will be installed:
  tcpdump
0 upgraded, 1 newly installed, 0 to remove and 81 not upgraded.
Need to get 0 B/360 kB of archives.
After this operation, 1,179 kB of additional disk space will be used.
Selecting previously unselected package tcpdump.
(Reading database ... 63846 files and directories currently installed.)
Preparing to unpack .../tcpdump_4.6.2-4ubuntu1_amd64.deb ...
Unpacking tcpdump (4.6.2-4ubuntu1) ...
Processing triggers for man-db (2.7.0.2-5) ...
Setting up tcpdump (4.6.2-4ubuntu1) ...
```

 The Red Hat and CentOS version is

```
redhat# sudo yum install tcpdump
Loaded plugins: fastestmirror
Determining fastest mirrors
 * base: mirrors.xmission.com
 * epel: linux.mirrors.es.net
 * extras: centos.arvixe.com
 * updates: repos.lax.quadranet.com
Resolving Dependencies
--> Running transaction check
---> Package tcpdump.x86_64 14:4.5.1-2.el7 will be installed
--> Finished Dependency Resolution
tcpdump-4.5.1-2.el7.x86_64.rpm                | 387 kB  00:00
Running transaction check
Running transaction test
Transaction test succeeded
Running transaction
  Installing : 14:tcpdump-4.5.1-2.el7.x86_64   1/1
  Verifying  : 14:tcpdump-4.5.1-2.el7.x86_64   1/1
Installed:
  tcpdump.x86_64 14:4.5.1-2.el7
Complete!
```



The package manager for FreeBSD is **pkg**.

```
freebsd# sudo pkg install -y tcpdump
Updating FreeBSD repository catalogue...
Fetching meta.txz:           100%    944 B   0.9kB/s    00:01
Fetching packagesite.txz:    100%    5 MiB   5.5MB/s    00:01
Processing entries: 100%
FreeBSD repository update completed. 24632 packages processed.
All repositories are up-to-date.
The following 2 package(s) will be affected (of 0 checked):

New packages to be INSTALLED:
    tcpdump: 4.7.4
    libsmi: 0.4.8_1

The process will require 17 MiB more space.
2 MiB to be downloaded.
Fetching tcpdump-4.7.4.txz:   100%  301 KiB 307.7kB/s    00:01
Fetching libsmi-0.4.8_1.txz:  100%    2 MiB   2.0MB/s    00:01
Checking integrity... done (0 conflicting)
[1/2] Installing libsmi-0.4.8_1...
[1/2] Extracting libsmi-0.4.8_1: 100%
[2/2] Installing tcpdump-4.7.4...
[2/2] Extracting tcpdump-4.7.4: 100%
```

Building software from source code

As an illustration, here's how you build a version of **tcpdump** from the source code.

The first chore is to identify the code. Software maintainers sometimes keep an index of releases on the project's web site that are downloadable as tarballs. For open source projects, you're most likely to find the code in a Git repository.

The **tcpdump** source is kept on GitHub. Clone the repository in the **/tmp** directory, create a branch of the tagged version you want to build, then unpack, configure, build, and install it:

```
redhat$ cd /tmp
redhat$ git clone https://github.com/the-tcpdump-group/tcpdump.git
<status messages as repository is cloned>
redhat$ cd tcpdump
redhat$ git checkout tags/tcpdump-4.7.4 -b tcpdump-4.7.4
Switched to a new branch 'tcpdump-4.7.4'
redhat$ ./configure
checking build system type... x86_64-unknown-linux-gnu
checking host system type... x86_64-unknown-linux-gnu
checking for gcc... gcc
checking whether the C compiler works... yes
...
redhat$ make
<several pages of compilation output>
redhat$ sudo make install
<files are moved in to place>
```

This **configure**/**make**/**make install** sequence is common to most software written in C and works on all UNIX and Linux systems. It's always a good idea to check the package's **INSTALL** or **README** file for specifics. You must have the development environment and any package-specific prerequisites installed. (In the case of **tcpdump**, **libpcap** and its libraries are prerequisites.)

You'll often need to tweak the build configuration, so use **./configure** --**help** to see the options available for each particular package. Another useful **configure** option is --**prefix**=*directory*, which lets you compile the software for installation somewhere other than **/usr/local**, which is usually the default.

Installing from a web script

Cross-platform software bundles increasingly offer an expedited installation process that's driven by a shell script you download from the web with **curl**, **fetch**, or **wget**. These are all simple HTTP clients that download the contents of a URL to a local file or, optionally, print the contents to their standard output. For example, to set up a machine as a Salt client, you can run the following commands:

```
$ curl -o /tmp/saltboot -sL https://bootstrap.saltstack.com
$ sudo sh /tmp/saltboot
```

The bootstrap script investigates the local environment, then downloads, installs, and configures an appropriate version of the software. This type of installation is particularly common in cases where the process itself is somewhat complex, but the vendor is highly motivated to make things easy for users. (Another good example is RVM; see this page.)

See Chapter 6 for more information about package installation.

This installation method is perfectly fine, but it raises a couple of issues that are worth mentioning. To begin with, it leaves no proper record of the installation for future reference. If your operating system offers a packagized version of the software, it's usually preferable to install the package instead of running a web installer. Packages are easy to track, upgrade, and remove. (On the other hand, most OS-level packages are out of date. You probably won't end up with the most current version of the software.)

See this page for details on HTTPS's chain of trust.

Be very suspicious if the URL of the boot script is not secure (that is, it does not start with https:). Unsecured HTTP is trivial to hijack, and installation URLs are of particular interest to hackers because they know you're likely to run, as root, whatever code comes back. By contrast, HTTPS validates the identity of the server through a cryptographic chain of trust. Not foolproof, but reliable enough.

A few vendors publicize an HTTP installation URL that automatically redirects to an HTTPS version. This is dumb and is in fact no more secure than straight-up HTTP. There's nothing to prevent the initial HTTP exchange from being intercepted, so you might never reach the vendor's redirect. However, the existence of such redirects does mean it's worth trying your own substitution of https for http in insecure URLs. More often than not, it works just fine.

The shell accepts script text on its standard input, and this feature enables tidy, one-line installation procedures such as the following:

```
$ curl -L https://badvendor.com | sudo sh
```

However, there's a potential issue with this construction in that the root shell still runs even if **curl** outputs a partial script and then fails—say, because of a transient network glitch. The end result is unpredictable and potentially not good.

We are not aware of any documented cases of problems attributable to this cause. Nevertheless, it is a plausible failure mode. More to the point, piping the output of **curl** to a shell has entered the collective sysadmin unconscious as a prototypical rookie blunder, so if you must do it, at least keep it on the sly.

The fix is easy: just save the script to a temporary file, then run the script in a separate step after the download successfully completes.

1.11 Where to host

Operating systems and software can be hosted in private data centers, at co-location facilities, on a cloud platform, or on some combination of these. Most burgeoning startups choose the cloud. Established enterprises are likely to have existing data centers and may run a private cloud internally.

The most practical choice, and our recommendation for new projects, is a public cloud provider. These facilities offer numerous advantages over data centers:

- No capital expenses and low initial operating costs
- No need to install, secure, and manage hardware
- On-demand adjustment of storage, bandwidth, and compute capacity
- Ready-made solutions for common ancillary needs such as databases, load balancers, queues, monitoring, and more
- Cheaper and simpler implementation of highly available/redundant systems

Early cloud systems acquired a reputation for inferior security and performance, but these are no longer major concerns. These days, most of our administration work is in the cloud. See Chapter 9 for a general introduction to this space.

Our preferred cloud platform is the leader in the space: Amazon Web Services (AWS). Gartner, a leading technology research firm, found that AWS is ten times the size of all competitors combined. AWS innovates rapidly and offers a much broader array of services than does any other provider. It also has a reputation for excellent customer service and supports a large and engaged community. AWS offers a free service tier to cut your teeth on, including a year's use of a low powered cloud server.

Google Cloud Platform (GCP) is aggressively improving and marketing its products. Some claim that its technology is unmatched by other providers. GCP's growth has been slow, in part due to Google's reputation for dropping support for popular offerings. However, its customer-friendly pricing terms and unique features are appealing differentiators.

DigitalOcean is a simpler service with a stated goal of high performance. Its target market is developers, whom it woos with a clean API, low pricing, and extremely fast boot times. DigitalOcean is a strong proponent of open source software, and their tutorials and guides for popular Internet technologies are some of the best available.

1.12 Specialization and adjacent disciplines

System administrators do not exist in a vacuum; a team of experts is required to build and maintain a complex network. This section describes some of the roles with which system administrators overlap in skills and scope. Some administrators choose to specialize in one or more of these areas.

Your goal as a system administrator, or as a professional working in any of these related areas, is to achieve the objectives of the organization. Avoid letting politics or hierarchy interfere with progress. The best administrators solve problems and share information freely with others.

DevOps

See this page for more comments on DevOps.

DevOps is not so much a specific function as a culture or operational philosophy. It aims to improve the efficiency of building and delivering software, especially at large sites that have many interrelated services and teams. Organizations with a DevOps practice promote integration among engineering teams and may draw little or no distinction between development and operations. Experts who work in this area seek out inefficient processes and replace them with small shell scripts or large and unwieldy Chef repositories.

Site reliability engineers

Site reliability engineers value uptime and correctness above all else. Monitoring networks, deploying production software, taking pager duty, planning future expansion, and debugging outages all lie within the realm of these availability crusaders. Single points of failure are site reliability engineers' nemeses.

Security operations engineers

Security operations engineers focus on the practical, day-to-day side of an information security program. These folks install and operate tools that search for vulnerabilities and monitor for attacks on the network. They also participate in attack simulations to gauge the effectiveness of their prevention and detection techniques.

Network administrators

Network administrators design, install, configure, and operate networks. Sites that operate data centers are most likely to employ network administrators; that's because these facilities have a variety of physical switches, routers, firewalls, and other devices that need management. Cloud platforms also offer a variety of networking options, but these usually don't require a dedicated administrator because most of the work is handled by the provider.

Database administrators

Database administrators (sometimes known as DBAs) are experts at installing and managing database software. They manage database schemas, perform installations and upgrades, configure clustering, tune settings for optimal performance, and help users formulate efficient queries. DBAs are usually wizards with one or more query languages and have experience with both relational and nonrelational (NoSQL) databases.

Network operations center (NOC) engineers

NOC engineers monitor the real-time health of large sites and track incidents and outages. They troubleshoot tickets from users, perform routine upgrades, and coordinate actions among other teams. They can most often be found watching a wall of monitors that show graphs and measurements.

Data center technicians

Data center technicians work with hardware. They receive new equipment, track equipment inventory and life cycles, install servers in racks, run cabling, maintain power and air conditioning, and handle the daily operations of a data center. As a system administrator, it’s in your best interest to befriend data center technicians and bribe them with coffee, caffeinated soft drinks, and alcoholic beverages.

Architects

Systems architects have deep expertise in more than one area. They use their experience to design distributed systems. Their job descriptions may include defining security zones and segmentation, eliminating single points of failure, planning for future growth, ensuring connectivity among multiple networks and third parties, and other site-wide decision making. Good architects are technically proficient and generally prefer to implement and test their own designs.

1.13 RECOMMENDED READING

ABBOTT, MARTIN L., AND MICHAEL T. FISHER. *The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise (2nd Edition)*. Addison-Wesley Professional, 2015.

GANCARZ, MIKE. *Linux and the Unix Philosophy*. Boston: Digital Press, 2003.

LIMONCELLI, THOMAS A., AND PETER SALUS. *The Complete April Fools' Day RFCs.* Peer-to-Peer Communications LLC. 2007. Engineering humor. You can read this collection on-line for free at rfc-humor.com.

RAYMOND, ERIC S. *The Cathedral & The Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. Sebastopol, CA: O'Reilly Media, 2001.

SALUS, PETER H. *The Daemon, the GNU & the Penguin: How Free and Open Software is Changing the World*. Reed Media Services, 2008. This fascinating history of the open source movement by UNIX's best-known historian is also available at groklaw.com under the Creative Commons license. The URL for the book itself is quite long; look for a current link at groklaw.com or try this compressed equivalent: tinyurl.com/d6u7j.

SIEVER, ELLEN, STEPHEN FIGGINS, ROBERT LOVE, AND ARNOLD ROBBINS. *Linux in a Nutshell (6th Edition)*. Sebastopol, CA: O'Reilly Media, 2009.

System administration and DevOps

KIM, GENE, KEVIN BEHR, AND GEORGE SPAFFORD. *The Phoenix Project: A Novel about IT, DevOps, and Helping Your Business Win.* Portland, OR: IT Revolution Press, 2014. A guide to the philosophy and mindset needed to run a modern IT organization, written as a narrative. An instant classic.

KIM, GENE, JEZ HUMBLE, PATRICK DEBOIS, AND JOHN WILLIS. *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations.* Portland, OR: IT Revolution Press, 2016.

LIMONCELLI, THOMAS A., CHRISTINA J. HOGAN, AND STRATA R. CHALUP. *The Practice of System and Network Administration (2nd Edition).* Reading, MA: Addison-Wesley, 2008. This is a good book with particularly strong coverage of the policy and procedural aspects of system administration. The authors maintain a system administration blog at everythingsysadmin.com.

LIMONCELLI, THOMAS A., CHRISTINA J. HOGAN, AND STRATA R. CHALUP. *The Practice of Cloud System Administration.* Reading, MA: Addison-Wesley, 2014. From the same authors as the previous title, now with a focus on distributed systems and cloud computing.

Essential tools

Blum, Richard, and Christine Bresnahan. *Linux Command Line and Shell Scripting Bible (3rd Edition).* Wiley, 2015.

Dougherty, Dale, and Arnold Robins. *Sed & Awk (2nd Edition).* Sebastopol, CA: O'Reilly Media, 1997. Classic O'Reilly book on the powerful, indispensable text processors **sed** and **awk**.

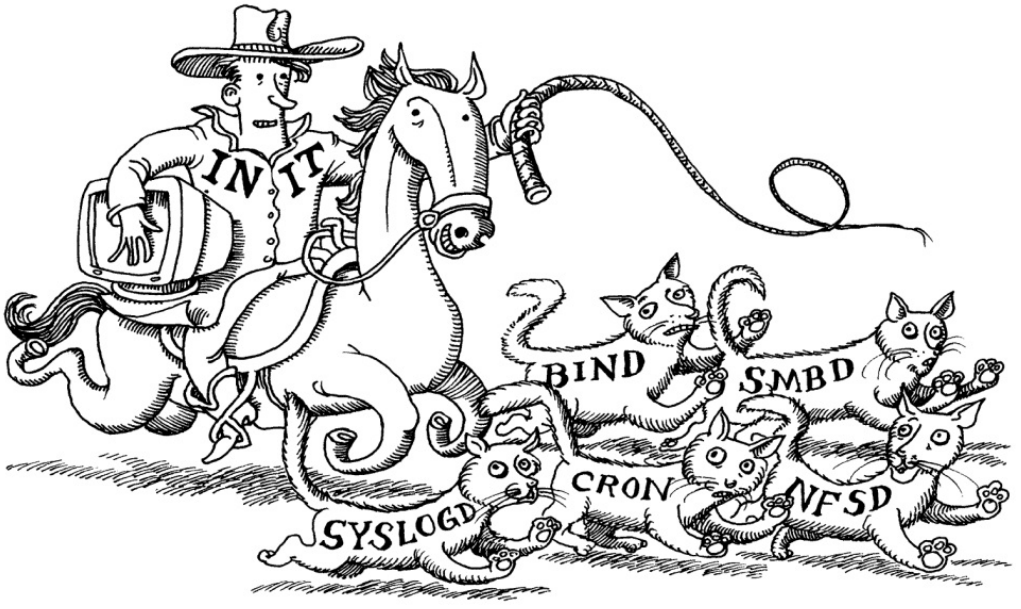
Kim, Peter. *The Hacker Playbook 2: Practical Guide To Penetration Testing.* CreateSpace Independent Publishing Platform, 2015.

Neil, Drew. *Practical Vim: Edit Text at the Speed of Thought*. Pragmatic Bookshelf, 2012.

Shotts, William E. *The Linux Command Line: A Complete Introduction.* San Francisco, CA: No Starch Press, 2012.

Sweigart, Al. *Automate the Boring Stuff with Python: Practical Programming for Total Beginners.* San Francisco, CA: No Starch Press, 2015.

2 Booting and System Management Daemons



“Booting” is the standard term for “starting up a computer.” It’s a shortened form of the word “bootstrapping,” which derives from the notion that the computer has to “pull itself up by its own bootstraps.”

The boot process consists of a few broadly defined tasks:

- Finding, loading, and running bootstrapping code
- Finding, loading, and running the OS kernel
- Running startup scripts and system daemons
- Maintaining process hygiene and managing system state transitions

The activities included in that last bullet point continue as long as the system remains up, so the line between bootstrapping and normal operation is inherently a bit blurry.

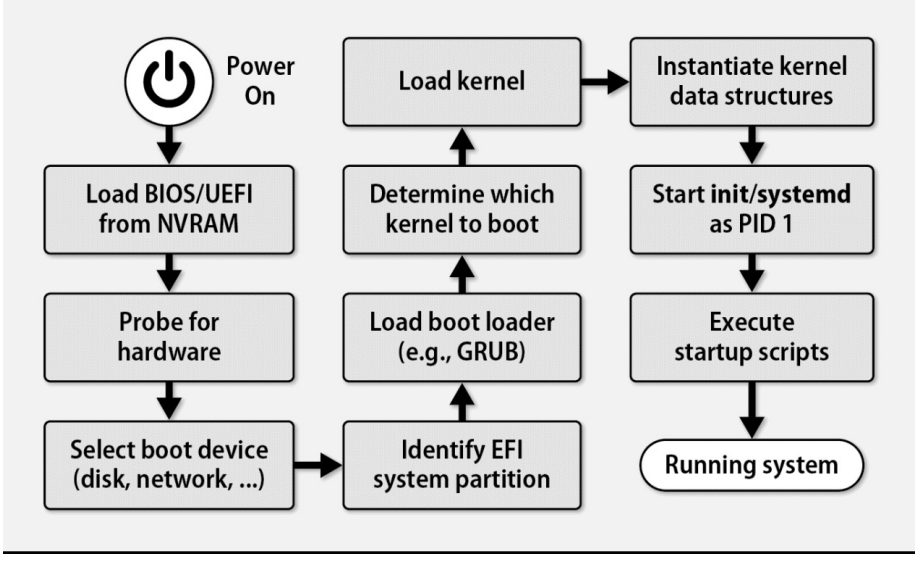
2.1 Boot process overview

Startup procedures have changed a lot in recent years. The advent of modern (UEFI) BIOSs has simplified the early stages of booting, at least from a conceptual standpoint. In later stages, most Linux distributions now use a system manager daemon called **systemd** instead of the traditional UNIX **init**. **systemd** streamlines the boot process by adding dependency management, support for concurrent startup processes, and a comprehensive approach to logging, among other features.

Boot management has also changed as systems have migrated into the cloud. The drift toward virtualization, cloud instances, and containerization has reduced the need for administrators to touch physical hardware. Instead, we now have image management, APIs, and control panels.

During bootstrapping, the kernel is loaded into memory and begins to execute. A variety of initialization tasks are performed, and the system is then made available to users. The general overview of this process is shown in Exhibit A.

Exhibit A: Linux & UNIX boot process



Administrators have little direct, interactive control over most of the steps required to boot a system. Instead, admins can modify bootstrap configurations by editing config files for the system startup scripts or by changing the arguments the boot loader passes to the kernel.

Before the system is fully booted, filesystems must be checked and mounted and system daemons started. These procedures are managed by a series of shell scripts (sometimes called

“**init** scripts”) or unit files that are run in sequence by **init** or parsed by **systemd**. The exact layout of the startup scripts and the manner in which they are executed varies among systems. We cover the details later in this chapter.

2.2 System firmware

When a machine is powered on, the CPU is hardwired to execute boot code stored in ROM. On virtualized systems, this "ROM" may be imaginary, but the concept remains the same.

The system firmware typically knows about all the devices that live on the motherboard, such as SATA controllers, network interfaces, USB controllers, and sensors for power and temperature. (Virtual systems pretend to have this same set of devices.) In addition to allowing hardware-level configuration of these devices, the firmware lets you either expose them to the operating system or disable and hide them.

On physical (as opposed to virtualized) hardware, most firmware offers a user interface. However, it's generally crude and a bit tricky to access. You need control of the computer and console, and must press a particular key immediately after powering on the system. Unfortunately, the identity of the magic key varies by manufacturer; see if you can glimpse a cryptic line of instructions at the instant the system first powers on. (You might find it helpful to disable the monitor's power management features temporarily.) Barring that, try Delete, Control, F6, F8, F10, or F11. For the best chance of success, tap the key several times, then hold it down.

During normal bootstrapping, the system firmware probes for hardware and disks, runs a simple set of health checks, and then looks for the next stage of bootstrapping code. The firmware UI lets you designate a boot device, usually by prioritizing a list of available options (e.g., "try to boot from the DVD drive, then a USB drive, then a hard disk").

In most cases, the system's disk drives populate a secondary priority list. To boot from a particular drive, you must both set it as the highest-priority disk and make sure that "hard disk" is enabled as a boot medium.

BIOS vs. UEFI

Traditional PC firmware was called the BIOS, for Basic Input/Output System. Over the last decade, however, BIOS has been supplanted by a more formalized and modern standard, the Unified Extensible Firmware Interface (UEFI). You'll often see UEFI referred to as "UEFI BIOS," but for clarity, we'll reserve the term BIOS for the legacy standard in this chapter. Most systems that implement UEFI can fall back to a legacy BIOS implementation if the operating system they're booting doesn't support UEFI.

UEFI is the current revision of an earlier standard, EFI. References to the name EFI persist in some older documentation and even in some standard terms, such as "EFI system partition." In all but the most technically explicit situations, you can treat these terms as equivalent.

UEFI support is pretty much universal on new PC hardware these days, but plenty of BIOS systems remain in the field. Moreover, virtualized environments often adopt BIOS as their underlying boot mechanism, so the BIOS world isn't in danger of extinction just yet.

As much as we'd prefer to ignore BIOS and just talk about UEFI, it's likely that you'll encounter both types of systems for years to come. UEFI also builds-in several accommodations to the old BIOS regime, so a working knowledge of BIOS can be quite helpful for deciphering the UEFI documentation.

Legacy BIOS

Partitioning is a way to subdivide physical disks. See this page for a more detailed discussion.

Traditional BIOS assumes that the boot device starts with a record called the MBR (Master Boot Record). The MBR includes both a first-stage boot loader (aka “boot block”) and a primitive disk partitioning table. The amount of space available for the boot loader is so small (less than 512 bytes) that it’s not able to do much other than load and run a second-stage boot loader.

Neither the boot block nor the BIOS is sophisticated enough to read any type of standard filesystem, so the second-stage boot loader must be kept somewhere easy to find. In one typical scenario, the boot block reads the partitioning information from the MBR and identifies the disk partition marked as “active.” It then reads and executes the second-stage boot loader from the beginning of that partition. This scheme is known as a volume boot record.

Alternatively, the second-stage boot loader can live in the dead zone that lies between the MBR and the beginning of the first disk partition. For historical reasons, the first partition doesn’t start until the 64th disk block, so this zone normally contains at least 32KB of storage: still not a lot, but enough to store a filesystem driver. This storage scheme is commonly used by the GRUB boot loader; see *GRUB: the GRand Unified Boot loader*.

To effect a successful boot, all components of the boot chain must be properly installed and compatible with one another. The MBR boot block is OS-agnostic, but because it assumes a particular location for the second stage, there may be multiple versions that can be installed. The second-stage loader is generally knowledgeable about operating systems and filesystems (it may support several of each), and usually has configuration options of its own.

UEFI

See this page for more information about GPT partitions.

The UEFI specification includes a modern disk partitioning scheme known as GPT (GUID Partition Table, where GUID stands for “globally unique identifier”). UEFI also understands FAT (File Allocation Table) filesystems, a simple but functional layout that originated in MS-DOS. These features combine to define the concept of an EFI System Partition (ESP). At boot time, the firmware consults the GPT partition table to identify the ESP. It then reads the configured target application directly from a file in the ESP and executes it.

Because the ESP is just a generic FAT filesystem, it can be mounted, read, written, and maintained by any operating system. No “mystery meat” boot blocks are required anywhere on the disk. (Truth be told, UEFI does maintain an MBR-compatible record at the beginning of each disk to facilitate interoperability with BIOS systems. BIOS systems can’t see the full GPT-style partition table, but they at least recognize the disk as having been formatted. Be careful not to run MBR-specific administrative tools on GPT disks. They may think they understand the disk layout, but they do not.)

In the UEFI system, no boot loader at all is technically required. The UEFI boot target can be a UNIX or Linux kernel that has been configured for direct UEFI loading, thus effecting a loader-less bootstrap. In practice, though, most systems still use a boot loader, partly because that makes it easier to maintain compatibility with legacy BIOSes.

UEFI saves the pathname to load from the ESP as a configuration parameter. With no configuration, it looks for a standard path, usually **/efi/boot/bootx64.efi** on modern Intel systems. A more typical path on a configured system (this one for Ubuntu and the GRUB boot loader) would be **/efi/ubuntu/grubx64.efi**. Other distributions follow a similar convention.

UEFI defines standard APIs for accessing the system’s hardware. In this respect, it’s something of a miniature operating system in its own right. It even provides for UEFI-level add-on device drivers, which are written in a processor-independent language and stored in the ESP. Operating systems can use the UEFI interface, or they can take over direct control of the hardware.

Because UEFI has a formal API, you can examine and modify UEFI variables (including boot menu entries) on a running system. For example, **efibootmgr -v** shows the following summary of the boot configuration:

```
$ efibootmgr -v
BootCurrent: 0004
BootOrder: 0000,0001,0002,0004,0003
Boot0000* EFI DVD/CDROM PciRoot(0x0)/Pci(0x1f,0x2)/Sata(1,0,0)
Boot0001* EFI Hard Drive PciRoot(0x0)/Pci(0x1f,0x2)/Sata(0,0,0)
Boot0002* EFI Network PciRoot(0x0)/Pci(0x5,0x0)/MAC(001c42fb5baf,0)
Boot0003* EFI Internal Shell MemoryMapped(11,0x7ed5d000,0x7f0dcfff)/
    FvFile(c57ad6b7-0515-40a8-9d21-551652854e37)
Boot0004* ubuntu HD(1,GPT,020c8d3e-fd8c-4880-9b61-
    ef4cffc3d76c,0x800,0x100000)/File(\EFI\ubuntu\shimx64.efi)
```

efibootmgr lets you change the boot order, select the next configured boot option, or even create and destroy boot entries. For example, to set the boot order to try the system drive before trying the network, and to ignore other boot options, we could use the command

```
$ sudo efibootmgr -o 0004,0002
```

Here, we're specifying the options Boot0004 and Boot0002 from the output above.

The ability to modify the UEFI configuration from user space means that the firmware's configuration information is mounted read/write—a blessing and a curse. On systems (typically, those with **systemd**) that allow write access by default, **rm -rf /** can be enough to permanently destroy the system at the firmware level; in addition to removing files, **rm** also removes variables and other UEFI information accessible through **/sys**. Yikes! Don't try this at home! (See goo.gl/QMSiSG, a link to a Phoronix article, for some additional details.)

2.3 Boot loaders

Most bootstrapping procedures include the execution of a boot loader that is distinct from both the BIOS/UEFI code and the OS kernel. It's also separate from the initial boot block on a BIOS system, if you're counting steps.

The boot loader's main job is to identify and load an appropriate operating system kernel. Most boot loaders can also present a boot-time user interface that lets you select which of several possible kernels or operating systems to invoke.

Another task that falls to the boot loader is the marshaling of configuration arguments for the kernel. The kernel doesn't have a command line per se, but its startup option handling will seem eerily similar from the shell. For example, the argument **single** or **-s** usually tells the kernel to enter single-user mode instead of completing the normal boot process.

Such options can be hard-wired into the boot loader's configuration if you want them used on every boot, or they can be provided on the fly through the boot loader's UI.

In the next few sections, we discuss GRUB (the Linux world's predominant boot loader) and the boot loaders used with FreeBSD.

2.4 GRUB: THE GRAND UNIFIED BOOT LOADER



GRUB, developed by the GNU Project, is the default boot loader on most Linux distributions. The GRUB lineage has two main branches: the original GRUB, now called GRUB Legacy, and the newer, extra-crispy GRUB 2, which is the current standard. Make sure you know which GRUB you're dealing with, as the two versions are quite different.

GRUB 2 has been the default boot manager for Ubuntu since version 9.10, and it recently became the default for RHEL 7. All our example Linux distributions use it as their default. In this book we discuss only GRUB 2, and we refer to it simply as GRUB.

FreeBSD has its own boot loader (covered in more detail in *The FreeBSD boot process*). However, GRUB is perfectly happy to boot FreeBSD, too. This might be an advantageous configuration if you're planning to boot multiple operating systems on a single computer. Otherwise, the FreeBSD boot loader is more than adequate.

GRUB configuration

See this page for more about operating modes.

GRUB lets you specify parameters such as the kernel to boot (specified as a GRUB "menu entry") and the operating mode to boot into.

Since this configuration information is needed at boot time, you might imagine that it would be stored somewhere strange, such as the system's NVRAM or the disk blocks reserved for the boot loader. In fact, GRUB understands most of the filesystems in common use and can usually find its way to the root filesystem on its own. This feat lets GRUB read its configuration from a regular text file.

The config file is called **grub.cfg**, and it's usually kept in **/boot/grub** (**/boot/grub2** in Red Hat and CentOS) along with a selection of other resources and code modules that GRUB might need to access at boot time. Changing the boot configuration is a simple matter of updating the **grub.cfg** file.

Although you can create the **grub.cfg** file yourself, it's more common to generate it with the **grub-mkconfig** utility, which is called **grub2-mkconfig** on Red Hat and CentOS and wrapped as **update-grub** on Debian and Ubuntu. In fact, most distributions assume that **grub.cfg** can be regenerated at will, and they do so automatically after updates. If you don't take steps to prevent this, your handcrafted **grub.cfg** file will get clobbered.

As with all things Linux, distributions configure **grub-mkconfig** in a variety of ways. Most commonly, the configuration is specified in **/etc/default/grub** in the form of **sh** variable assignments. Table 2.1 shows some of the commonly modified options.

Table 2.1: Common GRUB configuration options from /etc/default/grub

Shell variable name	Contents or function
GRUB_BACKGROUND	Background image [a]
GRUB_CMDLINE_LINUX	Kernel parameters to add to menu entries for Linux [b]
GRUB_DEFAULT	Number or title of the default menu entry
GRUB_DISABLE_RECOVERY	Prevents the generation of recovery mode entries
GRUB_PRELOAD_MODULES	List of GRUB modules to be loaded as early as possible
GRUB_TIMEOUT	Seconds to display the boot menu before autoboot

a. The background image must be a **.png**, **.tga**, **.jpg**, or **.jpeg** file.
b. Table 2.3 lists some of the available options.

After editing **/etc/default/grub**, run **update-grub** or **grub2-mkconfig** to translate your configuration into a proper **grub.cfg** file. As part of the configuration-building process, these

commands inventory the system's bootable kernels, so they can be useful to run after you make kernel changes even if you haven't explicitly changed the GRUB configuration.

You may need to edit the **/etc/grub.d/40_custom** file to change the order in which kernels are listed in the boot menu (after you create a custom kernel, for example), set a boot password, or change the names of boot menu items. As usual, run **update-grub** or **grub2-mkconfig** after making changes.

As an example, here's a **40_custom** file that invokes a custom kernel on an Ubuntu system:

```
#!/bin/sh

exec tail -n +3 $0

# This file provides an easy way to add custom menu entries. Just type
# the menu entries you want to add after this comment. Be careful not to
# change the 'exec tail' line above.

menuentry 'My Awesome Kernel' {
    set root='(hd0,msdos1)'
    linux    /awesome_kernel root=UUID=XXX-XXX-XXX ro quiet
    initrd   /initrd.img-awesome_kernel
}
```

See this page for more information about mounting filesystems.

In this example, GRUB loads the kernel from **/awesome_kernel**. Kernel paths are relative to the boot partition, which historically was mounted as **/boot** but with the advent of UEFI now is likely an unmounted EFI System Partition. Use **gpart show** and **mount** to examine your disk and determine the state of the boot partition.

The GRUB command line

GRUB supports a command-line interface for editing config file entries on the fly at boot time. To enter command-line mode, type **c** at the GRUB boot screen.

From the command line, you can boot operating systems that aren't listed in the **grub.cfg** file**,** display system information, and perform rudimentary filesystem testing. Anything that can be done through **grub.cfg** can also be done through the command line.

Press the <Tab> key to see a list of possible commands. Table 2.2 shows some of the more useful ones.

Table 2.2: GRUB commands

Cmd	Function
boot	Boots the system from the specified kernel image
help	Gets interactive help for a command
linux	Loads a Linux kernel
reboot	Reboots the system
search	Searches devices by file, filesystem label, or UUID
usb	Tests USB support

For detailed information about GRUB and its command-line options, refer to the official manual at gnu.org/software/grub/manual.

Linux kernel options

See Chapter 11 for more about kernel parameters.

Kernel startup options typically modify the values of kernel parameters, instruct the kernel to probe for particular devices, specify the path to the **init** or **systemd** process, or designate a particular root device. Table 2.3 shows a few examples.

Table 2.3: Examples of kernel boot time options

Option	Meaning
`debug`	Turns on kernel debugging
`init=/bin/bash`	Starts only the **bash** shell; useful for emergency recovery
`root=/dev/foo`	Tells the kernel to use **/dev/foo** as the root device
`single`	Boots to single-user mode

When specified at boot time, kernel options are not persistent. Edit the appropriate kernel line in **/etc/grub.d/40_custom** or **/etc/defaults/grub** (the variable named GRUB_CMDLINE_LINUX) to make the change permanent across reboots.

Security patches, bug fixes, and features are all regularly added to the Linux kernel. Unlike other software packages, however, new kernel releases typically do not replace old ones. Instead, the new kernels are installed side by side with the previous versions so that you can return to an older kernel in the event of problems.

This convention helps administrators back out of an upgrade if a kernel patch breaks their system, although it also means that the boot menu tends to get cluttered with old versions of the kernel. Try choosing a different kernel if your system won’t boot after an update.

2.5 The FreeBSD boot process



FreeBSD's boot system is a lot like GRUB in that the final-stage boot loader (called **loader**) uses a filesystem-based configuration file, supports menus, and offers an interactive, command-line-like mode. **loader** is the final common pathway for both the BIOS and UEFI boot paths.

The BIOS path: boot0

As with GRUB, the full **loader** environment is too large to fit in an MBR boot block, so a chain of progressively more sophisticated preliminary boot loaders get **loader** up and running on a BIOS system.

GRUB bundles all of these components under the umbrella name “GRUB,” but in FreeBSD, the early boot loaders are part of a separate system called **boot0** that’s used only on BIOS systems. **boot0** has options of its own, mostly because it stores later stages of the boot chain in a volume boot record (see *Legacy BIOS*) rather than in front of the first disk partition.

For that reason, the MBR boot record needs a pointer to the partition it should use to continue the boot process. Normally, all this is automatically set up for you as part of the FreeBSD installation process, but if you should ever need to adjust the configuration, you can do so with the **boot0cfg** command.

The UEFI path

On UEFI systems, FreeBSD creates an EFI system partition and installs boot code there under the path **/boot/bootx64.efi**. This is the default path that UEFI systems check at boot time (at least on modern PC platforms), so no firmware-level configuration should be needed other than ensuring that device boot priorities are properly set.

Don't confuse the **/boot** directory in the EFI system partition with the **/boot** directory in the FreeBSD root filesystem. They are separate and serve different purposes, although of course both are bootstrapping related.

By default, FreeBSD doesn't keep the EFI system partition mounted after booting. You can inspect the partition table with **gpart** to identify it:

```
$ gpart show
=>         40  134217648  ada0  GPT  (64G)
           40       1600     1  efi  (800K)
         1640  127924664     2  freebsd-ufs  (61G)
    127926304    6291383     3  freebsd-swap  (3.0G)
    134217687          1        - free -  (512B)
```

See [this page] for more information about mounting filesystems.

Although you can mount the ESP if you're curious to see what's in it (use **mount**'s **-t msdos** option), the whole filesystem is actually just a copy of an image found in **/boot/boot1.efifat** on the root disk. No user-serviceable parts inside.

If the ESP partition gets damaged or removed, you can re-create it by setting up the partition with **gpart** and then copying in the filesystem image with **dd**:

```
$ sudo dd if=/boot/boot1.efifat of=/dev/ada0p1
```

Once the first-stage UEFI boot loader finds a partition of type freebsd-ufs, it loads a UEFI version of the **loader** software from **/boot/loader.efi**. From there, booting proceeds as under BIOS, with **loader** determining the kernel to load, the kernel parameters to set, and so on. (As of FreeBSD 10.1, it is possible to use ZFS as the root partition on a UEFI system as well.)

loader configuration

loader is actually a scripting environment, and the scripting language is Forth.This is a remarkable and interesting fact if you're a historian of programming languages, and unimportant otherwise. There's a bunch of Forth code stored under **/boot** that controls **loader**'s operations, but it's designed to be self-contained—you needn't learn Forth.

The Forth scripts execute **/boot/loader.conf** to obtain the values of configuration variables, so that's where your customizations should go. Don't confuse this file with **/boot/defaults/loader.conf**, which contains the configuration defaults and isn't intended for modification. Fortunately, the variable assignments in **loader.conf** are syntactically similar to standard **sh** assignments.

The man pages for **loader** and **loader.conf** give the dirt on all the boot loader options and the configuration variables that control them. Some of the more interesting options include those for protecting the boot menu with a password, changing the splash screen displayed at boot, and passing kernel options.

loader commands

loader understands a variety of interactive commands. For example, to locate and boot an alternate kernel, you'd use a sequence of commands like this:

```
Type '?' for a list of commands, 'help' for more detailed help.
OK ls
/
 d .snap
 d dev
...
d rescue
l home
...
OK unload
OK load /boot/kernel/kernel.old
/boot/kernel/kernel.old text=0xf8f898 data=0x124 ... b077]
OK boot
```

Here, we listed the contents of the (default) root filesystem, unloaded the default kernel (**/boot/kernel/kernel**), loaded an older kernel (**/boot/kernel/kernel.old**), and then continued the boot process.

See **man loader** for complete documentation of the available commands.

2.6 System management daemons

Once the kernel has been loaded and has completed its initialization process, it creates a complement of “spontaneous” processes in user space. They’re called spontaneous processes because the kernel starts them autonomously—in the normal course of events, new processes are created only at the behest of existing processes.

Most of the spontaneous processes are really part of the kernel implementation. They don’t necessarily correspond to programs in the filesystem. They’re not configurable, and they don’t require administrative attention. You can recognize them in **ps** listings (see this page) by their low PIDs and by the brackets around their names (for example, [pagedaemon] on FreeBSD or [kdump] on Linux).

The exception to this pattern is the system management daemon. It has process ID 1 and usually runs under the name **init**. The system gives **init** a couple of special privileges, but for the most part it’s just a user-level program like any other daemon.

Responsibilities of init

init has multiple functions, but its overarching goal is to make sure the system runs the right complement of services and daemons at any given time.

To serve this goal, **init** maintains a notion of the mode in which the system should be operating. Some commonly defined modes:

- Single-user mode, in which only a minimal set of filesystems is mounted, no services are running, and a root shell is started on the console
- Multiuser mode, in which all customary filesystems are mounted and all configured network services have been started, along with a window system and graphical login manager for the console
- Server mode, similar to multiuser mode, but with no GUI running on the console

Don't take these mode names or descriptions too literally; they're just examples of common operating modes that most systems define in one way or another.

Every mode is associated with a defined complement of system services, and the initialization daemon starts or stops services as needed to bring the system's actual state into line with the currently active mode. Modes can also have associated milepost tasks that run whenever the mode begins or ends.

As an example, **init** normally takes care of many different startup chores as a side effect of its transition from bootstrapping to multiuser mode. These may include

- Setting the name of the computer
- Setting the time zone
- Checking disks with **fsck**
- Mounting filesystems
- Removing old files from the **/tmp** directory
- Configuring network interfaces
- Configuring packet filters
- Starting up other daemons and network services

init has very little built-in knowledge about these tasks. In simply runs a set of commands or scripts that have been designated for execution in that particular context.

Implementations of init

Today, three very different flavors of system management processes are in widespread use:

- An **init** styled after the **init** from AT&T's System V UNIX, which we refer to as "traditional **init**." This was the predominant **init** used on Linux systems until the debut of **systemd**.
- An **init** variant that derives from BSD UNIX and is used on most BSD-based systems, including FreeBSD, OpenBSD, and NetBSD. This one is just as tried-and-true as the SysV **init** and has just as much claim to being called "traditional," but for clarity we refer to it as "BSD **init**." This variant is quite simple in comparison with SysV-style **init**. We discuss it separately starting on this page.
- A more recent contender called **systemd** which aims to be one-stop-shopping for all daemon- and state-related issues. As a consequence, **systemd** carves out a significantly larger territory than any historical version of **init**. That makes it somewhat controversial, as we discuss below. Nevertheless, all our example Linux distributions have now adopted **systemd**.

Although these implementations are the predominant ones today, they're far from being the only choices. Apple's macOS, for example, uses a system called **launchd**. Until it adopted **systemd**, Ubuntu used another modern **init** variant called Upstart.



On Linux systems, you can theoretically replace your system's default **init** with whichever variant you prefer. But in practice, **init** is so fundamental to the operation of the system that a lot of add-on software is likely to break. If you can't abide **systemd**, standardize on a distribution that doesn't use it.

Traditional init

In the traditional **init** world, system modes (e.g., single-user or multiuser) are known as “run levels.” Most run levels are denoted by a single letter or digit.

Traditional **init** has been around since the early 80s, and grizzled folks in the anti-**systemd** camp often cite the principle, “If it ain’t broke, don’t fix it.” That said, traditional **init** does have a number of notable shortcomings.

To begin with, the traditional **init** on its own is not really powerful enough to handle the needs of a modern system. Most systems that use it actually have a standard and fixed **init** configuration that never changes. That configuration points to a second tier of shell scripts that do the actual work of changing run levels and letting administrators make configuration changes.

The second layer of scripts maintains yet a third layer of daemon- and system-specific scripts, which are cross-linked to run-level-specific directories that indicate what services are supposed to be running at what run level. It’s all a bit hackish and unsightly.

More concretely, this system has no general model of dependencies among services, so it requires that all startups and takedowns be run in a numeric order that’s maintained by the administrator. Later actions can’t run until everything ahead of them has finished, so it’s impossible to execute actions in parallel, and the system takes a long time to change states.

systemd vs. the world

Few issues in the Linux space have been more hotly debated than the migration from traditional **init** to **systemd**. For the most part, complaints center on **systemd**'s seemingly ever-increasing scope.

systemd takes all the **init** features formerly implemented with sticky tape, shell script hacks, and the sweat of administrators and formalizes them into a unified field theory of how services should be configured, accessed, and managed.

See Chapter 6, Software Installation and Management, for more information about package management.

Much like a package management system, **systemd** defines a robust dependency model, not only among services but also among "targets," **systemd**'s term for the operational modes that traditional **init** calls run levels. **systemd** not only manages processes in parallel, but also manages network connections (**networkd**), kernel log entries (**journald**), and logins (**logind**).

The anti-**systemd** camp argues that the UNIX philosophy is to keep system components small, simple, and modular. A component as fundamental as **init**, they say, should not have monolithic control over so many of the OS's other subsystems. **systemd** not only breeds complexity, but also introduces potential security weaknesses and muddies the distinction between the OS platform and the services that run on top of it.

systemd has also received criticism for imposing new standards and responsibilities on the Linux kernel, for its code quality, for the purported unresponsiveness of its developers to bug reports, for the functional design of its basic features, and for looking at people funny. We can't fairly address these issues here, but you may find it informative to peruse the *Arguments against systemd* section at without-systemd.org, the Internet's premier **systemd** hate site.

inits judged and assigned their proper punishments

The architectural objections to **systemd** outlined above are all reasonable points. **systemd** does indeed display most of the telltale stigmata of an overengineered software project.

In practice, however, many administrators quite like **systemd**, and we fall squarely into this camp. Ignore the controversy for a bit and give **systemd** a chance to win your love. Once you've become accustomed to it, you will likely find yourself appreciating its many merits.

At the very least, keep in mind that the traditional **init** that **systemd** displaces was no national treasure. If nothing else, **systemd** delivers some value just by eliminating a few of the unnecessary differences among Linux distributions.

The debate really doesn't matter anymore because the **systemd** coup is over. The argument was effectively settled when Red Hat, Debian, and Ubuntu switched. Many other Linux distributions are now adopting **systemd**, either by choice or by being dragged along, kicking and screaming, by their upstream distributions.

Traditional **init** still has a role to play when a distribution either targets a small installation footprint or doesn't need **systemd**'s advanced process management functions. There's also a sizable population of revanchists who disdain **systemd** on principle, so some Linux distributions are sure to keep traditional **init** alive indefinitely as a form of protest theater.

Nevertheless, we don't think that traditional **init** has enough of a future to merit a detailed discussion in this book. For Linux, we mostly limit ourselves to **systemd**. We also discuss the mercifully simple system used by FreeBSD, starting on this page.

2.7 SYSTEMD IN DETAIL

The configuration and control of system services is an area in which Linux distributions have traditionally differed the most from one another. **systemd** aims to standardize this aspect of system administration, and to do so, it reaches further into the normal operations of the system than any previous alternative.

systemd is not a single daemon but a collection of programs, daemons, libraries, technologies, and kernel components. A post on the **systemd** blog at 0pointer.de/blog notes that a full build of the project generates 69 different binaries. Think of it as a scrumptious buffet at which you are forced to consume everything.

Since **systemd** depends heavily on features of the Linux kernel, it’s a Linux-only proposition. You won’t see it ported to BSD or to any other variant of UNIX within the next five years.

Units and unit files

An entity that is managed by **systemd** is known generically as a unit. More specifically, a unit can be "a service, a socket, a device, a mount point, an automount point, a swap file or partition, a startup target, a watched filesystem path, a timer controlled and supervised by **systemd**, a resource management slice, a group of externally created processes, or a wormhole into an alternate universe." OK, we made up the part about the alternate universe (the rest is from the **systemd.unit** man page), but that still covers a lot of territory.

Within **systemd**, the behavior of each unit is defined and configured by a unit file. In the case of a service, for example, the unit file specifies the location of the executable file for the daemon, tells **systemd** how to start and stop the service, and identifies any other units that the service depends on.

We explore the syntax of unit files in more detail soon, but here's a simple example from an Ubuntu system as an appetizer. This unit file is **rsync.service**; it handles startup of the **rsync** daemon that mirrors filesystems.

---

*See this page for more information about **rsync**.*

```
[Unit]
Description=fast remote file copy program daemon
ConditionPathExists=/etc/rsyncd.conf

[Service]
ExecStart=/usr/bin/rsync --daemon --no-detach

[Install]
WantedBy=multi-user.target
```

If you recognize this as the file format used by MS-DOS **.ini** files, you are well on your way to understanding both **systemd** and the anguish of the **systemd** haters.

Unit files can live in several different places. **/usr/lib/systemd/system** is the main place where packages deposit their unit files during installation; on some systems, the path is **/lib/systemd/system** instead. The contents of this directory are considered stock, so you shouldn't modify them. Your local unit files and customizations can go in **/etc/systemd/system**. There's also a unit directory in **/run/systemd/system** that's a scratch area for transient units.

systemd maintains a telescopic view of all these directories, so they're pretty much equivalent. If there's any conflict, the files in **/etc** have the highest priority.

By convention, unit files are named with a suffix that varies according to the type of unit being configured. For example, service units have a **.service** suffix and timers use **.timer**. Within the

unit file, some sections (e.g., [Unit]) apply generically to all kinds of units, but others (e.g., [Service]) can appear only in the context of a particular unit type.

systemctl: manage systemd

systemctl is an all-purpose command for investigating the status of **systemd** and making changes to its configuration. As with Git and several other complex software suites, **systemctl**'s first argument is typically a subcommand that sets the general agenda, and subsequent arguments are specific to that particular subcommand. The subcommands could be top-level commands in their own right, but for consistency and clarity, they're bundled into the **systemctl** omnibus.

See this page for more information about Git.

Running **systemctl** without any arguments invokes the default **list-units** subcommand, which shows all loaded and active services, sockets, targets, mounts, and devices**.** To show only loaded and active services, use the **--type=service** qualifier:

```
$ systemctl list-units --type=service
UNIT                      LOAD   ACTIVE  SUB      DESCRIPTION
accounts-daemon.service   loaded active  running  Accounts Service
...
wpa_supplicant.service    loaded active  running  WPA supplicant
```

It's also sometimes helpful to see all the installed unit files, regardless of whether or not they're active:

```
$ systemctl list-unit-files --type=service
UNIT FILE                    STATE
...
cron.service                 enabled
cryptdisks-early.service     masked
cryptdisks.service           masked
cups-browsed.service         enabled
cups.service                 disabled
...
wpa_supplicant.service       disabled
x11-common.service           masked

188 unit files listed.
```

For subcommands that act on a particular unit (e.g., **systemctl status**) **systemctl** can usually accept a unit name without a unit-type suffix (e.g., **cups** instead of **cups.service**). However, the default unit type with which simple names are fleshed out varies by subcommand.

Table 2.4 shows the most common and useful **systemctl** subcommands. See the **systemctl** man page for a complete list.

Table 2.4: Commonly used systemctl subcommands

Subcommand	Function
list-unit-files [*pattern*]	Shows installed units; optionally matching *pattern*
enable *unit*	Enables *unit* to activate at boot
disable *unit*	Prevents *unit* from activating at boot
isolate *target*	Changes operating mode to *target*
start *unit*	Activates *unit* immediately
stop *unit*	Deactivates *unit* immediately
restart *unit*	Restarts (or starts, if not running) *unit* immediately
status *unit*	Shows *unit*'s status and recent log entries
kill *pattern*	Sends a signal to units matching *pattern*
reboot	Reboots the computer
daemon-reload	Reloads unit files and **systemd** configuration

Unit statuses

In the output of **systemctl list-unit-files** above, we can see that **cups.service** is disabled. We can use **systemctl status** to find out more details:

```
$ sudo systemctl status -l cups
cups.service - CUPS Scheduler
   Loaded: loaded (/lib/systemd/system/cups.service; disabled; vendor
    preset: enabled)
   Active: inactive (dead) since Sat 2016-12-12 00:51:40 MST; 4s ago
     Docs: man:cupsd(8)
 Main PID: 10081 (code=exited, status=0/SUCCESS)

Dec 12 00:44:39 ulsah systemd[1]: Started CUPS Scheduler.
Dec 12 00:44:45 ulsah systemd[1]: Started CUPS Scheduler.
Dec 12 00:51:40 ulsah systemd[1]: Stopping CUPS Scheduler...
Dec 12 00:51:40 ulsah systemd[1]: Stopped CUPS Scheduler.
```

Here, **systemctl** shows us that the service is currently inactive (dead) and tells us when the process died. (Just a few seconds ago; we disabled it for this example.) It also shows (in the section marked Loaded) that the service defaults to being enabled at startup, but that it is presently disabled.

The last four lines are recent log entries. By default, the log entries are condensed so that each entry takes only one line. This compression often makes entries unreadable, so we included the **-l** option to request full entries. It makes no difference in this case, but it's a useful habit to acquire.

Table 2.5 shows the statuses you'll encounter most frequently when checking up on units.

Table 2.5: Unit file statuses

State	Meaning
`bad`	Some kind of problem within **systemd**; usually a bad unit file
`disabled`	Present, but not configured to start autonomously
`enabled`	Installed and runnable; will start autonomously
`indirect`	Disabled, but has peers in `Also` clauses that may be enabled
`linked`	Unit file available through a symlink
`masked`	Banished from the **systemd** world from a logical perspective
`static`	Depended upon by another unit; has no install requirements

The enabled and disabled states apply only to unit files that live in one of **systemd's system** directories (that is, they are not linked in by a symbolic link) and that have an [Install] section in their unit files. "Enabled" units should perhaps really be thought of as "installed," meaning that the directives in the [Install] section have been executed and that the unit is wired up to its normal

activation triggers. In most cases, this state causes the unit to be activated automatically once the system is bootstrapped.

Likewise, the disabled state is something of a misnomer because the only thing that's actually disabled is the normal activation path. You can manually activate a unit that is disabled by running **systemctl start**; **systemd** won't complain.

Many units have no installation procedure, so they can't truly be said to be enabled or disabled; they're just available. Such units' status is listed as static. They only become active if activated by hand (**systemctl start**) or named as a dependency of other active units.

Unit files that are linked were created with **systemctl link**. This command creates a symbolic link from one of **systemd**'s **system** directories to a unit file that lives elsewhere in the filesystem. Such unit files can be addressed by commands or named as dependencies, but they are not full citizens of the ecosystem and have some notable quirks. For example, running **systemctl disable** on a linked unit file deletes the link and all references to it.

Unfortunately, the exact behavior of linked unit files is not well documented. Although the idea of keeping local unit files in a separate repository and linking them into **systemd** has a certain appeal, it's probably not the best approach at this point. Just make copies.

The masked status means "administratively blocked." **systemd** knows about the unit, but has been forbidden from activating it or acting on any of its configuration directives by **systemctl mask**. As a rule of thumb, turn off units whose status is enabled or linked with **systemctl disable** and reserve **systemctl mask** for static units.

Returning to our investigation of the **cups** service, we could use the following commands to reenable and start it:

```
$ sudo systemctl enable cups
Synchronizing state of cups.service with SysV init with /lib/systemd/
    systemd-sysv-install...
Executing /lib/systemd/systemd-sysv-install enable cups
insserv: warning: current start runlevel(s) (empty) of script `cups'
    overrides LSB defaults (2 3 4 5).
insserv: warning: current stop runlevel(s) (1 2 3 4 5) of script `cups'
    overrides LSB defaults (1).
Created symlink from /etc/systemd/system/sockets.target.wants/cups.socket
    to /lib/systemd/system/cups.socket.
Created symlink from /etc/systemd/system/multi-user.target.wants/cups.
    path to /lib/systemd/system/cups.path.
$ sudo systemctl start cups
```

Targets

Unit files can declare their relationships to other units in a variety of ways. In the example on this page, for example, the WantedBy clause says that if the system has a **multi-user.target** unit, that unit should depend on this one (**rsync.service**) when this unit is enabled.

Because units directly support dependency management, no additional machinery is needed to implement the equivalent of **init**'s run levels. For clarity, **systemd** does define a distinct class of units (of type **.target**) to act as well-known markers for common operating modes. However, targets have no real superpowers beyond the dependency management that's available to any other unit.

Traditional **init** defines at least seven numeric run levels, but many of those aren't actually in common use. In a perhaps-ill-advised gesture toward historical continuity, **systemd** defines targets that are intended as direct analogs of the **init** run levels (**runlevel0.target**, etc.). It also defines mnemonic targets for day-to-day use such as **poweroff.target** and **graphical.target**. Table 2.6 shows the mapping between **init** run levels and **systemd** targets.

Table 2.6: Mapping between init run levels and systemd targets

Run level	Target	Description
0	**poweroff.target**	System halt
emergency	**emergency.target**	Bare-bones shell for system recovery
1, s, single	**rescue.target**	Single-user mode
2	**multi-user.target** [a]	Multiuser mode (command line)
3	**multi-user.target** [a]	Multiuser mode with networking
4	**multi-user.target** [a]	Not normally used by **init**
5	**graphical.target**	Multiuser mode with networking and GUI
6	**reboot.target**	System reboot

a. By default, **multi-user.target** maps to **runlevel3.target**, multiuser mode with networking.

The only targets to really be aware of are **multi-user.target** and **graphical.target** for day-to-day use, and **rescue.target** for accessing single-user mode. To change the system's current operating mode, use the **systemctl isolate** command:

```
$ sudo systemctl isolate multi-user.target
```

The **isolate** subcommand is so-named because it activates the stated target and its dependencies but deactivates all other units.

Under traditional **init**, you use the **telinit** command to change run levels once the system is booted. Some distributions now define **telinit** as a symlink to the **systemctl** command, which recognizes how it's being invoked and behaves appropriately.

To see the target the system boots into by default, run the **get-default** subcommand:

```
$ systemctl get-default
graphical.target
```

Most Linux distributions boot to **graphical.target** by default, which isn't appropriate for servers that don't need a GUI. But that's easily changed:

```
$ sudo systemctl set-default multi-user.target
```

To see all the system's available targets, run **systemctl list-units**:

```
$ systemctl list-units --type=target
```

Dependencies among units

Linux software packages generally come with their own unit files, so administrators don't need a detailed knowledge of the entire configuration language. However, they do need a working knowledge of **systemd**'s dependency system to diagnose and fix dependency problems.

To begin with, not all dependencies are explicit. **systemd** takes over the functions of the old **inetd** and also extends this idea into the domain of the D-Bus interprocess communication system. In other words, **systemd** knows which network ports or IPC connection points a given service will be hosting, and it can listen for requests on those channels without actually starting the service. If a client does materialize, **systemd** simply starts the actual service and passes off the connection. The service runs if it's actually used and remains dormant otherwise.

Second, **systemd** makes some assumptions about the normal behavior of most kinds of units. The exact assumptions vary by unit type. For example, **systemd** assumes that the average service is an add-on that shouldn't be running during the early phases of system initialization. Individual units can turn off these assumptions with the line

```
DefaultDependencies=false
```

in the [Unit] section of their unit file; the default is true. See the man page for **systemd.***unit-type* to see the exact assumptions that apply to each type of unit (e.g., **man systemd.service**).

A third class of dependencies are those explicitly declared in the [Unit] sections of unit files. Table 2.7 shows the available options.

Table 2.7: Explicit dependencies in the [Unit] section of unit files

Option	Meaning
`Wants`	Units that should be coactivated if possible, but are not required
`Requires`	Strict dependencies; failure of any prerequisite terminates this service
`Requisite`	Like `Requires`, but must already be active
`BindsTo`	Similar to `Requires`, but even more tightly coupled
`PartOf`	Similar to `Requires`, but affects only starting and stopping
`Conflicts`	Negative dependencies; cannot be coactive with these units

With the exception of Conflicts, all the options in Table 2.7 express the basic idea that the unit being configured depends on some set of other units. The exact distinctions among these options are subtle and primarily of interest to service developers. The least restrictive variant, Wants, is preferred when possible.

You can extend a unit's Wants or Requires cohorts by creating a *unit-file*.**wants** or *unit-file*.**requires** directory in **/etc/systemd/system** and adding symlinks there to other unit files. Better yet, just let

systemctl do it for you. For example, the command

```
$ sudo systemctl add-wants multi-user.target my.local.service
```

adds a dependency on **my.local.service** to the standard multiuser target, ensuring that the service will be started whenever the system enters multiuser mode.

In most cases, such ad hoc dependencies are automatically taken care of for you, courtesy of the [Install] sections of unit files. This section includes WantedBy and RequiredBy options that are read only when a unit is enabled with **systemctl enable** or disabled with **systemctl disable**. On enablement, they make **systemctl** perform the equivalent of an **add-wants** for every WantedBy or an **add-requires** for every RequiredBy.

The [Install] clauses themselves have no effect in normal operation, so if a unit doesn't seem to be started when it should be, make sure that it has been properly enabled and symlinked.

Execution order

You might reasonably guess that if unit A Requires unit B, then unit B will be started or configured before unit A. But in fact that is not the case. In **systemd**, the order in which units are activated (or deactivated) is an *entirely separate* question from that of which units to activate.

When the system transitions to a new state, **systemd** first traces the various sources of dependency information outlined in the previous section to identify the units that will be affected. It then uses Before and After clauses from the unit files to sort the work list appropriately. To the extent that units have no Before or After constraints, they are free to be adjusted in parallel.

Although potentially surprising, this is actually a praiseworthy design feature. One of the major design goals of **systemd** was to facilitate parallelism, so it makes sense that units do not acquire serialization dependencies unless they explicitly ask for them.

In practice, After clauses are typically used more frequently than Wants or Requires. Target definitions (and in particular, the reverse dependencies encoded in WantedBy and RequiredBy clauses) establish the general outlines of the services running in each operating mode, and individual packages worry only about their immediate and direct dependencies.

A more complex unit file example

Now for a closer look at a few of the directives used in unit files. Here's a unit file for the NGINX web server, **nginx.service**:

```
[Unit]
Description=The nginx HTTP and reverse proxy server
After=network.target remote-fs.target nss-lookup.target

[Service]
Type=forking
PIDFile=/run/nginx.pid
ExecStartPre=/usr/bin/rm -f /run/nginx.pid
ExecStartPre=/usr/sbin/nginx -t
ExecStart=/usr/sbin/nginx
ExecReload=/bin/kill -s HUP $MAINPID
KillMode=process
KillSignal=SIGQUIT
TimeoutStopSec=5
PrivateTmp=true

[Install]
WantedBy=multi-user.target
```

This service is of type forking, which means that the startup command is expected to terminate even though the actual daemon continues running in the background. Since **systemd** won't have directly started the daemon, the daemon records its PID (process ID) in the stated PIDFile so that **systemd** can determine which process is the daemon's primary instance.

The Exec lines are commands to be run in various circumstances. ExecStartPre commands are run before the actual service is started; the ones shown here validate the syntax of NGINX's configuration file and ensure that any preexisting PID file is removed. ExecStart is the command that actually starts the service. ExecReload tells **systemd** how to make the service reread its configuration file. (**systemd** automatically sets the environment variable MAINPID to the appropriate value.)

See this page for more information about signals.

Termination for this service is handled through KillMode and KillSignal, which tell **systemd** that the service daemon interprets a QUIT signal as an instruction to clean up and exit. The line

```
ExecStop=/bin/kill -s HUP $MAINPID
```

would have essentially the same effect. If the daemon doesn't terminate within TimeoutStopSec seconds, **systemd** will force the issue by pelting it with a TERM signal and then an uncatchable KILL signal.

The PrivateTmp setting is an attempt at increasing security. It puts the service's **/tmp** directory somewhere other than the actual **/tmp**, which is shared by all the system's processes and users.

Local services and customizations

As you can see from the previous examples, it's relatively trivial to create a unit file for a home-grown service. Browse the examples in **/usr/lib/systemd/system** and adapt one that's close to what you want. See the man page for **systemd.service** for a complete list of configuration options for services. For options common to all types of units, see the page for **systemd.unit**.

Put your new unit file in **/etc/systemd/system**. You can then run

```
$ sudo systemctl enable custom.service
```

to activate the dependencies listed in the service file's [Install] section.

As a general rule, you should never edit a unit file you didn't write. Instead, create a configuration directory in **/etc/systemd/system/***unit-file***.d** and add one or more configuration files there called *xxx***.conf**. The *xxx* part doesn't matter; just make sure the file has a **.conf** suffix and is in the right location. **override.conf** is the standard name.

.conf files have the same format as unit files, and in fact **systemd** smooshes them all together with the original unit file. However, override files have priority over the original unit file should both sources try to set the value of a particular option.

One point to keep in mind is that many **systemd** options are allowed to appear more than once in a unit file. In these cases, the multiple values form a list and are all active simultaneously. If you assign a value in your **override.conf** file, that value joins the list but does not replace the existing entries. This may or may not be what you want. To remove the existing entries from a list, just assign the option an empty value before adding your own.

Let's look at an example. Suppose that your site keeps its NGINX configuration file in a nonstandard place, say, **/usr/local/www/nginx.conf**. You need to run the **nginx** daemon with a **-c /usr/local/www/nginx.conf** option so that it can find the proper configuration file.

You can't just add this option to **/usr/lib/systemd/system/nginx.service** because that file will be replaced whenever the NGINX package is updated or refreshed. Instead, you can use the following command sequence:

```
$ sudo mkdir /etc/systemd/system/nginx.service.d
$ sudo cat > !$/override.conf
[Service]
ExecStart=
ExecStart=/usr/sbin/nginx -c /usr/local/www/nginx.conf
<Control-D>
$ sudo systemctl daemon-reload
$ sudo systemctl restart nginx.service
```

The `>` and `!$` are shell metacharacters. The `>` redirects output to a file, and the `!$` expands to the last component of the previous command line so that you don't have to retype it. All shells understand this notation. See *Shell basics* for some other handy features.

The first `ExecStart=` removes the current entry, and the second sets an alternative start command. **systemctl daemon-reload** makes **systemd** re-parse unit files. However, it does not restart daemons automatically, so you'll also need an explicit **systemctl restart** to make the change take effect immediately.

This command sequence is such a common idiom that **systemctl** now implements it directly:

```
$ sudo systemctl edit nginx.service
<edit the override file in the editor>
$ sudo systemctl restart nginx.service
```

As shown, you must still do the **restart** by hand.

One last thing to know about override files is that they can't modify the `[Install]` section of a unit file. Any changes you make are silently ignored. Just add dependencies directly with **systemctl add-wants** or **systemctl add-requires**.

Service and startup control caveats

systemd has many architectural implications, and adopting it is not a simple task for the teams that build Linux distributions. Current releases are mostly Frankenstein systems that adopt much of **systemd** but also retain a few links to the past. Sometimes the holdovers just haven't yet been fully converted. In other cases, various forms of glue have been left behind to facilitate compatibility.

Though **systemctl** can and should be used for managing services and daemons, don't be surprised when you run into traditional **init** scripts or their associated helper commands. If you attempt to use **systemctl** to disable the network on a CentOS or Red Hat system, for example, you'll receive the following output:

```
$ sudo systemctl disable network
network.service is not a native service, redirecting to /sbin/chkconfig.
Executing /sbin/chkconfig network off
```

See [this page] for more information about Apache.

Traditional **init** scripts often continue to function on a **systemd** system. For example, an **init** script **/etc/rc.d/init.d/my-old-service** might be automatically mapped to a unit file such as **my-old-service.service** during system initialization or when **systemctl daemon-reload** is run. Apache 2 on Ubuntu 17.04 is a case in point: attempting to disable the **apache2.service** results in the following output:

```
$ sudo systemctl disable apache2
Synchronizing state of apache2.service with SysV service script with
    /lib/systemd/systemd-sysv-install.
Executing: /lib/systemd/systemd-sysv-install disable apache2
```

The end result is what you wanted, but it goes through a rather circuitous route.

RHEL  Red Hat, and by extension CentOS, still run the **/etc/rc.d/rc.local** script at boot time if you configure it to be executable. In theory, you can use this script to hack in site-specific tweaks or post-boot tasks if desired. (At this point, though, you should really skip the hacks and do things **systemd**'s way by creating an appropriate set of unit files.)

Some Red Hat and CentOS boot chores continue to use config files found in the **/etc/sysconfig** directory. Table 2.8 summarizes these.

Table 2.8: Files and subdirectories of Red Hat's /etc/sysconfig directory

File or directory	Contents
console/	A directory that historically allowed for custom keymapping
crond	Arguments to pass to the **crond** daemon
init	Configuration for handling messages from startup scripts
iptables-config	Loads additional **iptables** modules such as NAT helpers
network-scripts/	Accessory scripts and network config files
nfs	Optional RPC and NFS arguments
ntpd	Command-line options for **ntpd**
selinux	Symlink to **/etc/selinux/config** [a]

a. Sets arguments for SELinux or allows you to disable it altogether

A couple of the items in Table 2.8 merit additional comment:

- The **network-scripts** directory contains additional material related to network configuration. The only things you might need to change here are the files named **ifcfg-***interface*. For example, **network-scripts/ifcfg-eth0** contains the configuration parameters for the interface eth0. It sets the interface's IP address and networking options. See *Red Hat and CentOS network configuration* for more information about configuring network interfaces.
- The **iptables-config** file doesn't actually allow you to modify the **iptables** (firewall) rules themselves. It just provides a way to load additional modules such as those for network address translation (NAT) if you're going to be forwarding packets or using the system as a router. See this page for more information about configuring **iptables**.

systemd logging

Capturing the log messages produced by the kernel has always been something of a challenge. It became even more important with the advent of virtual and cloud-based systems, since it isn't possible to simply stand in front of these systems' consoles and watch what happens. Frequently, crucial diagnostic information was lost to the ether.

systemd alleviates this problem with a universal logging framework that includes all kernel and service messages from early boot to final shutdown. This facility, called the journal, is managed by the **journald** daemon.

System messages captured by **journald** are stored in the **/run** directory. **rsyslog** can process these messages and store them in traditional log files or forward them to a remote syslog server. You can also access the logs directly with the **journalctl** command**.**

Without arguments, **journalctl** displays all log entries (oldest first):

```
$ journalctl
-- Logs begin at Fri 2016-02-26 15:01:25 UTC, end at Fri 2016-02-26
    15:05:16 UTC. --
Feb 26 15:01:25 ubuntu systemd-journal[285]: Runtime journal is using
    4.6M (max allowed 37.0M, t
Feb 26 15:01:25 ubuntu systemd-journal[285]: Runtime journal is using
    4.6M (max allowed 37.0M, t
Feb 26 15:01:25 ubuntu kernel: Initializing cgroup subsys cpuset
Feb 26 15:01:25 ubuntu kernel: Initializing cgroup subsys cpu
Feb 26 15:01:25 ubuntu kernel: Linux version 3.19.0-43-generic (buildd@
    lcy01-02) (gcc version 4.
Feb 26 15:01:25 ubuntu kernel: Command line: BOOT_IMAGE=/boot/vmlinuz-
    3.19.0-43-generic root=UUI
Feb 26 15:01:25 ubuntu kernel: KERNEL supported cpus:
Feb 26 15:01:25 ubuntu kernel:   Intel GenuineIntel
...
```

You can configure **journald** to retain messages from prior boots. To do this, edit **/etc/systemd/journald.conf** and configure the Storage attribute:

```
[Journal]
Storage=persistent
```

Once you've configured **journald**, you can obtain a list of prior boots with

```
$ journalctl --list-boots
-1 a73415fade0e4e7f4bea60913883d180dc Fri 2016-02-26 15:01:25 UTC
    Fri 2016-02-26 15:05:16 UTC
0 0c563fa3830047ecaa2d2b053d4e661d Fri 2016-02-26 15:11:03 UTC  Fri
    2016-02-26 15:12:28 UTC
```

You can then access messages from a prior boot by referring to its index or by naming its long-form ID:

```
$ journalctl -b -1
$ journalctl -b a73415fade0e4e7f4bea60913883d180dc
```

To restrict the logs to those associated with a specific unit, use the **-u** flag:

```
$ journalctl -u ntp
-- Logs begin at Fri 2016-02-26 15:11:03 UTC, end at Fri 2016-02-26
    15:26:07 UTC. --
Feb 26 15:11:07 ub-test-1 systemd[1]: Stopped LSB: Start NTP daemon.
Feb 26 15:11:08 ub-test-1 systemd[1]: Starting LSB: Start NTP daemon...
Feb 26 15:11:08 ub-test-1 ntp[761]: * Starting NTP server ntpd
...
```

System logging is covered in more detail in Chapter 10, *Logging*.

2.8 FreeBSD init and startup scripts

FreeBSD uses a BSD-style **init**, which does not support the concept of run levels. To bring the system to its fully booted state, FreeBSD's **init** just runs **/etc/rc**. This program is a shell script, but it should not be directly modified. Instead, the **rc** system implements a couple of standardized ways for administrators and software packages to extend the startup system and make configuration changes.

See Chapter 7 for more information about shell scripting.

/etc/rc is primarily a wrapper that runs other startup scripts, most of which live in **/usr/local/etc/rc.d**. and **/etc/rc.d**. Before it runs any of those scripts, however, **rc** executes three files that hold configuration information for the system:

- **/etc/defaults/config**
- **/etc/rc.conf**
- **/etc/rc.conf.local**

These files are themselves scripts, but they typically contain only definitions for the values of shell variables. The startup scripts then check these variables to determine how to behave. (**/etc/rc** uses some shell magic to ensure that the variables defined in these files are visible everywhere.)

/etc/defaults/rc.conf lists all the configuration parameters and their default settings. Never edit this file, lest the startup script bogeyman hunt you down and overwrite your changes the next time the system is updated. Instead, just override the default values by setting them again in **/etc/rc.conf** or **/etc/rc.conf.local**. The **rc.conf** man page has an extensive list of the variables you can specify.

In theory, the **rc.conf** files can also specify other directories in which to look for startup scripts by your setting the value of the local_startup variable. The default value is **/usr/local/etc/rc.d**, and we recommend leaving it that way. For local customizations, you have the option of either creating standard **rc.d**-style scripts that go in **/usr/local/etc/rc.d** or editing the system-wide **/etc/rc.local** script. The former is preferred.

As you can see from peeking at **/etc/rc.d**, there are many different startup scripts, more than 150 on a standard installation. **/etc/rc** runs these scripts in the order calculated by the **rcorder** command, which reads the scripts and looks for dependency information that's been encoded in a standard way.

FreeBSD's startup scripts for garden-variety services are fairly straightforward. For example, the top of the **sshd** startup script is as follows:

```
#!/bin/sh
# PROVIDE: sshd
# REQUIRE: LOGIN FILESYSTEMS
# KEYWORD: shutdown
. /etc/rc.subr
name="sshd"
rcvar="sshd_enable"
command="/usr/sbin/${name}"
...
```

The rcvar variable contains the name of a variable that's expected to be defined in one of the **rc.conf** scripts, in this case, sshd_enable. If you want **sshd** (the real daemon, not the startup script; both are named **sshd**) to run automatically at boot time, put the line

```
sshd_enable="YES"
```

into **/etc/rc.conf**. If this variable is set to "NO" or commented out, the **sshd** script will not start the daemon or check to see whether it should be stopped when the system is shut down.

The **service** command provides a real-time interface into FreeBSD's **rc.d** system. To stop the **sshd** service manually, for example, you could run the command

```
$ sudo service sshd stop
```

Note that this technique works only if the service is enabled in the **/etc/rc.conf** files. If it is not, use the subcommand **onestop**, **onestart**, or **onerestart**, depending on what you want to do. (**service** is generally forgiving and will remind you if need be, however.)

2.9 Reboot and shutdown procedures

Historically, UNIX and Linux machines were touchy about how they were shut down. Modern systems have become less sensitive, especially when a robust filesystem is used, but it’s always a good idea to shut down a machine nicely when possible.

Consumer operating systems of yesteryear trained many sysadmins to reboot the system as the first step in debugging any problem. It was an adaptive habit back then, but these days it more commonly wastes time and interrupts service. Focus on identifying the root cause of problems, and you’ll probably find yourself rebooting less often.

That said, it’s a good idea to reboot after modifying a startup script or making significant configuration changes. This check ensures that the system can boot successfully. If you’ve introduced a problem but don’t discover it until several weeks later, you’re unlikely to remember the details of your most recent changes.

Shutting down physical systems

The **halt** command performs the essential duties required for shutting down the system. **halt** logs the shutdown, kills nonessential processes, flushes cached filesystem blocks to disk, and then halts the kernel. On most systems, **halt -p** powers down the system as a final flourish.

reboot is essentially identical to **halt**, but it causes the machine to reboot instead of halting.

The **shutdown** command is a layer over **halt** and **reboot** that provides for scheduled shutdowns and ominous warnings to logged-in users. It dates back to the days of time-sharing systems and is now largely obsolete. **shutdown** does nothing of technical value beyond **halt** or **reboot**, so feel free to ignore it if you don’t have multiuser systems.

Shutting down cloud systems

You can halt or restart a cloud system either from within the server (with **halt** or **reboot**, as described in the previous section) or from the cloud provider's web console (or its equivalent API).

Generally speaking, powering down from the cloud console is akin to turning off the power. It's better if the virtual server manages its own shutdown, but feel free to kill a virtual server from the console if it becomes unresponsive. What else can you do?

Either way, make sure you understand what a shutdown means from the perspective of the cloud provider. It would be a shame to destroy your system when all you meant to do was reboot it.

In the AWS universe, the Stop and Reboot operations do what you'd expect. "Terminate" decommissions the instance and removes it from your inventory. If the underlying storage device is set to "delete on termination," not only will your instance be destroyed, but the data on the root disk will also be lost. That's perfectly fine, as long as it's what you expect. You can enable "termination protection" if you consider this a bad thing.

2.10 STRATAGEMS FOR A NONBOOTING SYSTEM

A variety of problems can prevent a system from booting, ranging from faulty devices to kernel upgrades gone wrong. There are three basic approaches to this situation, listed here in rough order of desirability:

- Don’t debug; just restore the system to a known-good state.
- Bring the system up just enough to run a shell, and debug interactively.
- Boot a separate system image, mount the sick system’s filesystems, and investigate from there.

The first option is the one most commonly used in the cloud, but it can be helpful on physical servers, too, as long as you have access to a recent image of the entire boot disk. If your site does backups by filesystem, a whole-system restore may be more trouble than it’s worth. We discuss the whole-system restore option in *Recovery of cloud systems*.

The remaining two approaches focus on giving you a way to access the system, identify the underlying issue, and make whatever fix is needed. Booting the ailing system to a shell is by far the preferable option, but problems that occur very early in the boot sequence may stymie this approach.

The “boot to a shell” mode is known generically as single-user mode or rescue mode. Systems that use **systemd** have an even more primitive option available in the form of emergency mode; it’s conceptually similar to single-user mode, but does an absolute minimum of preparation before starting a shell.

Because single-user, rescue, and emergency modes don’t configure the network or start network-related services, you’ll generally need physical access to the console to make use of them. As a result, single-user mode normally isn’t available for cloud-hosted systems. We review some options for reviving broken cloud images starting on this page.

Single-user mode

In single-user mode, also known as **rescue.target** on systems that use **systemd**, only a minimal set of processes, daemons, and services are started. The root filesystem is mounted (as is **/usr**, in most cases), but the network remains uninitialized.

At boot time, you request single-user mode by passing an argument to the kernel, usually **single** or **-s**. You can do this through the boot loader's command-line interface. In some cases, it may be set up for you automatically as a boot menu option.

If the system is already running, you can bring it down to single-user mode with a **shutdown** (FreeBSD), **telinit** (traditional **init**), or **systemctl** (**systemd**) command.

See Chapter 3, for more information about the root account.

Sane systems prompt for the root password before starting the single-user root shell. Unfortunately, this means that it's virtually impossible to reset a forgotten root password through single-user mode. If you need to reset the password, you'll have to access the disk by way of separate boot media.

See Chapter 5 for more information about filesystems and mounting.

From the single-user shell, you can execute commands in much the same way as when logged in on a fully booted system. However, sometimes only the root partition is mounted; you must mount other filesystems manually to use programs that don't live in **/bin**, **/sbin**, or **/etc**.

You can often find pointers to the available filesystems by looking in **/etc/fstab**. Under Linux, you can run **fdisk -l** (lowercase L option) to see a list of the local system's disk partitions. The analogous procedure on FreeBSD is to run **camcontrol devlist** to identify disk devices and then run **fdisk -s** *device* for each disk.

In many single-user environments, the filesystem root directory starts off being mounted read-only. If **/etc** is part of the root filesystem (the usual case), it will be impossible to edit many important configuration files. To fix this problem, you'll have to begin your single-user session by remounting / in read/write mode. Under Linux, the command

```
# mount -o rw,remount /
```

usually does the trick. On FreeBSD systems, the remount option is implicit when you repeat an existing mount, but you'll need to explicitly specify the source device. For example,

```
# mount -o rw /dev/gpt/rootfs /
```



Single-user mode in Red Hat and CentOS is a bit more aggressive than normal. By the time you reach the shell prompt, these systems have tried to mount all local filesystems. Although this default is usually helpful, it can be problematic if you have a sick filesystem. In that case, you can boot to emergency mode by adding **systemd.unit=emergency.target** to the kernel arguments from within the boot loader (usually GRUB). In this mode, no local filesystems are mounted and only a few essential services are started.

The **fsck** command is run during a normal boot to check and repair filesystems. Depending on what filesystem you're using for the root, you may need to run **fsck** manually when you bring the system up in single-user or emergency mode. See this page for more details about **fsck**.

Single-user mode is just a waypoint on the normal booting path, so you can terminate the single-user shell with **exit** or <Control-D> to continue with booting. You can also type <Control-D> at the password prompt to bypass single-user mode entirely.

Single-user mode on FreeBSD

 FreeBSD includes a single-user option in its boot menu:

```
1. Boot Multi User [Enter]
2. Boot Single User
3. Escape to loader prompt
4. Reboot

Options
5. Kernel: default/kernel (1 of 2)
6. Configure Boot Options...
```

One nice feature of FreeBSD's single-user mode is that it asks you what program to use as the shell. Just press <Enter> for **/bin/sh**.

If you choose option 3, "Escape to loader prompt," you'll drop into a boot-level command-line environment implemented by FreeBSD's final-common-stage boot loader, **loader**.

Single-user mode with GRUB



On systems that use **systemd**, you can boot into rescue mode by appending **systemd.unit=rescue.target** to the end of the existing Linux kernel line. At the GRUB splash screen, highlight your desired kernel and press the “e” key to edit its boot options. Similarly, for emergency mode, use **systemd.unit=emergency.target**.

Here’s an example of a typical configuration:

```
linux16 /vmlinuz-3.10.0-229.el7.x86_64 root=/dev/mapper/rhel_rhel-root
    ro crashkernel=auto rd.lvm.lv=rhel_rhel/swap rd.lvm.lv=rhel_rhel/
    root rhgb quiet LANG=en_US.UTF-8 systemd.unit=rescue.target
```

Type <Control-X> to start the system after you’ve made your changes.

Recovery of cloud systems

See Chapter 9 for a broader introduction to cloud computing.

It's inherent in the nature of cloud systems that you can't hook up a monitor or USB stick when boot problems occur. Cloud providers do what they can to facilitate problem solving, but basic limitations remain.

Backups are important for all systems, but cloud servers are particularly easy to snapshot. Providers charge extra for backups, but they're inexpensive. Be liberal with your snapshots and you'll always have a reasonable system image to fall back on at short notice.

From a philosophical perspective, you're probably doing something wrong if your cloud servers require boot-time debugging. Pets and physical servers receive veterinary care when they're sick, but cattle get euthanized. Your cloud servers are cattle; replace them with known-good copies when they misbehave. Embracing this approach helps you not only avoid critical failures but also facilitates scaling and system migration.

That said, you will inevitably need to attempt to recover cloud systems or drives, so we briefly discuss that process below.

Within AWS, single-user and emergency modes are unavailable. However, EC2 filesystems can be attached to other virtual servers if they're backed by Elastic Block Storage (EBS) devices. This is the default for most EC2 instances, so it's likely that you can use this method if you need to. Conceptually, it's similar to booting from a USB drive so that you can poke around on a physical system's boot disk.

Here's what to do:

1. Launch a new instance in the same availability zone as the instance you're having issues with. Ideally, this recovery instance should be launched from the same base image and should use the same instance type as the sick system.
2. Stop the problem instance. (But be careful not to "terminate" it; that operation deletes the boot disk image.)
3. With the AWS web console or CLI, detach the volume from the problem system and attach the volume to the recovery instance.
4. Log in to the recovery system. Create a mount point and mount the volume, then do whatever's necessary to fix the issue. Then unmount the volume. (Won't unmount? Make sure you're not **cd**'ed there.)

5. In the AWS console, detach the volume from the recovery instance and reattach it to the problem instance. Start the problem instance and hope for the best.

DigitalOcean droplets offer a VNC-enabled console that you can access through the web, although the web app's behavior is a bit wonky on some browsers. DigitalOcean does not afford a way to detach storage devices and migrate them to a recovery system the way Amazon does. Instead, most system images let you boot from an alternate recovery kernel.

To access the recovery kernel, first power off the droplet and then mount the recovery kernel and reboot. If all went well, the virtual terminal will give you access to a single-user-like mode. More detailed instructions for this process are available at digitalocean.com.

The recovery kernel is not available on all modern distributions. If you're running a recent release and the recovery tab tells you that "The kernel for this Droplet is managed internally and cannot be changed from the control panel" you'll need to open a support ticket with DigitalOcean to have them associate your instance with a recovery ISO, allowing you to continue your recovery efforts.

Boot issues within a Google Compute Engine instance should first be investigated by examination of the instance's serial port information:

```
$ gcloud compute instances get-serial-port-output instance
```

The same information is available through GCP web console.

A disk-shuffling process similar to that described above for the Amazon cloud is also available on Google Compute Engine. You use the CLI to remove the disk from the defunct instance and boot a new instance that mounts the disk as an add-on filesystem. You can then run filesystem checks, modify boot parameters, and select a new kernel if necessary. This process is nicely detailed in Google's documentation at cloud.google.com/compute/docs/troubleshooting.

3 Access Control and Rootly Powers



This chapter is about “access control,” as opposed to “security,” by which we mean that it focuses on the mechanical details of how the kernel and its delegates make security-related decisions. Chapter 27, *Security*, addresses the more general question of how to set up a system or network to minimize the chance of unwelcome access by intruders.

Access control is an area of active research, and it has long been one of the major challenges of operating system design. Over the last decade, UNIX and Linux have seen a Cambrian explosion of new options in this domain. A primary driver of this surge has been the advent of kernel APIs that allow third party modules to augment or replace the traditional UNIX access control system. This modular approach creates a variety of new frontiers; access control is now just as open to change and experimentation as any other aspect of UNIX.

Nevertheless, the traditional system remains the UNIX and Linux standard, and it’s adequate for the majority of installations. Even for administrators who want to venture into the new frontier, a thorough grounding in the basics is essential.

3.1 Standard UNIX access control

The standard UNIX access control model has remained largely unchanged for decades. With a few enhancements, it continues to be the default for general-purpose OS distributions. The scheme follows a few basic rules:

- Access control decisions depend on which user is attempting to perform an operation, or in some cases, on that user's membership in a UNIX group.
- Objects (e.g., files and processes) have owners. Owners have broad (but not necessarily unrestricted) control over their objects.
- You own the objects you create.
- The special user account called "root" can act as the owner of any object.
- Only root can perform certain sensitive administrative operations.

Certain system calls (e.g., **settimeofday**) are restricted to root; the implementation simply checks the identity of the current user and rejects the operation if the user is not root. Other system calls (e.g., **kill**) implement different calculations that involve both ownership matching and special provisions for root. Finally, filesystems have their own access control systems, which they implement in cooperation with the kernel's VFS layer. These are generally more elaborate than the access controls found elsewhere in the kernel. For example, filesystems are much more likely to make use of UNIX groups for access control.

See this page for more information about device files.

Complicating this picture is that the kernel and the filesystem are intimately intertwined. For example, you control and communicate with most devices through files that represent them in **/dev**. Since device files are filesystem objects, they are subject to filesystem access control semantics. The kernel uses that fact as its primary form of access control for devices.

Keep in mind that we are here describing the original design of the access control system. These days, not all of the statements above remain literally true. For example, a Linux process that bears appropriate capabilities (see this page) can now perform some operations that were previously restricted to root.

Filesystem access control

In the standard model, every file has both an owner and a group, sometimes referred to as the "group owner." The owner can set the permissions of the file. In particular, the owner can set them so restrictively that no one else can access it. We talk more about file permissions in Chapter 5, *The Filesystem* (see this page).

See this page for more information about groups.

Although the owner of a file is always a single person, many people can be group owners of the file, as long as they are all part of a single group. Groups are traditionally defined in the **/etc/group** file, but these days group information is often stored in a network database system such as LDAP; see Chapter 17, *Single Sign-On*, for details.

The owner of a file gets to specify what the group owners can do with it. This scheme allows files to be shared among members of the same project.

You can determine the ownerships of a file with **ls -l**:

```
$ ls -l ~garth/todo
-rw-r-----  1 garth  staff  1259 May 29 19:55 /Users/garth/todo
```

This file is owned by user garth and group staff. The letters and dashes in the first column symbolize the permissions on the file; see this page for details on how to decode the information. In this case, the codes mean that garth can read or write the file and that members of the staff group can read it.

---

*See Chapter 8 for more information about the **passwd** and **group** files.*

Both the kernel and the filesystem track owners and groups as numbers rather than as text names. In the most basic case, user identification numbers (UIDs for short) are mapped to usernames in the **/etc/passwd** file, and group identification numbers (GIDs) are mapped to group names in **/etc/group**. (See Chapter 17, *Single Sign-On*, for information about the more sophisticated options.)

The text names that correspond to UIDs and GIDs are defined only for the convenience of the system's human users. When commands such as **ls** should display ownership information in a human-readable format, they must look up each name in the appropriate file or database.

Process ownership

The owner of a process can send the process signals (see this page) and can also reduce (degrade) the process's scheduling priority. Processes actually have multiple identities associated with them: a real, effective, and saved UID; a real, effective, and saved GID; and under Linux, a "filesystem UID" that is used only to determine file access permissions. Broadly speaking, the real numbers are used for accounting (now largely vestigial), and the effective numbers are used for the determination of access permissions. The real and effective numbers are normally the same.

The saved UID and GID are parking spots for IDs that are not currently in use but that remain available for the process to invoke. The saved IDs allow a program to repeatedly enter and leave a privileged mode of operation; this precaution reduces the risk of unintended misbehavior.

See Chapter 21 for more about NFS.

The filesystem UID is generally explained as an implementation detail of NFS, the Network File System. It is usually the same as the effective UID.

The root account

The root account is UNIX's omnipotent administrative user. It's also known as the superuser account, although the actual username is "root".

The defining characteristic of the root account is its UID of 0. Nothing prevents you from changing the username on this account or from creating additional accounts whose UIDs are 0; however, these are both bad ideas. (Jennine Townsend, one of our stalwart technical reviewers, commented, "Such bad ideas that I fear even mentioning them might encourage someone!") Changes like these have a tendency to create inadvertent breaches of system security. They also create confusion when other people have to deal with the strange way you've configured your system.

Traditional UNIX allows the superuser (that is, any process for which the effective UID is 0) to perform any valid operation on any file or process. "Valid" is the operative word here; certain operations (such as executing a file on which the execute permission bit is not set) are forbidden even to the superuser.

Some examples of restricted operations are

- Creating device files
- Setting the system clock
- Raising resource usage limits and process priorities
- Setting the system's hostname
- Configuring network interfaces
- Opening privileged network ports (those numbered below 1,024)
- Shutting down the system

An example of superuser powers is the ability of a process owned by root to change its UID and GID. The **login** program and its GUI equivalents are a case in point; the process that prompts you for your password when you log in to the system initially runs as root. If the password and username that you enter are legitimate, the login program changes its UID and GID to your UID and GID and starts up your shell or GUI environment. Once a root process has changed its ownerships to become a normal user process, it can't recover its former privileged state.

Setuid and setgid execution

Traditional UNIX access control is complemented by an identity substitution system that's implemented by the kernel and the filesystem in collaboration. This scheme allows specially marked executable files to run with elevated permissions, usually those of root. It lets developers and administrators set up structured ways for unprivileged users to perform privileged operations.

When the kernel runs an executable file that has its "setuid" or "setgid" permission bits set, it changes the effective UID or GID of the resulting process to the UID or GID of the file containing the program image rather than the UID and GID of the user that ran the command. The user's privileges are thus promoted for the execution of that specific command only.

For example, users must be able to change their passwords. But since passwords are (traditionally) stored in the protected **/etc/master.passwd** or **/etc/shadow** file, users need a setuid **passwd** command to mediate their access. The **passwd** command checks to see who's running it and customizes its behavior accordingly: users can change only their own passwords, but root can change any password.

Programs that run setuid, especially ones that run setuid to root, are prone to security problems. The setuid commands distributed with the system are theoretically secure; however, security holes have been discovered in the past and will undoubtedly be discovered in the future.

The surest way to minimize the number of setuid *problems* is to minimize the number of setuid *programs*. Think twice before installing software that needs to run setuid, and avoid using the setuid facility in your own home-grown software. Never use setuid execution on programs that were not explicitly written with setuid execution in mind.

See this page for more information about filesystem mount options.

You can disable setuid and setgid execution on individual filesystems by specifying the **nosuid** option to **mount**. It's a good idea to use this option on filesystems that contain users' home directories or that are mounted from less trustworthy administrative domains.

3.2 Management of the root account

Root access is required for system administration, and it’s also a pivot point for system security. Proper husbandry of the root account is a crucial skill.

Root account login

Since root is just another user, most systems let you log in directly to the root account. However, this turns out to be a bad idea, which is why Ubuntu forbids it by default.

To begin with, root logins leave no record of what operations were performed as root. That's bad enough when you realize that you broke something last night at 3:00 a.m. and can't remember what you changed; it's even worse when an access was unauthorized and you are trying to figure out what an intruder has done to your system. Another disadvantage is that the log-in-as-root scenario leaves no record of who was actually doing the work. If several people have access to the root account, you won't be able to tell who used it and when.

For these reasons, most systems allow root logins to be disabled on terminals, through window systems, and across the network—everywhere but on the system console. We suggest that you use these features. See *PAM: cooking spray or authentication wonder?* to see how to implement this policy on your particular system.

If root does have a password (that is, the root account is not disabled; see this page), that password must be of high quality. See this page for some additional comments regarding password selection.

su: substitute user identity

A marginally better way to access the root account is to use the **su** command. If invoked without arguments, **su** prompts for the root password and then starts up a root shell. Root privileges remain in effect until you terminate the shell by typing <Control-D> or the **exit** command. **su** doesn't record the commands executed as root, but it does create a log entry that states who became root and when.

The **su** command can also substitute identities other than root. Sometimes, the only way to reproduce or debug a user's problem is to **su** to their account so that you reproduce the environment in which the problem occurs.

If you know someone's password, you can access that person's account directly by executing **su - *username***. As with an **su** to root, you are prompted for the password for *username*. The - (dash) option makes **su** spawn the shell in login mode.

The exact implications of login mode vary by shell, but login mode normally changes the number or identity of the files that the shell reads when it starts up. For example, **bash** reads **~/.bash_profile** in login mode and **~/.bashrc** in nonlogin mode. When diagnosing other users' problems, it helps to reproduce their login environments as closely as possible by running in login mode.

On some systems, the root password allows an **su** or **login** to any account. On others, you must first **su** explicitly to root before **su**ing to another account; root can **su** to any account without entering a password.

Get in the habit of typing the full pathname to **su** (e.g., **/bin/su** or **/usr/bin/su**) rather than relying on the shell to find the command for you. This precaution gives you some protection against arbitrary programs called **su** that might have been sneaked into your search path with the intention of harvesting passwords. (For the same reason, do not include ".", the current directory, in your shell's search path. Although convenient, including "." makes it easy to inadvertently run "special" versions of system commands that an intruder has left lying around as a trap. Naturally, this advice goes double for root.)

On most systems, you must be a member of the group "wheel" to use **su**.

We consider **su** to have been largely superseded by **sudo**, described in the next section. **su** is best reserved for emergencies. It's also helpful for fixing situations in which **sudo** has been broken or misconfigured.

sudo: limited su

Without one of the advanced access control systems outlined starting on this page, it's hard to enable someone to do one task (backups, for example) without giving that person free run of the system. And if the root account is used by several administrators, you really have only a vague idea of who's using it or what they've done.

The most widely used solution to these problems is a program called **sudo** that is currently maintained by Todd Miller. It runs on all our example systems and is also available in source code form from sudo.ws. We recommend it as the primary method of access to the root account.

sudo takes as its argument a command line to be executed as root (or as another restricted user). **sudo** consults the file **/etc/sudoers** (**/usr/local/etc/sudoers** on FreeBSD), which lists the people who are authorized to use **sudo** and the commands they are allowed to run on each host. If the proposed command is permitted, **sudo** prompts for the *user's own* password and executes the command.

Additional **sudo** commands can be executed without the "doer" having to type a password until a five-minute period (configurable) has elapsed with no further **sudo** activity. This timeout serves as a modest protection against users with **sudo** privileges who leave terminals unattended.

See Chapter 10 for more information about syslog.

---

sudo keeps a log of the command lines that were executed, the hosts on which they were run, the people who ran them, the directories from which they were run, and the times at which they were invoked. This information can be logged by syslog or placed in the file of your choice. We recommend using syslog to forward the log entries to a secure central host.

A log entry for randy's executing **sudo /bin/cat /etc/sudoers** might look like this:

```
Dec 7 10:57:19 tigger sudo: randy: TTY=ttyp0 ; PWD=/tigger/users/randy;
    USER=root ; COMMAND=/bin/cat /etc/sudoers
```

Example configuration

The **sudoers** file is designed so that a single version can be used on many different hosts at once. Here's a typical example:

```
# Define aliases for machines in CS & Physics departments
Host_Alias  CS = tigger, anchor, piper, moet, sigi
Host_Alias  PHYSICS = eprince, pprince, icarus

# Define collections of commands
Cmnd_Alias  DUMP = /sbin/dump, /sbin/restore
Cmnd_Alias  WATCHDOG = /usr/local/bin/watchdog
Cmnd_Alias  SHELLS = /bin/sh, /bin/dash, /bin/bash

# Permissions
mark, ed    PHYSICS = ALL
herb        CS = /usr/sbin/tcpdump : PHYSICS = (operator) DUMP
lynda       ALL = (ALL) ALL, !SHELLS
%wheel      ALL, !PHYSICS = NOPASSWD: WATCHDOG
```

The first two sets of lines define groups of hosts and commands that are referred to in the permission specifications later in the file. The lists could be included literally in the specifications, but aliases make the **sudoers** file easier to read and understand; they also make the file easier to update in the future. It's also possible to define aliases for sets of users and for sets of users as whom commands may be run.

Each permission specification line includes information about

- The users to whom the line applies
- The hosts on which the line should be heeded
- The commands that the specified users can run
- The users as whom the commands can be executed

The first permission line applies to the users mark and ed on the machines in the PHYSICS group (eprince, pprince, and icarus). The built-in command alias ALL allows them to run any command. Since no list of users is specified in parentheses, **sudo** will run commands as root.

The second permission line allows herb to run **tcpdump** on CS machines and dump-related commands on PHYSICS machines. However, the dump commands can be run only as operator, not as root. The actual command line that herb would type would be something like

```
ubuntu$ sudo -u operator /usr/sbin/dump 0u /dev/sda1
```

The user lynda can run commands as any user on any machine, except that she can't run several common shells. Does this mean that lynda really can't get a root shell? Of course not:

```
ubuntu$ cp -p /bin/sh /tmp/sh
ubuntu$ sudo /tmp/sh
```

Generally speaking, any attempt to allow "all commands except …" is doomed to failure, at least in a technical sense. However, it might still be worthwhile to set up the **sudoers** file this way as a reminder that root shells are strongly discouraged.

The final line allows users in group wheel to run the local **watchdog** command as root on all machines except eprince, pprince, and icarus. Furthermore, no password is required to run the command.

Note that commands in the **sudoers** file are specified with full pathnames to prevent people from executing their own programs and scripts as root. Though no examples are shown above, it is possible to specify the arguments that are permissible for each command as well.

To manually modify the **sudoers** file, use the **visudo** command, which checks to be sure no one else is editing the file, invokes an editor on it (**vi**, or whichever editor you specify in your EDITOR environment variable), and then verifies the syntax of the edited file before installing it. This last step is particularly important because an invalid **sudoers** file might prevent you from **sudo**ing again to fix it.

sudo pros and cons

The use of **sudo** has the following advantages:

- Accountability is much improved because of command logging.
- Users can do specific chores without having unlimited root privileges.
- The real root password can be known to only one or two people.
- Using **sudo** is faster than using **su** or logging in as root.
- Privileges can be revoked without the need to change the root password.
- A canonical list of all users with root privileges is maintained.
- The chance of a root shell being left unattended is lessened.
- A single file can control access for an entire network.

sudo has a couple of disadvantages as well. The worst of these is that any breach in the security of a sudoer's personal account can be equivalent to breaching the root account itself. You can't do much to counter this threat other than caution your sudoers to protect their own accounts as they would the root account. You can also run a password cracker regularly on sudoers' passwords to ensure that they are making good password selections. All the comments on password selection from this page apply here as well.

sudo's command logging can easily be subverted by tricks such as shell escapes from within an allowed program, or by **sudo sh** and **sudo su**. (Such commands do show up in the logs, so you'll at least know they've been run.)

sudo vs. advanced access control

If you think of **sudo** as a way of subdividing the privileges of the root account, it is superior in some ways to many of the drop-in access control systems outlined starting on this page:

- You decide exactly how privileges will be subdivided. Your division can be coarser or finer than the privileges defined for you by an off-the-shelf system.
- Simple configurations—the most common—are simple to set up, maintain, and understand.
- **sudo** runs on all UNIX and Linux systems. You do need not worry about managing different solutions on different platforms.
- You can share a single configuration file throughout your site.
- You get consistent, high-quality logging for free.

Because the system is vulnerable to catastrophic compromise if the root account is penetrated, a major drawback of **sudo**-based access control is that the potential attack surface expands to include the accounts of all administrators.

sudo works well as a tool for well-intentioned administrators who need general access to root privileges. It's also great for allowing non-administrators to perform a few specific operations. Despite a configuration syntax that suggests otherwise, it is unfortunately not a safe way to define limited domains of autonomy or to place certain operations out of bounds.

Don't even attempt these configurations. If you need this functionality, you are much better off enabling one of the drop-in access control systems described starting on this page.

Typical setup

sudo's configuration system has accumulated a lot of features over the years. It has also expanded to accommodate a variety of unusual situations and edge cases. As a result, the current documentation conveys an impression of complexity that isn't necessarily warranted.

Since it's important that **sudo** be reliable and secure, it's natural to wonder if you might be exposing your systems to additional risk if you don't make use of **sudo**'s advanced features and set exactly the right values for all options. The answer is no. 90% of **sudoers** files look something like this:

```
User_Alias  ADMINS = alice, bob, charles
ADMINS      ALL = (ALL) ALL
```

This is a perfectly respectable configuration, and in many cases there's no need to complicate it further. We've mentioned a few extras you can play with in the sections below, but they're all

problem-solving tools that are helpful for specific situations. Nothing more is required for general robustness.

Environment management

Many commands consult the values of environment variables and modify their behavior depending on what they find. In the case of commands run as root, this mechanism can be both a useful convenience and a potential route of attack.

For example, several commands run the program specified in your EDITOR environment variable to spawn a text editor. If this variable points to a hacker's malicious program instead of an editor, it's likely that you'll eventually end up running that program as root. (Just to be clear, the scenario in this case is that your account has been compromised, but the attacker does not know your actual password and so cannot run **sudo** directly. Unfortunately, this is a common situation—all it takes is a terminal window left momentarily unattended.)

To minimize this risk, **sudo**'s default behavior is to pass only a minimal, sanitized environment to the commands that it runs. If your site needs additional environment variables to be passed, you can whitelist them by adding them to the **sudoers** file's env_keep list. For example, the lines

```
Defaults    env_keep += "SSH_AUTH_SOCK"
Defaults    env_keep += "DISPLAY XAUTHORIZATION XAUTHORITY"
```

preserve several environment variables used by X Windows and by SSH key forwarding.

It's possible to set different env_keep lists for different users or groups, but the configuration rapidly becomes complicated. We suggest sticking to a single, universal list and being relatively conservative with the exceptions you enshrine in the **sudoers** file.

If you need to preserve an environment variable that isn't listed in the **sudoers** file, you can set it explicitly on the **sudo** command line. For example, the command

```
$ sudo EDITOR=emacs vipw
```

edits the system password file with **emacs**. This feature has some potential restrictions, but they're waived for users who can run ALL commands.

***sudo** without passwords*

It's distressingly common to see **sudo** set up to allow command execution as root without the need to enter a password. Just for reference, that configuration is achieved with the NOPASSWD keyword in the **sudoers** file. For example:

```
ansible     ALL = (ALL) NOPASSWD: ALL    # Don't do this
```

See Chapter 23 for more information about Ansible.

Sometimes this is done out of laziness, but more typically, the underlying need is to allow some type of unattended **sudo** execution. The most common cases are when performing remote configuration through a system such as Ansible, or when running commands out of **cron**.

Needless to say, this configuration is dangerous, so avoid it if you can. At the very least, restrict passwordless execution to a specific set of commands if you can.

See this page for more information about PAM configuration.

Another option that works well in the context of remote execution is to replace manually entered passwords with authentication through **ssh-agent** and forwarded SSH keys. You can configure this method of authentication through PAM on the server where **sudo** will actually run.

Most systems don't include the PAM module that implements SSH-based authentication by default, but it is readily available. Look for a pam_ssh_agent_auth package.

SSH key forwarding has its own set of security concerns, but it's certainly an improvement over no authentication at all.

Precedence

A given invocation of **sudo** might potentially be addressed by several entries in the **sudoers** file. For example, consider the following configuration:

```
User_Alias      ADMINS = alice, bob, charles
User_Alias      MYSQL_ADMINS = alice, bob

%wheel          ALL = (ALL) ALL
MYSQL_ADMINS    ALL = (mysql) NOPASSWD: ALL
ADMINS          ALL = (ALL) NOPASSWD: /usr/sbin/logrotate
```

Here, administrators can run the **logrotate** command as any user without supplying a password. MySQL administrators can run any command as mysql without a password. Anyone in the wheel group can run any command under any UID, but must authenticate with a password first.

If user alice is in the wheel group, she is potentially covered by each of the last three lines. How do you know which one will determine **sudo**'s behavior?

The rule is that **sudo** always obeys the *last* matching line, with matching being determined by the entire 4-tuple of user, host, target user, and command. Each of those elements must match the configuration line, or the line is simply ignored.

Therefore, NOPASSWD exceptions must follow their more general counterparts, as shown above. If the order of the last three lines were reversed, poor alice would have to type a password no matter what **sudo** command she attempted to run.

__sudo__ without a control terminal

In addition to raising the issue of passwordless authentication, unattended execution of **sudo** (e.g., from **cron**) often occurs without a normal control terminal. There's nothing inherently wrong with that, but it's an odd situation that **sudo** can check for and reject if the requiretty option is turned on in the **sudoers** file.

This option is not the default from **sudo**'s perspective, but some OS distributions include it in their default **sudoers** files, so it's worth checking for and removing. Look for a line of the form

```
Defaults    requiretty
```

and invert its value:

```
Defaults    !requiretty
```

The requiretty option does offer a small amount of symbolic protection against certain attack scenarios. However, it's easy to work around and so offers little real security benefit. In our opinion, requiretty should be disabled as a matter of course because it is a common source of problems.

Site-wide __sudo__ configuration

Because the **sudoers** file includes the current host as a matching criterion for configuration lines, you can use one master **sudoers** file throughout an administrative domain (that is, a region of your site in which hostnames and user accounts are guaranteed to be name-equivalent). This approach makes the initial **sudoers** setup a bit more complicated, but it's a great idea, for multiple reasons. You should do it.

The main advantage of this approach is that there's no mystery about who has what permissions on what hosts. Everything is recorded in one authoritative file. When an administrator leaves your organization, for example, there's no need to track down all the hosts on which that user might have had **sudo** permissions. When changes are needed, you simply modify the master **sudoers** file and redistribute it.

A natural corollary of this approach is that **sudo** permissions might be better expressed in terms of user accounts rather than UNIX groups. For example,

```
%wheel      ALL = ALL
```

has some intuitive appeal, but it defers the enumeration of privileged users to each local machine. You can't look at this line and determine who's covered by it without an excursion to the

machine in question. Since the idea is to keep all relevant information in one place, it's best to avoid this type of grouping option when sharing a **sudoers** file on a network. Of course, if your group memberships are tightly coordinated site-wide, it's fine to use groups.

Distribution of the **sudoers** file is best achieved through a broader system of configuration management, as described in Chapter 23. But if you haven't yet reached that level of organization, you can easily roll your own. Be careful, though: installing a bogus **sudoers** file is a quick route to disaster. This is also a good file to keep an eye on with a file integrity monitoring solution of some kind; see this page.

In the absence of a configuration management system, it's best to use a "pull" script that runs out of **cron** on each host. Use **scp** to copy the current **sudoers** file from a known central repository, then validate it with **visudo -c -f** *newsudoers* before installation to verify that the format is acceptable to the local **sudo**. **scp** checks the remote server's host key for you, ensuring that the **sudoers** file is coming from the host you intended and not from a spoofed server.

Hostname specifications can be a bit subtle when sharing the **sudoers** file. By default, **sudo** uses the output of the **hostname** command as the text to be matched. Depending on the conventions in use at your site, this name may or may not include a domain portion (e.g., anchor vs. anchor.cs.colorado.edu). In either case, the hostnames specified in the **sudoers** file must match the hostnames as they are returned on each host. (You can turn on the `fqdn` option in the **sudoers** file to attempt to normalize local hostnames to their fully qualified forms.)

Hostname matching gets even stickier in the cloud, where instance names often default to algorithmically generated patterns. **sudo** understands simple pattern-matching characters (globbing) in hostnames, so consider adopting a naming scheme that incorporates some indication of each host's security classification from **sudo**'s perspective.

Alternatively, you can use your cloud provider's virtual networking features to segregate hosts by IP address, and then match on IP addresses instead of hostnames from within the **sudoers** file.

Disabling the root account

If your site standardizes on the use of **sudo**, you'll have surprisingly little use for actual root passwords. Most of your administrative team will never have occasion to use them.

That fact raises the question of whether a root password is necessary at all. If you decide that it isn't, you can disable root logins entirely by setting root's encrypted password to * or to some other fixed, arbitrary string. On Linux, **passwd -l** "locks" an account by prepending a ! to the encrypted password, with equivalent results.

The * and the ! are just conventions; no software checks for them explicitly. Their effect derives from their not being valid password hashes. As a result, attempts to verify root's password simply fail.

The main effect of locking the root account is that root cannot log in, even on the console. Neither can any user successfully run **su**, because that requires a root password check as well. However, the root account continues to exist, and all the software that usually runs as root continues to do so. In particular, **sudo** works normally.

The main advantage of disabling the root account is that you needn't record and manage root's password. You're also eliminating the possibility of the root password being compromised, but that's more a pleasant side effect than a compelling reason to go passwordless. Rarely used passwords are already at low risk of violation.

It's particularly helpful to have a real root password on physical computers (as opposed to cloud or virtual instances; see Chapters 9 and 24). Real computers are apt to require rescuing when hardware or configuration problems interfere with **sudo** or the boot process. In these cases, it's nice to have the traditional root account available as an emergency fallback.


Ubuntu ships with the root account locked, and all administrative access is funneled through **sudo** or a GUI equivalent. If you prefer, it's fine to set a root password on Ubuntu and then unlock the account with **sudo passwd -u root**.

System accounts other than root

Root is generally the only user that has special status in the eyes of the kernel, but several other pseudo-users are defined by most systems. You can identify these sham accounts by their low UIDs, usually less than 100. Most often, UIDs under 10 are system accounts, and UIDs between 10 and 100 are pseudo-users associated with specific pieces of software.

---

*See this page for more information about **shadow** and **master.passwd**.*

It's customary to replace the encrypted password field of these special users in the **shadow** or **master.passwd** file with a star so that their accounts cannot be logged in to. Their shells should be set to **/bin/false** or **/bin/nologin** as well, to protect against remote login exploits that use password alternatives such as SSH key files.

As with user accounts, most systems define a variety of system-related groups that have similarly low GIDs.

Files and processes that are part of the operating system but that need not be owned by root are sometimes assigned to the users bin or daemon. The theory was that this convention would help avoid the security hazards associated with ownership by root. It's not a compelling argument, however, and current systems often just use the root account instead.

The main advantage of defining pseudo-accounts and pseudo-groups is that they can be used more safely than the root account to provide access to defined groups of resources. For example, databases often implement elaborate access control systems of their own. From the perspective of the kernel, they run as a pseudo-user such as "mysql" that owns all database-related resources.

See this page for more information about the nobody account.

The Network File System (NFS) uses an account called "nobody" to represent root users on other systems. For remote roots to be stripped of their rootly powers, the remote UID 0 has to be mapped to something other than the local UID 0. The nobody account acts as a generic alter ego for these remote roots. In NFSv4, the nobody account can be applied to remote users that correspond to no valid local account as well.

Since the nobody account is supposed to represent a generic and relatively powerless user, it shouldn't own any files. If nobody does own files, remote roots can take control of them.

3.3 Extensions to the standard access control model

The preceding sections outline the major concepts of the traditional access control model. Even though this model can be summarized in a couple of pages, it has stood the test of time because it's simple, predictable, and capable of handling the requirements of the average site. All UNIX and Linux variants continue to support this model, and it remains the default approach and the one that's most widely used today.

As actually implemented and shipped on modern operating systems, the model includes a number of important refinements. Three layers of software contribute to the current status quo:

- The standard model as described to this point
- Extensions that generalize and fine-tune this basic model
- Kernel extensions that implement alternative approaches

These categories are not architectural layers so much as historical artifacts. Early UNIX derivatives all used the standard model, but its deficiencies were widely recognized even then. Over time, the community began to develop workarounds for a few of the more pressing issues. In the interest of maintaining compatibility and encouraging widespread adoption, changes were usually structured as refinements of the traditional system. Some of these tweaks (e.g., PAM) are now considered UNIX standards.

Over the last decade, great strides have been made toward modularization of access control systems. This evolution enables even more radical changes to access control. We review some of the more common pluggable options for Linux and FreeBSD in *Modern access control*.

For now, we look at some of the more prosaic extensions that are bundled with most systems. First, we consider the problems those extensions attempt to address.

Drawbacks of the standard model

Despite its elegance, the standard model has some obvious shortcomings.

- To begin with, the root account represents a potential single point of failure. If it's compromised, the integrity of the entire system is violated, and there is essentially no limit to the damage an attacker can inflict.
- The only way to subdivide the privileges of the root account is to write setuid programs. Unfortunately, as the steady flow of security-related software updates demonstrates, it's difficult to write secure software. Every setuid program is a potential target.
- The standard model has little to say about security on a network. No computer to which an unprivileged user has physical access can be trusted to accurately represent the ownerships of the processes it's running. Who's to say that someone hasn't reformatted the disk and installed their own operating system, with UIDs of their choosing?
- In the standard model, group definition is a privileged operation. For example, there's no way for a generic user to express the intent that only alice and bob should have access to a particular file.
- Because many access control rules are embedded in the code of individual commands and daemons (the classic example being the **passwd** program), you cannot redefine the system's behavior without modifying the source code and recompiling. In the real world, that's impractical and error prone.
- The standard model also has little or no support for auditing or logging. You can see which UNIX groups a user belongs to, but you can't necessarily determine what those group memberships permit a user to do. In addition, there's no real way to track the use of elevated privileges or to see what operations they have performed.

PAM: Pluggable Authentication Modules

---

*See this page for more information about the **shadow** and **master.passwd** files.*

User accounts are traditionally secured by passwords stored (in encrypted form) in the **/etc/shadow** or **/etc/master.passwd** file or an equivalent network database. Many programs may need to validate accounts, including **login**, **sudo**, **su**, and any program that accepts logins on a GUI workstation.

These programs really shouldn't have hard-coded expectations about how passwords are to be encrypted or verified. Ideally, they shouldn't even assume that passwords are in use at all. What if you want to use biometric identification, a network identity system, or some kind of two-factor authentication? Pluggable Authentication Modules to the rescue!

PAM is a wrapper for a variety of method-specific authentication libraries. Administrators specify the authentication methods they want the system to use, along with the appropriate contexts for each one. Programs that require user authentication simply call the PAM system rather than implement their own forms of authentication. PAM in turn calls the authentication library specified by the system administrator.

Strictly speaking, PAM is an authentication technology, not an access control technology. That is, instead of addressing the question "Does user X have permission to perform operation Y?", it helps answer the precursor question, "How do I know this is really user X?"

PAM is an important component of the access control chain on most systems, and PAM configuration is a common administrative task. You can find more details on PAM in the *Single Sign-On* chapter starting on this page.

Kerberos: network cryptographic authentication

Like PAM, Kerberos deals with authentication rather than access control per se. But whereas PAM is an authentication *framework*, Kerberos is a specific authentication *method*. At sites that use Kerberos, PAM and Kerberos generally work together, PAM being the wrapper and Kerberos the actual implementation.

Kerberos uses a trusted third party (a server) to perform authentication for an entire network. You don't authenticate yourself to the machine you are using, but provide your credentials to the Kerberos service. Kerberos then issues cryptographic credentials that you can present to other services as evidence of your identity.

Kerberos is a mature technology that has been in widespread use for decades. It's the standard authentication system used by Windows, and is part of Microsoft's Active Directory system. Read more about Kerberos here.

Filesystem access control lists

Since filesystem access control is so central to UNIX and Linux, it was an early target for elaboration. The most common addition has been support for access control lists (ACLs), a generalization of the traditional user/group/other permission model that permits permissions to be set for multiple users and groups at once.

ACLs are part of the filesystem implementation, so they have to be explicitly supported by whatever filesystem you are using. However, all major UNIX and Linux filesystems now support ACLs in one form or another.

See Chapter 21, The Network File System, for more information about NFS.

ACL support generally comes in one of two forms: an early POSIX draft standard that never quite made its way to formal adoption but was widely implemented anyway, and the system standardized by NFSv4, which adapts Microsoft Windows ACLs. Both ACL standards are described in more detail in the filesystem chapter, starting on this page.

Linux capabilities



Capability systems divide the powers of the root account into a handful (~30) of separate permissions.

The Linux version of capabilities derives from the defunct POSIX 1003.1e draft, which totters on despite never having been formally approved as a standard. In addition to bearing this zombie stigma, Linux capabilities raise the hackles of theorists because of nonconformance to the academic conception of a capability system. No matter; they're here, and Linux calls them capabilities, so we will too.

Capabilities can be inherited from a parent process. They can also be enabled or disabled by attributes set on an executable file, in a process reminiscent of setuid execution. Processes can renounce capabilities that they don't plan to use.

The traditional powers of root are simply the union of all possible capabilities, so there's a fairly direct mapping between the traditional model and the capability model. The capability model is just more granular.

As an example, the Linux capability called CAP_NET_BIND_SERVICE controls a process's ability to bind to privileged network ports (those numbered under 1,024). Some daemons that traditionally run as root need only this one particular superpower. In the capability world, such a daemon can theoretically run as an unprivileged user and pick up the port-binding capability from its executable file. As long as the daemon does not explicitly check to be sure that it's running as root, it needn't even be capability aware.

Is all this actually done in the real world? Well, no. As it happens, capabilities have evolved to become more an enabling technology than a user-facing system. They're widely employed by higher-level systems such as AppArmor (see this page) and Docker (see Chapter 25) but are rarely used on their own.

For administrators, it's helpful to review the **capabilities**(7) man page just to get a sense of what's included in each of the capability buckets.

Linux namespaces



Linux can segregate processes into hierarchical partitions (“namespaces”) from which they see only a subset of the system’s files, network ports, and processes. Among other effects, this scheme acts as a form of preemptive access control. Instead of having to base access control decisions on potentially subtle criteria, the kernel simply denies the existence of objects that are not visible from inside a given box.

Inside a partition, normal access control rules apply, and in most cases jailed processes are not even aware that they have been confined. Because confinement is irreversible, processes can run as root within a partition without fear that they might endanger other parts of the system.

This clever trick is one of the foundations of software containerization and its best-known implementation, Docker. The full system is a lot more sophisticated and includes extensions such as copy-on-write filesystem access. We have quite a bit more to say about containers in Chapter 25.

As a form of access control, namespacing is a relatively coarse approach. The construction of properly configured nests for processes to live in is also somewhat tricky. Currently, this technology is applied primarily to add-on services as opposed to intrinsic components of the operating system.

3.4 Modern access control

Given the world's wide range of computing environments and the mixed success of efforts to advance the standard model, kernel maintainers have been reluctant to act as mediators in the larger debate over access control. In the Linux world, the situation came to a head in 2001, when the U.S. National Security Agency proposed to integrate its Security-Enhanced Linux (SELinux) system into the kernel as a standard facility.

For several reasons, the kernel maintainers resisted this merge. Instead of adopting SELinux or another, alternative system, they developed the Linux Security Modules API, a kernel-level interface that allows access control systems to integrate themselves as loadable kernel modules.

LSM-based systems have no effect unless users load them and turn them on. This fact lowers the barriers for inclusion in the standard kernel, and Linux now ships with SELinux and four other systems (AppArmor, Smack, TOMOYO, and Yama) ready to go.

Developments on the BSD side have roughly paralleled those of Linux, thanks largely to Robert Watson's work on TrustedBSD. This code has been included in FreeBSD since version 5. It also provides the application sandboxing technology used in Apple's macOS and iOS.

When multiple access control modules are active simultaneously, an operation must be approved by all of them to be permitted. Unfortunately, the LSM system requires explicit cooperation among active modules, and none of the current modules include this feature. For now, Linux systems are effectively limited to a choice of one LSM add-on module.

Separate ecosystems

Access control is inherently a kernel-level concern. With the exception of filesystem access control lists (see this page), there is essentially no standardization among systems with regard to alternative access control mechanisms. As a result, every kernel has its own array of available implementations, and none of them are cross-platform.



Because Linux distributions share a common kernel lineage, all Linux distributions are theoretically compatible with all the various Linux security offerings. But in practical terms, they're not: these systems all need user-level support in the form of additional commands, modifications to user-level components, and securement profiles for daemons and services. Ergo, every distribution has only one or two access control mechanisms that it actively supports (if that).

Mandatory access control

The standard UNIX model is considered a form of "discretionary access control" because it allows the owners of access-controlled entities to set the permissions on them. For example, you might allow other users to view the contents of your home directory, or you might write a setuid program that lets other people send signals to your processes.

Discretionary access control provides no particular guarantee of security for user-level data. The downside of letting users set permissions is that users can set permissions; there's no telling what they might do with their own files. And even with the best intentions and training, users can make mistakes.

Mandatory access control (aka MAC) systems let administrators write access control policies that override or supplement the discretionary permissions of the traditional model. For example, you might establish the rule that users' home directories are accessible only by their owners. It then doesn't matter if a user makes a private copy of a sensitive document and is careless with the document's permissions; nobody else can see into that user's home directory anyway.

MAC capabilities are an enabling technology for implementing security models such as the Department of Defense's "multilevel security" system. In this model, security policies control access according to the perceived sensitivity of the resources being controlled. Users are assigned a security classification from a structured hierarchy. They can read and write items at the same classification level or lower but cannot access items at a higher classification. For example, a user with "secret" access can read and write "secret" objects but cannot read objects that are classified as "top secret."

Unless you're handling sensitive data for a government entity, it is unlikely that you will ever encounter or need to deploy such comprehensive "foreign" security models. More commonly, MAC is used to protect individual services, and it otherwise stays out of users' way.

A well-implemented MAC policy relies on the principle of least privilege (allowing access only when necessary), much as a properly designed firewall allows only specifically recognized services and clients to pass. MAC can prevent software with code execution vulnerabilities (e.g., buffer overflows) from compromising the system by limiting the scope of the breach to the few specific resources required by that software.

MAC has unfortunately become something of a buzzword synonymous with "advanced access control." Even FreeBSD's generic security API is called the MAC interface, despite the fact that some plug-ins offer no actual MAC features.

Available MAC systems range from wholesale replacements for the standard model to lightweight extensions that address specific domains and use cases. The common thread among MAC implementations is that they generally add centralized, administrator-written (or vendor-supplied) policies into the access control system along with the usual mix of file permissions, access controls lists, and process attributes.

Regardless of scope, MAC represents a potentially significant departure from the standard system, and it's one that programs expecting to deal with the standard UNIX security model may find surprising. Before committing to a full-scale MAC deployment, make sure you understand the module's logging conventions and know how to identify and troubleshoot MAC-related problems.

Role-based access control

Another feature commonly name-checked by access control systems is role-based access control (aka RBAC), a theoretical model formalized in 1992 by David Ferraiolo and Rick Kuhn. The basic idea is to add a layer of indirection to access control calculations. Permissions, instead of being assigned directly to users, are assigned to intermediate constructs known as “roles,” and roles in turn are assigned to users. To make an access control decision, the system enumerates the roles of the current user and checks to see if any of those roles have the appropriate permissions.

Roles are similar in concept to UNIX groups, but they’re more general because they can be used outside the context of the filesystem. Roles can also have a hierarchical relationship to one another, a fact that greatly simplifies administration. For example, you might define a “senior administrator” role that has all the permissions of an “administrator” plus the additional permissions X, Y, and Z.

Many UNIX variants, including Solaris, HP-UX, and AIX, include some form of built-in RBAC system. Linux and FreeBSD have no distinct, native RBAC facility. However, it is built into several of the more comprehensive MAC options.

SELinux: Security-Enhanced Linux

SELinux is one of the oldest Linux MAC implementations and is a product of the U.S. National Security Agency. Depending on one's perspective, that might be a source of either comfort or suspicion. (If your tastes incline toward suspicion, it's worth noting that as part of the Linux kernel distribution, the SELinux code base is open to inspection.)

SELinux takes a maximalist approach, and it implements pretty much every flavor of MAC and RBAC one might envision. Although it has gained footholds in a few distributions, it is notoriously difficult to administer and troubleshoot. This unattributed quote from a former version of the SELinux Wikipedia page vents the frustration felt by many sysadmins:

> *Intriguingly, although the stated raison d'être of SELinux is to facilitate the creation of individualized access control policies specifically attuned to organizational data custodianship practices and rules, the supportive software tools are so sparse and unfriendly that the vendors survive chiefly on "consulting," which typically takes the form of incremental modifications to boilerplate security policies.*

Despite its administrative complexity, SELinux adoption has been slowly growing, particularly in environments such as government, finance, and health care that enforce strong and specific security requirements. It's also a standard part of the Android platform.

Our general opinion regarding SELinux is that it's capable of delivering more harm than benefit. Unfortunately, that harm can manifest not only as wasted time and aggravation for system administrators, but also, ironically, as security lapses. Complex models are hard to reason about, and SELinux isn't really a level playing field; hackers that focus on it understand the system far more thoroughly than the average sysadmin.

In particular, SELinux policy development is a complicated endeavor. To protect a new daemon, for example, a policy must carefully enumerate all the files, directories, and other objects to which the process needs access. For complicated software like **sendmail** or **httpd**, this task can be quite complex. At least one company offers a three-day class on policy development.

Fortunately, many general policies are available on-line, and most SELinux-enabled distributions come with reasonable defaults. These can easily be installed and configured for your particular environment. A full-blown policy editor that aims to ease policy application can be found at seedit.sourceforge.net.

RHEL  SELinux is well supported by both Red Hat (and hence, CentOS) and Fedora. Red Hat enables it by default.

 Debian and SUSE Linux also have some available support for SELinux, but you must install additional packages, and the system is less aggressive in its default configuration.



Ubuntu inherits some SELinux support from Debian, but over the last few releases, Ubuntu's focus has been on AppArmor (see this page). Some vestigial SELinux-related packages are still available, but they are generally not up to date.

/etc/selinux/config is the top-level control for SELinux. The interesting lines are

```
SELINUX=enforcing
SELINUXTYPE=targeted
```

The first line has three possible values: enforcing, permissive, or disabled. The enforcing setting ensures that the loaded policy is applied and prohibits violations. permissive allows violations to occur but logs them through syslog, which is valuable for debugging and policy development. disabled turns off SELinux entirely.

SELINUXTYPE refers to the name of the policy database to be applied. This is essentially the name of a subdirectory within **/etc/selinux**. Only one policy can be active at a time, and the available policy sets vary by system.

RHEL  Red Hat's default policy is targeted, which defines additional security for a few daemons that Red Hat has explicitly protected but leaves the rest of the system alone. There used to be a separate policy called strict that applied MAC to the entire system, but that policy has now been merged into targeted. Remove the unconfined and unconfineduser modules with **semodule -d** to achieve full-system MAC.

Red Hat also defines an mls policy that implements DoD-style multilevel security. You must install it separately with **yum install selinux-policy-mls**.

If you're interested in developing your own SELinux policies, check out the **audit2allow** utility. It builds policy definitions from logs of violations. The idea is to permissively protect a subsystem so that its violations are logged but not enforced. You can then put the subsystem through its paces and build a policy that allows everything the subsystem actually did. Unfortunately, it's hard to guarantee complete coverage of all code paths with this sort of ad hoc approach, so the autogenerated profiles are unlikely to be perfect.

AppArmor



AppArmor is a product of Canonical, Ltd., releasers of the Ubuntu distribution. It's supported by Debian and Ubuntu, but has also been adopted as a standard by SUSE distributions. Ubuntu and SUSE enable it on default installs, although the complement of protected services is not extensive.

AppArmor implements a form of MAC and is intended as a supplement to the traditional UNIX access control system. Although any configuration is possible, AppArmor is not designed to be a user-facing system. Its main goal is service securement; that is, limiting the damage that individual programs can do if they should be compromised or run amok.

Protected programs continue to be subject to all the limitations imposed by the standard model, but in addition, the kernel filters their activities through a designated and task-specific AppArmor profile. By default, AppArmor denies all requests, so the profile must explicitly name everything the process is allowed to do. Programs without profiles, such as user shells, have no special restrictions and run as if AppArmor were not installed.

This service securement role is essentially the same configuration that's implemented by SELinux in Red Hat's targeted environment. However, AppArmor is designed more specifically for service securement, so it sidesteps some of the more puzzling nuances of SELinux.

AppArmor profiles are stored in **/etc/apparmor.d**, and they're relatively readable even without detailed knowledge of the system. For example, here's the profile for the **cups-browsed** daemon, part of the printing system on Ubuntu:

```
#include <tunables/global>

/usr/sbin/cups-browsed {

    #include <abstractions/base>
    #include <abstractions/nameservice>
    #include <abstractions/cups-client>
    #include <abstractions/dbus>
    #include <abstractions/p11-kit>

    /etc/cups/cups-browsed.conf r,
    /etc/cups/lpoptions r,
    /{var/,}run/cups/certs/* r,
    /var/cache/cups/* rw,
    /tmp/** rw,

    # Site-specific additions and overrides. See local/README.
    #include <local/usr.sbin.cups-browsed>
}
```

Most of this code is modular boilerplate. For example, this daemon needs to perform hostname lookups, so the profile interpolates **abstractions/nameservice**, which gives access to name

resolution libraries, **/etc/nsswitch.conf**, **/etc/hosts**, the network ports used with LDAP, and so on.

The profiling information that's specific to this daemon consists (in this case) of a list of files the daemon can access, along with the permissions allowed on each file. The pattern matching syntax is a bit idiosyncratic: `**` can match multiple pathname components, and `{var/,}` matches whether `var/` appears at that location or not.

Even this simple profile is quite complex under the hood. With all the `#include` instructions expanded, the profile is nearly 750 lines long. (And we chose this example for its brevity. Yikes!)

AppArmor refers to files and programs by pathname, which makes profiles readable and independent of any particular filesystem implementation. This approach is something of a compromise, however. For example, AppArmor doesn't recognize hard links as pointing to the same underlying entity.

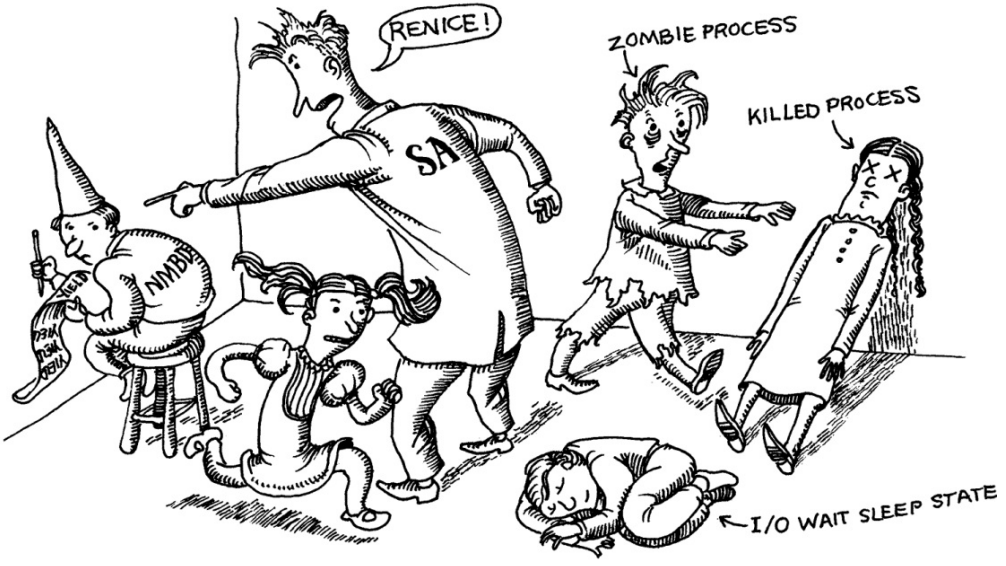
3.5 Recommended reading

Ferraiolo, David F., D. Richard Kuhn, and Ramaswamy Chandramouli. *Role-Based Access Control (2nd Edition).* Boston, MA: Artech House, 2007.

Haines, Richard. *The SELinux Notebook (4th Edition).* 2014. This compendium of SELinux-related information is the closest thing to official documentation. It's available for download from freecomputerbooks.com.

Vermeulen, Sven. *SELinux Cookbook.* Birmingham, UK: Packt Publishing, 2014. This book includes a variety of practical tips for dealing with SELinux. It covers both service securement and user-facing security models.

4 Process Control



A process represents a running program. It's the abstraction through which memory, processor time, and I/O resources can be managed and monitored.

It is an axiom of the UNIX philosophy that as much work as possible be done within the context of processes rather than being handled specially by the kernel. System and user processes follow the same rules, so you can use a single set of tools to control them both.

4.1 Components of a process

A process consists of an address space and a set of data structures within the kernel. The address space is a set of memory pages that the kernel has marked for the process's use. (Pages are the units in which memory is managed. They are usually 4KiB or 8KiB in size.) These pages contain the code and libraries that the process is executing, the process's variables, its stacks, and various extra information needed by the kernel while the process is running. The process's virtual address space is laid out randomly in physical memory and tracked by the kernel's page tables.

The kernel's internal data structures record various pieces of information about each process. Here are some of the more important of these:

- The process's address space map
- The current status of the process (sleeping, stopped, runnable, etc.)
- The execution priority of the process
- Information about the resources the process has used (CPU, memory, etc.)
- Information about the files and network ports the process has opened
- The process's signal mask (a record of which signals are blocked)
- The owner of the process

A "thread" is an execution context within a process. Every process has at least one thread, but some processes have many. Each thread has its own stack and CPU context but operates within the address space of its enclosing process.

Modern computer hardware includes multiple CPUs and multiple cores per CPU. A process's threads can run simultaneously on different cores. Multithreaded applications such as BIND and Apache benefit quite a bit from this architecture because it lets them farm out requests to individual threads.

Many of the parameters associated with a process directly affect its execution: the amount of processor time it gets, the files it can access, and so on. In the following sections, we discuss the meaning and significance of the parameters that are most interesting from a system administrator's point of view. These attributes are common to all versions of UNIX and Linux.

PID: process ID number

The kernel assigns a unique ID number to every process. Most commands and system calls that manipulate processes require you to specify a PID to identify the target of the operation. PIDs are assigned in order as processes are created.



Linux now defines the concept of process "namespaces," which further restrict processes' ability to see and affect each other. Container implementations use this feature to keep processes segregated. One side effect is that a process might appear to have different PIDs depending on the namespace of the observer. It's kind of like Einsteinian relativity for process IDs. Refer to Chapter 25, *Containers*, for more information.

PPID: parent PID

Neither UNIX nor Linux has a system call that initiates a new process running a particular program. Instead, it's done in two separate steps. First, an existing process must clone itself to create a new process. The clone can then exchange the program it's running for a different one.

When a process is cloned, the original process is referred to as the parent, and the copy is called the child. The PPID attribute of a process is the PID of the parent from which it was cloned, at least initially. (If the original parent dies, **init** or **systemd** becomes the new parent. See this page.)

The parent PID is a useful piece of information when you're confronted with an unrecognized (and possibly misbehaving) process. Tracing the process back to its origin (whether that is a shell or some other program) may give you a better idea of its purpose and significance.

UID and EUID: real and effective user ID

See this page for more information about UIDs.

A process's UID is the user identification number of the person who created it, or more accurately, it is a copy of the UID value of the parent process. Usually, only the creator (aka, the owner) and the superuser can manipulate a process.

See this page for more information about setuid execution.

The EUID is the "effective" user ID, an extra UID that determines what resources and files a process has permission to access at any given moment. For most processes, the UID and EUID are the same, the usual exception being programs that are setuid.

Why have both a UID and an EUID? Simply because it's useful to maintain a distinction between identity and permission, and because a setuid program might not wish to operate with expanded permissions all the time. On most systems, the effective UID can be set and reset to enable or restrict the additional permissions it grants.

Most systems also keep track of a "saved UID," which is a copy of the process's EUID at the point at which the process first begins to execute. Unless the process takes steps to obliterate this saved UID, it remains available for use as the real or effective UID. A conservatively written setuid program can therefore renounce its special privileges for the majority of its execution and access them only at the points where extra privileges are needed.



Linux also defines a nonstandard FSUID process parameter that controls the determination of filesystem permissions. It is infrequently used outside the kernel and is not portable to other UNIX systems.

GID and EGID: real and effective group ID

See this page for more information about groups.

The GID is the group identification number of a process. The EGID is related to the GID in the same way that the EUID is related to the UID in that it can be “upgraded” by the execution of a setgid program. As with the saved UID, the kernel maintains a saved GID for each process.

The GID attribute of a process is largely vestigial. For purposes of access determination, a process can be a member of many groups at once. The complete group list is stored separately from the distinguished GID and EGID. Determinations of access permissions normally take into account the EGID and the supplemental group list, but not the GID itself.

The only time at which the GID is actually significant is when a process creates new files. Depending on how the filesystem permissions have been set, new files might default to adopting the GID of the creating process. See this page for details.

Niceness

A process's scheduling priority determines how much CPU time it receives. The kernel computes priorities with a dynamic algorithm that takes into account the amount of CPU time that a process has recently consumed and the length of time it has been waiting to run. The kernel also pays attention to an administratively set value that's usually called the "nice value" or "niceness," so called because it specifies how nice you are planning to be to other users of the system. We discuss niceness in detail here.

Control terminal

See this page for more information about the standard communication channels.

Most nondaemon processes have an associated control terminal. The control terminal determines the default linkages for the standard input, standard output, and standard error channels. It also distributes signals to processes in response to keyboard events such as <Control-C>; see the discussion starting here.

Of course, actual terminals are rare outside of computer museums these days. Nevertheless, they live on in the form of pseudo-terminals, which are still widely used throughout UNIX and Linux systems. When you start a command from the shell, for example, your terminal window typically becomes the process's control terminal.

4.2 THE LIFE CYCLE OF A PROCESS

To create a new process, a process copies itself with the **fork** system call. **fork** creates a copy of the original process, and that copy is largely identical to the parent. The new process has a distinct PID and has its own accounting information. (Technically, Linux systems use **clone**, a superset of **fork** that handles threads and includes additional features. **fork** remains in the kernel for backward compatibility but calls **clone** behind the scenes.)

fork has the unique property of returning two different values. From the child's point of view, it returns zero. The parent receives the PID of the newly created child. Since the two processes are otherwise identical, they must both examine the return value to figure out which role they are supposed to play.

After a **fork**, the child process often uses one of the **exec** family of routines to begin the execution of a new program. These calls change the program that the process is executing and reset the memory segments to a predefined initial state. The various forms of **exec** differ only in the ways in which they specify the command-line arguments and environment to be given to the new program.

When the system boots, the kernel autonomously creates and installs several processes. The most notable of these is **init** or **systemd**, which is always process number 1. This process executes the system's startup scripts, although the exact manner in which this is done differs slightly between UNIX and Linux. All processes other than the ones the kernel creates are descendants of this primordial process. See Chapter 2, *Booting and System Management Daemons*, for more information about booting and the various flavors of **init** daemon.

init (or **systemd**) also plays another important role in process management. When a process completes, it calls a routine named **_exit** to notify the kernel that it is ready to die. It supplies an exit code (an integer) that tells why it's exiting. By convention, zero indicates a normal or "successful" termination.

Before a dead process can be allowed to disappear completely, the kernel requires that its death be acknowledged by the process's parent, which the parent does with a call to **wait**. The parent receives a copy of the child's exit code (or if the child did not exit voluntarily, an indication of why it was killed) and can also obtain a summary of the child's resource use if it wishes.

This scheme works fine if parents outlive their children and are conscientious about calling **wait** so that dead processes can be disposed of. If a parent dies before its children, however, the kernel recognizes that no **wait** is forthcoming. The kernel adjusts the orphan processes to make them children of **init** or **systemd**, which politely performs the **wait** needed to get rid of them when they die.

Signals

Signals are process-level interrupt requests. About thirty different kinds are defined, and they're used in a variety of ways:

- They can be sent among processes as a means of communication.
- They can be sent by the terminal driver to kill, interrupt, or suspend processes when keys such as <Control-C> and <Control-Z> are pressed.
- They can be sent by an administrator (with **kill**) to achieve various ends.
- They can be sent by the kernel when a process commits an infraction such as division by zero.
- They can be sent by the kernel to notify a process of an "interesting" condition such as the death of a child process or the availability of data on an I/O channel.

When a signal is received, one of two things can happen. If the receiving process has designated a handler routine for that particular signal, the handler is called with information about the context in which the signal was delivered. Otherwise, the kernel takes some default action on behalf of the process. The default action varies from signal to signal. Many signals terminate the process; some also generate core dumps (if core dumps have not been disabled; a core dump is a copy of a process's memory image, which is sometimes useful for debugging).

Specifying a handler routine for a signal is referred to as catching the signal. When the handler completes, execution restarts from the point at which the signal was received.

To prevent signals from arriving, programs can request that they be either ignored or blocked. A signal that is ignored is simply discarded and has no effect on the process. A blocked signal is queued for delivery, but the kernel doesn't require the process to act on it until the signal has been explicitly unblocked. The handler for a newly unblocked signal is called only once, even if the signal was received several times while reception was blocked.

Table 4.1 lists some signals that administrators should be familiar with. The uppercase convention for the names derives from C language tradition. You might also see signal names written with a SIG prefix (e.g., SIGHUP) for similar reasons.

Table 4.1: Signals every administrator should know

# [b]	Name	Description	Default	Can catch?	Can block?	Dump core?
1	HUP	Hangup	Terminate	Yes	Yes	No
2	INT	Interrupt	Terminate	Yes	Yes	No
3	QUIT	Quit	Terminate	Yes	Yes	Yes
9	KILL	Kill	Terminate	No	No	No
10	BUS	Bus error	Terminate	Yes	Yes	Yes
11	SEGV	Segmentation fault	Terminate	Yes	Yes	Yes
15	TERM	Software termination	Terminate	Yes	Yes	No
17	STOP	Stop	Stop	No	No	No
18	TSTP	Keyboard stop	Stop	Yes	Yes	No
19	CONT	Continue after stop	Ignore	Yes	No	No
28	WINCH	Window changed	Ignore	Yes	Yes	No
30	USR1	User-defined #1	Terminate	Yes	Yes	No
31	USR2	User-defined #2	Terminate	Yes	Yes	No

a. A list of signal names and numbers is also available from the **bash** built-in command **kill -l**.
b. May vary on some systems. See **/usr/include/signal.h** or **man signal** for more information.

Other signals, not shown in Table 4.1, mostly report obscure errors such as “illegal instruction.” The default handling for such signals is to terminate with a core dump. Catching and blocking are generally allowed because some programs are smart enough to try to clean up whatever problem caused the error before continuing.

The BUS and SEGV signals are also error signals. We’ve included them in the table because they’re so common: when a program crashes, it’s usually one of these two signals that finally brings it down. By themselves, the signals are of no specific diagnostic value. Both of them indicate an attempt to use or access memory improperly.

The signals named KILL and STOP cannot be caught, blocked, or ignored. The KILL signal destroys the receiving process, and STOP suspends its execution until a CONT signal is received. CONT can be caught or ignored, but not blocked.

TSTP is a “soft” version of STOP that might be best described as a request to stop. It’s the signal generated by the terminal driver when <Control-Z> is typed on the keyboard. Programs that catch this signal usually clean up their state, then send themselves a STOP signal to complete the stop operation. Alternatively, programs can ignore TSTP to prevent themselves from being stopped from the keyboard.

The signals KILL, INT, TERM, HUP, and QUIT all sound as if they mean approximately the same thing, but their uses are actually quite different. It’s unfortunate that such vague terminology was selected for them. Here’s a decoding guide:

- KILL is unblockable and terminates a process at the kernel level. A process can never actually receive or handle this signal.

- INT is sent by the terminal driver when the user presses <Control-C>. It's a request to terminate the current operation. Simple programs should quit (if they catch the signal) or simply allow themselves to be killed, which is the default if the signal is not caught. Programs that have interactive command lines (such as shells) should stop what they're doing, clean up, and wait for user input again.

- TERM is a request to terminate execution completely. It's expected that the receiving process will clean up its state and exit.

- HUP has two common interpretations. First, it's understood as a reset request by many daemons. If a daemon is capable of rereading its configuration file and adjusting to changes without restarting, a HUP can generally trigger this behavior.

  Second, HUP signals are sometimes generated by the terminal driver in an attempt to "clean up" (i.e., kill) the processes attached to a particular terminal. This behavior is largely a holdover from the days of wired terminals and modem connections, hence the name "hangup."

  Shells in the C shell family (**tcsh** et al.) usually make background processes immune to HUP signals so that they can continue to run after the user logs out. Users of Bourne-ish shells (**ksh**, **bash**, etc.) can emulate this behavior with the **nohup** command.

- QUIT is similar to TERM, except that it defaults to producing a core dump if not caught. A few programs cannibalize this signal and interpret it to mean something else.

The signals USR1 and USR2 have no set meaning. They're available for programs to use in whatever way they'd like. For example, the Apache web server interprets a HUP signal as a request for an immediate restart. A USR1 signal initiates a more graceful transition in which existing client conversations are allowed to finish.

kill: send signals

As its name implies, the **kill** command is most often used to terminate a process. **kill** can send any signal, but by default it sends a TERM. **kill** can be used by normal users on their own processes or by root on any process. The syntax is

```
kill [-signal] pid
```

where *signal* is the number or symbolic name of the signal to be sent (as shown in Table 4.1) and *pid* is the process identification number of the target process.

A **kill** without a signal number does not guarantee that the process will die, because the TERM signal can be caught, blocked, or ignored. The command

```
$ kill -9 pid
```

"guarantees" that the process will die because signal 9, KILL, cannot be caught. Use **kill -9** only if a polite request fails. We put quotes around "guarantees" because processes can on occasion become so wedged that even KILL does not affect them, usually because of some degenerate I/O vapor lock such as waiting for a volume that has disappeared. Rebooting is usually the only way to get rid of these processes.

killall kills processes by name. For example, the following command kills all Apache web server processes:

```
$ sudo killall httpd
```

The **pkill** command searches for processes by name (or other attributes, such as EUID) and sends the specified signal. For example, the following command sends a TERM signal to all processes running as the user ben:

```
$ sudo pkill -u ben
```

Process and thread states

As you saw in the previous section, a process can be suspended with a STOP signal and returned to active duty with a CONT signal. The state of being suspended or runnable applies to the process as a whole and is inherited by all the process's threads. (Individual threads can in fact be managed similarly. However, those facilities are primarily of interest to developers; system administrators needn't concern themselves.)

Even when nominally runnable, threads must often wait for the kernel to complete some background work for them before they can continue execution. For example, when a thread reads data from a file, the kernel must request the appropriate disk blocks and then arrange for their contents to be delivered into the requesting process's address space. During this time, the requesting thread enters a short-term sleep state in which it is ineligible to execute. Other threads in the same process can continue to run, however.

You'll sometimes see entire processes described as "sleeping" (for example, in **ps** output—see the next section). Since sleeping is a thread-level attribute, this convention is a bit deceptive. A process is generally reported as "sleeping" when all its threads are asleep. Of course, the distinction is moot in the case of single-threaded processes, which remain the most common case.

Interactive shells and system daemons spend most of their time sleeping, waiting for terminal input or network connections. Since a sleeping thread is effectively blocked until its request has been satisfied, its process generally receives no CPU time unless it receives a signal or a response to one of its I/O requests.

See this page for more information about hard-mounting NFS filesystems.

Some operations can cause processes or threads to enter an uninterruptible sleep state. This state is usually transient and is not observed in **ps** output (denoted by a D in the STAT column; see Table 4.2). However, a few degenerate situations can cause it to persist. The most common cause involves server problems on an NFS filesystem mounted with the hard option. Since processes in the uninterruptible sleep state cannot be roused even to service a signal, they cannot be killed. To get rid of them, you must fix the underlying problem or reboot.

In the wild, you might occasionally see "zombie" processes that have finished execution but that have not yet had their status collected by their parent process (or by **init** or **systemd**). If you see zombies hanging around, check their PPIDs with **ps** to find out where they're coming from.

4.3 PS: MONITOR PROCESSES

The **ps** command is the system administrator's main tool for monitoring processes. Although versions of **ps** differ in their arguments and display, they all deliver essentially the same information. Part of the enormous variation among versions of **ps** can be traced back to differences in the development history of UNIX. However, **ps** is also a command that vendors tend to customize for other reasons. It's closely tied to the kernel's handling of processes, so it tends to reflect all a vendor's underlying kernel changes.

ps can show the PID, UID, priority, and control terminal of processes. It also informs you how much memory a process is using, how much CPU time it has consumed, and what its current status is (running, stopped, sleeping, etc.). Zombies show up in a **ps** listing as <exiting> or <defunct>.

Implementations of **ps** have become hopelessly complex over the years. Several vendors have abandoned the attempt to define meaningful displays and made their **ps**es completely configurable. With a little customization work, almost any desired output can be produced.



As a case in point, the **ps** used by Linux is a highly polymorphous version that understands option sets from multiple historical lineages. Almost uniquely among UNIX commands, Linux's **ps** accepts command-line flags with or without dashes but might assign different interpretations to those forms. For example, **ps -a** is not the same as **ps a**.

Do not be alarmed by all this complexity: it's there mainly for developers, not for system administrators. Although you will use **ps** frequently, you only need to know a few specific incantations.

You can obtain a useful overview of all the processes running on the system with **ps aux**. The **a** option says show all processes, and **x** says show even processes that don't have a control terminal; **u** selects the "user oriented" output format. Here's an example of **ps aux** output on a machine running Red Hat:

```
redhat$ ps aux
  USER   PID %CPU %MEM    VSZ   RSS  TTY STAT  TIME  COMMAND
  root     1  0.1   0.2   3356   560  ?   S     0:00  init [5]
  root     2    0     0      0     0  ?   SN    0:00  [ksoftirqd/0]
  root     3    0     0      0     0  ?   S<    0:00  [events/0]
  root     4    0     0      0     0  ?   S<    0:00  [khelper]
  root     5    0     0      0     0  ?   S<    0:00  [kacpid]
  root    18    0     0      0     0  ?   S<    0:00  [kblockd/0]
  root    28    0     0      0     0  ?   S     0:00  [pdflush]
...
  root   196    0     0      0     0  ?   S     0:00  [kjournald]
  root  1050    0   0.1   2652   448  ?   S<s   0:00  udevd
  root  1472    0   0.3   3048  1008  ?   S<s   0:00  /sbin/dhclient -1
  root  1646    0   0.3   3012  1012  ?   S<s   0:00  /sbin/dhclient -1
  root  1733    0     0      0     0  ?   S     0:00  [kjournald]
  root  2124    0   0.3   3004  1008  ?   Ss    0:00  /sbin/dhclient -1
  root  2182    0   0.2   2264   596  ?   Ss    0:00  rsyslog -m 0
  root  2186    0   0.1   2952   484  ?   Ss    0:00  klogd -x
  root  2519  0.0   0.0  17036   380  ?   Ss    0:00  /usr/sbin/atd
  root  2384    0   0.6   4080  1660  ?   Ss    0:00  /usr/sbin/sshd
  root  2419    0   1.1   7776  3004  ?   Ss    0:00  sendmail: accept
...
```

Command names in brackets are not really commands at all but rather kernel threads scheduled as processes. The meaning of each field is shown in Table 4.2.

Table 4.2: Explanation of ps aux output

Field	Contents
USER	Username of the process's owner
PID	Process ID
%CPU	Percentage of the CPU this process is using
%MEM	Percentage of real memory this process is using
VSZ	Virtual size of the process
RSS	Resident set size (number of pages in memory)
TTY	Control terminal ID
STAT	Current process status: R = Runnable D = In uninterruptible sleep S = Sleeping (< 20 sec) T = Traced or stopped Z = Zombie Additional flags: W = Process is swapped out < = Process has higher than normal priority N = Process has lower than normal priority L = Some pages are locked in core s = Process is a session leader
TIME	CPU time the process has consumed
COMMAND	Command name and arguments [a]

a. Programs can modify this information, so it's not necessarily an accurate representation of the actual command line.

Another useful set of arguments is **lax**, which gives more technical information. The **a** and **x** options are as above (show every process), and **l** selects the "long" output format. **ps lax** might be slightly faster to run than **ps aux** because it doesn't have to translate every UID to a username —efficiency can be important if the system is already bogged down.

Shown here in an abbreviated example, **ps lax** includes fields such as the parent process ID (PPID), niceness (NI), and the type of resource on which the process is waiting (WCHAN, short for "wait channel").

```
redhat$ ps lax
  F  UID   PID  PPID  PRI   NI    VSZ   RSS  WCHAN  STAT  TIME  COMMAND
  4    0     1     0   16    0   3356   560  select  S     0:00  init [5]
  1    0     2     1   34   19      0     0  ksofti  SN    0:00  [ksoftirqd/0
  1    0     3     1    5  -10      0     0  worker  S<    0:00  [events/0]
  1    0     4     3    5  -10      0     0  worker  S<    0:00  [khelper]
  5    0  2186     1   16    0   2952   484  syslog  Ss    0:00  klogd -x
  5   32  2207     1   15    0   2824   580  -       Ss    0:00  portmap
  5   29  2227     1   18    0   2100   760  select  Ss    0:00  rpc.statd
  1    0  2260     1   16    0   5668  1084  -       Ss    0:00  rpc.idmapd
  1    0  2336     1   21    0   3268   556  select  Ss    0:00  acpid
  5    0  2384     1   17    0   4080  1660  select  Ss    0:00  sshd
  1    0  2399     1   15    0   2780   828  select  Ss    0:00  xinetd -sta
  5    0  2419     1   16    0   7776  3004  select  Ss    0:00  sendmail: a
...
```

Commands with long argument lists may have the command-line output cut off. Add **w** to the list of flags to display more columns in the output. Add **w** twice for unlimited column width, handy for those processes that have exceptionally long command-line arguments, such as some **java** applications.

Administrators frequently need to identify the PID of a process. You can find the PID by **grep**ping the output of **ps**:

```
$ ps aux | grep sshd
root     6811 0.0 0.0  78056 1340 ?     Ss 16:04 0:00 /usr/sbin/sshd
bwhaley 13961 0.0 0.0 110408  868 pts/1 S+ 20:37 0:00 grep /usr/sbin/sshd
```

Note that the **ps** output includes the **grep** command itself, since the **grep** was active in the process list at the time **ps** was running. You can remove this line from the output with **grep -v**:

```
$ ps aux | grep -v grep | grep sshd
root     6811 0.0 0.0  78056 1340 ?      Ss 16:04 0:00 /usr/sbin/sshd
```

You can also determine the PID of a process with the **pidof** command:

```
$ pidof /usr/sbin/sshd
6811
```

Or with the **pgrep** utility:

```
$ pgrep sshd
6811
```

pidof and **pgrep** show all processes that match the passed string. We often find a simple **grep** to offer the most flexibility, though it can be a bit more verbose.

4.4 Interactive monitoring with top

Commands like **ps** show you a snapshot of the system as it was at the time. Often, that limited sample is insufficient to convey the big picture of what's really going on. **top** is a sort of real-time version of **ps** that gives a regularly updated, interactive summary of processes and their resource usage. For example:

```
redhat$ top
top - 20:07:43 up  1:59,  3 users,  load average: 0.45, 0.16, 0.09
Tasks: 251 total,   1 running, 250 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.7 us, 1.2 sy, 0.0 ni, 98.0 id, 0.0 wa, 0.0 hi, 0.2 si, 0.0 st
KiB Mem :  1013672 total,  128304 free,  547176 used,  338192 buff/cache
KiB Swap:  2097148 total, 2089188 free,    7960 used.  242556 avail Mem

 PID  USER   PR  NI    VIRT   RES   SHR S %CPU %MEM    TIME+  COMMAND
 2731  root   20   0  193316 34848 15184 S  1.7  3.4  0:30.39  Xorg
25721  ulsah  20   0  619412 27216 17636 S  1.0  2.7  0:03.67  konsole
25296  ulsah  20   0  260724  6068  3268 S  0.7  0.6  0:17.78  prlcc
  747  root   20   0    4372   604   504 S  0.3  0.1  0:02.68  rngd
  846  root   20   0  141744   384   192 S  0.3  0.0  0:01.74  prltoolsd
 1647  root   20   0  177436  3656  2632 S  0.3  0.4  0:04.47  cupsd
10246  ulsah  20   0  130156  1936  1256 R  0.3  0.2  0:00.10  top
    1  root   20   0   59620  5472  3348 S  0.0  0.5  0:02.09  systemd
    2  root   20   0       0     0     0 S  0.0  0.0  0:00.02  kthreadd
    3  root   20   0       0     0     0 S  0.0  0.0  0:00.03  ksoftirqd/0
    9  root   20   0       0     0     0 S  0.0  0.0  0:00.00  rcuob/0
...
    7  root   rt   0       0     0     0 S  0.0  0.0  0:00.20  migration/0
    8  root   20   0       0     0     0 S  0.0  0.0  0:00.00  rcu_bh
    5  root    0 -20       0     0     0 S  0.0  0.0  0:00.00  kworker/0:+
```

By default, the display updates every 1–2 seconds, depending on the system. The most CPU-consumptive processes appear at the top. **top** also accepts input from the keyboard to send signals and **renice** processes (see the next section). You can then observe how your actions affect the overall condition of the machine.

Learn how to interpret the CPU, memory, and load details in Chapter 29.

The summary information in the first few lines of **top** output is one of the first places to look at to analyze the health of the system. It shows a condensed view of the system load, memory usage, number of processes, and a breakdown of how the CPU is being used.

On multicore systems, CPU usage is an average of all the cores in the system. Under Linux, press 1 (numeral one) while **top** is open to switch to a display of the individual cores. On FreeBSD, run **top -P** to achieve the same effect.

On FreeBSD systems, you can set the TOP environment variable to pass additional arguments to **top**. We recommend **-H** to show all threads for multithreaded processes rather than just a summary, plus **-P** to display all CPU cores. Add `export TOP="-HP"` to your shell initialization file to make these changes persistent between shell sessions.

Root can run **top** with the **-q** option to goose it up to the highest possible priority. This option can be useful when you are trying to track down a process that has already brought the system to its knees.

We also like **htop**, an open source, cross-platform, interactive process viewer that offers more features and has a nicer interface than that of **top**. It is not yet available as a package for our example systems, but you can download a binary or source version from the developer's web site at hisham.hm/htop.

4.5 NICE AND RENICE: INFLUENCE SCHEDULING PRIORITY

The “niceness” of a process is a numeric hint to the kernel about how the process should be treated in relation to other processes contending for the CPU. The strange name is derived from the fact that it determines how nice you are going to be to other users of the system. A high niceness means a low priority for your process: you are going to be nice. A low or negative value means high priority: you are not very nice.

It’s highly unusual to set priorities by hand these days. On the puny systems where UNIX originated, performance was significantly affected by which process was on the CPU. Today, with more than adequate CPU power on every desktop, the scheduler does a good job of managing most workloads. The addition of scheduling classes gives developers additional control when fast response is essential.

The range of allowable niceness values varies among systems. In Linux the range is -20 to +19, and in FreeBSD it’s -20 to +20.

Unless the user takes special action, a newly created process inherits the niceness of its parent process. The owner of the process can increase its niceness but cannot lower it, even to return the process to the default niceness. This restriction prevents processes running at low priority from bearing high-priority children. However, the superuser can set nice values arbitrarily.

I/O performance has not kept up with increasingly fast CPUs. Even with today’s high-performance SSDs, disk bandwidth remains the primary bottleneck on most systems. Unfortunately, a process’s niceness has no effect on the kernel’s management of its memory or I/O; high-nice processes can still monopolize a disproportionate share of these resources.

A process’s niceness can be set at the time of creation with the **nice** command and adjusted later with the **renice** command. **nice** takes a command line as an argument, and **renice** takes a PID or (sometimes) a username.

Some examples:

```
$ nice -n 5 ~/bin/longtask   // Lowers priority (raise nice) by 5
$ sudo renice -5 8829        // Sets niceness to -5
$ sudo renice 5 -u boggs     // Sets niceness of boggs's procs to 5
```

Unfortunately, there is little agreement among systems about how the desired priorities should be specified; in fact, even **nice** and **renice** from the same system usually don’t agree. To complicate things, a version of **nice** is built into the C shell and some other common shells (but not **bash**). If you don’t type the full path to **nice**, you’ll get the shell’s version rather than the operating system’s. To sidestep this ambiguity, we suggest using the fully qualified path to the system’s version, found at **/usr/bin/nice**.

Table 4.3 summarizes the variations. A *prio* is an absolute niceness, while an *incr* is relative to the niceness of the shell from which **nice** or **renice** is run. Only the shell **nice** understands plus signs (in fact, it requires them); leave them out in all other circumstances.

Table 4.3: How to express priorities for nice and renice

System	Range	OS nice	csh nice	renice
Linux	-20 to 19	**-n** *incr*	*+incr* or *-incr*	*prio* or **-n** *prio*
FreeBSD	-20 to 20	**-n** *incr*	*+incr* or *-incr*	*incr* or **-n** *incr*

4.6 THE /PROC FILESYSTEM



The Linux versions of **ps** and **top** read their process status information from the **/proc** directory, a pseudo-filesystem in which the kernel exposes a variety of interesting information about the system's state.

Despite the name **/proc** (and the name of the underlying filesystem type, "proc"), the information is not limited to process information—a variety of status information and statistics generated by the kernel are represented here. You can even modify some parameters by writing to the appropriate **/proc** file. See this page for some examples.

Although a lot of the information is easiest to access through front-end commands such as **vmstat** and **ps**, some of the more obscure nuggets must be read directly from **/proc**. It's worth poking around in this directory to familiarize yourself with everything that's there. **man proc** has a comprehensive explanation of its contents.

Because the kernel creates the contents of **/proc** files on the fly (as they are read), most appear to be empty, 0-byte files when listed with **ls -l**. You'll have to **cat** or **less** the contents to see what they actually contain. But be cautious—a few files contain or link to binary data that can confuse your terminal emulator if viewed directly.

Process-specific information is divided into subdirectories named by PID. For example, **/proc/1** is always the directory that contains information about **init**. Table 4.4 lists the most useful per-process files.

Table 4.4: Process information files in Linux /proc (numbered subdirectories)

File	Contents
cgroup	The control groups to which the process belongs
cmd	Command or program the process is executing
cmdline[a]	Complete command line of the process (null-separated)
cwd	Symbolic link to the process's current directory
environ	The process's environment variables (null-separated)
exe	Symbolic link to the file being executed
fd	Subdirectory containing links for each open file descriptor
fdinfo	Subdirectory containing further info for each open file descriptor
maps	Memory mapping information (shared segments, libraries, etc.)
ns	Subdirectory with links to each namespace used by the process.
root	Symbolic link to the process's root directory (set with **chroot**)
stat	General process status information (best decoded with **ps**)
statm	Memory usage information

a. Might be unavailable if the process is swapped out of memory

The individual components contained within the **cmdline** and **environ** files are separated by null characters rather than newlines. You can filter their contents through **tr "\000" "\n"** to make them more readable.

The **fd** subdirectory represents open files in the form of symbolic links. File descriptors that are connected to pipes or network sockets don't have an associated filename. The kernel supplies a generic description as the link target instead.

The **maps** file can be useful for determining what libraries a program is linked to or depends on.



FreeBSD includes a similar-but-different implementation of **/proc**. However, its use has been deprecated because of neglect in the code base and a history of security issues. It's still available for compatibility but is not mounted by default. To mount it, use the command

```
freebsd$ sudo mount -t procfs proc /proc
```

(To automatically mount the **/proc** filesystem at boot time, append the line `proc /proc procfs rw 0 0` to **/etc/fstab**.)

The filesystem layout is similar—but not identical—to the Linux version of procfs. The information for a process includes its status, a symbolic link to the file being executed, details about the process's virtual memory, and other low-level information. See also **man procfs**.

4.7 STRACE AND TRUSS: TRACE SIGNALS AND SYSTEM CALLS

It's often difficult to figure out what a process is actually doing. The first step is generally to make an educated guess based on indirect data collected from the filesystem, logs, and tools such as **ps**.

If those sources of information prove insufficient, you can snoop on the process at a lower level with the **strace** (Linux; usually an optional package) or **truss** (FreeBSD) command. These commands display every system call that a process makes and every signal it receives. You can attach **strace** or **truss** to a running process, snoop for a while, and then detach from the process without disturbing it. (Well, usually. **strace** can interrupt system calls. The monitored process must then be prepared to restart them. This is a standard rule of UNIX software hygiene, but it's not always observed.)

Although system calls occur at a relatively low level of abstraction, you can usually tell quite a bit about a process's activity from the call trace. For example, the following log was produced by **strace** run against an active copy of **top** (which was running as PID 5810):

```
redhat$ sudo strace -p 5810
gettimeofday({1116193814, 213881}, {300, 0})   = 0
open("/proc", O_RDONLY|O_NONBLOCK|O_LARGEFILE|O_DIRECTORY) = 7
fstat64(7, {st_mode=S_IFDIR|0555, st_size=0, ...})      = 0
fcntl64(7, F_SETFD, FD_CLOEXEC)          = 0
getdents64(7, /* 36 entries */, 1024)        = 1016
getdents64(7, /* 39 entries */, 1024)        = 1016
stat64("/proc/1", {st_mode=S_IFDIR|0555, st_size=0, ...}) = 0
open("/proc/1/stat", O_RDONLY)           = 8
read(8, "1 (init) S 0 0 0 0 -1 4194560 73"..., 1023)     = 191
close(8)                                 = 0
...
```

Not only does **strace** show you the name of every system call made by the process, but it also decodes the arguments and shows the result code that the kernel returns.

In the example above, **top** starts by checking the current time. It then opens and stats the **/proc** directory and reads the directory's contents, thereby obtaining a list of running processes. **top** goes on to stat the directory representing the **init** process and then opens **/proc/1/stat** to read **init**'s status information.

System call output can often reveal errors that are not reported by the process itself. For example, filesystem permission errors or socket conflicts are usually quite obvious in the output of **strace** or **truss**. Look for system calls that return error indications, and check for nonzero values.

strace is packed with goodies, most of which are documented in the man page. For example, the **-f** flag follows forked processes. This feature is useful for tracing daemons (such as **httpd**) that

spawn many children. The **-e trace=file** option displays only file-related operations. This feature is especially handy for discovering the location of evasive configuration files.

Here's a similar example from FreeBSD that uses **truss**. In this case, we trace how **cp** copies a file:

```
freebsd$ truss cp /etc/passwd /tmp/pw
...
lstat("/etc/passwd",{ mode=-rw-r--r-- ,inode=13576,size=2380,
    blksize=4096 }) = 0 (0x0)
umask(0x1ff)                            = 18 (0x12)
umask(0x12)                             = 511 (0x1ff)
fstatat(AT_FDCWD,"/etc/passwd",{ mode=-rw-r--r-- ,inode=13576,
    size=2380,blksize=4096 },0x0)       = 0 (0x0)
stat("/tmp/pw",0x7fffffffe440)      ERR#2 'No such file or directory'
openat(AT_FDCWD,"/etc/passwd",O_RDONLY,00)  = 3 (0x3)
openat(AT_FDCWD,"/tmp/pw",O_WRONLY|O_CREAT,0100644) = 4 (0x4)
mmap(0x0,2380,PROT_READ,MAP_SHARED,3,0x0)     = 34366304256
    (0x800643000)
write(4,"# $FreeBSD: releng/11.0/etc/mast"...,2380)  = 2380 (0x94c)
close(4)                                = 0 (0x0)
close(3)                                = 0 (0x0)
...
```

After allocating memory and opening library dependencies (not shown), **cp** uses the **lstat** system call to check the current status of the **/etc/passwd** file. It then runs **stat** on the path of the prospective copy, **/tmp/pw**. That file does not yet exist, so the **stat** fails and **truss** decodes the error for you as "No such file or directory."

cp then invokes the **openat** system call (with the O_RDONLY option) to read the contents of **/etc/passwd**, followed by an **openat** of **/tmp/pw** with O_WRONLY to create the new destination file. It then maps the contents of **/etc/passwd** into memory (with **mmap**) and writes out the data with **write**. Finally, **cp** cleans up after itself by closing both file handles.

System call tracing is a powerful debugging tool for administrators. Turn to these tools after more traditional routes such as examining log files and configuring a process for verbose output have been exhausted. Do not be intimidated by the dense output; it's usually sufficient to focus on the human-readable portions.

4.8 RUNAWAY PROCESSES

"Runaway" processes are those that soak up significantly more of the system's CPU, disk, or network resources than their usual role or behavior would lead you to expect. Sometimes, such programs have their own bugs that have led to degenerate behavior. In other cases, they fail to deal appropriately with upstream failures and get stuck in maladaptive loops. For example, a process might reattempt the same failing operation over and over again, flooring the CPU. In yet another category of cases, there is no bug per se, but the software is simply inefficient in its implementation and greedy with the system's resources.

All these situations merit investigation by a system administrator, not only because the runaway process is most likely malfunctioning but also because it typically interferes with the operation of other processes that are running on the system.

The line between pathological behavior and normal behavior under heavy workload is vague. Often, the first step in diagnosis is to figure out which of these phenomena you are actually observing. Generally, system processes should always behave reasonably, so obvious misbehavior on the part of one of these processes is automatically suspicious. User processes such as web servers and databases might simply be overloaded.

You can identify processes that are using excessive CPU time by looking at the output of **ps** or **top**. Also check the system load averages as reported by the **uptime** command. Traditionally, these values quantify the average number of processes that have been runnable over the previous 1-, 5-, and, 15-minute intervals. Under Linux, the load average also takes account of busyness caused by disk traffic and other forms of I/O.

For CPU bound systems, the load averages should be less than the total number of CPU cores available on your system. If they are not, the system is overloaded. Under Linux, check total CPU utilization with **top** or **ps** to determine whether high load averages relate to CPU load or to I/O. If CPU utilization is near 100%, that is probably the bottleneck.

Processes that use excessive memory relative to the system's physical RAM can cause serious performance problems. You can check the memory size of processes by running **top**. The VIRT column shows the total amount of virtual memory allocated by each process, and the RES column shows the portion of that memory currently mapped to specific memory pages (the "resident set").

Both of these numbers can include shared resources such as libraries and thus are potentially misleading. A more direct measure of process-specific memory consumption is found in the DATA column, which is not shown by default. To add this column to **top**'s display, type the **f** key once **top** is running and select DATA from the list by pressing the space bar. The DATA value indicates the amount of memory in each process's data and stack segments, so it's relatively specific to individual processes (modulo shared memory segments). Look for growth

over time as well as absolute size. On FreeBSD, SIZE is the equivalent column and is shown by default.

Make a concerted effort to understand what's going on before you terminate a seemingly runaway process. The best route to debugging the issue and preventing a recurrence is to have a live example you can investigate. Once you kill a misbehaving process, most of the available evidence disappears.

Keep the possibility of hacking in mind as well. Malicious software is typically not tested for correctness in a variety of environments, so it's more likely than average to enter some kind of degenerate state. If you suspect misfeasance, obtain a system call trace with **strace** or **truss** to get a sense of what the process is doing (e.g., cracking passwords) and where its data is stored.

Runaway processes that produce output can fill up an entire filesystem, causing numerous problems. When a filesystem fills up, lots of messages will be logged to the console and attempts to write to the filesystem will produce error messages.

The first thing to do in this situation is to determine which filesystem is full and which file is filling it up. The **df -h** command shows filesystem disk use in human-readable units. Look for a filesystem that's 100% or more full. (Most filesystem implementations reserve ~5% of the storage space for "breathing room," but processes running as root can encroach on this space, resulting in a reported usage that is greater than 100%.) Use the **du -h** command on the identified filesystem to determine which directory is using the most space. Rinse and repeat with **du** until the large files are discovered.

df and **du** report disk usage in subtly different manners. **df** reports the disk space used by a mounted filesystem according to disk block totals in the filesystem's metadata. **du** sums the sizes of all files in a given directory. If a file is unlinked (deleted) from the filesystem but is still referenced by some running process, **df** reports the space but **du** does not. This disparity persists until the open file descriptor is closed or the file is truncated. If you can't determine which process is using a file, try running the **fuser** and **lsof** commands (covered in detail here) to get more information.

4.9 Periodic processes

It’s often useful to have a script or command executed without any human intervention. Common use cases include scheduled backups, database maintenance activities, or the execution of nightly batch jobs. As is typical of UNIX and Linux, there’s more than one way to achieve this goal.

cron: schedule commands

The **cron** daemon is the traditional tool for running commands on a predetermined schedule. It starts when the system boots and runs as long as the system is up. There are multiple implementations of **cron**, but fortunately for administrators, the syntax and functionality of the various versions is nearly identical.

RHEL  For reasons that are unclear, **cron** has been renamed **crond** on Red Hat. But it is still the same **cron** we all know and love.

cron reads configuration files containing lists of command lines and times at which they are to be invoked. The command lines are executed by **sh**, so almost anything you can do by hand from the shell can also be done with **cron**. If you prefer, you can even configure **cron** to use a different shell.

A **cron** configuration file is called a "crontab," short for "cron table." Crontabs for individual users are stored under **/var/spool/cron** (Linux) or **/var/cron/tabs** (FreeBSD). There is at most one crontab file per user. Crontab files are plain text files named with the login names of the users to whom they belong. **cron** uses these filenames (and the file ownership) to figure out which UID to use when running the commands contained in each file. The **crontab** command transfers crontab files to and from this directory.

cron tries to minimize the time it spends reparsing configuration files and making time calculations. The **crontab** command helps maintain **cron**'s efficiency by notifying **cron** when crontab files change. Ergo, you shouldn't edit crontab files directly, because this approach might result in **cron** not noticing your changes. If you do get into a situation where **cron** doesn't seem to acknowledge a modified crontab, a HUP signal sent to the **cron** process forces it to reload on most systems.

See Chapter 10 for more information about syslog.

---

cron normally does its work silently, but most versions can keep a log file (usually **/var/log/cron**) that lists the commands that were executed and the times at which they ran. Glance at the **cron** log file if you're having problems with a **cron** job and can't figure out why.

The format of crontab files

All the crontab files on a system share a similar format. Comments are introduced with a pound sign (#) in the first column of a line. Each non-comment line contains six fields and represents one command:

```
minute hour dom month weekday command
```

The first five fields tell **cron** when to run the *command*. They're separated by whitespace, but within the *command* field, whitespace is passed along to the shell. The fields in the time specification are interpreted as shown in Table 4.5. An entry in a crontab is colloquially known as a "cron job."

Table 4.5: Crontab time specifications

Field	Description	Range
minute	Minute of the hour	0 to 59
hour	Hour of the day	0 to 23
dom	Day of the month	1 to 31
month	Month of the year	1 to 12
weekday	Day of the week	0 to 6 (0 = Sunday)

Each of the time-related fields can contain

- A star, which matches everything
- A single integer, which matches exactly
- Two integers separated by a dash, matching a range of values
- A range followed by a slash and a step value, e.g., `1-10/2`
- A comma-separated list of integers or ranges, matching any value

For example, the time specification

```
45 10 * * 1-5
```

means "10:45 a.m., Monday through Friday." A hint: never use stars in every field unless you want the command to be run every minute, which is useful only in testing scenarios. One minute is the finest granularity available to **cron** jobs.

Time ranges in crontabs can include a step value. For example, the series `0,3,6,9,12,15,18` can be written more concisely as `0-18/3`. You can also use three-letter text mnemonics for the names of months and days, but not in combination with ranges. As far as we know, this feature works only with English names.

There is a potential ambiguity to watch out for with the *weekday* and *dom* fields. Every day is both a day of the week and a day of the month. If both *weekday* and *dom* are specified, a day need satisfy only one of the two conditions to be selected.

For example,

```
0,30 * 13 * 5
```

means “every half-hour on Friday, and every half-hour on the 13th of the month,” not “every half-hour on Friday the 13th.”

The *command* is the **sh** command line to be executed. It can be any valid shell command and should not be quoted. The *command* is considered to continue to the end of the line and can contain blanks or tabs.

Percent signs (%) indicate newlines within the *command* field. Only the text up to the first percent sign is included in the actual command. The remaining lines are given to the command as standard input. Use a backslash (\) as an escape character in commands that have a meaningful percent sign, for example, `date +\%s`.

Although **sh** is involved in executing the *command*, the shell does not act as a login shell and does not read the contents of **~/.profile** or **~/.bash_profile**. As a result, the command’s environment variables might be set up somewhat differently from what you expect. If a command seems to work fine when executed from the shell but fails when introduced into a crontab file, the environment is the likely culprit. If need be, you can always wrap your command with a script that sets up the appropriate environment variables.

We also suggest using the fully qualified path to the command, ensuring that the job will work properly even if the PATH is not set as expected. For example, the following command logs the date and uptime to a file in the user’s home directory every minute:

```
* * * * * echo $(/bin/date) - $(/usr/bin/uptime) >> ~/uptime.log
```

Alternatively, you can set environment variables explicitly at the top of the crontab:

```
PATH=/bin:/usr/bin
* * * * * echo $(date) - $(uptime) >> ~/uptime.log
```

Here are a few more examples of valid crontab entries:

```
*/10 * * * 1,3,5 echo ruok | /usr/bin/nc localhost 2181 |
    mail -s "TCP port 2181 status" ben@admin.com
```

This line emails the results of a connectivity check on port 2181 every 10 minutes on Mondays, Wednesdays, and Fridays. Since **cron** executes *command* by way of **sh**, special shell characters like pipes and redirects function as expected.

```
0 4 * * Sun (/usr/bin/mysqlcheck -u maintenance --optimize --all-databases)
```

This entry runs the **mysqlcheck** maintenance program on Sundays at 4:00 a.m. Since the output is not saved to a file or otherwise discarded, it will be emailed to the owner of the crontab.

```
20  1  *  *  *   find /tmp -mtime +7 -type f -exec rm -f { } ';'
```

This command runs at 1:20 each morning. It removes all files in the **/tmp** directory that have not been modified in 7 days. The ';' at the end of the line marks the end of the subcommand arguments to **find**.

cron does not try to compensate for commands that are missed while the system is down. However, it is smart about time adjustments such as shifts into and out of daylight saving time.

If your cron job is a script, be sure to make it executable (with **chmod +x**) or **cron** won't be able to execute it. Alternatively, set the cron command to invoke a shell on your script directly (e.g., **bash -c ~/bin/myscript.sh).**

Crontab management

crontab *filename* installs *filename* as your crontab, replacing any previous version. **crontab -e** checks out a copy of your crontab, invokes your editor on it (as specified by the EDITOR environment variable), and then resubmits it to the crontab directory. **crontab -l** lists the contents of your crontab to standard output, and **crontab -r** removes it, leaving you with no crontab file at all.

Root can supply a *username* argument to edit or view other users' crontabs. For example, **crontab -r jsmith** erases the crontab belonging to the user jsmith, and **crontab -e jsmith** edits it. Linux allows both a *username* and a *filename* argument in the same command, so the username must be prefixed with **-u** to disambiguate (e.g., **crontab -u jsmith crontab.new**).

Without command-line arguments, most versions of **crontab** try to read a crontab from standard input. If you enter this mode by accident, don't try to exit with <Control-D>; doing so erases your entire crontab. Use <Control-C> instead. FreeBSD requires you to supply a dash as the *filename* argument to make **crontab** pay attention to its standard input. Smart.

Many sites have experienced subtle but recurrent network glitches that occur because administrators have configured **cron** to run the same command on hundreds of machines at exactly the same time, causing delays or excessive load. Clock synchronization with NTP exacerbates the problem. This issue is easy to fix with a random delay script.

cron logs its activities through syslog using the facility "cron," with most messages submitted at level "info." Default syslog configurations generally send **cron** log data to its own file.

Other crontabs

In addition to looking for user-specific crontabs, **cron** also obeys system crontab entries found in **/etc/crontab** and in the **/etc/cron.d** directory. These files have a slightly different format from the per-user crontab files: they allow commands to be run as an arbitrary user. An extra *username* field comes before the command name. The *username* field is not present in garden-variety crontab files because the crontab's filename supplies this same information.

In general, **/etc/crontab** is a file for system administrators to maintain by hand, whereas **/etc/cron.d** is a sort of depot into which software packages can install any crontab entries they might need. Files in **/etc/cron.d** are by convention named after the packages that install them, but **cron** doesn't care about or enforce this convention.



Linux distributions also pre-install crontab entries that run the scripts in a set of well-known directories, thereby providing another way for software packages to install periodic jobs without any editing of a crontab file. For example, scripts in **/etc/cron.hourly**, **/etc/cron.daily**, and **/etc/cron.weekly** are run hourly, daily, and weekly, respectively.

cron *access control*

Two config files specify which users may submit crontab files. For Linux, the files are **/etc/cron.** {**allow**,**deny**}, and on FreeBSD they are **/var/cron/**{**allow**,**deny**}. Many security standards require that crontabs be available only to service accounts or to users with a legitimate business need. The **allow** and **deny** files facilitate compliance with these requirements.

If the **cron.allow** file exists, then it contains a list of all users that may submit crontabs, one per line. No unlisted person can invoke the **crontab** command. If the **cron.allow** file doesn't exist, then the **cron.deny** file is checked. It, too, is just a list of users, but the meaning is reversed: everyone except the listed users is allowed access.

If neither the **cron.allow** file nor the **cron.deny** file exists, systems default (apparently at random, there being no dominant convention) either to allowing all users to submit crontabs or to limiting crontab access to root. In practice, a starter configuration is typically included in the default OS installation, so the question of how **crontab** might behave without configuration files is moot. Most default configurations allow all users to access **cron** by default.

It's important to note that on most systems, access control is implemented by **crontab**, not by **cron**. If a user is able to sneak a crontab file into the appropriate directory by other means, **cron** will blindly execute the commands it contains. Therefore it is vital to maintain root ownership of **/var/spool/cron** and **/var/cron/tabs**. OS distributions always set the permissions correctly by default.

systemd timers

See Chapter 2, for an introduction to systemd and units.

In accordance with its mission to duplicate the functions of all other Linux subsystems, **systemd** includes the concept of timers, which activate a given **systemd** service on a predefined schedule. Timers are more powerful than crontab entries, but they are also more complicated to set up and manage. Some Linux distributions (e.g., CoreOS) have abandoned **cron** entirely in favor of **systemd** timers, but our example systems all continue to include **cron** and to run it by default.

We have no useful advice regarding the choice between **systemd** timers and crontab entries. Use whichever you prefer for any given task. Unfortunately, you do not really have the option to standardize on one system or the other, because software packages add their jobs to a random system of their own choice. You'll always have to check both systems when you are trying to figure out how a particular job gets run.

*Structure of **systemd** timers*

A **systemd** timer comprises two files:

- A timer unit that describes the schedule and the unit to activate
- A service unit that specifies the details of what to run

In contrast to crontab entries, **systemd** timers can be described both in absolute calendar terms ("Wednesdays at 10:00 a.m.") and in terms that are relative to other events ("30 seconds after system boot"). The options combine to allow powerful expressions that don't suffer the same constraints as **cron** jobs. Table 4.6 describes the time expression options.

Table 4.6: systemd timer types

Type	Time basis
`OnActiveSec`	Relative to the time at which the timer itself is activated
`OnBootSec`	Relative to system boot time
`OnStartupSec`	Relative to the time at which **systemd** was started
`OnUnitActiveSec`	Relative to the time the specified unit was last active
`OnUnitInactiveSec`	Relative to the time the specified unit was last inactive
`OnCalendar`	A specific day and time

As their names suggest, values for these timer options are given in seconds. For example, `OnActiveSec=30` is 30 seconds after the timer activates. The value can actually be any valid **systemd** time expression, as discussed in more detail starting here.

***systemd** timer example*

Red Hat and CentOS include a preconfigured **systemd** timer that cleans up the system's temporary files once a day. Below, we take a more detailed look at an example. First, we enumerate all the defined timers with the **systemctl** command. (We rotated the output table below to make it readable. Normally, each timer produces one long line of output.)

```
redhat$ systemctl list-timers
NEXT         Sun 2017-06-18 10:24:33 UTC
LEFT         18h left
LAST         Sat 2017-06-17 00:45:29 UTC
PASSED       15h ago
UNIT         systemd-tmpfiles-clean.timer
ACTIVATES    systemd-tmpfiles-clean.service
```

The output lists both the name of the timer unit and the name of the service unit it activates. Since this is a default system timer, the unit file lives in the standard systemd unit directory, **/usr/lib/systemd/system**. Here's the timer unit file:

```
redhat$ cat /usr/lib/systemd/system/systemd-tmpfiles-clean.timer
[Unit]
Description=Daily Cleanup of Temporary Directories
[Timer]
OnBootSec=15min
OnUnitActiveSec=1d
```

The timer first activates 15 minutes after boot and then fires once a day thereafter. Note that some kind of trigger for the initial activation (here, OnBootSec) is always necessary. There is no single specification that achieves an "every X minutes" effect on its own.

Astute observers will notice that the timer does not actually specify which unit to run. By default, **systemd** looks for a service unit that has the same name as the timer. You can specify a target unit explicitly with the Unit option.

In this case, the associated service unit holds no surprises:

```
redhat$ cat /usr/lib/systemd/system/systemd-tmpfiles-clean.service
[Unit]
Description=Cleanup of Temporary Directories
DefaultDependencies=no
Conflicts=shutdown.target
After=systemd-readahead-collect.service systemd-readahead-replay.
    service local-fs.target time-sync.target
Before=shutdown.target

[Service]
Type=simple
ExecStart=/usr/bin/systemd-tmpfiles --clean
IOSchedulingClass=idle
```

You can run the target service directly (that is, independently of the timer) with **systemctl start systemd-tmpfiles-clean**, just like any other service. This fact greatly facilitates the debugging of scheduled tasks, which can be a source of much administrative anguish when you are using **cron**.

To create your own timer, drop **.timer** and **.service** files in **/etc/systemd/system**. If you want the timer to run at boot, add

```
[Install]
WantedBy=multi-user.target
```

to the end of the timer's unit file. Don't forget to enable the timer at boot time with **systemctl enable**. (You can also start the timer immediately with **systemctl start**.)

A timer's AccuracySec option delays its activation by a random amount of time within the specified time window. This feature is handy when a timer runs on a large group of networked machines and you want to avoid having all the timers fire at exactly the same moment. (Recall that with **cron**, you need to use a random delay script to achieve this feat.)

AccuracySec defaults to 60 seconds. If you want your timer to execute at exactly the scheduled time, use AccuracySec=1ns. (A nanosecond is probably close enough. Note that you won't actually obtain nanosecond accuracy.)

systemd time expressions

Timers allow for flexible specification of dates, times, and intervals. The **systemd.time** man page is the authoritative reference for the specification grammar.

You can use interval-valued expressions instead of seconds for relative timings such as those used as the values of OnActiveSec and OnBootSec. For example, the following forms are all valid:

```
OnBootSec=2h 1m
OnStartupSec=1week 2days 3hours
OnActiveSec=1hr20m30sec10msec
```

Spaces are optional in time expressions. The minimum granularity is nanoseconds, but if your timer fires too frequently (more than once every two seconds) **systemd** temporarily disables it.

In addition to triggering at periodic intervals, timers can be scheduled to activate at specific times by including the OnCalendar option. This feature offers the closest match to the syntax of a traditional **cron** job, but its syntax is more expressive and flexible. Table 4.7 shows some examples of time specifications that could be used as the value of OnCalendar.

Table 4.7: systemd time and date encoding examples

Time specification	Meaning
2017-07-04	July 4th, 2017 at 00:00:00 (midnight)

Fri-Mon *-7-4	July 4th each year, but only if it falls on Fri–Mon
Mon-Wed *-*-* 12:00:00	Mondays, Tuesdays, and Wednesdays at noon
Mon 17:00:00	Mondays at 5:00 p.m.
weekly	Mondays at 00:00:00 (midnight)
monthly	The 1st day of the month at 00:00:00 (midnight)
*:0/10	Every 10 minutes, starting at the 0th minute
--* 11/12:10:0	At 11:10 and 23:10 every day

In time expressions, stars are placeholders that match any plausible value. As in crontab files, slashes introduce an increment value. The exact syntax is a bit different from that used in crontabs, however: crontabs want the incremented object to be a range (e.g., 9-17/2, "every two hours between 9:00 a.m. and 5:00 p.m."), but **systemd** time expressions take only a start value (e.g., 9/2, "every two hours starting at 9:00 a.m.").

Transient timers

You can use the **systemd-run** command to schedule the execution of a command according to any of the normal **systemd** timer types, but without creating task-specific timer and service unit files. For example, to pull a Git repository every ten minutes:

```
$ systemd-run --on-calendar '*:0/10' /bin/sh -c "cd /app && git pull"
Running timer as unit run-8823.timer.
Will run service as unit run-8823.service.
```

systemd returns a transient unit identifier that you can list with **systemctl**. (Once again, we futzed with the output format below...)

```
$ systemctl list-timers run-8823.timer
NEXT        Sat 2017-06-17 20:40:07 UTC
LEFT        9min left
LAST        Sat 2017-06-17 20:30:07 UTC
PASSED      18s ago
```

```
$ systemctl list-units run-8823.timer
UNIT        run-8823.timer
LOAD        loaded
ACTIVE      active
SUB         waiting
DESCRIPTION /bin/sh -c "cd /app && git pull"
```

To cancel and remove a transient timer, just stop it by running **systemctl stop**:

```
$ sudo systemctl stop run-8823.timer
```

systemd-run functions by creating timer and unit files for you in subdirectories of **/run/systemd/system.** However, transient timers do not persist after a reboot. To make them permanent, you can fish them out of **/run**, tweak them as necessary, and install them in

/etc/systemd/system. Be sure to stop the transient timer before starting or enabling the permanent version.

Common uses for scheduled tasks

In this section, we look at a couple of common chores that are often automated through **cron** or **systemd**.

Sending mail

The following crontab entry implements a simple email reminder. You can use an entry like this to automatically email the output of a daily report or the results of a command execution. (Lines have been folded to fit the page. In reality, this is one long line.)

```
30  4  25  *  *  /usr/bin/mail -s "Time to do the TPS reports"
    ben@admin.com%TPS reports are due at the end of the month! Get
    busy!%%Sincerely,%cron%
```

Note the use of the `%` character both to separate the command from the input text and to mark line endings within the input. This entry sends email at 4:30 a.m. on the 25th day of each month.

Cleaning up a filesystem

When a program crashes, the kernel may write out a file (usually named **core**.*pid*, **core**, or *program*.**core**) that contains an image of the program's address space. Core files are useful for developers, but for administrators they are usually a waste of space. Users often don't know about core files, so they tend not to disable their creation or delete them on their own. You can use a **cron** job to clean up these core files or other vestiges left behind by misbehaving and crashed processes.

Rotating a log file

Systems vary in the quality of their default log file management, and you will probably need to adjust the defaults to conform to your local policies. To "rotate" a log file means to divide it into segments by size or by date, keeping several older versions of the log available at all times. Since log rotation is a recurrent and regularly occurring event, it's an ideal task to be scheduled. See *Management and rotation of log files*, for more details.

Running batch jobs

Some long-running calculations are best run as batch jobs. For example, messages can accumulate in a queue or database. You can use a **cron** job to process all the queued messages at once as an ETL (extract, transform, and load) to another location, such as a data warehouse.

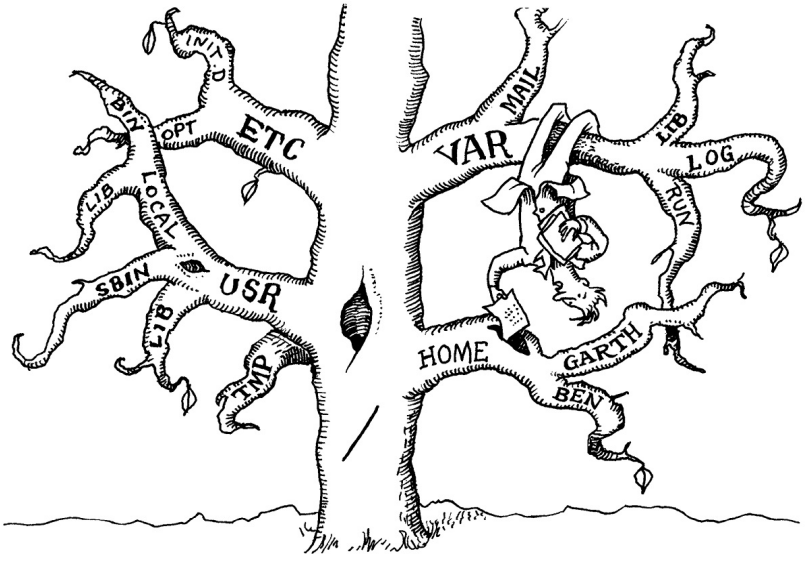
Some databases benefit from routine maintenance. For example, the open source distributed database Cassandra has a repair function that keeps the nodes in a cluster in sync. These maintenance tasks are good candidates for execution through **cron** or **systemd**.

Backing up and mirroring

You can use a scheduled task to automatically back up a directory to a remote system. We suggest running a full backup once a week, with incremental differences each night. Run backups late at night when the load on the system is likely to be low.

Mirrors are byte-for-byte copies of filesystems or directories that are hosted on another system. They can be used as a form of backup or as a way to make files available at more than one location. Web sites and software repositories are often mirrored to offer better redundancy and to offer faster access for users that are physically distant from the primary site. Use periodic execution of the **rsync** command to maintain mirrors and keep them up to date.

5 The Filesystem



Quick: which of the following would you expect to find in a “filesystem”?

- Processes
- Audio devices
- Kernel data structures and tuning parameters
- Interprocess communication channels

If the system is UNIX or Linux, the answer is “all the above, and more!” And yes, you might find some files in there, too. (It’s perhaps more accurate to say that these entities are *represented* within the filesystem. In most cases, the filesystem is used as a rendezvous point to connect clients with the drivers they are seeking.)

The basic purpose of a filesystem is to represent and organize the system’s storage resources. However, programmers have been eager to avoid reinventing the wheel when it comes to managing other types of objects. It has often proved convenient to map these objects into the filesystem namespace. This unification has some advantages (consistent programming interface, easy access from the shell) and some disadvantages (filesystem implementations suggestive of Frankenstein’s monster), but like it or not, this is the UNIX (and hence, the Linux) way.

The filesystem can be thought of as comprising four main components:

- A namespace – a way to name things and organize them in a hierarchy
- An API – a set of system calls for navigating and manipulating objects

- Security models – schemes for protecting, hiding, and sharing things
- An implementation – software to tie the logical model to the hardware

Modern kernels define an abstract interface that accommodates many different back-end filesystems. Some portions of the file tree are handled by traditional disk-based implementations. Others are fielded by separate drivers within the kernel. For example, network filesystems are handled by a driver that forwards the requested operations to a server on another computer.

Unfortunately, the architectural boundaries are not clearly drawn, and quite a few special cases exist. For example, "device files" define a way for programs to communicate with drivers inside the kernel. Device files are not really data files, but they're handled through the filesystem and their characteristics are stored on disk.

Another complicating factor is that the kernel supports more than one type of disk-based filesystem. The predominant standards are the ext4, XFS, and UFS filesystems, along with Oracle's ZFS and Btrfs. However, many others are available, including, Veritas's VxFS and JFS from IBM.

"Foreign" filesystems are also widely supported, including the FAT and NTFS filesystems used by Microsoft Windows and the ISO 9660 filesystem used on older CD-ROMs.

The filesystem is a rich topic that we approach from several different angles. This chapter tells where to find things on your system and describes the characteristics of files, the meanings of permission bits, and the use of some basic commands that view and set attributes. Chapter 20, *Storage*, is where you'll find the more technical filesystem topics such as disk partitioning.

Chapter 21, *The Network File System*, describes NFS, a file sharing system that is commonly used for remote file access between UNIX and Linux systems. Chapter 22, *SMB*, describes an analogous system from the Windows world.

See the sections starting here for more information about specific filesystems.

With so many different filesystem implementations available, it may seem strange that this chapter reads as if there were only a single filesystem. We can be vague about the underlying code because most modern filesystems either try to implement the traditional filesystem functionality in a faster and more reliable manner, or they add extra features as a layer on top of the standard filesystem semantics. Some filesystems do both. For better or worse, too much existing software depends on the model described in this chapter for that model to be discarded.

5.1 PATHNAMES

The filesystem is presented as a single unified hierarchy that starts at the directory / and continues downward through an arbitrary number of subdirectories. / is also called the root directory. This single-hierarchy system differs from the one used by Windows, which retains the concept of partition-specific namespaces.

Graphical user interfaces often refer to directories as “folders,” even on Linux systems. Folders and directories are exactly the same thing; “folder” is just linguistic leakage from the worlds of Windows and macOS. Nevertheless, it’s worth noting that the word “folder” tends to raise the hackles of some techies. Don’t use it in technical contexts unless you’re prepared to receive funny looks.

The list of directories that must be traversed to locate a particular file plus that file’s filename form a pathname. Pathnames can be either absolute (e.g., **/tmp/foo**) or relative (e.g., **book4/filesystem**). Relative pathnames are interpreted starting at the current directory. You might be accustomed to thinking of the current directory as a feature of the shell, but every process has one.

The terms *filename*, *pathname*, and *path* are more or less interchangeable—or at least, we use them interchangeably in this book. *Filename* and *path* can be used for both absolute and relative paths; *pathname* usually suggests an absolute path.

The filesystem can be arbitrarily deep. However, each component of a pathname (that is, each directory) must have a name no more than 255 characters long. There’s also a limit on the total path length you can pass into the kernel as a system call argument (4,095 bytes on Linux, 1,024 bytes on BSD). To access a file with a pathname longer than this, you must **cd** to an intermediate directory and use a relative pathname.

5.2 Filesystem mounting and unmounting

The filesystem is composed of smaller chunks—also called filesystems—each of which consists of one directory and its subdirectories and files. It's normally apparent from context which type of "filesystem" is being discussed, but for clarity in the following discussion, we use the term "file tree" to refer to the overall layout and reserve the word "filesystem" for the branches attached to the tree.

Some filesystems live on disk partitions or on logical volumes backed by physical disks, but as mentioned earlier, filesystems can be anything that obeys the proper API: a network file server, a kernel component, a memory-based disk emulator, etc. Most kernels have a nifty "loop" filesystem that lets you mount individual files as if they were distinct devices. It's useful for mounting DVD-ROM images stored on disk or for developing filesystem images without having to worry about repartitioning. Linux systems can even treat existing portions of the file tree as filesystems. This trick lets you duplicate, move, or hide portions of the file tree.

In most situations, filesystems are attached to the tree with the **mount** command. **mount** maps a directory within the existing file tree, called the mount point, to the root of the newly attached filesystem. The previous contents of the mount point become temporarily inaccessible as long as another filesystem is mounted there. Mount points are usually empty directories, however.

For example,

```
$ sudo mount /dev/sda4 /users
```

installs the filesystem stored on the disk partition represented by **/dev/sda4** under the path **/users**. You could then use **ls /users** to see that filesystem's contents.

On some systems, **mount** is a just a wrapper that calls filesystem-specific commands such as **mount.ntfs** or **mount_smbfs**. You're free to call these helper commands directly if you need to; they sometimes offer additional options that the **mount** wrapper does not understand. On the other hand, the generic **mount** command suffices for day-to-day use.

You can run the **mount** command without any arguments to see all the filesystems that are currently mounted. On Linux systems, there might be 30 or more, most of which represent various interfaces to the kernel.

The **/etc/fstab** file lists filesystems that are normally mounted on the system. The information in this file allows filesystems to be automatically checked (with **fsck**) and mounted (with **mount**) at boot time, with options you specify. The **fstab** file also serves as documentation for the layout of the filesystems on disk and enables short commands such as **mount /usr**. See this page for a discussion of **fstab**.

You detach filesystems with the **umount** command. **umount** complains if you try to unmount a filesystem that's in use. The filesystem to be detached must not have open files or processes whose current directories are located there, and if the filesystem contains executable programs, none of them can be running.

Linux has a "lazy" unmount option (**umount -l**) that removes a filesystem from the naming hierarchy but does not truly unmount it until all existing file references have been closed. It's debatable whether this is a useful option. To begin with, there's no guarantee that existing references will ever close on their own. In addition, the "semi-unmounted" state can present inconsistent filesystem semantics to the programs that are using it; they can read and write through existing file handles but cannot open new files or perform other filesystem operations.



umount -f force-unmounts a busy filesystem and is supported on all our example systems. However, it's almost always a bad idea to use it on non-NFS mounts, and it may not work on certain types of filesystems (e.g., those that keep journals, such as XFS or ext4).

Instead of reaching for **umount -f** when a filesystem you're trying to unmount turns out to be busy, run the **fuser** command to find out which processes hold references to that filesystem. **fuser -c** *mountpoint* prints the PID of every process that's using a file or directory on that filesystem, plus a series of letter codes that show the nature of the activity. For example,

```
freebsd$ fuser -c /usr/home
/usr/home: 15897c  87787c  67124x  11201x  11199x  11198x  972x
```

The exact letter codes vary from system to system. In this example from a FreeBSD system, `c` indicates that a process has its current working directory on the filesystem and `x` indicates a program being executed. However, the details are usually unimportant—the PIDs are what you want.

To investigate the offending processes, just run **ps** with the list of PIDs returned by **fuser**. For example,

```
nutrient:~$ ps up "87787 11201"
USER    PID  %CPU  %MEM  STARTED     TIME  COMMAND
fnd   11201   0.0   0.2  14Jul16  2:32.49  ruby: slave_audiochannelbackend
fnd   87787   0.0   0.0  Thu07PM  0:00.93  -bash (bash)
```

Here, the quotation marks force the shell to pass the list of PIDs to **ps** as a single argument.



On Linux systems, you can avoid the need to launder PIDs through **ps** by running **fuser** with the **-v** flag. This option produces a more readable display that includes the command name.

```
$ fuser -cv /usr
                     USER   PID  ACCESS    COMMAND
                /usr  root   444   ....m    atd
                      root   499   ....m    sshd
                      root   520   ....m    lpd
...
```

The letter codes in the ACCESS column are the same ones used in **fuser**'s nonverbose output.

A more elaborate alternative to **fuser** is the **lsof** utility. **lsof** is a more complex and sophisticated program than **fuser**, and its output is correspondingly verbose. **lsof** comes installed by default on all our example Linux systems and is available as a package on FreeBSD.



Under Linux, scripts in search of specific information about processes' use of filesystems can also read the files in **/proc** directly. However, **lsof -F**, which formats **lsof**'s output for easy parsing, is an easier and more portable solution. Use additional command-line flags to request just the information you need.

5.3 Organization of the file tree

UNIX systems have never been well organized. Various incompatible naming conventions are used simultaneously, and different types of files are scattered randomly around the namespace. In many cases, files are divided by function and not by how likely they are to change, making it difficult to upgrade the operating system. The **/etc** directory, for example, contains some files that are never customized and some that are entirely local. How do you know which files to preserve during an upgrade? Well, you just have to know…or trust the installation software to make the right decisions.

As a logically minded sysadmin, you may be tempted to improve the default organization. Unfortunately, the file tree has many hidden dependencies, so such efforts usually end up creating problems. Just let everything stay where the OS installation and the system packages put it. When offered a choice of locations, always accept the default unless you have a specific and compelling reason to do otherwise.

See Chapter 11 for more information about configuring the kernel.

The root filesystem includes at least the root directory and a minimal set of files and subdirectories. The file that contains the OS kernel usually lives under **/boot**, but its exact name and location can vary. Under BSD and some other UNIX systems, the kernel is not really a single file so much as a set of components.

Also part of the root filesystem are **/etc** for critical system and configuration files, **/sbin** and **/bin** for important utilities, and sometimes **/tmp** for temporary files. The **/dev** directory was traditionally part of the root filesystem, but these days it's a virtual filesystem that's mounted separately. (See this page for more information.)

Some systems keep shared library files and a few other oddments, such as the C preprocessor, in the **/lib** or **/lib64** directory. Others have moved these items into **/usr/lib**, sometimes leaving **/lib** as a symbolic link.

The directories **/usr** and **/var** are also of great importance. **/usr** is where most standard-but-not-system-critical programs are kept, along with various other booty such as on-line manuals and most libraries. FreeBSD stores quite a bit of local configuration under **/usr/local**. **/var** houses spool directories, log files, accounting information, and various other items that grow or change rapidly and that vary on each host. Both **/usr** and **/var** must be available to enable the system to come up all the way to multiuser mode.

See this page for some reasons why partitioning might be desirable and some rules of thumb to guide it.

In the past, it was standard practice to partition the system disk and to put some parts of the file tree on their own partitions, most commonly **/usr**, **/var**, and **/tmp**. That's not uncommon even now, but the secular trend is toward having one big root filesystem. Large hard disks and increasingly sophisticated filesystem implementations have reduced the value of partitioning.

In cases where partitioning is used, it's most frequently an attempt to prevent one part of the file tree from consuming all available space and bringing the entire system to a halt. Accordingly, **/var** (which contains log files that are apt to grow in times of trouble), **/tmp**, and user home directories are some of the most common candidates for having their own partitions. Dedicated filesystems can also store bulky items such as source code libraries and databases.

Table 5.1 lists some of the more important standard directories. (Alternate rows have been shaded to improve readability.)

Table 5.1: Standard directories and their contents

Pathname	Contents
/bin	Core operating system commands
/boot	Boot loader, kernel, and files needed by the kernel
/compat	On FreeBSD, files and libraries for Linux binary compatibility
/dev	Device entries for disks, printers, pseudo-terminals, etc.
/etc	Critical startup and configuration files
/home	Default home directories for users
/lib	Libraries, shared libraries, and commands used by **/bin** and **/sbin**
/media	Mount points for filesystems on removable media
/mnt	Temporary mount points, mounts for removable media
/opt	Optional software packages (rarely used, for compatibility)
/proc	Information about all running processes
/root	Home directory of the superuser (sometimes just /)
/run	Rendezvous points for running programs (PIDs, sockets, etc.)
/sbin	Core operating system commands[a]
/srv	Files held for distribution through web or other servers
/sys	A plethora of different kernel interfaces (Linux)
/tmp	Temporary files that may disappear between reboots
/usr	Hierarchy of secondary files and commands
/usr/bin	Most commands and executable files
/usr/include	Header files for compiling C programs
/usr/lib	Libraries; also, support files for standard programs
/usr/local	Local software or configuration data; mirrors **/usr**
/usr/sbin	Less essential commands for administration and repair
/usr/share	Items that might be common to multiple systems
/usr/share/man	On-line manual pages
/usr/src	Source code for nonlocal software (not widely used)
/usr/tmp	More temporary space (preserved between reboots)
/var	System-specific data and a few configuration files
/var/adm	Varies: logs, setup records, strange administrative bits
/var/log	System log files
/var/run	Same function as **/run**; now often a symlink
/var/spool	Spooling (that is, storage) directories for printers, mail, etc.
/var/tmp	More temporary space (preserved between reboots)

a. The distinguishing characteristic of **/sbin** was originally that its contents were statically linked and so had fewer dependencies on other parts of the system. These days, all binaries are dynamically linked and there is no real difference between **/bin** and **/sbin**.

On most systems, a **hier** man page outlines some general guidelines for the layout of the filesystem. Don't expect the actual system to conform to the master plan in every respect, however.



For Linux systems, the Filesystem Hierarchy Standard (see wiki.linuxfoundation.org/en/FHS) attempts to codify, rationalize, and explain the standard

directories. It’s an excellent resource to consult when you confront an unusual situation and need to figure out where to put something. Despite its status as a “standard,” it’s more a reflection of real-world practice than a prescriptive document. It also hasn’t undergone much updating recently, so it doesn’t describe the exact filesystem layout found on current distributions.

5.4 File types

Most filesystem implementations define seven types of files. Even when developers add something new and wonderful to the file tree (such as the process information under **/proc**), it must still be made to look like one of these seven types:

- Regular files
- Directories
- Character device files
- Block device files
- Local domain sockets
- Named pipes (FIFOs)
- Symbolic links

You can determine the type of an existing file with the **file** command. Not only does **file** know about the standard types of files, but it also knows a thing or two about common formats used within regular files.

```
$ file /usr/include
/usr/include: directory
$ file /bin/sh
/bin/sh: ELF 64-bit LSB executable, x86-64, version 1 (FreeBSD),
    dynamically linked, interpreter /libexec/ld-elf.so.1, for FreeBSD
    11.0 (1100122), FreeBSD-style, stripped
```

All that hoo-hah about **/bin/sh** means “it’s an executable command.”

Another option for investigating files is **ls -ld**. The **-l** flag shows detailed information, and the **-d** flag forces **ls** to show the information for a directory rather than showing the directory’s contents.

The first character of the **ls** output encodes the type. For example, the circled `d` in the following output demonstrates that **/usr/include** is a directory:

```
$ ls -ld /usr/include
drwxr-xr-x   27 root     root         4096  Jul 15 20:57  /usr/include
```

Table 5.2 shows the codes **ls** uses to represent the various types of files.

Table 5.2: File-type encoding used by ls

File type	Symbol	Created by	Removed by
Regular file	–	editors, **cp**, etc.	**rm**
Directory	d	**mkdir**	**rmdir**, **rm -r**
Character device file	c	**mknod**	**rm**
Block device file	b	**mknod**	**rm**
Local domain socket	s	**socket** system call	**rm**
Named pipe	p	**mknod**	**rm**
Symbolic link	l	**ln -s**	**rm**

As Table 5.2 shows, **rm** is the universal tool for deleting files. But how would you delete a file named, say, **-f**? It's a legitimate filename under most filesystems, but **rm -f** doesn't work because **rm** interprets the **-f** as a flag. The answer is either to refer to the file by a pathname that doesn't start with a dash (such as **./-f**) or to use **rm**'s -- argument to tell it that everything that follows is a filename and not an option (i.e., **rm -- -f**).

Filenames that contain control or Unicode characters present a similar problem since reproducing these names from the keyboard can be difficult or impossible. In this situation, you can use shell globbing (pattern matching) to identify the files to delete. When you use pattern matching, it's a good idea to get in the habit of using **rm**'s **-i** option to make **rm** confirm the deletion of each file. This feature protects you against deleting any "good" files that your pattern inadvertently matches. To delete the file named **foo**<Control-D>**bar** in the following example, you could use

```
$ ls
foo?bar     foose    kde-root

$ rm -i foo*
rm: remove 'foo\004bar'? y
rm: remove 'foose'? n
```

Note that **ls** shows the control character as a question mark, which can be a bit deceptive. If you don't remember that **?** is a shell pattern-matching character and try to **rm foo?bar**, you might potentially remove more than one file (although not in this example). **-i** is your friend!

ls -b shows control characters as octal numbers, which can be helpful if you need to identify them specifically. <Control-A> is 1 (\001 in octal), <Control-B> is 2, and so on, in alphabetical order. **man ascii** and the Wikipedia page for ASCII both include a nice table of control characters and their octal equivalents.

To delete the most horribly named files, you might need to resort to **rm -i ***.

Another option for removing files with squirrelly names is to use an alternative interface to the filesystem such as **emacs**'s dired mode or a visual tool such as Nautilus.

Regular files

Regular files consist of a series of bytes; filesystems impose no structure on their contents. Text files, data files, executable programs, and shared libraries are all stored as regular files. Both sequential access and random access are allowed.

Directories

A directory contains named references to other files. You can create directories with **mkdir** and delete them with **rmdir** if they are empty. You can recursively delete nonempty directories—including all their contents—with **rm -r**.

The special entries "**.**" and "**..**" refer to the directory itself and to its parent directory; they cannot be removed. Since the root directory has no real parent directory, the path "**/..**" is equivalent to the path "**/.**" (and both are equivalent to /).

Hard links

A file's name is stored within its parent directory, not with the file itself. In fact, more than one directory (or more than one entry in a single directory) can refer to a file at one time, and the references can have different names. Such an arrangement creates the illusion that a file exists in more than one place at the same time.

These additional references ("links," or "hard links" to distinguish them from symbolic links, discussed below) are synonymous with the original file; as far as the filesystem is concerned, all links to the file are equivalent. The filesystem maintains a count of the number of links that point to each file and does not release the file's data blocks until its last link has been deleted. Hard links cannot cross filesystem boundaries.

You create hard links with **ln** and remove them with **rm**. It's easy to remember the syntax of **ln** if you keep in mind that it mirrors the syntax of **cp**. The command **cp oldfile newfile** creates a copy of **oldfile** called **newfile**, and **ln oldfile newfile** makes the name **newfile** an additional reference to **oldfile**.

In most filesystem implementations, it is technically possible to make hard links to directories as well as to flat files. However, directory links often lead to degenerate conditions such as filesystem loops and directories that don't have a single, unambiguous parent. In most cases, a symbolic link (see this page) is a better option.

You can use **ls -l** to see how many links to a given file exist. See the **ls** example output here for some additional details. Also note the comments regarding **ls -i** on this page, as this option is particularly helpful for identifying hard links.

Hard links are not a distinct type of file. Instead of defining a separate "thing" called a hard link, the filesystem simply allows more than one directory entry to point to the same file. In addition to the file's contents, the underlying attributes of the file (such as ownerships and permissions) are also shared.

Character and block device files

See Chapter 11 for more information about devices and drivers.

Device files let programs communicate with the system's hardware and peripherals. The kernel includes (or loads) driver software for each of the system's devices. This software takes care of the messy details of managing each device so that the kernel itself can remain relatively abstract and hardware-independent.

Device drivers present a standard communication interface that looks like a regular file. When the filesystem is given a request that refers to a character or block device file, it simply passes the request to the appropriate device driver. It's important to distinguish device *files* from device *drivers*, however. The files are just rendezvous points that communicate with drivers. They are not drivers themselves.

The distinction between character and block devices is subtle and not worth reviewing in detail. In the past, a few types of hardware were represented by both block and character device files, but that configuration is rare today. As a matter of practice, FreeBSD has done away with block devices entirely, though their spectral presence can still be glimpsed in man pages and header files.

Device files are characterized by two numbers, called the major and minor device numbers. The major device number tells the kernel which driver the file refers to, and the minor device number typically tells the driver which physical unit to address. For example, major device number 4 on a Linux system denotes the serial driver. The first serial port (**/dev/tty0**) would have major device number 4 and minor device number 0.

Drivers can interpret the minor device numbers that are passed to them in whatever way they please. For example, tape drivers use the minor device number to determine whether the tape should be rewound when the device file is closed.

In the distant past, **/dev** was a generic directory and the device files within it were created with **mknod** and removed with **rm**. Unfortunately, this crude system was ill-equipped to deal with the endless sea of drivers and device types that have appeared over the last few decades. It also facilitated all sorts of potential configuration mismatches: device files that referred to no actual device, devices inaccessible because they had no device files, and so on.

These days, the **/dev** directory is normally mounted as a special filesystem type, and its contents are automatically maintained by the kernel in concert with a user-level daemon. There are a couple of different versions of this same basic system. See Chapter 11, *Drivers and the Kernel*, for more information about each system's approach to this task.

Local domain sockets

Sockets are connections between processes that allow them to communicate hygienically. UNIX defines several kinds of sockets, most of which involve the network.

See Chapter 10 for more information about syslog.

Local domain sockets are accessible only from the local host and are referred to through a filesystem object rather than a network port. They are sometimes known as “UNIX domain sockets.” Syslog and the X Window System are examples of standard facilities that use local domain sockets, but there are many more, including many databases and app servers.

Local domain sockets are created with the **socket** system call and removed with the **rm** command or the **unlink** system call once they have no more users.

Named pipes

Like local domain sockets, named pipes allow communication between two processes running on the same host. They're also known as "FIFO files" (As in financial accounting, FIFO is short for the phrase "first in, first out"). You can create named pipes with **mknod** and remove them with **rm**.

Named pipes and local domain sockets serve similar purposes, and the fact that both exist is essentially a historical artifact. Most likely, neither of them would exist if UNIX and Linux were designed today; network sockets would stand in for both.

Symbolic links

A symbolic or “soft” link points to a file by name. When the kernel comes upon a symbolic link in the course of looking up a pathname, it redirects its attention to the pathname stored as the contents of the link. The difference between hard links and symbolic links is that a hard link is a direct reference, whereas a symbolic link is a reference by name. Symbolic links are distinct from the files they point to.

You create symbolic links with **ln -s** and remove them with **rm**. Since symbolic links can contain arbitrary paths, they can refer to files on other filesystems or to nonexistent files. A series of symbolic links can also form a loop.

A symbolic link can contain either an absolute or a relative path. For example,

```
$ sudo ln -s archived/secure /var/data/secure
```

links **/var/data/secure** to **/var/data/archived/secure** with a relative path. It creates the symbolic link **/var/data/secure** with a target of **archived/secure**, as demonstrated by this output from **ls**:

```
$ ls -l /var/data/secure
lrwxrwxrwx 1 root root 18 Aug 3 12:54 /var/data/secure -> archived/secure
```

The entire **/var/data** directory could then be moved elsewhere without causing the symbolic link to stop working.

The file permissions that **ls** shows for a symbolic link, lrwxrwxrwx, are dummy values. Permission to create, remove, or follow the link is controlled by the containing directory, whereas read, write, and execute permission on the link target are granted by the target’s own permissions. Therefore, symbolic links do not need (and do not have) any permission information of their own.

A common mistake is to think that the first argument to **ln -s** is interpreted relative to the current working directory. However, that argument is not actually resolved as a filename by **ln**: it’s simply a literal string that becomes the target of the symbolic link.

5.5 File attributes

Under the traditional UNIX and Linux filesystem model, every file has a set of nine permission bits that control who can read, write, and execute the contents of the file. Together with three other bits that primarily affect the operation of executable programs, these bits constitute the file's "mode."

The twelve mode bits are stored along with four bits of file-type information. The four file-type bits are set when the file is first created and cannot be changed, but the file's owner and the superuser can modify the twelve mode bits with the **chmod** (change mode) command. Use **ls -l** (or **ls -ld** for a directory) to inspect the values of these bits. See this page for an example.

The permission bits

Nine permission bits determine what operations can be performed on a file and by whom. Traditional UNIX does not allow permissions to be set per user (although all systems now support access control lists of one sort or another; see this page). Instead, three sets of permissions define access for the owner of the file, the group owners of the file, and everyone else (in that order). Each set has three bits: a read bit, a write bit, and an execute bit (also in that order).

If you think of the owner as "the user" and everyone else as "other," you can remember the order of the permission sets by thinking of the name H**ugo**. **u**, **g**, and **o** are also the letter codes used by the mnemonic version of **chmod**.

It's convenient to discuss file permissions in terms of octal (base 8) numbers because each digit of an octal number represents three bits and each group of permission bits consists of three bits. The topmost three bits (with octal values of 400, 200, and 100) control access for the owner. The second three (40, 20, and 10) control access for the group. The last three (4, 2, and 1) control access for everyone else ("the world"). In each triplet, the high bit is the read bit, the middle bit is the write bit, and the low bit is the execute bit.

Although a user might fit into two of the three permission categories, only the most specific permissions apply. For example, the owner of a file always has access determined by the owner permission bits and never by the group permission bits. It is possible for the "other" and "group" categories to have more access than the owner, although this configuration would be highly unusual.

On a regular file, the read bit allows the file to be opened and read. The write bit allows the contents of the file to be modified or truncated; however, the ability to delete or rename (or delete and then re-create!) the file is controlled by the permissions on its parent directory, where the name-to-dataspace mapping is actually stored.

The execute bit allows the file to be executed. Two types of executable files exist: binaries, which the CPU runs directly, and scripts, which must be interpreted by a shell or some other program. By convention, scripts begin with a line similar to

```
#!/usr/bin/perl
```

that specifies an appropriate interpreter. Nonbinary executable files that do not specify an interpreter are assumed to be **sh** scripts.

The kernel understands the `#!` ("shebang") syntax and acts on it directly. However, if the interpreter is not specified completely and correctly, the kernel will refuse to execute the file. The shell then makes a second attempt to execute the script by calling **/bin/sh**, which is usually a link to the Almquist shell or to **bash**; see this page. Sven Mascheck maintains an excruciatingly

detailed page about the history, implementation, and cross-platform behavior of the shebang at goo.gl/J7izhL.

For a directory, the execute bit (often called the “search” or “scan” bit in this context) allows the directory to be entered or passed through as a pathname is evaluated, but not to have its contents listed. The combination of read and execute bits allows the contents of the directory to be listed. The combination of write and execute bits allows files to be created, deleted, and renamed within the directory.

A variety of extensions such as access control lists (see this page), SELinux (see this page), and “bonus” permission bits defined by individual filesystems (see this page) complicate or override the traditional 9-bit permission model. If you’re having trouble explaining the system’s observed behavior, check to see whether one of these factors might be interfering.

The setuid and setgid bits

The bits with octal values 4000 and 2000 are the setuid and setgid bits. When set on executable files, these bits allow programs to access files and processes that would otherwise be off-limits to the user that runs them. The setuid/setgid mechanism for executables is described here.

When set on a directory, the setgid bit causes newly created files within the directory to take on the group ownership of the directory rather than the default group of the user that created the file. This convention makes it easier to share a directory of files among several users, as long as they belong to a common group. This interpretation of the setgid bit is unrelated to its meaning when set on an executable file, but no ambiguity can exist as to which meaning is appropriate.

The sticky bit

The bit with octal value 1000 is called the sticky bit. It was of historical importance as a modifier for executable files on early UNIX systems. However, that meaning of the sticky bit is now obsolete and modern systems silently ignore the sticky bit when it’s set on regular files.

If the sticky bit is set on a directory, the filesystem won’t allow you to delete or rename a file unless you are the owner of the directory, the owner of the file, or the superuser. Having write permission on the directory is not enough. This convention helps make directories like **/tmp** a little more private and secure.

ls: list and inspect files

The filesystem maintains about forty separate pieces of information for each file, but most of them are useful only to the filesystem itself. As a system administrator, you will be concerned mostly with the link count, owner, group, mode, size, last access time, last modification time, and type. You can inspect all these with **ls -l** (or **ls -ld** for a directory; without the **-d** flag, **ls** lists the directory's contents).

An attribute change time is also maintained for each file. The conventional name for this time (the "ctime," short for "change time") leads some people to believe that it is the file's creation time. Unfortunately, it is not; it just records the time at which the attributes of the file (owner, mode, etc.) were last changed (as opposed to the time at which the file's contents were modified).

Consider the following example:

```
$ ls -l /usr/bin/gzip
-rwxr-xr-x  4 root  wheel    37432 Nov 11  2016 /usr/bin/gzip
```

The first field specifies the file's type and mode. The first character is a dash, so the file is a regular file. (See Table 5.2 for other codes.)

The next nine characters in this field are the three sets of permission bits. The order is owner-group-other, and the order of bits within each set is read-write-execute. Although these bits have only binary values, **ls** shows them symbolically with the letters `r`, `w`, and `x` for read, write, and execute. In this case, the owner has all permissions on the file and everyone else has read and execute permission.

If the setuid bit had been set, the `x` representing the owner's execute permission would have been replaced with an `s`, and if the setgid bit had been set, the `x` for the group would also have been replaced with an `s`. The last character of the permissions (execute permission for "other") is shown as `t` if the sticky bit of the file is turned on. If either the setuid/setgid bit or the sticky bit is set but the corresponding execute bit is not, these bits are shown as `S` or `T`.

The next field in the listing is the file's link count. In this case it is 4, indicating that **/usr/bin/gzip** is just one of four names for this file (the others on this system are **gunzip**, **gzcat**, and **zcat**, all in **/usr/bin**). Each time a hard link is made to a file, the file's link count is incremented by 1. Symbolic links do not affect the link count.

All directories have at least two hard links: the link from the parent directory and the link from the special file called . inside the directory itself.

The next two fields in the **ls** output are the owner and group owner of the file. In this example, the file's owner is root, and the file belongs to the group named wheel. The filesystem actually stores these as the user and group ID numbers rather than as names. If the text versions (names) can't be determined, **ls** shows the fields as numbers. This might happen if the user or group that

owns the file has been deleted from the **/etc/passwd** or **/etc/group** file. It could also suggest a problem with your LDAP database (if you use one); see Chapter 17.

The next field is the size of the file in bytes. This file is 37,432 bytes long. Next comes the date of last modification: November 11, 2016. The last field in the listing is the name of the file, **/usr/bin/gzip**.

ls output is slightly different for a device file. For example:

```
$ ls -l /dev/tty0
crw--w----. 1 root tty 4, 0 Aug  3 15:12 /dev/tty0
```

Most fields are the same, but instead of a size in bytes, **ls** shows the major and minor device numbers. **/dev/tty0** is the first virtual console on this (Red Hat) system and is controlled by device driver 4 (the terminal driver). The dot at the end of the mode indicates the absence of an access control list (ACL, discussed starting here). Some systems show this by default and some don't.

One **ls** option that's useful for scoping out hard links is **-i**, which tells **ls** to show each file's "inode number." Briefly, the inode number is an integer associated with the contents of a file. Inodes are the "things" that are pointed to by directory entries; entries that are hard links to the same file have the same inode number. To figure out a complex web of links, you need both **ls -li**, to show link counts and inode numbers, and **find**, to search for matches. (Try **find** *mountpoint* **-xdev -inum** *inode* **-print**.)

Some other **ls** options that are important to know are **-a** to show all entries in a directory (even files whose names start with a dot), **-t** to sort files by modification time (or **-tr** to sort in reverse chronological order), **-F** to show the names of files in a way that distinguishes directories and executable files, **-R** to list recursively, and **-h** to show file sizes in human-readable form (e.g., 8K or 53M).

Most versions of **ls** now default to color-coding files if your terminal program supports this (most do). **ls** specifies colors according to a limited and abstract palette ("red," "blue," etc.), and it's up to the terminal program to map these requests to specific colors. You may need to tweak both **ls** (the LSCOLORS or LS_COLORS environment variable) and the terminal emulator to achieve colors that are readable and unobtrusive. Alternatively, you can just remove the default configuration for colorization (usually **/etc/profile.d/colorls***) to eliminate colors entirely.

chmod: change permissions

The **chmod** command changes the permissions on a file. Only the owner of the file and the superuser can change a file's permissions. To use the command on early UNIX systems, you had to learn a bit of octal notation, but current versions accept both octal notation and a mnemonic syntax. The octal syntax is generally more convenient for administrators, but it can only be used to specify an absolute value for the permission bits. The mnemonic syntax can modify some bits but leave others alone.

The first argument to **chmod** is a specification of the permissions to be assigned, and the second and subsequent arguments are names of files on which permissions should be changed. In the octal case, the first octal digit of the specification is for the owner, the second is for the group, and the third is for everyone else. If you want to turn on the setuid, setgid, or sticky bits, you use four octal digits rather than three, with the three special bits forming the first digit.

Table 5.3 illustrates the eight possible combinations for each set of three bits, where `r`, `w`, and `x` stand for read, write, and execute.

Table 5.3: Permission encoding for chmod

Octal	Binary	Perms	Octal	Binary	Perms
0	000	---	4	100	r--
1	001	--x	5	101	r-x
2	010	-w-	6	110	rw-
3	011	-wx	7	111	rwx

For example, **chmod 711 myprog** gives all permissions to the user (owner) and execute-only permission to everyone else. (If **myprog** were a shell script, it would need both read and execute permission turned on. For the script to be run by an interpreter, it must be opened and read like a text file. Binary files are executed directly by the kernel and therefore do not need read permission turned on.)

For the mnemonic syntax, you combine a set of targets (**u**, **g**, or **o** for user, group, other, or **a** for all three) with an operator (+, -, = to add, remove, or set) and a set of permissions. The **chmod** man page gives the details, but the syntax is probably best learned by example. Table 5.4 exemplifies some mnemonic operations.

Table 5.4: Examples of chmod's mnemonic syntax

Spec	Meaning
u+w	Adds write permission for the owner of the file
ug=rw,o=r	Gives r/w permission to owner and group, and read permission to others
a-x	Removes execute permission for all categories (owner/group/other)
ug=srx,o=	Makes setuid/setgid and gives r/x permission to only owner and group
g=u	Makes the group permissions be the same as the owner permissions

The hard part about using the mnemonic syntax is remembering whether **o** stands for "owner" or "other"; "other" is correct. Just remember **u** and **g** by analogy to UID and GID; only one possibility is left. Or remember the order of letters in the name H**ugo**.



On Linux systems, you can also specify the modes to be assigned by copying them from an existing file. For example, **chmod --reference=filea fileb** makes **fileb**'s mode the same as **filea**'s.

With the **-R** option, **chmod** recursively updates the file permissions within a directory. However, this feat is trickier than it looks because the enclosed files and directories may not share the same attributes; for example, some might be executable files; others, text files. Mnemonic syntax is particularly useful with **-R** because it preserves bits whose values you don't set explicitly. For example,

```
$ chmod -R g+w mydir
```

adds group write permission to **mydir** and all its contents without messing up the execute bits of directories and programs.

If you *want* to adjust execute bits, be wary of **chmod -R**. It's blind to the fact that the execute bit has a different interpretation on a directory than it does on a flat file. Therefore, **chmod -R a-x** probably won't do what you intend. Use **find** to select only the regular files:

```
$ find mydir -type f -exec chmod a-x {} ';'
```

chown and chgrp: change ownership and group

The **chown** command changes a file's ownership, and the **chgrp** command changes its group ownership. The syntax of **chown** and **chgrp** mirrors that of **chmod**, except that the first argument is the new owner or group, respectively.

To change a file's group, you must either be the superuser or be the owner of the file and belong to the group you're changing to. Older systems in the SysV lineage allowed users to give away their own files with **chown**, but that's unusual these days; **chown** is now a privileged operation.

Like **chmod**, **chown** and **chgrp** offer the recursive **-R** flag to change the settings of a directory and all the files underneath it. For example, the sequence

```
$ sudo chown -R matt ~matt/restore
$ sudo chgrp -R staff ~matt/restore
```

could reset the owner and group of files restored from a backup for the user matt. Don't try to **chown** dot files with a command such as

```
$ sudo chown -R matt ~matt/.*
```

since the pattern matches **~matt/..** and therefore ends up changing the ownerships of the parent directory and probably the home directories of other users.

chown can change both the owner and group of a file at once with the syntax

```
chown user:group file ...
```

For example,

```
$ sudo chown -R matt:staff ~matt/restore
```

You can actually omit either *user* or *group*, which makes the **chgrp** command superfluous. If you include the colon but name no specific *group*, the Linux version of **chown** uses the user's default group.

Some systems accept the notation *user.group* as being equivalent to *user:group*. This is just a nod to historical variation among systems; it means the same thing.

umask: assign default permissions

You can use the built-in shell command **umask** to influence the default permissions given to the files you create. Every process has its own **umask** attribute; the shell's built-in **umask** command sets the shell's own **umask**, which is then inherited by commands that you run.

The **umask** is specified as a three-digit octal value that represents the permissions to *take away*. When a file is created, its permissions are set to whatever the creating program requests minus whatever the **umask** forbids. Thus, the individual digits of the **umask** allow the permissions shown in Table 5.5.

Table 5.5: Permission encoding for umask

Octal	Binary	Perms	Octal	Binary	Perms
0	000	rwx	4	100	-wx
1	001	rw-	5	101	-w-
2	010	r-x	6	110	--x
3	011	r--	7	111	---

For example, **umask 027** allows all permissions for the owner but forbids write permission to the group and allows no permissions for anyone else. The default **umask** value is often 022, which denies write permission to the group and world but allows read permission.

See Chapter 8 for more information about startup files.

In the standard access control model, you cannot force users to have a particular **umask** value because they can always reset it to whatever they want. However, you can put a suitable default in the sample startup files that you give to new users. If you require more control over the permissions on user-created files, you'll need to graduate to a mandatory access control system such as SELinux; see this page.

Linux bonus flags

Linux defines a set of supplemental flags that can be set on files to request special handling. For example, the **a** flag makes a file append-only, and the **i** flag makes it immutable and undeletable.

Flags have binary values, so they are either present or absent for a given file. The underlying filesystem implementation must support the corresponding feature, so not all flags can be used on all filesystem types. In addition, some flags are experimental, unimplemented, or read-only.

Linux uses the commands **lsattr** and **chattr** to view and change file attributes. Table 5.6 lists some of the more mainstream flags.

Table 5.6: Linux file attribute flags

Flag	FS[a]	Meaning
A	XBE	Never update access time (`st_atime`; for performance)
a	XBE	Allow writing only in append mode[b]
C	B	Disable copy-on-write updates
c	B	Compress contents
D	BE	Force directory updates to be written synchronously
d	XBE	Do not back up; backup utilities should ignore this file
i	XBE	Make file immutable and undeletable[b]
j	E	Keep a journal for data changes as well as metadata
S	XBE	Force changes to be written synchronously (no buffering)
X	B	Avoid data compression if it is the default

a. X = XFS, B = Btrfs, E = ext3 and ext4
b. Can be set only by root

As might be expected from such a random grab bag of features, the value of these flags to administrators varies. The main thing to remember is that if a particular file seems to be behaving strangely, check it with **lsattr** to see if it has one or more flags enabled.

Waiving maintenance of last-access times (the **A** flag) can boost performance in some situations. However, its value depends on the filesystem implementation and access pattern; you'll have to do your own benchmarking. In addition, modern kernels now default to mounting filesystems with the relatime option, which minimizes updates to st_atime and makes the **A** flag largely obsolete.

The immutable and append-only flags (**i** and **a**) were largely conceived as ways to make the system more resistant to tampering by hackers or hostile code. Unfortunately, they can confuse software and protect only against hackers that don't know enough to use **chattr -ia**. Real-world experience has shown that these flags are more often used *by* hackers than against them.

See Chapter 5 for more information about configuration management.

We have seen several cases in which admins have used the **i** (immutable) flag to prevent changes that would otherwise be imposed by a configuration management system such as Ansible or Salt. Needless to say, this hack creates confusion once the details have been forgotten and no one can figure out why configuration management isn't working. Never do this—just think of the shame your mother would feel if she knew what you'd been up to. Fix the issue within the configuration management system like Mom would want.

The "no backup" flag (**d**) is potentially of interest to administrators, but since it's an advisory flag, make sure that your backup system honors it.

Flags that affect journaling and write synchrony (**D**, **j**, and **S**) exist primarily to support databases. They are not of general use for administrators. All these options can reduce filesystem performance significantly. In addition, tampering with write synchrony has been known to confuse **fsck** on ext* filesystems.

5.6 Access control lists

The traditional 9-bit owner/group/other access control system is powerful enough to accommodate the vast majority of administrative needs. Although the system has clear limitations, it's very much in keeping with the UNIX traditions (some might say, "former traditions") of simplicity and predictability.

Access control lists, aka ACLs, are a more powerful but also more complicated way of regulating access to files. Each file or directory can have an associated ACL that lists the permission rules to be applied to it. Each of the rules within an ACL is called an access control entry or ACE.

An access control entry identifies the user or group to which it applies and specifies a set of permissions to be applied to those entities. ACLs have no set length and can include permission specifications for multiple users or groups. Most OSes limit the length of an individual ACL, but the limit is high enough (usually at least 32 entries) that it rarely comes into play.

The more sophisticated ACL systems let administrators specify partial sets of permissions or negative permissions. Most also have inheritance features that allow access specifications to propagate to newly created filesystem entities.

A cautionary note

ACLs are widely supported and occupy our attention for the rest of this chapter. However, neither of these facts should be interpreted as an encouragement to embrace them. ACLs have a niche, but it lies outside the mainstream of UNIX and Linux administration.

ACLs exist primarily to facilitate Windows compatibility and to serve the needs of the small segment of enterprises that actually require ACL-level flexibility. They are not the shiny next generation of access control and are not intended to supplant the traditional model.

ACLs' complexity creates several potential problems. Not only are ACLs tedious to use, but they can also cause unexpected interactions with ACL-unaware backup systems, network file service peers, and even simple programs such as text editors.

ACLs also tend to become increasingly unmaintainable as the number of entries grows. Real-world ACLs frequently include vestigial entries and entries that serve only to compensate for issues caused by previous entries. It's possible to refactor and simplify these complex ACLs, but that's risky and time consuming, so it rarely gets done.

In the past, copies of this chapter that we've sent out to professional administrators for review have often come back with notes such as, "This part looks fine, but I can't really say, because I've never used ACLs."

ACL types

Two types of ACLs have emerged as the predominant standards for UNIX and Linux: POSIX ACLs and NFSv4 ACLs.

The POSIX version dates back to specification work done in the mid-1990s. Unfortunately, no actual standard was ever issued, and initial implementations varied widely. These days, we are in much better shape. Systems have largely converged on a common framing for POSIX ACLs and a common command set, **getfacl** and **setfacl**, for manipulating them.

To a first approximation, the POSIX ACL model simply extends the traditional UNIX `rwx` permission system to accommodate permissions for multiple groups and users.

As POSIX ACLs were coming into focus, it became increasingly common for UNIX and Linux to share filesystems with Windows, which has its own set of ACL conventions. Here the plot thickens, because Windows makes a variety of distinctions that are not found in either the traditional UNIX model or its POSIX ACL equivalent. Windows ACLs are semantically more complex, too; for example, they allow negative permissions (“deny” entries) and have a complicated inheritance scheme.

See Chapter 21 for more information about NFS.

The architects of version 4 of NFS—a common file-sharing protocol—wanted to incorporate ACLs as a first-class entity. Because of the UNIX/Windows split and the inconsistencies among UNIX ACL implementations, it was clear that the systems on the ends of an NFSv4 connection might often be of different types. Each system might understand NFSv4 ACLs, POSIX ACLs, Windows ACLs, or no ACLs at all. The NFSv4 standard would have to be interoperable with all these various worlds without causing too many surprises or security problems.

Given this constraint, it’s perhaps not surprising that NFSv4 ACLs are essentially a union of all preexisting systems. They are a strict superset of POSIX ACLs, so any POSIX ACL can be represented as an NFSv4 ACL without loss of information. At the same time, NFSv4 ACLs accommodate all the permission bits found on Windows systems, and they have most of Windows’ semantic features as well.

Implementation of ACLs

In theory, responsibility for maintaining and enforcing ACLs could be assigned to several different components of the operating system. ACLs could be implemented by the kernel on behalf of all the system's filesystems, by individual filesystems, or perhaps by higher-level software such as NFS and SMB servers.

In practice, ACL support is both OS-dependent and filesystem-dependent. A filesystem that supports ACLs on one system might not support them on another, or it might feature a somewhat different implementation managed by different commands.

See Chapter 22 for more information about SMB.

File service daemons map their host's native ACL scheme (or schemes) to and from the conventions appropriate to the filing protocol: NFSv4 ACLs for NFS, and Windows ACLs for SMB. The details of that mapping depend on the implementation of the file server. Usually, the rules are complicated and somewhat tunable with configuration options.

Because ACL implementations are filesystem-specific and because systems support multiple filesystem implementations, some systems end up supporting multiple types of ACLs. Even a given filesystem might offer several ACL options, as seen in the various ports of ZFS. If multiple ACL systems are available, the commands to manipulate them might be the same or different; it depends on the system. Welcome to sysadmin hell.

Linux ACL support


Linux has standardized on POSIX-style ACLs. NFSv4 ACLs are not supported at the filesystem level, though of course Linux systems can mount and share NFSv4 filesystems over the network.

An advantage of this standardization is that nearly all Linux filesystems now include POSIX ACL support, including XFS, Btrfs, and the ext* family. Even ZFS, whose native ACL system is NFSv4-ish, has been ported to Linux with POSIX ACLs. The standard **getfacl** and **setfacl** commands can be used everywhere, without regard to the underlying filesystem type. (You may, however, need to ensure that the correct **mount** option has been used to mount the filesystem. Filesystems generally support an `acl` option, a `noacl` option, or both, depending on their defaults.)

Linux does have a command suite (**nfs4_getfacl**, **nfs4_setfacl**, and **nfs4_editfacl**) for grooming the NFSv4 ACLs of files mounted from NFS servers. However, these commands cannot be used on locally stored files. Moreover, they are rarely included in distributions' default software inventory; you'll have to install them separately.

FreeBSD ACL support



FreeBSD supports both POSIX ACLs and NFSv4 ACLs. Its native **getfacl** and **setfacl** commands have been extended to include NFSv4-style ACL wrangling. NSFv4 ACL support is a relatively recent (as of 2017) development.

At the filesystem level, both UFS and ZFS support NFSv4-style ACLs, and UFS supports POSIX ACLs as well. The potential point of confusion here is ZFS, which is NFSv4-only on BSD (and on Solaris, its system of origin) and POSIX-only on Linux.

For UFS, use one of the **mount** options `acls` or `nfsv4acls` to specify which world you want to live in. These options are mutually exclusive.

POSIX ACLs

POSIX ACLs are a mostly straightforward extension of the standard 9-bit UNIX permission model. Read, write, and execute permission are the only capabilities that the ACL system deals with. Embellishments such as the setuid and sticky bits are handled exclusively through the traditional mode bits.

ACLs allow the rwx bits to be set independently for any combination of users and groups. Table 5.7 shows what the individual entries in an ACL can look like.

Table 5.7: Entries that can appear in POSIX ACLs

Format	Example	Sets permissions for
user::*perms*	`user::rw-`	The file's owner
user:*username*:*perms*	`user:trent:rw-`	A specific user
group::*perms*	`group::r-x`	The group that owns the file
group:*groupname*:*perms*	`group:staff:rw-`	A specific group
other::*perms*	`other::---`	All others
mask::*perms*	`mask::rwx`	All but owner and other[a]

a. Masks are somewhat tricky and are explained later in this section.

Users and groups can be identified by name or by UID/GID. The exact number of entries that an ACL can contain varies with the filesystem implementation but is usually at least 32. That's probably about the practical limit for manageability, anyway.

Interaction between traditional modes and ACLs

Files with ACLs retain their original mode bits, but consistency is automatically enforced and the two sets of permissions can never conflict. The following example demonstrates that the ACL entries automatically update in response to changes made with the standard **chmod** command:

```
$ touch example
$ ls -l example
-rw-rw-r--  1 garth  garth    0 Jun 14 15:57 example
$ getfacl example
# file: example
# owner: garth
# group: garth
user::rw-
group::rw-
other::r--
$ chmod 640 example
$ ls -l example
-rw-r-----  1 garth  garth    0 Jun 14 15:57 example
$ getfacl --omit-header example
user::rw-
group::r--
other::---
```

(This example is from Linux. The FreeBSD version of **getfacl** uses **-q** instead of **--omit-header** to suppress the comment-like lines in the output.)

This enforced consistency allows older software with no awareness of ACLs to play reasonably well in the ACL world. However, there's a twist. Even though the group:: ACL entry in the example above appears to be tracking the middle set of traditional mode bits, that will not always be the case.

To understand why, suppose that a legacy program clears the write bits within all three permission sets of the traditional mode (e.g., **chmod ugo-w** *file*). The intention is clearly to make the file unwritable by anyone. But what if the resulting ACL were to look like this?

```
user::r--
group::r--
group:staff:rw-
other::r--
```

From the perspective of legacy programs, the file appears to be unmodifiable, yet it is actually writable by anyone in group staff. Not good. To reduce the chance of ambiguity and misunderstandings, the following rules are enforced:

- The user:: and other:: ACL entries are by definition identical to the "owner" and "other" permission bits from the traditional mode. Changing the mode changes the corresponding ACL entries, and vice versa.
- In all cases, the effective access permission afforded to the file's owner and to users not mentioned in another way are those specified in the user:: and other:: ACL entries, respectively.

- If a file has no explicitly defined ACL or has an ACL that consists of only one `user::`, one `group::`, and one `other::` entry, these ACL entries are identical to the three sets of traditional permission bits. This is the case illustrated in the **getfacl** example above. (Such an ACL is termed "minimal" and need not actually be implemented as a logically separate ACL.)
- In more complex ACLs, the traditional group permission bits correspond to a special ACL entry called `mask` rather than the `group::` ACL entry. The mask limits the access that the ACL can confer on *all* named users, *all* named groups, *and* the default group.

In other words, the mask specifies an upper bound on the access that the ACL can assign to individual groups and users. It is conceptually similar to the **umask**, except that the ACL mask is always in effect and that it specifies the allowed permissions rather than the permissions to be denied. ACL entries for named users, named groups, and the default group can include permission bits that are not present in the mask, but filesystems simply ignore them.

As a result, the traditional mode bits can never understate the access allowed by the ACL as a whole. Furthermore, clearing a bit from the group portion of the traditional mode clears the corresponding bit in the ACL mask and thereby forbids this permission to everyone but the file's owner and those who fall in the category of "other."

When the ACL shown in the previous example is expanded to include entries for a specific user and group, **setfacl** automatically supplies an appropriate mask:

```
$ ls -l example
-rw-r-----    1 garth  garth   0 Jun 14 15:57 example
$ setfacl -m user::r,user:trent:rw,group:admin:rw example
$ ls -l example
-r--rw----+   1 garth  garth   0 Jun 14 15:57 example
$ getfacl --omit-header example
user::r--
user:trent:rw-
group::r--
group:admin:rw-
mask::rw-
other::---
```

The **-m** option to **setfacl** means "modify": it adds entries that are not already present and adjusts those that are already there. Note that **setfacl** automatically generates a mask that allows all the permissions granted in the ACL to take effect. If you want to set the mask by hand, include it in the ACL entry list given to **setfacl** or use the **-n** option to prevent **setfacl** from regenerating it.

Note that after the **setfacl** command, **ls -l** shows a `+` sign at the end of the file's mode to denote that it now has a real ACL associated with it. The first **ls -l** shows no `+` because at that point the ACL is "minimal."

If you use the traditional **chmod** command to manipulate an ACL-bearing file, be aware that your settings for the "group" permissions affect only the mask. To continue the previous example:

```
$ chmod 770 example
$ ls -l example
-rwxrwx---+ 1 garth  staff    0 Jun 14 15:57 example
$ getfacl --omit-header example
user::rwx
user:trent:rw-
group::r--
group:admin:rw-
mask::rwx
other::---
```

The **ls** output in this case is misleading. Despite the apparently generous group permissions, no one actually has permission to execute the file by reason of group membership. To grant such permission, you must edit the ACL itself.

To remove an ACL entirely and revert to the standard UNIX permission system, use **setfacl -bn**. (Strictly speaking, the **-n** flag is needed only on FreeBSD. Without it, FreeBSD's **setfacl** leaves you with a vestigial `mask` entry that will screw up later group-mode changes. However, you can include the **-n** on Linux without creating problems.)

POSIX access determination

When a process attempts to access a file, its effective UID is compared to the UID that owns the file. If they are the same, access is determined by the ACL's `user::` permissions. Otherwise, if a matching user-specific ACL entry exists, permissions are determined by that entry in combination with the ACL mask.

If no user-specific entry is available, the filesystem tries to locate a valid group-related entry that authorizes the requested access; these entries are processed in conjunction with the ACL mask. If no matching entry can be found, the `other::` entry prevails.

POSIX ACL inheritance

In addition to the ACL entry types listed in Table 5.7, the ACLs for directories can include `default` entries that are propagated to the ACLs of newly created files and subdirectories created within them. Subdirectories receive these entries both in the form of active ACL entries and in the form of copies of the default entries. Therefore, the original default entries may eventually propagate down through several layers of the directory hierarchy.

Once default entries have been copied to new subdirectories, there is no ongoing connection between the parent and child ACLs. If the parent's default entries change, those changes are not reflected in the ACLs of existing subdirectories.

You can set default ACL entries with **setfacl -dm**. Alternatively, you can include default entries within a regular access control entry list by prefixing them with default:.

If a directory has *any* default entries, it must include a full set of defaults for user::, group::, other::, and mask::. **setfacl** will fill in any default entries you don’t specify by copying them from the current permissions ACL, generating a summary mask as usual.

NFSv4 ACLs


In this section, we discuss the characteristics of NFSv4 ACLs and briefly review the command syntax used to set and inspect them on FreeBSD. They aren't supported on Linux (other than by NFS service daemons).

From a structural perspective, NFSv4 ACLs are similar to Windows ACLs. The main difference between them lies in the specification of the entity to which an access control entry refers.

In both systems, the ACL stores this entity as a string. For Windows ACLs, the string typically contains a Windows security identifier (SID), whereas for NFSv4, the string is typically of the form user:*username* or group:*groupname*. It can also be one of the special tokens owner@, group@, or everyone@. These latter entries are the most common because they correspond to the mode bits found on every file.

Systems such as Samba that share files between UNIX and Windows systems must provide some way of mapping between Windows and NFSv4 identities.

The NFSv4 and Windows permission models are more granular than the traditional UNIX read-write-execute model. In the case of NFSv4, the main refinements are as follows:

- NFSv4 distinguishes permission to create files within a directory from permission to create subdirectories.
- NFSv4 has a separate "append" permission bit.
- NFSv4 has separate read and write permissions for data, file attributes, extended attributes, and ACLs.
- NFSv4 controls a user's ability to change the ownership of a file through the standard ACL system. In traditional UNIX, the ability to change the ownership of files is usually reserved for root.

Table 5.8 shows the various permissions that can be assigned in the NFSv4 system. It also shows the one-letter codes used to represent them and the more verbose canonical names.

Table 5.8: NFSv4 file permissions

Code	Verbose name	Permission
r	read_data	Read data (file) or list directory contents (directory)
w	write_data	Write data (file) or create file (directory)
x	execute	Execute as a program
p	append_data	Append data (file) or create subdirectory (directory)
D	delete_child	Delete child within a directory
d	delete	Delete
a	read_attributes	Read nonextended attributes
A	write_attributes	Write nonextended attributes
R	read_xattr	Read named ("extended") attributes
W	write_xattr	Write named ("extended") attributes
c	read_acl	Read access control list
C	write_acl	Write access control list
o	write_owner	Change ownership
s	synchronize	Allow requests for synchronous I/O (usually ignored)

Although the NFSv4 permission model is fairly detailed, the individual permissions should mostly be self-explanatory. (The "synchronize" permission allows a client to specify that its modifications to a file should be synchronous—that is, calls to **write** should not return until the data has actually been saved on disk.)

An extended attribute is a named chunk of data that is stored along with a file; most modern filesystems support such attributes. At this point, the predominant use of extended attributes is to store ACLs themselves. However, the NFSv4 permission model treats ACLs separately from other extended attributes.

In FreeBSD's implementation, a file's owner always has read_acl, write_acl, read_attributes, and write_attributes permissions, even if the file's ACL itself specifies otherwise.

NFSv4 entities for which permissions can be specified

In addition to the garden-variety user:*username* and group:*groupname* specifiers, NFSv4 defines several special entities that may be assigned permissions in an ACL. Most important among these are owner@, group@, and everyone@, which correspond to the traditional categories in the 9-bit permission model.

NFSv4 has several differences from POSIX. For one thing, it has no default entity, used in POSIX to control ACL inheritance. Instead, any individual access control entry (ACE) can be flagged as inheritable (see *ACL inheritance in NFSv4*, below). NFSv4 also does not use a mask to reconcile the permissions specified in a file's mode with its ACL. The mode is required to be consistent with the settings specified for owner@, group@, and everyone@, and filesystems that implement NFSv4 ACLs must preserve this consistency when either the mode or the ACL is updated.

NFSv4 access determination

The NFSv4 system differs from POSIX in that an ACE specifies only a partial set of permissions. Each ACE is either an “allow” ACE or a “deny” ACE; it acts more like a mask than an authoritative specification of all possible permissions. Multiple ACEs can apply to any given situation.

When deciding whether to allow a particular operation, the filesystem reads the ACL in order, processing ACEs until either all requested permissions have been granted or some requested permission has been denied. Only ACEs whose entity strings are compatible with the current user’s identity are considered.

This iterative evaluation process means that owner@, group@, and everyone@ are not exact analogs of the corresponding traditional mode bits. An ACL can contain multiple copies of these elements, and their precedence is determined by their order of appearance in the ACL rather than by convention. In particular, everyone@ really does apply to everyone, not just users who aren’t addressed more specifically.

It’s possible for the filesystem to reach the end of an ACL without having obtained a definitive answer to a permission query. The NFSv4 standard considers the result to be undefined, but real-world implementations deny access, both because this is the convention used by Windows and because it’s the only option that makes sense.

ACL inheritance in NFSv4

Like POSIX ACLs, NFSv4 ACLs allow newly created objects to inherit access control entries from their enclosing directory. However, the NFSv4 system is a bit more powerful and a lot more confusing. Here are the important points:

- You can flag any ACE as inheritable. Inheritance for newly created subdirectories (dir_inherit or d) and inheritance for newly created files (file_inherit or f) are flagged separately.
- You can apply different access control entries to new files and new directories by creating separate access control entries on the parent directory and flagging them appropriately. You can also apply a single ACE to all new child entities (of whatever type) by turning on both the d and f flags.
- From the perspective of access determination, access control entries have the same effect on the parent (source) directory whether or not they are inheritable. If you want an entry to apply to children but not to the parent directory itself, turn on the ACE’s inherit_only (i) flag.
- New subdirectories normally inherit two copies of each ACE: one with the inheritance flags turned off, which applies to the subdirectory itself; and one with the inherit_only flag turned on, which sets up the new subdirectory to propagate its inherited ACEs. You can suppress the creation of this second ACE by turning on the no_propagate

(n) flag on the parent directory's copy of the ACE. The end result is that the ACE propagates only to immediate children of the original directory.

- Don't confuse the propagation of access control entries with true inheritance. Your setting an inheritance-related flag on an ACE simply means that the ACE will be copied to new entities. It does not create any ongoing relationship between the parent and its children. If you later change the ACE entries on the parent directory, the children are not updated.

Table 5.9 summarizes these various inheritance flags.

Table 5.9: NFSv4 ACE inheritance flags

Code	Verbose name	Meaning
f	file_inherit	Propagate this ACE to newly created files.
d	dir_inherit	Propagate this ACE to newly created subdirectories.
i	inherit_only	Propagate, but don't apply to the current directory.
n	no_propagate	Propagate to new subdirectories, but turn off inheritance.

NFSv4 ACL viewing

FreeBSD has extended the standard **setfacl** and **getfacl** commands used with POSIX ACLs to handle NFSv4 ACLs as well. For example, here's the ACL for a newly created directory:

```
freebsd$ mkdir example
freebsd$ ls -ld example
drwxr-xr-x  2 garth  staff  2 Aug 16 18:52 example/

$ getfacl -q example
         owner@:rwxp--aARWcCos:------:allow
         group@:r-x---a-R-c--s:------:allow
      everyone@:r-x---a-R-c--s:------:allow
```

The **-v** flag requests verbose permission names. (We indented the following output lines and wrapped them at slashes to clarify structure.)

```
freebsd$ getfacl -qv example
    owner@:read_data/write_data/execute/append_data/read_attributes/
        write_attributes/read_xattr/write_xattr/read_acl/write_acl/
        write_owner/synchronize::allow
    group@:read_data/execute/read_attributes/read_xattr/read_acl/
        synchronize::allow
    everyone@:read_data/execute/read_attributes/read_xattr/read_acl/
        synchronize::allow
```

This newly created directory seems to have a complex ACL, but in fact this is just the 9-bit mode translated into ACLese. It is not necessary for the filesystem to store an actual ACL, because the ACL and the mode are equivalent. (As with POSIX ACLS, such lists are termed “minimal” or “trivial.”) If the directory had an actual ACL, **ls** would show the mode bits with a `+` on the end (i.e., `drwxr-xr-x+`) to mark its presence.

Each clause represents one access control entry. The format is

```
entity:permissions:inheritance_flags:type
```

The *entity* can be the keywords `owner@`, `group@`, or `everyone@`, or a form such as `user:`*username* or `group:`*groupname*. Both the *permissions* and the *inheritance_flags* are slash-separated lists of options in the verbose output and **ls**-style bitmaps in short output. The *type* of an ACE is either `allow` or `deny`.

The use of a colon as a subdivider within the *entity* field makes it tricky for scripts to parse **getfacl** output no matter which output format you use. If you need to process ACLs programmatically, it’s best to do so through a modular API rather than by parsing command output.

Interactions between ACLs and modes

The mode and the ACL must remain consistent, so whenever you adjust one of these entities, the other automatically updates to conform to it. It’s easy for the system to determine the appropriate mode for a given ACL. However, emulating the traditional behavior of the mode with a series of access control entries is trickier, especially in the context of an existing ACL. The system must often generate multiple and seemingly inconsistent sets of entries for `owner@`, `group@`, and `everyone@` that depend on evaluation order for their aggregate effect.

As a rule, it’s best to avoid tampering with a file’s or directory’s mode once you’ve applied an ACL.

NFSv4 ACL setup

Because the permission system enforces consistency between a file’s mode and its ACL, all files have at least a trivial ACL. Ergo, ACL changes are always updates.

You make ACL changes with the **setfacl** command, much as you do under the POSIX ACL regime. The main difference is that order of access control entries is significant for an NFSv4 ACL, so you might need to insert new entries at a particular point within the existing ACL. You can do that with the **-a** flag:

```
setfacl -a position entries file ...
```

Here, *position* is the index of the existing access control entry (numbered starting at zero) in front of which the new entries should be inserted. For example, the command

```
$ setfacl -a 0 user:ben:full_set::deny ben_keep_out
```

installs an access control entry on the file **ben_keep_out** that denies all permissions to the user ben. The notation **full_set** is a shorthand notation that includes all possible permissions. (Written out, those would currently be **rwxpDdaARWcCos**; compare with Table 5.8.)

Because the new access control entry is inserted at position zero, it's the first one consulted and takes precedence over later entries. Ben will be denied access to the file even if, for example, the everyone@ permissions grant access to other users.

You can also use long names such as **write_data** to identify permissions. Separate multiple long names with slashes. You cannot mix single-letter codes and long names in a single command.

As with POSIX ACLs, you can use the **-m** flag to add new entries to the end of the existing ACL.

As for complex changes to existing ACLs, you can best achieve them by dumping the ACL to a text file, editing the access control entries in a text editor, and then reloading the entire ACL. For example:

```
$ getfacl -q file > /tmp/file.acl
$ vi /tmp/file.acl                    # Make any required changes
$ setfacl -b -M /tmp/file.acl file
```

setfacl's **-b** option removes the existing ACL before adding the access control entries listed in **file.acl**. Its inclusion lets you delete entries simply by removing them from the text file.

6 Software Installation and Management



The installation, configuration, and management of software is a large part of most sysadmins' jobs. Administrators respond to installation and configuration requests from users, apply updates to fix security problems, and supervise transitions to new software releases that may offer both new features and incompatibilities. Generally, administrators perform all the following tasks:

- Automating mass installations of operating systems
- Maintaining custom OS configurations
- Keeping systems and applications patched and up to date
- Tracking software licenses
- Managing add-on software packages

The process of configuring an off-the-shelf distribution or software package to conform to your needs (and to your local conventions for security, file placement, and network topology) is often referred to as "localization." This chapter explores some techniques and software that help reduce the pain of software installation and make these tasks scale more gracefully. We also discuss the installation procedure for each of our example operating systems, including some options for automated deployment that use common (platform-specific) tools.

6.1 Operating system installation

Linux distributions and FreeBSD have straightforward procedures for basic installation. For physical hosts, installation typically involves booting from external USB storage or optical media, answering a few basic questions, optionally configuring disk partitions, and then telling the installer which software packages to install. Most systems, including all our example distributions, include a "live" option on the installation media that lets you run the operating system without actually installing it on a local disk.

Installing the base operating system from local media is fairly trivial thanks to the GUI applications that shepherd you through the process. Table 6.1 lists pointers to detailed installation instructions for each of our example systems.

Table 6.1: Installation documentation

System	Documentation source
Red Hat	redhat.com/docs/manuals/enterprise
CentOS	wiki.centos.org/Manuals/ReleaseNotes/CentOS7
Debian	debian.org/releases/stable/installmanual
Ubuntu	help.ubuntu.com/lts/serverguide/installation.html
FreeBSD	freebsd.org/doc/handbook/bsdinstall.html

Installing from the network

If you have to install an operating system on more than one computer, you will quickly reach the limits of interactive installation. It's time consuming, error prone, and boring to repeat the standard installation process on hundreds of systems. You can minimize human errors with a localization checklist, but even this measure does not remove all potential sources of variation.

To alleviate some of these problems, you can use network installation options to simplify deployments. Network installations are appropriate for sites with more than ten or so systems. The most common methods use DHCP and TFTP to boot the system sans physical media. They then retrieve the OS installation files from a network server with HTTP, NFS, or FTP.

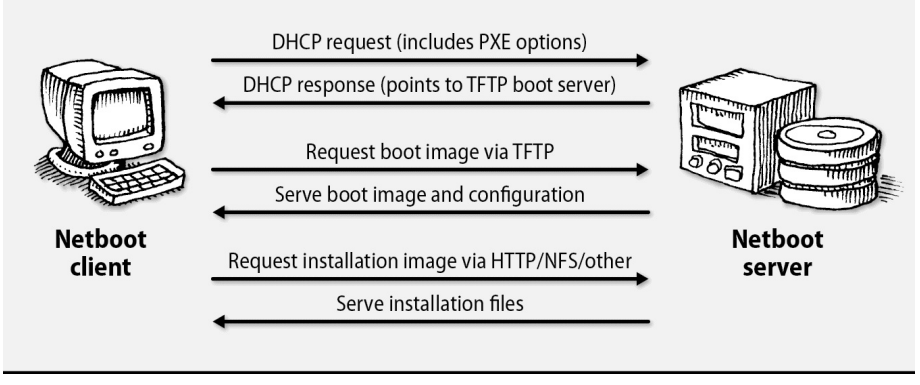
You can set up completely hands-free installations through PXE, the Preboot eXecution Environment. This scheme is a standard from Intel that lets systems boot from a network interface. It works especially well in virtualized environments.

PXE acts like a miniature OS that sits in a ROM on your network card. It exposes its network capabilities through a standardized API for the system BIOS to use. This cooperation makes it possible for a single boot loader to netboot any PXE-enabled PC without having to supply special drivers for each network card.

See this page for more information about DHCP.

The external (network) portion of the PXE protocol is straightforward and is similar to the netboot procedures used on other architectures. A host broadcasts a DHCP "discover" request with the PXE flag turned on, and a DHCP server or proxy responds with a DHCP packet that includes PXE options (the name of a boot server and boot file). The client downloads its boot file over TFTP (or, optionally, multicast TFTP) and then executes it. The PXE boot procedure is depicted in Exhibit A.

Exhibit A: PXE boot and installation process



The DHCP, TFTP, and file servers can all be located on different hosts. The TFTP-provided boot file includes a menu with pointers to the available OS boot images, which can then be retrieved from the file server with HTTP, FTP, NFS, or some other network protocol.

PXE booting is most commonly used in conjunction with unattended installation tools such as Red Hat's kickstart or Debian's preseeding system, as discussed in the upcoming sections. You can also use PXE to boot diskless systems such as thin clients.

Cobbler, discussed here, includes some glue that makes netbooting much easier. However, you will still need a working knowledge of the tools that underlie Cobbler, beginning with PXE.

Setting up PXE

The most widely used PXE boot system is H. Peter Anvin's PXELINUX, which is part of his SYSLINUX suite of boot loaders for every occasion. Check it out at syslinux.org. Another option is iPXE (ipxe.org), which supports additional bootstrapping modes, including support for wireless networks.

PXELINUX supplies a boot file that you install in the TFTP server's **tftpboot** directory. To boot from the network, a PC downloads the PXE boot loader and its configuration from the TFTP server. The configuration file lists one or more options for operating systems to boot. The system can boot through to a specific OS installation without any user intervention, or it can display a custom boot menu.

PXELINUX uses the PXE API for its downloads and is therefore hardware independent all the way through the boot process. Despite the name, PXELINUX is not limited to booting Linux. You can deploy PXELINUX to install FreeBSD and other operating systems, including Windows.

See this page for more information about DHCP server software.

On the DHCP side, ISC's (the Internet Systems Consortium's) DHCP server is your best bet for serving PXE information. Alternatively, try Dnsmasq (goo.gl/FNk7a), a lightweight server with DNS, DHCP, and netboot support. Or simply use Cobbler, discussed below.

Using kickstart, the automated installer for Red Hat and CentOS



Kickstart is a Red Hat-developed tool for performing automated installations. It is really just a scripting interface to the standard Red Hat installer software, Anaconda, and depends both on the base distribution and on RPM packages. Kickstart is flexible and quite smart about autodetecting the system's hardware, so it works well for bare-metal and virtual machines alike. Kickstart installs can be performed from optical media, the local hard drive, NFS, FTP, or HTTP.

Setting up a kickstart configuration file

Kickstart's behavior is controlled by a single configuration file, generally called **ks.cfg**. The format of this file is straightforward. If you're visually inclined, Red Hat's handy GUI tool **system-config-kickstart** lets you point and click your way to **ks.cfg** nirvana.

A kickstart config file consists of three ordered parts. The first part is the command section, which specifies options such as language, keyboard, and time zone. This section also specifies the source of the distribution with the `url` option. In the following example, it's a host called installserver.

Here's an example of a complete command section:

```
text
lang en_US                      # lang is used during the installation...
langsupport en_US               #   ...and langsupport at run time
keyboard us                     # Use an American keyboard
timezone --utc America/EST      # --utc means hardware clock is on GMT
mouse
rootpw --iscrypted $6$NaCl$X5jRlREy9DqNTCXjHp075/
reboot                          # Reboot after installation. Always wise.
bootloader --location=mbr       # Install default boot loader in the MBR
install                         # Install a new system, don't upgrade
url --url http://installserver/redhat
clearpart --all --initlabel     # Clear all existing partitions
part / --fstype ext3 --size 4096
part swap --size 1024
part /var --fstype ext3 -size 1 --grow
network --bootproto dhcp
auth --useshadow --enablemd5
firewall --disabled
xconfig --defaultdesktop=GNOME --startxonboot --resolution 1280x1024
    --depth 24
```

Kickstart uses graphical mode by default, which defeats the goal of unattended installation. The `text` keyword at the top of the example fixes this.

The rootpw option sets the new machine's root password. The default is to specify the password in cleartext, which presents a serious security problem. Always use the --iscrypted flag to specify a hashed password. To encrypt a password for use with kickstart, use **openssl passwd -1**. Still, this option leaves all your systems with the same root password. Consider running a postboot process to change the password at build time.

The clearpart and part directives specify a list of disk partitions and their sizes. You can include the --grow option to expand one of the partitions to fill any remaining space on the disk. This feature makes it easy to accommodate systems that have different sizes of hard disk. Advanced partitioning options, such as the use of LVM, are supported by kickstart but not by the **system-config-kickstart** tool. Refer to Red Hat's on-line documentation for a complete list of disk layout options.

The second section is a list of packages to install. It begins with a %packages directive. The list can contain individual packages, collections such as @ GNOME, or the notation @ Everything to include the whole shebang. When selecting individual packages, specify only the package name, not the version or the **.rpm** extension. Here's an example:

```
%packages
@ Networked Workstation
@ X Window System
@ GNOME
mylocalpackage
```

In the third section of the kickstart configuration file, you can specify arbitrary shell commands for kickstart to execute. There are two possible sets of commands: one introduced with %pre that runs before installation, and one introduced with %post that runs afterward. Both sections have some restrictions on the ability of the system to resolve hostnames, so it's safest to use IP addresses if you want to access the network. In addition, the postinstall commands are run in a **chroot**ed environment, so they cannot access the installation media.

The **ks.cfg** file is quite easy to generate programmatically. One option is to use the pykickstart Python library, which can read and write kickstart configurations.

For example, suppose you wanted to install different sets of packages on servers and clients and that you also have two separate physical locations that require slightly different customizations. You could use pykickstart to write a script that transforms a master set of parameters into a set of four separate configuration files, one for servers and one for clients in each office.

Changing the complement of packages would then be just a matter of changing the master configuration file rather than of changing every possible configuration file. There may even be cases in which you need to generate individualized configuration files for specific hosts. In this situation, you would certainly want the final **ks.cfg** files to be automatically generated.

Building a kickstart server

Kickstart expects its installation files, called the installation tree, to be laid out as they are on the distribution media, with packages stored in a directory called **RedHat/RPMS** on the server. If you're installing over the network via FTP, NFS, or HTTP, you can either copy the contents of the distribution (leaving the tree intact), or you can simply use the distribution's ISO images. You can also add your own packages to this directory. There are, however, a couple of issues to be aware of.

First, if you tell kickstart to install all packages (with an `@ Everything` in the packages section of your **ks.cfg**), it installs add-on packages in alphabetical order once all the base packages have been laid down. If your package depends on other packages that are not in the base set, you might want to call your package something like **zzmypackage.rpm** to make sure that it's installed last.

If you don't want to install all packages, either list your supplemental packages individually in the `%packages` section of the **ks.cfg** file or add your packages to one or more of the collection lists. Collection lists are specified by entries such as `@ GNOME` and stand for a predefined set of packages whose members are enumerated in the file **RedHat/base/comps** on the server. The collections are the lines that begin with 0 or 1; the number specifies whether the collection is selected by default.

In general, it's not a good idea to tamper with the standard collections. Leave them as Red Hat defined them and explicitly name all your supplemental packages in the **ks.cfg** file.

Pointing kickstart at your config file

See this page for more information about PXE.

Once you've created a config file, you have a couple of ways to get kickstart to use it. The officially sanctioned method is to boot from external media (USB or DVD) and ask for a kickstart installation by specifying **linux inst.ks** at the initial `boot:` prompt. PXE boot is also an option.

If you don't specify additional arguments, the system determines its network address with DHCP. It then obtains the DHCP boot server and boot file options, attempts to mount the boot server with NFS, and uses the value of the boot file option as its kickstart configuration file. If no boot file has been specified, the system looks for a file called **/kickstart/***host_ip_address*-**kickstart**.

Alternatively, you can tell kickstart to get its configuration file in some other way by supplying a path as an argument to the **inst.ks** option. There are several possibilities. For example, the instruction

```
boot: linux inst.ks=http:server:/path
```

tells kickstart to use HTTP to download the file instead of NFS. (Prior to RHEL 7, the option was called **ks** instead of **inst.ks**. Both are understood for now, but future versions may drop **ks**.)

To eliminate the use of boot media entirely, you'll need to graduate to PXE. See this page for more information about that.

Automating installation for Debian and Ubuntu



Debian and Ubuntu can use the **debian-installer** for "preseeding," the recommended method for automated installation. As with Red Hat's kickstart, a preconfiguration file answers questions asked by the installer.

All the interactive parts of the Debian installer use the **debconf** utility to decide which questions to ask and what default answers to use. By giving **debconf** a database of preformulated answers, you fully automate the installer. You can either generate the database by hand (it's a text file), or you can perform an interactive installation on an example system and then dump out your **debconf** answers with the following commands:

```
$ sudo debconf-get-selections --installer > preseed.cfg
$ sudo debconf-get-selections >> preseed.cfg
```

Make the config file available on the net and then pass it to the kernel at installation time with the following kernel argument:

```
preseed/url=http://host/path/to/preseed
```

The syntax of the preseed file, usually called **preseed.cfg**, is simple and is reminiscent of Red Hat's **ks.cfg**. The sample below has been shortened for simplicity.

```
d-i debian-installer/locale string en_US
d-i console-setup/ask_detect boolean false
d-i console-setup/layoutcode string us
d-i netcfg/choose_interface select auto
d-i netcfg/get_hostname string unassigned-hostname
d-i netcfg/get_domain string unassigned-domain
...
d-i partman-auto/disk string /dev/sda
d-i partman-auto/method string lvm
d-i partman-auto/choose_recipe select atomic
...
d-i passwd/user-fullname string Daffy Duck
d-i passwd/username string dduck
d-i passwd/user-password-crypted password $6$/mkq9/$G//i6tN.x6670.95lVSM/
d-i user-setup/encrypt-home boolean false
tasksel tasksel/first multiselect ubuntu-desktop
d-i grub-installer/only_debian boolean true
d-i grub-installer/with_other_os boolean true
d-i finish-install/reboot_in_progress note
xserver-xorg xserver-xorg/autodetect_monitor boolean true
...
```

Several options in this list simply disable dialogs that would normally require user interaction. For example, the console-setup/ask_detect clause disables manual keymap selection.

This configuration tries to identify a network interface that's actually connected to a network (choose_interface select auto) and obtains network information through DHCP. The system hostname and domain values are presumed to be furnished by DHCP and are not overridden.

Preseeded installations cannot use existing partitions; they must either use existing free space or repartition the entire disk. The partman* lines in the code above are evidence that the partman-auto package is being used for disk partitioning. You must specify a disk to install to unless the system has only one. In this case, **/dev/sda** is used.

Several partitioning recipes are available.

- atomic puts all the system's files in one partition.
- home creates a separate partition for **/home**.
- multi creates separate partitions for **/home**, **/usr**, **/var**, and **/tmp**.

You can create users with the passwd series of directives. As with kickstart configuration, we strongly recommend the use of encrypted (hashed) password values. Preseed files are often stored on HTTP servers and are apt to be discovered by curious users. (Of course, a hashed password is still subject to brute force attack. Use a long, complex password.)

The task selection (tasksel) option chooses the type of Ubuntu system to install. Available values include standard, ubuntu-desktop, dns-server, lamp-server, kubuntu-desktop, edubuntu-desktop, and xubuntu-desktop.

The sample preseed file shown above comes from the Ubuntu installation documentation found at help.ubuntu.com. This guide contains full documentation for the syntax and usage of the preseed file.

Although Ubuntu does not descend from the Red Hat lineage, it has grafted compatibility with kickstart control files onto its own underlying installer. Ubuntu also includes the **system-config-kickstart** tool for creating these files. However, the kickstart functionality in Ubuntu's installer is missing a number of important features that are supported by Red Hat's Anaconda, such as LVM and firewall configuration. We recommend sticking with the Debian installer unless you have a good reason to choose kickstart (e.g., to maintain compatibility with your Red Hat systems).

Netbooting with Cobbler, the open source Linux provisioning server

By far the easiest way to bring netbooting services to your network is with Cobbler, a project originally written by Michael DeHaan, prolific open source developer. Cobbler enhances kickstart to remove some of its most tedious and repetitive administrative elements. It bundles all the important netboot features, including DHCP, DNS, and TFTP, and helps you manage the OS images used to build physical and virtual machines. Cobbler includes command-line and web interfaces for administration.

Templates are perhaps Cobbler's most interesting and useful feature. You'll frequently need different kickstart and preseed settings for different host profiles. For example, you might have web servers in two data centers that, apart from network settings, require the same configuration. You can use Cobbler "snippets" to share sections of the configuration between the two types of hosts.

A snippet is just a collection of shell commands. For example, this snippet adds a public key to the authorized SSH keys for the root user:

```
mkdir -p --mode=700 /root/.ssh
cat >> /root/.ssh/authorized_keys << EOF
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQDKErzVdarNkL4bzAZotSzU/
... Rooy2R6TCzc1Bt/oqUK1R1kuV
EOF
chmod 600 /root/.ssh/authorized_keys
```

You save the snippet to Cobbler's snippet directory, then refer to it in a kickstart template. For example, if you saved the snippet above as **root_pubkey_snippet**, you could refer to it in a template as follows.

```
%post
SNIPPET::root_pubkey_snippet
$kickstart_done
```

Use Cobbler templates to customize disk partitions, conditionally install packages, customize time zones, add custom package repositories, and perform any other kind of localization requirement.

Cobbler can also create new virtual machines under a variety of hypervisors. It can integrate with a configuration management system to provision machines once they boot.

Cobbler packages are available in the standard repositories for our sample Linux distributions. You can also obtain packages and documentation from the Cobbler GitHub project at cobbler.github.io.

Automating FreeBSD installation

The FreeBSD **bsdinstall** utility is a text-based installer that kicks off when you boot a computer from a FreeBSD installation CD or DVD. Its automation facilities are rudimentary compared to Red Hat's kickstart or Debian's preseed, and the documentation is limited. The best source of information is the **bsdinstall** man page.

Creating a customized, unattended installation image is a tedious affair that involves the following steps.

1. Download the latest installation ISO (CD image) from ftp.freebsd.org.
2. Unpack the ISO image to a local directory.
3. Make any desired edits in the cloned directory.
4. Create a new ISO image from your customized layout and burn it to media, or create a PXE boot image for netbooting.

FreeBSD's version of **tar** understands ISO format in addition to many other formats, so you can simply extract the CD image files to a scratch directory. Create a subdirectory before extracting, because the ISO file unpacks to the current directory.

```
freebsd$ sudo mkdir FreeBSD
freebsd$ sudo tar xpCf FreeBSD FreeBSD-11.0.iso
```

Once you've extracted the contents of the image, you can customize them to reflect your desired installation settings. For example, you could add custom DNS resolvers by editing **FreeBSD/etc/resolv.conf** to include your own name servers.

bsdinstall normally requires users to select settings such as the type of terminal in use, the keyboard mapping, and the desired style of disk partitioning. You can bypass the interactive questions by putting a file called **installerconfig** in the **etc** directory of the system image.

This file's format is described in the **bsdinstall** man page. It has two sections:

- The preamble, which sets certain installation settings
- A shell script which executes after installation completes

We refer you to the man page rather than regurgitate its contents here. Among other settings, it contains options for installing directly to a ZFS root and to other custom partitioning schemes.

Once your customizations are complete, you can create a new ISO file with the **mkisofs** command. Create a PXE image or burn the ISO to optical media for an unattended installation.

The mfsBSD project (mfsbsd.vx.sk) is a set of scripts that generate a PXE-friendly ISO image. The basic FreeBSD 11 image weighs in at a lean 47MiB. See the source scripts at github.com/mmatuska/mfsbsd.

6.2 MANAGING PACKAGES

UNIX and Linux software assets (source code, build files, documentation, and configuration templates) were traditionally distributed as compressed archives, usually gzipped tarballs (**.tar.gz** or **.tgz** files). This was OK for developers but inconvenient for end users and administrators. These source archives had to be manually compiled and built for each system on each release of the software, a tedious and error prone process.

Packaging systems emerged to simplify and facilitate the job of software management. Packages include all the files needed to run a piece of software, including precompiled binaries, dependency information, and configuration file templates that can be customized by administrators. Perhaps most importantly, packaging systems try to make the installation process as atomic as possible. If an error occurs during installation, a package can be backed out or reapplied. New versions of software can be installed with a simple package update.

Package installers are typically aware of configuration files and will not normally overwrite local customizations made by a system administrator. They either back up existing config files that they change or supply example config files under a different name. If you find that a newly installed package breaks something on your system, you can, theoretically, back it out to restore your system to its original state. Of course, theory != practice, so don't try this on a production system without testing it first.

Packaging systems define a dependency model that allows package maintainers to ensure that the libraries and support infrastructure on which their applications depend are properly installed. Unfortunately, the dependency graphs are sometimes imperfect. Unlucky administrators can find themselves in package dependency hell, a state where it's impossible to update a package because of version incompatibilities among its dependencies. Fortunately, recent versions of packaging software seem to be less susceptible to this effect.

Packages can run scripts at various points during the installation, so they can do much more than just disgorge new files. Packages frequently add new users and groups, run sanity checks, and customize settings according to the environment.

Confusingly, package versions do not always correspond directly to the versions of the software that they install. For example, consider the following RPM package for **docker-engine**:

```
$ rpm -qa | grep -i docker
docker-engine-1.13.0-1.el7.centos.x86_64
$ docker version | grep Version
Version:      1.13.1
```

The package itself claims version 1.13.0, but the **docker** binary reports version 1.13.1. In this case, the distribution maintainers backported changes and incremented the minor package

version. Be aware that the package version string is not necessarily an accurate indication of the software version that is actually installed.

You can create packages to facilitate the distribution of your own localizations or software. For example, you can create a package that, when installed, reads localization information for a machine (or gets it from a central database) and uses that information to set up local configuration files.

You can also bundle local applications as packages (complete with dependencies) or create packages for third party applications that aren't normally distributed in package format. You can version your packages and use the dependency mechanism to upgrade machines automatically when a new version of your localization package is released. We refer you to **fpm**, the Effing Package Manager, which is the easiest way to get started building packages for multiple platforms. You can find it at github.com/jordansissel/fpm.

You can also use the dependency mechanism to create groups of packages. For example, you can create a package that installs nothing of its own but depends on many other packages. Installing the package with dependencies turned on results in all the packages being installed in a single step.

6.3 LINUX PACKAGE MANAGEMENT SYSTEMS

Two package formats are in common use on Linux systems. Red Hat, CentOS, SUSE, Amazon Linux, and several other distributions use RPM, a recursive acronym that expands to "RPM Package Manager." Debian and Ubuntu use the separate but equally popular **.deb** format. The two formats are functionally similar.

Both the RPM and **.deb** packaging systems now function as dual-layer soup-to-nuts configuration management tools. At the lowest level are the tools that install, uninstall, and query packages: **rpm** for RPM and **dpkg** for **.deb**.

On top of these commands are systems that know how to find and download packages from the Internet, analyze interpackage dependencies, and upgrade all the packages on a system. **yum**, the Yellowdog Updater, Modified, works with the RPM system. APT, the Advanced Package Tool, originated in the **.deb** universe but works well with both **.deb** and RPM packages.

On the next couple of pages, we review the low-level commands **rpm** and **dpkg**. In the section *High-level Linux package management systems*, we discuss the comprehensive update systems APT and **yum**, which build on these low-level facilities. Your day-to-day administration activities will usually involve the high-level tools, but you'll occasionally need to wade into the deep end of the pool with **rpm** and **dpkg**.

rpm: manage RPM packages

RHEL  The **rpm** command installs, verifies, and queries the status of packages. It formerly built them as well, but this function has now been relegated to a separate command called **rpmbuild**. **rpm** options have complex interactions and can be used together only in certain combinations. It's most useful to think of **rpm** as if it were several different commands that happen to share the same name.

The mode you tell **rpm** to enter (such as **-i** or **-q**) specifies which of **rpm**'s multiple personalities you are hoping to access. **rpm --help** lists all the options broken down by mode, but it's worth your time to read the man page in some detail if you will frequently be dealing with RPM packages.

The bread-and-butter options are **-i** (install), **-U** (upgrade), **-e** (erase), and **-q** (query). The **-q** option is a bit tricky; you must supply an additional command-line flag to pose a specific question. For example, the command **rpm -qa** lists all the packages installed on the system.

Let's look at an example. We need to install a new version of OpenSSH because of a recent security fix. Once we've downloaded the package, we'll run **rpm -U** to replace the older version with the newer.

```
redhat$ sudo rpm -U openssh-6.6.1p1-33.el7_2.x86_64.rpm
error: failed dependencies:
openssh = 6.6.1p1-23 is needed by openssh-clients-6.6.1p1-23
openssh = 6.6.1p1-23 is needed by openssh-server-6.6.1p1-23
```

D'oh! Perhaps it's not so simple after all. Here we see that the currently installed version of OpenSSH, 6.6.1p1-23, is required by several other packages. **rpm** won't let us upgrade OpenSSH to 6.6.1p1-33 because the change might affect the operation of these other packages. This type of conflict happens all the time, and it's a major motivation for the development of systems like APT and **yum**. In real life we wouldn't attempt to untangle the dependencies by hand, but let's continue with **rpm** alone for the purpose of this example.

We could force the upgrade with the **--force** option, but that's usually a bad idea. The dependency information is there to save time and trouble, not just to get in the way. There's nothing like a broken SSH on a remote system to ruin a sysadmin's morning.

Instead, we'll grab updated versions of the dependent packages as well. If we were smart, we could have determined that other packages depended on OpenSSH before we even attempted the upgrade:

```
redhat$ rpm -q --whatrequires openssh
openssh-server-6.6.1p1-23.el7_2.x86_64
openssh-clients-6.6.1p1-23.el7_2.x86_64
```

Suppose that we've obtained updated copies of all the packages. We could install them one at a time, but **rpm** is smart enough to handle them all at once. If multiple RPMs are listed on the command line, **rpm** sorts them by dependency before installation.

```
redhat$ sudo rpm -U openssh-*
...
redhat$ rpm -q openssh
openssh-6.6.1p1-33.el7_3
```

Cool! Looks like it succeeded. Note that **rpm** understands which package we are talking about even though we didn't specify the package's full name or version. (Unfortunately, **rpm** does not restart **sshd** after the installation. You'd need to manually restart it to complete the upgrade.)

dpkg: manage .deb packages



Just as RPM packages have the all-in-one **rpm** command, Debian packages have the **dpkg** command. Useful options include **--install**, **--remove**, and **-l** to list the packages that have been installed on the system. A **dpkg --install** of a package that's already on the system removes the previous version before installing.

Running **dpkg -l | grep** *package* is a convenient way to determine if a particular package is installed. For example, to search for an HTTP server, try

```
ubuntu$ dpkg -l | grep -i http
ii lighttpd  1.4.35-4+deb8u1  amd64        fast webserver with minimal
    memory footprint
```

This search found the **lighttpd** software, an excellent, open source, lightweight web server. The leading ii indicates that the software is installed.

Suppose that the Ubuntu security team recently released a fix to **nvi** to patch a potential security problem. After grabbing the patch, we run **dpkg** to install it. As you can see, it's much chattier than **rpm** and tells us exactly what it's doing.

```
ubuntu$ sudo dpkg --install ./nvi_1.81.6-12_amd64.deb
(Reading database ... 24368 files and directories currently installed.)
Preparing to replace nvi 1.79-14 (using ./nvi_1.81.6-12_amd64.deb) ...
Unpacking replacement nvi ...
Setting up nvi (1.81.6-12) ...
Checking available versions of ex, updating links in /etc/alternatives ...
(You may modify the symlinks there yourself if desired - see 'man ln'.)
Leaving ex (/usr/bin/ex) pointing to /usr/bin/nex.
Leaving ex.1.gz (/usr/share/man/man1/ex.1.gz) pointing to /usr/share/
    man/man1/nex.1.gz.
...
```

We can now use **dpkg -l** to verify that the installation worked. The **-l** flag accepts an optional prefix pattern to match, so we can just search for **nvi**.

```
ubuntu$ dpkg -l nvi
    Name    Version         Description
ii  nvi     1.81.6-12       4.4BSD re-implementation of vi.
```

Our installation seems to have gone smoothly.

6.4 High-level Linux package management systems

Metapackage management systems such as APT and **yum** share several goals:

- To simplify the task of locating and downloading packages
- To automate the process of updating or upgrading systems
- To facilitate the management of interpackage dependencies

Clearly, these systems include more than just client-side commands. They all require that distribution maintainers organize their offerings in an agreed-on way so that the software can be accessed and reasoned about by clients.

Since no single supplier can encompass the entire "world of Linux software," the systems all allow for the existence of multiple software repositories. Repositories can be local to your network, so these systems make a dandy foundation for creating your own internal software distribution system.

RHEL The Red Hat Network is closely tied to Red Hat Enterprise Linux. It's a commercial service that costs money and offers more in terms of attractive GUIs, site-wide system management, and automation ability than do APT and **yum**. It is a shiny, hosted version of Red Hat's expensive and proprietary Satellite Server. The client side can reference **yum** and APT repositories, and this ability has allowed distributions such as CentOS to adapt the client GUI for nonproprietary use.

APT is better documented than the Red Hat Network, is significantly more portable, and is free. It's also more flexible in terms of what you can do with it. APT originated in the world of Debian and **dpkg**, but it has been extended to encompass RPMs, and versions that work with all our example distributions are available. It's the closest thing we have at this point to a universal standard for software distribution.

yum is an RPM-specific analog of APT. It's included by default on Red Hat Enterprise Linux and CentOS, although it runs on any RPM-based system, provided that you can point it toward appropriately formatted repositories.

We like APT and consider it a solid choice if you run Debian or Ubuntu and want to set up your own automated package distribution network. See the section *APT: the Advanced Package Tool* for more information.

Package repositories

Linux distributors maintain software repositories that work hand-in-hand with their chosen package management systems. The default configuration for the package management system usually points to one or more well-known web or FTP servers that are under the distributor's control.

However, it isn't immediately obvious what such repositories should contain. Should they include only the sets of packages blessed as formal, major releases? Formal releases plus current security updates? Up-to-date versions of all the packages that existed in the formal releases? Useful third party software not officially supported by the distributor? Source code? Binaries for multiple hardware architectures? When you run **apt upgrade** or **yum upgrade** to bring the system up to date, what exactly should that mean?

In general, package management systems must answer all these questions and must make it easy for sites to select the cross-sections they want to include in their software "world." The following concepts help structure this process.

- A "release" is a self-consistent snapshot of the package universe. Before the Internet era, named OS releases were more or less immutable and were associated with one specific time; security patches were made available separately. These days, a release is a more nebulous concept. Releases evolve over time as packages are updated. Some releases, such as Red Hat Enterprise Linux, are specifically designed to evolve slowly; by default, only security updates are incorporated. Other releases, such as beta versions, change frequently and dramatically. But in all cases, the release is the baseline, the target, the "thing I want to update my system to look like."

- A "component" is a subset of the software within a release. Distributions partition themselves differently, but one common distinction is that between core software blessed by the distributor and extra software made available by the broader community. Another distinction that's common in the Linux world is the one between the free, open source portions of a release and the parts that are tainted by some kind of restrictive licensing agreement.

  Of particular note from an administrative standpoint are minimally active components that include only security fixes. Some releases allow you to combine a security component with an immutable baseline component to create a relatively stable version of the distribution, even though the mainline distribution may evolve much faster.

- An "architecture" represents a class of hardware. The expectation is that machines within an architecture class are similar enough that they can all run the same binaries. Architectures are instances of releases, for example, "Ubuntu Xenial Xerus for

x86_64." Since components are subdivisions of releases, there's a corresponding architecture-specific instance for each of them as well.

- Individual packages are the elements that make up components, and therefore, indirectly, releases. Packages are usually architecture-specific and are versioned independently of the main release and of other packages. The correspondence between packages and releases is implicit in the way the network repository is set up.

The existence of components that aren't maintained by the distributor (e.g., Ubuntu's "universe" and "multiverse") raises the question of how these components relate to the core OS release. Can they really be said to be "a component" of the specific release, or are they some other kind of animal entirely?

From a package management perspective, the answer is clear: extras are a true component. They are associated with a specific release, and they evolve in tandem with it. The separation of control is interesting from an administrative standpoint, but it doesn't affect the package distribution systems, except that multiple repositories might need to be manually added by the administrator.

RHN: the Red Hat Network

RHEL With Red Hat having gracefully departed from the consumer Linux business, the Red Hat Network has become the system management platform for Red Hat Enterprise Linux. You purchase the right to access the Red Hat Network by subscribing. At its simplest, you can use the Red Hat Network as a glorified web portal and mailing list. Used in this way, the Red Hat Network is not much different from the patch notification mailing lists that have been run by various UNIX vendors for years. But more features are available if you're willing to pay for them. For current pricing and features, see rhn.redhat.com.

The Red Hat Network presents a web-based interface for downloading new packages as well as a command-line alternative. Once you register, your machines get all the patches and bug fixes that they need without your ever having to intervene.

The downside of automatic registration is that Red Hat decides what updates you need. You might consider how much you really trust Red Hat (and the software maintainers whose products they package) not to screw things up.

A reasonable compromise might be to sign up one machine in your organization for automatic updates. You can take snapshots from that machine at periodic intervals to test as possible candidates for internal releases.

APT: the Advanced Package Tool

APT is one of the most mature package management systems. It's possible to upgrade an entire system full of software with a single **apt** command or even (as with the Red Hat Network) to have your boxes continuously keep themselves up to date without human intervention.

The first rule of using APT on Ubuntu systems (and indeed all management of Debian packages) is to ignore the existence of **dselect**, which acts as a front end for the Debian package system. It's not a bad idea, but the user interface is poor and can be intimidating to the novice user. Some documentation will try to steer you toward **dselect**, but stay strong and stick with **apt**.

If you are using APT to manage a stock Ubuntu installation from a standard repository mirror, the easiest way to see the available packages is to visit the master list at packages.ubuntu.com. The web site includes a nice search interface. If you set up your own APT server (see this page), then of course you will know what packages you have made available and you can list them in whatever way you want.

Distributions commonly include dummy packages that exist only to claim other packages as prerequisites. **apt** downloads and upgrades prerequisite packages as needed, so the dummy packages make it easy to install or upgrade several packages as a block. For example, installing the **gnome-desktop-environment** package obtains and installs all the packages necessary to run the GNOME UI.

APT includes a suite of low-level commands like **apt-get** and **apt-cache** that are wrapped for most purposes by an omnibus **apt** command. The wrapper is a later addition to the system, so you'll still see occasional references to the low-level commands on the web and in documentation. To a first approximation, commands that look similar are in fact the same command. There's no difference between **apt install** and **apt-get install**, for example.

Once you have set up your **/etc/apt/sources.list** file (described in detail below) and know the name of a package that you want, the only remaining task is to run **apt update** to refresh **apt**'s cache of package information. After that, just run **apt install** *package-name* as a privileged user to install the package. The same command updates a package that has already been installed.

Suppose you want to install a new version of the **sudo** package that fixes a security bug. First, it's always wise to do an **apt update**:

```
debian$ sudo apt update
Get:1 http://http.us.debian.org stable/main Packages [824kB]
Get:2 http://non-us.debian.org stable/non-US/main Release [102B]
...
```

Now you can actually fetch the package. Note the use of **sudo** to fetch the new **sudo** package —**apt** can even upgrade packages that are in use!

```
debian$ sudo apt install sudo
Reading Package Lists... Done
Building Dependency Tree... Done
1 packages upgraded, 0 newly installed, 0 to remove and 191 not upgraded.
Need to get 0B/122kB of archives. After unpacking 131kB will be used.
(Reading database ... 24359 files and directories currently installed.)
Preparing to replace sudo 1.6.2p2-2 (using .../sudo_1.8.10p3-1+deb8u3_
    amd64.deb) ...
Unpacking replacement sudo ...
Setting up sudo (1.8.10p3-1+deb8u3) ...
Installing new version of config file /etc/pam.d/sudo ...
```

Repository configuration

Configuring APT is straightforward; pretty much everything you need to know can be found in Ubuntu's community documentation on package management*:*

help.ubuntu.com/community/AptGet/Howto

The most important configuration file is **/etc/apt/sources.list**, which tells APT where to get its packages. Each line specifies the following:

- A type of package, currently `deb` or `deb-src` for Debian-style packages or `rpm` or `rpm-src` for RPMs
- A URL that points to a file, HTTP server, or FTP server from which to fetch packages
- A "distribution" (really, a release name) that lets you deliver multiple versions of packages
- A potential list of components (categories of packages within a release)

Unless you want to set up your own APT repository or cache, the default configuration generally works fine. Source packages are downloaded from the entries beginning with `deb-src`.

On Ubuntu systems, you'll almost certainly want to include the "universe" component, which accesses the larger world of Linux open source software. The "multiverse" packages include non-open-source content, such as some VMware tools and components.

As long as you're editing the **sources.list** file, you may want to retarget the individual entries to point to your closest mirror. A full list of Ubuntu mirrors can be found at launchpad.net/ubuntu/+archivemirrors. This is a dynamic (and long) list of mirrors that changes regularly, so be sure to keep an eye on it between releases.

Make sure that security.ubuntu.com is listed as a source so that you have access to the latest security patches.

An example /etc/apt/sources.list file

The following example uses archive.ubuntu.com as a package source for the “main” components of Ubuntu (those that are fully supported by the Ubuntu team). In addition, this **sources.list** file includes unsupported but open source “universe” packages, and non-free, unsupported packages in the “multiverse” component. There is also a repository for updates or bug-fixed packages in each component. Finally, the last six lines are for security updates.

```
# General format: type uri distribution [ components ]
deb http://archive.ubuntu.com/ubuntu xenial main restricted
deb-src http://archive.ubuntu.com/ubuntu xenial main restricted
deb http://archive.ubuntu.com/ubuntu xenial-updates main restricted
deb-src http://archive.ubuntu.com/ubuntu xenial-updates main restricted
deb http://archive.ubuntu.com/ubuntu xenial universe
deb-src http://archive.ubuntu.com/ubuntu xenial universe
deb http://archive.ubuntu.com/ubuntu xenial-updates universe
deb-src http://archive.ubuntu.com/ubuntu xenial-updates universe
deb http://archive.ubuntu.com/ubuntu xenial multiverse
deb-src http://archive.ubuntu.com/ubuntu xenial multiverse
deb http://archive.ubuntu.com/ubuntu xenial-updates multiverse
deb-src http://archive.ubuntu.com/ubuntu xenial-updates multiverse
deb http://archive.ubuntu.com/ubuntu xenial-backports main restricted
    universe multiverse
deb-src http://archive.ubuntu.com/ubuntu xenial-backports main restricted
    universe multiverse
deb http://security.ubuntu.com/ubuntu xenial-security main restricted
deb-src http://security.ubuntu.com/ubuntu xenial-security main restricted
deb http://security.ubuntu.com/ubuntu xenial-security universe
deb-src http://security.ubuntu.com/ubuntu xenial-security universe
deb http://security.ubuntu.com/ubuntu xenial-security multiverse
deb-src http://security.ubuntu.com/ubuntu xenial-security multiverse
```

The *distribution* and *components* fields help APT navigate the filesystem hierarchy of the Ubuntu repository, which has a standardized layout. The root distribution is the working title given to each release, such as trusty, xenial, or yakkety. The available components are typically called main, universe, multiverse, and restricted. Add the universe and multiverse repositories only if you are comfortable having unsupported (and license-restricted, in the case of multiverse) software in your environment.

After you update the **sources.list** file, run **apt-get update** to force APT to react to your changes.

Creation of a local repository mirror

If you plan to use **apt** on a large number of machines, you will probably want to cache packages locally. Downloading a copy of each package for every machine is not a sensible use of external bandwidth. A mirror of the repository is easy to configure and convenient for local administration. Just make sure to keep it updated with the latest security patches.

The best tool for the job is the handy **apt-mirror** package, which is available from apt-mirror.github.io. You can also install the package from the universe component with **sudo apt install apt-mirror**.

Once installed, **apt-mirror** drops a file called **mirror.list** in **/etc/apt**. It's a shadow version of **sources.list**, but it's used only as a source for mirroring operations. By default, **mirror.list** conveniently contains all the repositories for the running version of Ubuntu.

To actually mirror the repositories in **mirror.list**, just run **apt-mirror** as root:

```
ubuntu$ sudo apt-mirror
Downloading 162 index files using 20 threads...
Begin time: Sun Feb  5 22:34:58 2017
[20]... [19]... [18]... [17]... [16]... [15]... [14]...
```

By default, **apt-mirror** puts its repository copies in **/var/spool/apt-mirror**. Feel free to change this by uncommenting the set base_path directive in **mirror.list**, but be aware that you must then create **mirror**, **skel**, and **var** subdirectories under the new mirror root.

apt-mirror takes a long time to run on its first pass because it is mirroring many gigabytes of data (currently ~40GB per Ubuntu release). Subsequent executions are faster and should be run automatically out of **cron**. You can run the **clean.sh** script from the **var** subdirectory of your mirror to clean out obsolete files.

To start using your mirror, share the base directory through HTTP, using a web server of your choice. We like to use symbolic links to the web root. For instance:

```
ln -s /var/spool/apt-mirror/us.archive.ubuntu.com/ubuntu /var/www/ubuntu
```

To make clients use your local mirror, edit their **sources.list** files just as if you were selecting a nonlocal mirror.

APT automation

Use **cron** to schedule regular **apt** runs. Even if you don't install packages automatically, you may want to run **apt update** regularly to keep your package summaries up to date.

apt upgrade downloads and installs new versions of any packages that are currently installed on the local machine. Note that **apt upgrade** is defined slightly differently from the low-level command **apt-get upgrade**, but **apt upgrade** is usually what you want. (It's equivalent to **apt-get dist-upgrade --with-new-pkgs**.) **apt upgrade** might want to delete some packages that it views as irreconcilably incompatible with the upgraded system, so be prepared for potential surprises.

If you really want to play with fire, have machines perform the upgrade in an unattended fashion by including the **-y** option to **apt upgrade**. It answers any confirmation questions that **apt** might ask with an enthusiastic "Yes!" Be aware that some updates, such as kernel packages, might not take effect until after a system reboot.

It's probably not a good idea to perform automated upgrades directly from a distribution's mirror. However, in concert with your own APT servers, packages, and release control system, this is a perfect way to keep clients in sync. A one-liner like the following keeps a box up to date with its APT server.

```
# apt update && apt upgrade -y
```

Use this command in a **cron** job if you want it to run on a regular schedule. You can also refer to it from a system startup script to make the machine update at boot time. See this page for more information about **cron**; see Chapter 2, *Booting and System Management Daemons*, for more information about startup scripts.

If you run updates out of **cron** on many machines, it's a good idea to use time randomization to make sure that everyone doesn't try to update at once.

If you don't quite trust your source of packages, consider automatically downloading all changed packages without installing them. Use **apt**'s **--download-only** option to request this behavior, then review the packages by hand and install the ones you want to update. Downloaded packages are put in **/var/cache/apt**, and over time this directory can grow to be quite large. Clean out the unused files from this directory with **apt-get autoclean**.

yum: release management for RPM

yum, the Yellowdog Updater, Modified, is a metapackage manager based on RPM. It may be a bit unfair to call **yum** an APT clone, but it's thematically and implementationally similar, although cleaner and slower in practice.

On the server-side, the **yum-arch** command compiles a database of header information from a large set of packages (often an entire release). The header database is then shared along with the packages through HTTP. Clients use the **yum** command to fetch and install packages; **yum** figures out dependency constraints and does whatever additional work is needed to complete the installation of the requested packages. If a requested package depends on other packages, **yum** downloads and installs those packages as well.

The similarities between **apt** and **yum** extend to the command-line options they understand. For example, **yum install foo** downloads and installs the most recent version of the foo package (and its dependencies, if necessary). There is at least one treacherous difference, though: **apt update** refreshes **apt**'s package information cache, but **yum update** updates every package on the system (it's analogous to **apt upgrade**). To add to the confusion, **yum upgrade** is the same as **yum update** but with obsolescence processing enabled.

yum does not match on partial package names unless you include globbing characters (such as * and ?) to explicitly request this behavior. For example, **yum update 'lib*'** refreshes all packages whose name starts with "lib". Remember to quote the globbing characters so the shell doesn't interfere with them.

Unlike **apt**, **yum** defaults to validating its package information cache against the contents of the network repository every time you run it. Use the **-C** option to prevent the validation and **yum makecache** to update the local cache (it takes awhile to run). Unfortunately, the **-C** option doesn't do much to improve **yum**'s sluggish performance.

yum's configuration file is **/etc/yum.conf**. It includes general options and pointers to package repositories. Multiple repositories can be active at once, and each repository can be associated with multiple URLs.

A replacement for **yum** called DNF (for Dandified Yum) is under active development. It's already the default package manager for Fedora and will eventually replace **yum** completely. DNF sports better dependency resolution and an improved API, among other features. Visit dnf.baseurl.org to learn more.

6.5 FreeBSD software management



FreeBSD has had packaging facilities for several releases, but it's only now transitioning to a completely package-centric distribution model in which most elements of the core OS are defined as packages. FreeBSD's recent releases have segregated software into three general categories:

- A "base system," which includes a bundled set of core software and utilities
- A set of binary packages managed with the **pkg** command
- A separate "ports" system which downloads source code, applies FreeBSD-specific patches, then builds and installs it

As of FreeBSD 11, the lines between these territories have become even more muddled. The base system has been packagized, but the old scheme for managing the base system as one unit is still in place, too. Many software packages can be installed either as binary packages or as ports, with essentially similar results but different implications for future updates. However, cross-coverage is not complete; some things can only be installed as a port or as a package.

Part of the project definition for FreeBSD 12 is to shift the system more decisively toward universal package management. The base system and ports may both continue to exist in some form (it's currently too early to tell exactly how things will work out), but the future direction is clear.

Accordingly, try to manage add-on software with **pkg** to the extent possible. Avoid ports unless the software you want has no packagized version or you need to customize compile-time options.

Another peculiar remnant of the big-iron UNIX era is FreeBSD's insistence that add-on packages are "local," even though they are compiled by FreeBSD and released as part of an official package repository. Packages install binaries under **/usr/local**, and most configuration files end up in **/usr/local/etc** rather than **/etc**.

The base system

The base system is updated as a single unit and is functionally distinct from any add-on packages (at least in theory). The base system is maintained in a Subversion repository. You can browse the source tree, including all the source branches, at svnweb.freebsd.org.

Several development branches are defined:

- The CURRENT branch is meant only for active development purposes. It is the first to receive new features and fixes but is not widely tested by the user community.
- The STABLE branch is regularly updated with improvements intended for the next major release. It includes new features but maintains package compatibility and undergoes some testing. It may introduce bugs or breaking changes and is recommended only for the adventurous.
- The RELEASE branch is forked from STABLE when a release target is achieved. It remains mostly static. The only updates to RELEASE are security fixes and fixes for serious bugs. Official ISO images derive from the RELEASE branch, and that branch is the only one recommended for use on production systems.

View your system's current branch with **uname -r**.

```
$ uname -r
11.0-RELEASE
```

Run the **freebsd-update** command to keep your system updated with the latest packages. Fetching updates and installing them are separate operations, but you can combine the two into a single command line:

```
$ sudo freebsd-update fetch install
```

This command retrieves and installs the latest base binaries. It's available only for the RELEASE branch; binaries are not built for the STABLE and CURRENT branches. You can use the same tool to upgrade between releases of the system. For example:

```
$ sudo freebsd-update -r 11.1-RELEASE upgrade
```

pkg: the FreeBSD package manager

pkg is intuitive and fast. It's the easiest way to install software that isn't already included in the base system. Use **pkg help** for a quick reference on the available subcommands, or **pkg help** *command* to display the man page for a particular subcommand. Table 6.2 lists some of the most frequently used subcommands.

Table 6.2: Example pkg subcommands

Command	What it does
pkg install -y *package*	Installs without asking any "are you sure?" questions
pkg backup	Makes a backup of the local package database
pkg info	Lists all installed packages
pkg info *package*	Shows extended information for a package
pkg search -i *package*	Searches package repository (case insensitive)
pkg audit -F	Shows packages with known security vulnerabilities
pkg which *file*	Shows which package owns the named *file*
pkg autoremove	Removes unused packages
pkg delete *package*	Uninstalls a package (same as **remove**)
pkg clean -ay	Removes cached packages from **/var/cache/pkg**
pkg update	Updates local copy of the package catalog
pkg upgrade	Upgrades packages to the latest version

When you install packages with **pkg install**, **pkg** consults the local package catalog, then downloads the requested package from the repository at pkg.FreeBSD.org. Once the package is installed, it's registered in a SQLite database kept in **/var/db/pkg/local.sqlite**. Take care not to delete this file lest your system lose track of which packages have been installed. Create backups of the database with the **pkg backup** subcommand.

pkg version, a subcommand for comparing package versions, has an idiosyncratic syntax. It uses the =, <, and > characters to show packages that are current, older than the latest available version, or newer than the current version. Use the following command to list packages that have updates:

```
freebsd$ pkg version -vIL=
dri-11.2.2,2                    <   needs updating (index has 13.0.4,2)
gbm-11.2.2                      <   needs updating (index has 13.0.4)
harfbuzz-1.4.1                  <   needs updating (index has 1.4.2)
libEGL-11.2.2                   <   needs updating (index has 13.0.4_1)
```

This command compares all installed packages to the index (**-I**), looking for those that are not (**-L**) the current version (=), and printing verbose information (**-v**).

pkg search is faster than Google for finding packages. For example, **pkg search dns** finds all packages with “dns” in their names. The search term is a regular expression, so you can search for something like **pkg search ^apache**. See **pkg help search** for details.

The ports collection

FreeBSD ports are a collection of all the software that FreeBSD can build from source. After the ports tree is initialized, you'll find all the available software in categorized subdirectories of **/usr/ports**. To initialize the ports tree, use the **portsnap** utility:

```
freebsd$ portsnap fetch extract
```

To update the ports tree in one command, use **portsnap fetch update**.

It takes some time to download the ports metadata. The download includes pointers to the source code for all the ports, plus any associated patches for FreeBSD compatibility. When installation of the metadata is complete, you can search for software, then build and install anything you need.

For example, the **zsh** shell is not included in the FreeBSD base. Use the **whereis** utility to search for **zsh**, then build and install from the ports tree:

```
freebsd$ whereis zsh
bash: /usr/ports/shells/zsh
freebsd$ cd /usr/ports/shells/zsh
freebsd$ make install clean
```

To remove software installed through the ports system, run **make deinstall** from the appropriate directory.

There's more than one way to update ports, but we prefer the **portmaster** utility. First install **portmaster** from the ports collection:

```
freebsd$ cd /usr/ports/ports-mgmt/portmaster
freebsd$ make install clean
```

Run **portmaster -L** to see all the ports having updates available, and update them all at once with **portmaster -a**.

You can also install ports through the **portmaster**. In fact, it's somewhat more convenient than the typical **make**-based process because you don't need to leave your current directory. To install **zsh**:

```
freebsd$ portmaster shells/zsh
```

If you need to free up some disk space, clean up the ports' working directories with **portmaster -c**.

6.6 Software localization and configuration

Adapting systems to your local (or cloud) environment is one of the prime battlegrounds of system administration. Addressing localization issues in a structured and reproducible way helps avoid the creation of snowflake systems that are impossible to recover after a major incident.

We have more to say in this book about these issues. In particular, Chapter 23, *Configuration Management*, and Chapter 26, *Continuous Integration and Delivery*, discuss tools that structure these tasks. Configuration management systems are your go-to tools for installing and configuring software in a reproducible manner. They are the master key to sane localization.

Implementation issues aside, how do you know if your local environment is properly designed? Here are a few points to consider:

- Nonadministrators should not have root privileges. Any need for root privileges in the course of normal operations is suspicious and probably indicates that something is fishy with your local configuration.
- Systems should facilitate work and not get in users' way. Users do not wreck the system intentionally. Design internal security so that it guards against unintentional errors and the widespread dissemination of administrative privileges.
- Misbehaving users are learning opportunities. Interview them before you chastise them for not following proper procedures. Users frequently respond to inefficient administrative procedures by working around them, so always consider the possibility that noncompliance is an indication of architectural problems.
- Be customer-centered. Talk to users and ask them which tasks they find difficult in your current configuration. Find ways to make these tasks simpler.
- Your personal preferences are yours. Let your users have their own. Offer choices wherever possible.
- When administrative decisions affect users' experience of the system, be aware of the reasons for your decisions. Let your reasons be known.
- Keep your local documentation up to date and easily accessible. See this page for more information on this topic.

Organizing your localization

If your site has a thousand computers and each computer has its own configuration, you will spend a major portion of your working time figuring out why one box has a particular problem and another doesn't. Clearly, the solution is to make every computer the same…right? But real-world constraints and the varying needs of users typically make this solution impossible.

There's a big difference in administrability between *multiple* configurations and *countless* configurations. The trick is to split your setup into manageable bits. Some parts of the localization apply to all managed hosts, others apply to only a few, and still others are specific to individual boxes. Even with the convenience of configuration management tools, try not to allow too much drift among systems.

However you design your localization system, make sure that all original data is kept in a revision control system. This precaution lets you keep track of which changes have been thoroughly tested and are ready for deployment. In addition, it lets you identify the originator of any problematic changes. The more people involved in the process, the more important this last consideration becomes.

Structuring updates

In addition to performing initial installations, you will also need to continually roll out updates. This remains one of the most important security tasks. Keep in mind, though, that different hosts have different needs for concurrency, stability, and uptime.

Do not roll out new software releases en masse. Instead, stage rollouts according to a gradual plan that accommodates other groups' needs and allows time for problems to be discovered while their potential to cause damage is still limited. This sometimes referred to as a "canary" release process, named for the fabled "canary in the coal mine." In addition, never update critical servers until you have some confidence in the changes you are contemplating. Avoid rolling out changes on Fridays unless you're prepared for a long weekend in front of the terminal.

It's usually advantageous to separate the base OS release from the localization release. Depending on the stability needs of your environment, you might choose to use minor local releases only for bug fixing. However, we have found that adding new features in small doses yields a smoother operation than queuing up changes into "horse pill" releases that risk a major disruption of service. This principle is closely related to the idea of continuous integration and deployment; see Chapter 26.

Limiting the field of play

It's often a good idea to specify a maximum number of "releases" you are willing to have in play at any given time. Some administrators see no reason to fix software that isn't broken. They point out that gratuitously upgrading systems costs time and money and that "cutting edge" all too often means "bleeding edge." Those who put these principles into practice must be willing to collect an extensive catalog of active releases.

By contrast, the "lean and mean" crowd point to the inherent complexity of releases and the difficulty of comprehending (let alone managing) a random collection of releases dating years into the past. Their trump cards are security patches, which must typically be applied universally and on a strict schedule. Patching outdated versions of the operating system is often infeasible, so administrators are faced with the choice of skipping updates on some computers or crash-upgrading these machines to a newer internal release. Not good.

Neither of these perspectives is provably correct, but we tend to side with those who favor a limited number of releases. Better to perform your upgrades on your own schedule rather than one dictated by an external emergency.

Testing

It's important to test changes before unleashing them on the world. At a minimum, this means that you need to test your own local configuration changes. However, you should really test the software that your vendor releases as well. A major UNIX vendor once released a patch that performed an **rm -rf /**. Imagine installing this patch throughout your organization without testing it first.

Testing is an especially pertinent issue if you use a service that offers an automatic patching capability, such as most of the packaging systems discussed in this chapter. Never connect mission-critical systems directly to a vendor-sponsored update service. Instead, point most of your systems to an internal mirror that you control, and test updates on noncritical systems first.

See this page for more information about trouble tracking.

If you foresee that an update might cause user-visible problems or changes, notify users well in advance and give them a chance to communicate with you if they have concerns regarding your intended changes or timing. Make sure that users have an easy way to report bugs.

If your organization is geographically distributed, make sure that other offices help with testing. International participation is particularly valuable in multilingual environments. If no one in the U.S. office speaks Japanese, for example, you had better get the Tokyo office to test anything that might affect Unicode support. A surprising number of system parameters vary with location. Does the new version of software you're installing break UTF-8 encoding, rendering text illegible for some languages?

6.7 Recommended reading

Intel Corporation and SystemSoft. *Preboot Execution Environment (PXE) Specification, v2.1*. 1999. pix.net/software/pxeboot/archive/pxespec.pdf

Lawson, Nolan. *What it feels like to be an open-source maintainer*. wp.me/p1t8Ca-1ry

PXELinux Questions. syslinux.zytor.com/wiki/index.php/PXELINUX

Rodin, Josip. *Debian New Maintainers' Guide*. debian.org/doc/maint-guide
This document contains good information about **.deb** packages. See also Chapter 7 of the Debian FAQ and Chapter 2 of the Debian reference manual.

7 Scripting and the Shell



A scalable approach to system management requires that administrative changes be structured, reproducible, and replicable across multiple computers. In the real world, that means those changes should be mediated by software rather than performed by administrators working from checklists—or worse, from memory.

Scripts standardize administrative chores and free up admins' time for more important and more interesting tasks. Scripts also serve as a kind of low-rent documentation in that they record the steps needed to complete a particular task.

Sysadmins' main alternative to scripting is to use the configuration management systems described in Chapter 23. These systems offer a structured approach to administration that scales well to the cloud and to networks of machines. However, they are more complex, more formal, and less flexible than plain-vanilla scripting. In practice, most administrators use a combination of scripting and configuration management. Each approach has its strengths, and they work well together.

This chapter takes a quick look at **sh**, Python, and Ruby as languages for scripting. We cover some basic tips for using the shell and also discuss regular expressions as a general technology.

7.1 Scripting philosophy

This chapter includes a variety of scripting tidbits and language particulars. That information is useful, but more important than any of those details is the broader question of how to incorporate scripting (or more generally, automation) into your mental model of system administration.

Write microscripts

New sysadmins often wait to learn scripting until they're confronted with a particularly complex or tedious chore. For example, maybe it's necessary to automate a particular type of backup so that it's done regularly and so that the backup data is stored in two different data centers. Or perhaps there's a cloud server configuration that would be helpful to create, initialize, and deploy with a single command.

These are perfectly legitimate scripting projects, but they can leave the impression that scripting is an elephant gun to be unboxed only when big game is on the horizon. After all, that first 100-line script probably took several days to write and debug. You can't be spending days on every little task…can you?

Actually, you achieve most efficiencies by saving a few keystrokes here and a few commands there. Marquee-level scripts that are part of your site's formal procedures are just the visible portion of a much larger iceberg. Below the waterline lie many smaller forms of automation that are equally useful for sysadmins. As a general rule, approach every chore with the question, "How can I avoid having to deal with this issue again in the future?"

Most admins keep a selection of short scripts for personal use (aka scriptlets) in their **~/bin** directories. Use these quick-and-dirty scripts to address the pain points you encounter in day-to-day work. They are usually short enough to read at a glance, so they don't need documentation beyond a simple usage message. Keep them updated as your needs change.

For shell scripts, you also have the option of defining functions that live inside your shell configuration files (e.g., **.bash_profile**) rather than in freestanding script files. Shell functions work similarly to stand-alone scripts, but they are independent of your search path and automatically travel with you wherever you take your shell environment.

Just as a quick illustration, here's a simple Bash function that backs up files according to a standardized naming convention:

```
function backup () {
    newname=$1.`date +%Y-%m-%d.%H%M.bak`;
    mv $1 $newname;
    echo "Backed up $1 to $newname.";
    cp -p $newname $1;
}
```

Despite the function-like syntax, you use it just like a script or any other command:

```
$ backup afile
Backed up afile to afile.2017-02-05.1454.bak.
```

The main disadvantage of shell functions is that they're stored in memory and have to be reparsed every time you start a new shell. But on modern hardware, these costs are negligible.

At a smaller scale still are aliases, which are really just an extra-short variety of scriptlet. These can be defined either with shell functions or with your shell's built-in aliasing feature (usually called alias). Most commonly, they set default arguments for individual commands. For example,

```
alias ls='ls -Fh'
```

makes the **ls** command punctuate the names of directories and executables and requests human-readable file sizes for long listings (e.g., 2.4M).

Learn a few tools well

System administrators encounter a lot of software. They can't be experts at everything, so they usually become skilled at skimming documentation, running experiments, and learning just enough about new software packages to configure them for the local environment. Laziness is a virtue.

That said, some topics are valuable to study in detail because they amplify your power and effectiveness. In particular, you should know a shell, a text editor, and a scripting language thoroughly. (Not to spoil the rest of this chapter, but these should probably be Bash, **vim**, and Python.) Read the manuals from front to back, then regularly read books and blogs. There's always more to learn.

Enabling technologies like these reward up-front study for a couple of reasons. As tools, they are fairly abstract; it's hard to envision all the things they're capable of doing without reading about the details. You can't use features you're not aware of.

Another reason these tools reward exploration is that they're "made of meat"; most features are potentially valuable to most administrators. Compare that with the average server daemon, where your main challenge is often to identify the 80% of features that are irrelevant to your situation.

A shell or editor is a tool you use constantly. Every incremental improvement in your proficiency with these tools translates not only into increased productivity but also into greater enjoyment of the work. No one likes to waste time on repetitive details.

Automate all the things

Shell scripts aren't system administrators' only opportunity to benefit from automation. There's a whole world of programmable systems out there—just keep an eye out for them. Exploit these facilities aggressively and use them to impedance-match your tools to your workflow.

For example, we created this book in Adobe InDesign, which is ostensibly a GUI application. However, it's also scriptable in JavaScript, so we created a library of InDesign scripts to implement and enforce many of our conventions.

Such opportunities are everywhere:

- Microsoft Office apps are programmable in Visual Basic or C#. If your work involves analysis or reporting, make those TPS reports write themselves.
- Most Adobe applications are scriptable.
- If your responsibilities include database wrangling, you can automate many routine tasks with SQL stored procedures. Some databases even support additional languages; for example, PostgreSQL speaks Python.
- PowerShell is the mainstream scripting tool for Microsoft Windows systems. Third party add-ons like AutoHotKey go a long way toward facilitating the automation of Windows apps.
- On macOS systems, some applications can be controlled through AppleScript. At the system level, use the Automator app, the Services system, and folder actions to automate various chores and to connect traditional scripting languages to the GUI.

Within the world of system administration specifically, a few subsystems have their own approaches to automation. Many others play well with general-purpose automation systems such as Ansible, Salt, Chef, and Puppet, described in Chapter 23, *Configuration Management*. For everything else, there's general-purpose scripting.

Don’t optimize prematurely

There’s no real distinction between “scripting” and “programming.” Language developers sometimes take offense when their babies are lumped into the “scripting” category, not just because the label suggests a certain lack of completeness, but also because some scripting languages of the past have earned reputations for poor design.

We still like the term “scripting,” though; it evokes the use of software as a kind of universal glue that binds various commands, libraries, and configuration files into a more functional whole.

Administrative scripts should emphasize programmer efficiency and code clarity rather than computational efficiency. This is not an excuse to be sloppy, but simply a recognition that it rarely matters whether a script runs in half a second or two seconds. Optimization can have an amazingly low return on investment, even for scripts that run regularly out of **cron**.

Pick the right scripting language

For a long time, the standard language for administrative scripts was the one defined by the **sh** shell. Shell scripts are typically used for light tasks such as automating a sequence of commands or assembling several filters to process data.

The shell is always available, so shell scripts are relatively portable and have few dependencies other than the commands they invoke. Whether or not you choose the shell, the shell might choose you: most environments include a hefty complement of existing **sh** scripts, and administrators frequently need to read, understand, and tweak those scripts.

As a programming language, **sh** is somewhat inelegant. The syntax is idiosyncratic, and the shell lacks the advanced text processing features of modern languages—features that are often of particular use to system administrators.

Perl, designed in the late 1980s, was a major step forward for script-writing administrators. Its permissive syntax, extensive library of user-written modules, and built-in support of regular expressions made it an administrative favorite for many years. Perl permits (and some would say, encourages) a certain "get it done and damn the torpedoes" style of coding. Opinions differ on whether that's an advantage or a drawback.

These days, Perl is known as Perl 5 to distinguish it from the redesigned and incompatible Perl 6, which has finally reached general release after 15 years of gestation. Unfortunately, Perl 5 is showing its age in comparison with newer languages, and use of Perl 6 isn't yet widespread enough for us to recommend it as a safe choice. It might be that the world has moved on from Perl entirely. We suggest avoiding Perl for new work at this point.

JavaScript and PHP are best known as languages for web development, but they can be arm-twisted into service as general-purpose scripting tools, too. Unfortunately, both languages have design flaws that limit their appeal, and they lack many of the third party libraries that system administrators rely on.

If you come from the web development world, you might be tempted to apply your existing PHP or JavaScript skills to system administration. We recommend against this. Code is code, but living in the same ecosystem as other sysadmins brings a variety of long-term benefits. (At the very least, avoiding PHP means you won't have to endure the ridicule of your local sysadmin Meetup.)

Python and Ruby are modern, general-purpose programming languages that are both well suited for administrative work. These languages incorporate a couple of decades' worth of language design advancements relative to the shell, and their text processing facilities are so powerful that **sh** can only weep and cower in shame.

The main drawback to both Python and Ruby is that their environments can be a bit fussy to set up, especially when you start to use third party libraries that have compiled components written

in C. The shell skirts this particular issue by having no module structure and no third party libraries.

In the absence of outside constraints, Python is the most broadly useful scripting language for system administrators. It's well designed, widely used, and widely supported by other packages. Table 7.1 shows some general notes on other languages.

Table 7.1: Scripting language cheat sheet

Language	Designer	When to use it
Bourne shell	Stephen Bourne	Simple series of commands, portable scripts
bash	Brian Fox	Like Bourne shell; nicer but less portable
C shell	Bill Joy	Never for scripting
JavaScript	Brendan Eich	Web development, app scripting
Perl	Larry Wall	Quick hacks, one-liners, text processing
PHP	Rasmus Lerdorf	You've been bad and deserve punishment
Python	Guido van Rossum	General-purpose scripting, data wrangling
Ruby	"Matz" Matsumoto	General-purpose scripting, web

Follow best practices

Although the code fragments in this chapter contain few comments and seldom print usage messages, that's only because we've skeletonized each example to make specific points. Real scripts should behave better. There are whole books on best practices for coding, but here are a few basic guidelines:

- When run with inappropriate arguments, scripts should print a usage message and exit. For extra credit, implement --**help** this way, too.
- Validate inputs and sanity-check derived values. Before doing an **rm -rf** on a calculated path, for example, you might have the script double-check that the path conforms to the pattern you expect.
- Return a meaningful exit code: zero for success and nonzero for failure. You needn't necessarily give every failure mode a unique exit code, however; consider what callers will actually want to know.
- Use appropriate naming conventions for variables, scripts, and routines. Conform to the conventions of the language, the rest of your site's code base, and most importantly, the other variables and functions defined in the current project. Use mixed case or underscores to make long names readable. The naming of the scripts themselves is important, too. In this context, dashes are more common than underscores for simulating spaces, as in **system-config-printer**.
- Assign variable names that reflect the values they store, but keep them short. `number_of_lines_of_input` is way too long; try `n_lines`.
- Consider developing a style guide so you and your colleagues can write code according to the same conventions. A guide makes it easier for you to read other people's code and for them to read yours. (On the other hand, style guide construction can absorb a contentious team's attention for weeks. Don't fight over the style guide; cover the areas of agreement and avoid long negotiations over the placement of braces and commas. The main thing is to make sure everyone's on board with a consistent set of naming conventions.)
- Start every script with a comment block that tells what the script does and what parameters it takes. Include your name and the date. If the script requires nonstandard tools, libraries, or modules to be installed on the system, list those as well.
- Comment at the level you yourself will find helpful when you return to the script after a month or two. Some useful points to comment on are the following: choices of algorithm, web references used, reasons for not doing things in a more obvious way, unusual paths through the code, anything that was a problem during development.

- Don’t clutter code with useless comments; assume intelligence and language proficiency on the part of the reader.
- It’s OK to run scripts as root, but avoid making them setuid; it’s tricky to make setuid scripts completely secure. Use **sudo** to implement appropriate access control policies instead.
- Don’t script what you don’t understand. Administrators often view scripts as authoritative documentation of how a particular procedure should be handled. Don’t set a misleading example by distributing half-baked scripts.
- Feel free to adapt code from existing scripts for your own needs. But don’t engage in “copy, paste, and pray” programming when you don’t understand the code. Take the time to figure it out. This time is never wasted.
- With **bash**, use **-x** to echo commands before they are executed and **-n** to check commands for syntax without executing them.
- Remember that in Python, you are in debug mode unless you explicitly turn it off with a **-0** argument on the command line. You can test the special __debug__ variable before printing diagnostic output.

Tom Christiansen suggests the following five golden rules for producing useful error messages:

- Error messages should go to STDERR, not STDOUT (see this page).
- Include the name of the program that’s issuing the error.
- State what function or operation failed.
- If a system call fails, include the **perror** string.
- Exit with some code other than 0.

7.2 SHELL BASICS

UNIX has always offered users a choice of shells, but some version of the Bourne shell, **sh**, has been standard on every UNIX and Linux system. The code for the original Bourne shell never made it out of AT&T licensing limbo, so these days **sh** is most commonly manifested in the form of the Almquist shell (known as **ash**, **dash**, or simply **sh**) or the "Bourne-again" shell, **bash**.

The Almquist shell is a reimplementation of the original Bourne shell without extra frills. By modern standards, it's barely usable as a login shell. It exists only to run **sh** scripts efficiently.

bash focuses on interactive usability. Over the years, it has absorbed most of the useful features pioneered by other shells. It still runs scripts designed for the original Bourne shell, but it's not particularly tuned for scripting. Some systems (e.g., the Debian lineage) include both **bash** and **dash**. Others rely on **bash** for both scripting and interactive use.

The Bourne shell has various other offshoots, notably **ksh** (the Korn shell) and **ksh**'s souped-up cousin **zsh**. **zsh** features broad compatibility with **sh**, **ksh**, and **bash**, as well as many interesting features of its own, including spelling correction and enhanced globbing. It's not used as any system's default shell (as far as we are aware), but it does have something of a cult following.

Historically, BSD-derived systems favored the C shell, **csh**, as an interactive shell. It's now most commonly seen in an enhanced version called **tcsh**. Despite the formerly widespread use of **csh** as a login shell, it is not recommended for use as a scripting language. For a detailed explanation of why this is so, see Tom Christiansen's classic rant, "Csh Programming Considered Harmful." It's widely reproduced on the web. One copy is harmful.cat-v.org/software/csh.

tcsh is a fine and widely available shell, but it's not an **sh** derivative. Shells are complex; unless you're a shell connoisseur, there's not much value in learning one shell for scripting and a second one—with different features and syntax—for daily use. Stick to a modern version of **sh** and let it do double duty.

Among the **sh** options, **bash** is pretty much the universal standard these days. To move effortlessly among different systems, standardize your personal environment on **bash**.



FreeBSD retains **tcsh** as root's default and does not ship **bash** as part of the base system. But that's easily fixed: run **sudo pkg install bash** to install **bash**, and use **chsh** to change your shell or the shell of another user. You can set **bash** as the default for new users by running **adduser -C**. (Changing the default might seem presumptuous, but standard FreeBSD relegates new users to the Almquist **sh**. There's nowhere to go but up.)

Before taking up the details of shell scripting, we should review some of the basic features and syntax of the shell.

The material in this section applies to the major interactive shells in the **sh** lineage (including **bash** and **ksh**, but not **csh** or **tcsh**), regardless of the exact platform you are using. Try out the

forms you’re not familiar with and experiment!

Command editing

We've watched too many people edit command lines with the arrow keys. You wouldn't do that in your text editor, right?

If you like **emacs**, all the basic **emacs** commands are available to you when you're editing history. <Control-E> goes to the end of the line and <Control-A> to the beginning. <Control-P> steps backward through recently executed commands and recalls them for editing. <Control-R> searches incrementally through your history to find old commands.

If you like **vi**/**vim**, put your shell's command-line editing into **vi** mode like this:

```
$ set -o vi
```

As in **vi**, editing is modal; however, you start in input mode. Press <Esc> to leave input mode and "i" to reenter it. In edit mode, "w" takes you forward a word, "fX" finds the next X in the line, and so on. You can walk through past command history entries with <Esc> k. Want **emacs** editing mode back again?

```
$ set -o emacs
```

Pipes and redirection

Every process has at least three communication channels available to it: standard input (STDIN), standard output (STDOUT), and standard error (STDERR). Processes initially inherit these channels from their parents, so they don't necessarily know where they lead. They might connect to a terminal window, a file, a network connection, or a channel belonging to another process, to name a few possibilities.

UNIX and Linux have a unified I/O model in which each channel is named with a small integer called a file descriptor. The exact number assigned to a channel is not usually significant, but STDIN, STDOUT, and STDERR are guaranteed to correspond to file descriptors 0, 1, and 2, so it's safe to refer to these channels by number. In the context of an interactive terminal window, STDIN normally reads from the keyboard and both STDOUT and STDERR write their output to the screen.

Many traditional UNIX commands accept their input from STDIN and write their output to STDOUT. They write error messages to STDERR. This convention lets you string commands together like building blocks to create composite pipelines.

The shell interprets the symbols <, >, and >> as instructions to reroute a command's input or output to or from a file. A < symbol connects the command's STDIN to the contents of an existing file. The > and >> symbols redirect STDOUT; > replaces the file's existing contents, and >> appends to them. For example, the command

```
$ grep bash /etc/passwd > /tmp/bash-users
```

copies lines containing the word "bash" from **/etc/passwd** to **/tmp/bash-users**, creating the file if necessary. The command below sorts the contents of that file and prints them to the terminal.

```
$ sort < /tmp/bash-users
root:x:0:0:root:/root:/bin/bash
...
```

Truth be told, the **sort** command accepts filenames, so the < symbol is optional in this context. It's used here for illustration.

To redirect both STDOUT and STDERR to the same place, use the `>&` symbol. To redirect STDERR only, use `2>`.

The **find** command illustrates why you might want separate handling for STDOUT and STDERR because it tends to produce output on both channels, especially when run as an unprivileged user. For example, a command such as

```
$ find / -name core
```

usually results in so many “permission denied” error messages that genuine hits get lost in the clutter. To discard all the error messages, use

```
$ find / -name core 2> /dev/null
```

In this version, only real matches (where the user has read permission on the parent directory) come to the terminal window. To save the list of matching paths to a file, use

```
$ find / -name core > /tmp/corefiles 2> /dev/null
```

This command line sends matching paths to **/tmp/corefiles**, discards errors, and sends nothing to the terminal window.

To connect the STDOUT of one command to the STDIN of another, use the **|** symbol, commonly known as a pipe. For example:

```
$ find / -name core 2> /dev/null | less
```

The first command runs the same **find** operation as the previous example, but sends the list of discovered files to the **less** pager rather than to a file. Another example:

```
$ ps -ef | grep httpd
```

This one runs **ps** to generate a list of processes and pipes it to the **grep** command, which selects lines that contain the word **httpd**. The output of **grep** is not redirected, so the matching lines come to the terminal window.

```
$ cut -d: -f7 < /etc/passwd | sort -u
```

Here, the **cut** command picks out the path to each user’s shell from **/etc/passwd**. The list of shells is then sent through **sort -u** to produce a sorted list of unique values.

To execute a second command only if its precursor completes successfully, you can separate the commands with an **&&** symbol. For example,

```
$ mkdir foo && cd foo
```

attempts to create a directory called **foo**, and if the directory was successfully created, executes **cd**. Here, the success of the **mkdir** command is defined as its yielding an exit code of zero, so the use of a symbol that suggests “logical AND” for this purpose might be confusing if you’re accustomed to short-circuit evaluation in other programming languages. Don’t think about it too much; just accept it as a shell idiom.

Conversely, the || symbol executes the following command only if the preceding command fails (that is, it produces a nonzero exit status). For example,

```
$ cd foo || echo "No such directory"
```

In a script, you can use a backslash to break a command onto multiple lines. This feature can help to distinguish error-handling code from the rest of a command pipeline:

```
cp --preserve --recursive /etc/* /spare/backup \
    || echo "Did NOT make backup"
```

For the opposite effect—multiple commands combined onto one line—you can use a semicolon as a statement separator:

```
$ mkdir foo; cd foo; touch afile
```

Variables and quoting

Variable names are unmarked in *assignments* but prefixed with a dollar sign when their values are *referenced*. For example:

```
$ etcdir='/etc'
$ echo $etcdir
/etc
```

Omit spaces around the = symbol; otherwise, the shell mistakes your variable name for a command name and treats the rest of the line as a series of arguments to that command.

When referencing a variable, you can surround its name with curly braces to clarify to the parser and to human readers where the variable name stops and other text begins; for example, **${etcdir}** instead of just **$etcdir**. The braces are not normally required, but they can be useful when you want to expand variables inside double-quoted strings. Often, you'll want the contents of a variable to be followed by literal letters or punctuation. For example,

```
$ echo "Saved ${rev}th version of mdadm.conf."
Saved 8th version of mdadm.conf.
```

There's no standard convention for the naming of shell variables, but all-caps names typically suggest environment variables or variables read from global configuration files. More often than not, local variables are all-lowercase with components separated by underscores. Variable names are case sensitive.

The shell treats strings enclosed in single and double quotes similarly, except that double-quoted strings are subject to globbing (the expansion of filename-matching metacharacters such as * and ?) and variable expansion. For example:

```
$ mylang="Pennsylvania Dutch"
$ echo "I speak ${mylang}."
I speak Pennsylvania Dutch.
$ echo 'I speak ${mylang}.'
I speak ${mylang}.
```

Backquotes, also known as backticks, are treated similarly to double quotes, but they have the additional effect of executing the contents of the string as a shell command and replacing the string with the command's output. For example,

```
$ echo "There are `wc -l < /etc/passwd` lines in the passwd file."
There are 28 lines in the passwd file.
```

Environment variables

When a UNIX process starts up, it receives a list of command-line arguments and also a set of "environment variables." Most shells show you the current environment in response to the **printenv** command:

```
$ printenv
EDITOR=vi
USER=garth
ENV=/home/garth/.bashrc
LSCOLORS=exfxgxgxdxgxgxbxbxcxcx
PWD=/mega/Documents/Projects/Code/spl
HOME=/home/garth
... <total of about 50>
```

By convention, environment variables have all-caps names, but that is not technically required.

Programs that you run can consult these variables and change their behavior accordingly. For example, **vipw** checks the EDITOR environment variable to determine which text editor to run for you.

Environment variables are automatically imported into **sh**'s variable namespace, so they can be set and read with the standard syntax. Use **export** *varname* to promote a shell variable to an environment variable. You can also combine this syntax with a value assignment, as seen here:

```
$ export EDITOR=nano
$ vipw
<starts the nano editor>
```

Despite being called "environment" variables, these values don't exist in some abstract, ethereal place outside of space and time. The shell passes a snapshot of the current values to any program you run, but no ongoing connection exists. Moreover, every shell or program—and every terminal window—has its own distinct copy of the environment that can be separately modified.

Commands for environment variables that you want to set up at login time should be included in your **~/.profile** or **~/.bash_profile** file. Other environment variables, such as PWD for the current working directory, are automatically maintained by the shell.

Common filter commands

Any well-behaved command that reads STDIN and writes STDOUT can be used as a filter (that is, a component of a pipeline) to process data. In this section we briefly review some of the more widely used filter commands (including some used in passing above), but the list is practically endless. Filter commands are so team oriented that it's sometimes hard to show their use in isolation.

Most filter commands accept one or more filenames on the command line. Only if you do not specify a file do they read their standard input.

cut: separate lines into fields

The **cut** command prints selected portions of its input lines. It most commonly extracts delimited fields, as in the example on this page, but it can return segments defined by column boundaries as well. The default delimiter is <Tab>, but you can change delimiters with the **-d** option. The **-f** options specifies which fields to include in the output.

For an example of the use of **cut**, see the section on **uniq**, below.

sort: sort lines

sort sorts its input lines. Simple, right? Well, maybe not—there are a few potential subtleties regarding the exact parts of each line that are sorted (the "keys") and the collation order to be imposed. Table 7.2 shows a few of the more common options, but check the man page for others.

Table 7.2: sort options

Opt	Meaning
-b	Ignore leading whitespace
-f	Sort case-insensitively
-h	Sort "human readable" numbers (e.g., 2MB)
-k	Specify the columns that form the sort key
-n	Compare fields as integer numbers
-r	Reverse sort order
-t	Set field separator (the default is whitespace)
-u	Output only unique records

The commands below illustrate the difference between numeric and dictionary sorting, which is the default. Both commands use the **-t:** and **-k3,3** options to sort the **/etc/group** file by its third colon-separated field, the group ID. The first sorts numerically and the second alphabetically.

```
$ sort -t: -k3,3 -n /etc/group
root:x:0:
bin:x:1:daemon
daemon:x:2:
...
$ sort -t: -k3,3 /etc/group
root:x:0:
bin:x:1:daemon
users:x:100:
...
```

sort accepts the key specification **-k3** (rather than **-k3,3**), but it probably doesn't do what you expect. Without the terminating field number, the sort key continues to the end of the line.

Also useful is the **-h** option, which implements a numeric sort that understands suffixes such as M for mega and G for giga. For example, the following command correctly sorts the sizes of directories under **/usr** while maintaining the legibility of the output:

```
$ du -sh /usr/* | sort -h
16K     /usr/locale
128K    /usr/local
648K    /usr/games
15M     /usr/sbin
20M     /usr/include
117M    /usr/src
126M    /usr/bin
845M    /usr/share
1.7G    /usr/lib
```

***uniq**: print unique lines*

uniq is similar in spirit to **sort -u**, but it has some useful options that **sort** does not emulate: **-c** to count the number of instances of each line, **-d** to show only duplicated lines, and **-u** to show only nonduplicated lines. The input must be presorted, usually by being run through **sort**.

For example, the command below shows that 20 users have **/bin/bash** as their login shell and that 12 have **/bin/false**. (The latter are either pseudo-users or users whose accounts have been disabled.)

```
$ cut -d: -f7 /etc/passwd | sort | uniq -c
  20 /bin/bash
  12 /bin/false
```

***wc**: count lines, words, and characters*

Counting the number of lines, words, and characters in a file is another common operation, and the **wc** (word count) command is a convenient way of doing this. Run without options, it displays

all three counts:

```
$ wc /etc/passwd
 32  77 2003 /etc/passwd
```

In the context of scripting, it is more common to supply a **-l**, **-w**, or **-c** option to make **wc**'s output consist of a single number. This form is most commonly seen inside backquotes so that the result can be saved or acted on.

***tee**: copy input to two places*

A command pipeline is typically linear, but it's often helpful to tap into the data stream and send a copy to a file or to the terminal window. You can do this with the **tee** command, which sends its standard input both to standard out and to a file that you specify on the command line. Think of it as a tee fixture in plumbing.

The device **/dev/tty** is a synonym for the current terminal window. For example,

```
$ find / -name core | tee /dev/tty | wc -l
```

prints both the pathnames of files named **core** and a count of the number of **core** files that were found.

A common idiom is to terminate a pipeline that will take a long time to run with a **tee** command. That way, output goes both to a file and to the terminal window for inspection. You can preview the initial results to make sure everything is working as you expected, then leave while the command runs, knowing that the results will be saved.

***head** and **tail**: read the beginning or end of a file*

Reviewing lines from the beginning or end of a file is a common administrative operation. These commands display ten lines of content by default, but you can use the **-n** *numlines* option to specify more or fewer.

For interactive use, **head** is more or less obsoleted by the **less** command, which paginates files for display. But **head** still finds plenty of use within scripts.

tail also has a nifty **-f** option that's particularly useful for sysadmins. Instead of exiting immediately after printing the requested number of lines, **tail -f** waits for new lines to be added to the end of the file and prints them as they appear—great for monitoring log files. Be aware, however, that the program writing the file might be buffering its own output. Even if lines are being added at regular intervals from a logical perspective, they might only become visible in chunks of 1KiB or 4KiB.

head and **tail** accept multiple filenames on the command line. Even **tail -f** allows multiple files, and this feature can be quite handy; when new output appears, **tail** prints the name of the file in which it appeared.

Type <Control-C> to stop monitoring.

***grep**: search text*

grep searches its input text and prints the lines that match a given pattern. Its name derives from the **g**/*regular-expression*/**p** command in the **ed** editor, which came with the earliest versions of UNIX (and is still present on current systems).

“Regular expressions” are text-matching patterns written in a standard and well-characterized pattern-matching language. They’re a universal standard used by most programs that do pattern matching, although there are minor variations among implementations. The odd name stems from regular expressions’ origins in theory-of-computation studies. We discuss regular expression syntax in more detail starting here.

Like most filters, **grep** has many options, including **-c** to print a count of matching lines, **-i** to ignore case when matching, and **-v** to print nonmatching (rather than matching) lines. Another useful option is **-l** (lower case L), which makes **grep** print only the names of matching files rather than printing each line that matches. For example, the command

```
$ sudo grep -l mdadm /var/log/*
/var/log/auth.log
/var/log/syslog.0
```

shows that log entries from **mdadm** have appeared in two different log files.

grep is traditionally a fairly basic regular expression engine, but some versions permit the selection of other dialects. For example, **grep -P** on Linux selects Perl-style expressions, though the man page warns darkly that they are “highly experimental.” If you need full power, just use Ruby, Python, or Perl.

If you filter the output of **tail -f** with **grep**, add the **--line-buffered** option to make sure you see each matching line as soon as it becomes available:

```
$ tail -f /var/log/messages | grep --line-buffered ZFS
May  8 00:44:00 nutrient ZFS: vdev state changed, pool_
    guid=10151087465118396807 vdev_guid=7163376375690181882
...
```

7.3 SH SCRIPTING

sh is great for simple scripts that automate things you'd otherwise be typing on the command line. Your command-line skills carry over to **sh** scripting, and vice versa, which helps you extract maximum value from the learning time you invest in **sh** derivatives. But once an **sh** script gets above 50 lines or when you need features that **sh** doesn't have, it's time to move on to Python or Ruby.

For scripting, there's some value in limiting yourself to the dialect understood by the original Bourne shell, which is both an IEEE and a POSIX standard. **sh**-compatible shells often supplement this baseline with additional language features. It's fine to use these extensions if you do so deliberately and are willing to require a specific interpreter. But more commonly, scripters end up using these extensions inadvertently and are then surprised when their scripts don't run on other systems.

In particular, don't assume that the system's version of **sh** is always **bash**, or even that **bash** is available. Ubuntu replaced **bash** with **dash** as the default script interpreter in 2006, and as part of that conversion process they compiled a handy list of **bash**isms to watch out for. You can find it at wiki.ubuntu.com/DashAsBinSh.

Execution

sh comments start with a sharp (#) and continue to the end of the line. As on the command line, you can break a single logical line onto multiple physical lines by escaping the newline with a backslash. You can also put more than one statement on a line by separating the statements with semicolons.

An **sh** script can consist of nothing but a series of command lines. For example, the following **helloworld** script simply does an **echo**.

```
#!/bin/sh
echo "Hello, world!"
```

The first line is known as the “shebang” statement and declares the text file to be a script for interpretation by **/bin/sh** (which might itself be a link to **dash** or **bash**). The kernel looks for this syntax when deciding how to execute the file. From the perspective of the shell spawned to execute the script, the shebang line is just a comment.

In theory, you would need to adjust the shebang line if your system’s **sh** was in a different location. However, so many existing scripts assume **/bin/sh** that systems are compelled to support it, if only through a link.

If you need your script to run under **bash** or another interpreter that might not have the same command path on every system, you can use **/usr/bin/env** to search your PATH environment variable for a particular command. For example,

```
#!/usr/bin/env ruby
```

is a common idiom for starting Ruby scripts. Like **/bin/sh**, **/usr/bin/env** is such a widely-relied-on path that all systems are obliged to support it.

Path searching has security implications, particularly when running scripts under **sudo**. See this page for more information about **sudo**’s handling of environment variables.

To prepare a script for running, just turn on its execute bit (see this page).

```
$ chmod +x helloworld
$ ./helloworld
Hello, world!
```

If your shell understands the command **helloworld** without the **./** prefix, that means the current directory (**.**) is in your search path. This is bad because it gives other users the opportunity to lay traps for you in the hope that you’ll try to execute certain commands while **cd**’ed to a directory on which they have write access.

You can also invoke the shell as an interpreter directly:

```
$ sh helloworld
Hello, world!
$ source helloworld
Hello, world!
```

The first command runs **helloworld** in a new instance of **sh**, and the second makes your existing login shell read and execute the contents of the file. The latter option is useful when the script sets up environment variables or makes other customizations that apply only to the current shell. It's commonly used in scripting to incorporate the contents of a configuration file written as a series of variable assignments. The "dot" command is a synonym for **source**, e.g., **. helloworld**.

See this page for more information about permission bits.

If you come from the Windows world, you might be accustomed to a file's extension indicating what type of file it is and whether it can be executed. In UNIX and Linux, the file permission bits determine whether a file can be executed, and if so, by whom. If you wish, you can give your shell scripts a **.sh** suffix to remind you what they are, but you'll then have to type out the **.sh** when you run the command, since UNIX doesn't treat extensions specially.

From commands to scripts

Before we jump into **sh**'s scripting features, a note about methodology. Most people write **sh** scripts the same way they write Python or Ruby scripts: with a text editor. However, it's more productive to think of your regular shell command prompt as an interactive script development environment.

For example, suppose you have log files named with the suffixes **.log** and **.LOG** scattered throughout a directory hierarchy and that you want to change them all to the uppercase form. First, find all the files:

```
$ find . -name '*log'
.do-not-touch/important.log
admin.com-log/
foo.log
genius/spew.log
leather_flog
...
```

Oops, it looks like you need to include the dot in the pattern and to leave out directories as well. Do a <Control-P> to recall the command and then modify it:

```
$ find . -type f -name '*.log'
.do-not-touch/important.log
foo.log
genius/spew.log
...
```

OK, this looks better. That **.do-not-touch** directory looks dangerous, though; you probably shouldn't mess around in there:

```
$ find . -type f -name '*.log' | grep -v .do-not-touch
foo.log
genius/spew.log
...
```

All right, that's the exact list of files that need renaming. Try generating some new names:

```
$ find . -type f -name '*.log' | grep -v .do-not-touch | while read fname
> do
> echo mv $fname `echo $fname | sed s/.log/.LOG/`
> done
mv foo.log foo.LOG
mv genius/spew.log genius/spew.LOG
...
```

Yup, those are the commands to run to perform the renaming. So how to do it for real? You could recall the command and edit out the **echo**, which would make **sh** execute the **mv** commands instead of just printing them. However, piping the commands to a separate instance of **sh** is less error prone and requires less editing of the previous command.

When you do a <Control-P>, you'll find that **bash** has thoughtfully collapsed your mini-script into a single line. To this condensed command line, simply add a pipe that sends the output to **sh -x**.

```
$ find . -type f -name '*.log' | grep -v .do-not-touch | while read fname;
    do echo mv $fname `echo $fname | sed s/.log/.LOG/`; done | sh -x
+ mv foo.log foo.LOG
+ mv genius/spew.log genius/spew.LOG
...
```

The **-x** option to **sh** prints each command before executing it.

That completes the actual renaming, but save the script for future reuse. **bash**'s built-in command **fc** is a lot like <Control-P>, but instead of returning the last command to the command line, it transfers the command to your editor of choice. Add a shebang line and usage comment, write the file to a plausible location (**~/bin** or **/usr/local/bin**, perhaps), make the file executable, and you have a script.

To summarize this approach:

1. Develop the script (or script component) as a pipeline, one step at a time, entirely on the command line. Use **bash** for this process even though the eventual interpreter might be **dash** or another **sh** variant.
2. Send output to standard output and check to be sure it looks right.
3. At each step, use the shell's command history to recall pipelines and the shell's editing features to tweak them.
4. Until the output looks right, you haven't actually done anything, so there's nothing to undo if the command is incorrect.
5. Once the output is correct, execute the actual commands and verify that they worked as you intended.
6. Use **fc** to capture your work, then clean it up and save it.

In the example above, the command lines were printed and then piped to a subshell for execution. This technique isn't universally applicable, but it's often helpful. Alternatively, you can capture output by redirecting it to a file. No matter what, wait until you see the right stuff in the preview before doing anything that's potentially destructive.

Input and output

The **echo** command is crude but easy. For more control over your output, use **printf**. It is a bit less convenient because you must explicitly put newlines where you want them (use “\n”), but it gives you the option to use tabs and enhanced number formatting in your the output. Compare the output from the following two commands:

```
$ echo "\taa\tbb\tcc\n"
\taa\tbb\tcc\n
$ printf "\taa\tbb\tcc\n"
    aa   bb   cc
```

Some systems have OS-level **printf** and **echo** commands, usually in **/usr/bin** and **/bin**, respectively. Although the commands and the shell built-ins are similar, they may diverge subtly in their specifics, especially in the case of **printf**. Either adhere to **sh**’s syntax or call the external **printf** with a full pathname.

You can use the **read** command to prompt for input. Here’s an example:

```
#!/bin/sh

echo -n "Enter your name: "
read user_name

if [ -n "$user_name" ]; then
    echo "Hello $user_name!"
    exit 0
else
    echo "Greetings, nameless one!"
    exit 1
fi
```

The **-n** in the **echo** command suppresses the usual newline, but you could also have used **printf** here. We cover the `if` statement’s syntax shortly, but its effect should be obvious here. The `-n` in the `if` statement evaluates to true if its string argument is not null. Here’s what the script looks like when run:

```
$ sh readexample
Enter your name: Ron
Hello Ron!
```

Spaces in filenames

The naming of files and directories is essentially unrestricted, except that names are limited in length and must not contain slash characters or nulls. In particular, spaces are permitted. Unfortunately, UNIX has a long tradition of separating command-line arguments at whitespace, so legacy software tends to break when spaces appear within filenames.

Spaces in filenames were once found primarily on filesystems shared with Macs and PCs, but they have now metastasized into UNIX culture and are found in some standard software packages as well. There are no two ways about it: administrative scripts *must* be prepared to deal with spaces in filenames (not to mention apostrophes, asterisks, and various other menacing punctuation marks).

In the shell and in scripts, spaceful filenames can be quoted to keep their pieces together. For example, the command

```
$ less "My spacey file"
```

preserves **My spacey file** as a single argument to **less**. You can also escape individual spaces with a backslash:

```
$ less My\ spacey\ file
```

The filename completion feature of most shells (usually bound to the <Tab> key) normally adds the backslashes for you.

When you are writing scripts, a useful weapon to know about is **find**'s **-print0** option. In combination with **xargs -0**, this option makes the **find**/**xargs** combination work correctly regardless of the whitespace contained within filenames. For example, the command

```
$ find /home -type f -size +1M -print0 | xargs -0 ls -l
```

prints a long **ls** listing of every file beneath **/home** that's over one megabyte in size.

Command-line arguments and functions

Command-line arguments to a script become variables whose names are numbers. `$1` is the first command-line argument, `$2` is the second, and so on. `$0` is the name by which the script was invoked. That could be a strange construction such as **../bin/example.sh**, so it doesn't necessarily have the same value each time the script is run.

The variable `$#` contains the number of command-line arguments that were supplied, and the variable `$*` contains all the arguments at once. Neither of these variables includes or counts `$0`. Here's an example of the use of arguments:

```
#!/bin/sh

show_usage() {
    echo "Usage: $0 source_dir dest_dir" 1>&2
    exit 1
}

# Main program starts here

if [ $# -ne 2 ]; then
    show_usage
else # There are two arguments
    if [ -d $1 ]; then
        source_dir=$1
    else
        echo 'Invalid source directory' 1>&2
        show_usage
    fi
    if [ -d $2 ]; then
        dest_dir=$2
    else
        echo 'Invalid destination directory' 1>&2
        show_usage
    fi
fi

printf "Source directory is ${source_dir}\n"
printf "Destination directory is ${dest_dir}\n"
```

If you call a script without arguments or with inappropriate arguments, the script should print a short usage message to remind you how to use it. The example script above accepts two arguments, validates that the arguments are both directories, and prints their names. If the arguments are invalid, the script prints a usage message and exits with a nonzero return code. If the caller of the script checks the return code, it will know that this script failed to execute correctly.

We created a separate `show_usage` function to print the usage message. If the script were later updated to accept additional arguments, the usage message would have to be changed in only one

place. The 1>&2 notation on lines that print error messages makes the output go to STDERR.

```
$ mkdir aaa bbb
$ sh showusage aaa bbb
Source directory is aaa
Destination directory is bbb
$ sh showusage foo bar
Invalid source directory
Usage: showusage source_dir dest_dir
```

Arguments to **sh** functions are treated like command-line arguments. The first argument becomes $1, and so on. As you can see above, $0 remains the name of the script.

To make the example more robust, we could make the show_usage routine accept an error code as an argument. That would allow a more definitive code to be returned for each different type of failure. The next code excerpt shows how that might look.

```
show_usage() {
    echo "Usage: $0 source_dir dest_dir" 1>&2
    if [ $# -eq 0 ]; then
        exit 99 # Exit with arbitrary nonzero return code
    else
        exit $1
    fi
}
```

In this version of the routine, the argument is optional. Within a function, $# tells you how many arguments were passed in. The script exits with code 99 if no more-specific code is designated. But a specific value, for example,

```
show_usage 5
```

makes the script exit with that code after printing the usage message. (The shell variable $? contains the exit status of the last command executed, whether used inside a script or at the command line.)

The analogy between functions and commands is strong in **sh**. You can define useful functions in your **~/.bash_profile** file (**~/.profile** for vanilla **sh**) and then use them on the command line as if they were commands. For example, if your site has standardized on network port 7988 for the SSH protocol (a form of “security through obscurity”), you might define

```
ssh() {
    /usr/bin/ssh -p 7988 $*
}
```

in your **~/.bash_profile** to make sure **ssh** is always run with the option **-p 7988**.

Like many shells, **bash** has an aliasing mechanism that can reproduce this limited example even more concisely, but functions are more general and more powerful.

Control flow

We've seen several if-then and if-then-else forms in this chapter already; they do pretty much what you'd expect. The terminator for an `if` statement is `fi`. To chain your `if` clauses, you can use the `elif` keyword to mean "else if." For example:

```
if [ $base -eq 1 ] && [ $dm -eq 1 ]; then
    installDMBase
elif [ $base -ne 1 ] && [ $dm -eq 1 ]; then
    installBase
elif [ $base -eq 1 ] && [ $dm -ne 1 ]; then
    installDM
else
    echo '==> Installing nothing'
fi
```

Both the peculiar `[]` syntax for comparisons and the command-line-optionlike names of the integer comparison operators (e.g., `-eq`) are inherited from the original Bourne shell's channeling of **/bin/test**. The brackets are actually a shorthand way of invoking **test** and are not a syntactic requirement of the `if` statement. (In reality, these operations are now built into the shell and do not actually run **/bin/test**.)

Table 7.3 shows the **sh** comparison operators for numbers and strings. **sh** uses textual operators for numbers and symbolic operators for strings.

Table 7.3: Elementary sh comparison operators

String	Numeric	True if
`x = y`	`x -eq y`	x is equal to y
`x != y`	`x -ne y`	x is not equal to y
`x <`[a] `y`	`x -lt y`	x is less than y
n/a	`x -le y`	x is less than or equal to y
`x >`[a] `y`	`x -gt y`	x is greater than y
n/a	`x -ge y`	x is greater than or equal to y
`-n x`	*n/a*	x is not null
`-z x`	*n/a*	x is null

a. Must be backslash-escaped or double bracketed to prevent interpretation as an input or output redirection character.

sh shines in its options for evaluating the properties of files (once again, courtesy of its **/bin/test** legacy). Table 7.4 shows a few of **sh**'s many file testing and file comparison operators.

Table 7.4: sh file evaluation operators

Operator	True if
-d *file*	*file* exists and is a directory
-e *file*	*file* exists
-f *file*	*file* exists and is a regular file
-r *file*	User has read permission on *file*
-s *file*	*file* exists and is not empty
-w *file*	User has write permission on *file*
file1 -nt *file2*	*file1* is newer than *file2*
file1 -ot *file2*	*file1* is older than *file2*

Although the elif form is useful, a case selection is often a better choice for clarity. Its syntax is shown below in a sample routine that centralizes logging for a script. Of particular note are the closing parenthesis after each condition and the two semicolons that follow the statement block to be executed when a condition is met (except for the last condition). The case statement ends with esac.

```
# The log level is set in the global variable LOG_LEVEL. The choices
# are, from most to least severe, Error, Warning, Info, and Debug.

logMsg() {
    message_level=$1
    message_itself=$2
    if [ $message_level -le $LOG_LEVEL ]; then
        case $message_level in
            0) message_level_text="Error" ;;
            1) message_level_text="Warning" ;;
            2) message_level_text="Info" ;;
            3) message_level_text="Debug" ;;
            *) message_level_text="Other"
        esac
        echo "${message_level_text}: $message_itself"
    fi
}
```

This routine illustrates the common “log level” paradigm used by many administrative applications. The code of the script generates messages at many different levels of detail, but only the ones that pass a globally set threshold, $LOG_LEVEL, are actually logged or acted on. To clarify the importance of each message, the message text is preceded by a label that denotes its associated log level.

Loops

sh's for…in construct makes it easy to take some action for a group of values or files, especially when combined with filename globbing (the expansion of simple pattern-matching characters such as * and ? to form filenames or lists of filenames). The *.sh pattern in the for loop below returns a list of matching filenames in the current directory. The for statement then iterates through that list, assigning each filename in turn to the variable script.

```
#!/bin/sh

suffix=BACKUP--`date +%Y-%m-%d-%H%M`

for script in *.sh; do
    newname="$script.$suffix"
    echo "Copying $script to $newname..."
    cp -p $script $newname
done
```

The output looks like this:

```
$ sh forexample
Copying rhel.sh to rhel.sh.BACKUP--2017-01-28-2228...
Copying sles.sh to sles.sh.BACKUP--2017-01-28-2228...
...
```

The filename expansion is not magic in this context; it works exactly as it does on the command line. Which is to say, the expansion happens first and the line is then processed by the interpreter in its expanded form. (More accurately, the filename expansion is a little bit magical in that it does maintain a notion of the atomicity of each filename. Filenames that contain spaces go through the for loop in a single pass.) You could just as well have entered the filenames statically, as in the line

```
for script in rhel.sh sles.sh; do
```

In fact, any whitespace-separated list of things, including the contents of a variable, works as a target of for…in. You can also omit the list entirely (along with the in keyword), in which case the loop implicitly iterates over the script's command-line arguments (if at the top level) or the arguments passed to a function:

```
#!/bin/sh

for file; do
    newname="${file}.backup"
    echo "Copying $file to $newname..."
    cp -p $file $newname
done
```

bash, but not vanilla **sh**, also has the more familiar `for` loop from traditional programming languages in which you specify starting, increment, and termination clauses.

For example:

```
# bash-specific
for (( i=0 ; i < $CPU_COUNT ; i++ )); do
    CPU_LIST="$CPU_LIST $i"
done
```

The next example illustrates **sh**'s `while` loop, which is useful for processing command-line arguments and for reading the lines of a file.

```
#!/bin/sh

exec 0<$1
counter=1
while read line; do
    echo "$counter: $line"
    counter=$((counter + 1))
done
```

Here's what the output looks like:

```
$ sh whileexample /etc/passwd
1: root:x:0:0:Superuser:/root:/bin/bash
2: bin:x:1:1:bin:/bin:/bin/bash
3: daemon:x:2:2:Daemon:/sbin:/bin/bash
...
```

This scriptlet has a couple of interesting features. The `exec` statement redefines the script's standard input to come from whatever file is named by the first command-line argument. The file must exist or the script generates an error. Depending on the invocation, `exec` can also have the more familiar meaning "stop this script and transfer control to another script or expression." It's yet another shell oddity that both functions are accessed through the same statement.

The `read` statement within the `while` clause is a shell built-in, but it acts like an external command. You can put external commands in a `while` clause as well; in that form, the `while` loop terminates when the external command returns a nonzero exit status.

The `$((counter + 1))` expression is an odd duck, indeed. The `$((…))` notation forces numeric evaluation. It also makes optional the use of `$` to mark variable names. The expression is replaced with the result of the arithmetic calculation.

The `$((…))` shenanigans work in the context of double quotes, too. In **bash**, which supports C's ++ postincrement operator, the body of the loop can be collapsed down to one line.

```
while read line; do
    echo "$((counter++)): $line"
done
```

Arithmetic

All **sh** variables are string valued, so **sh** does not distinguish between the number 1 and the character string “1” in assignments. The difference lies in how the variables are used. The following code illustrates the distinction:

```
#!/bin/sh

a=1
b=$((2))

c=$a+$b
d=$((a + b))

echo "$a + $b = $c \t(plus sign as string literal)"
echo "$a + $b = $d \t(plus sign as arithmetic addition)"
```

This script produces the output

```
1 + 2 = 1+2 (plus sign as string literal)
1 + 2 = 3   (plus sign as arithmetic addition)
```

Note that the plus sign in the assignment to `$c` does not act as a concatenation operator for strings. It’s just a literal character. That line is equivalent to

```
c="$a+$b"
```

To force numeric evaluation, you enclose an expression in `$((...))`, as shown with the assignment to `$d` above. But even this precaution does not result in `$d` receiving a numeric value; the result of the calculation is the string “3”.

sh has the usual assortment of arithmetic, logical, and relational operators; see the man page for details.

7.4 Regular expressions

As we mentioned here, regular expressions are standardized patterns that parse and manipulate text. For example, the regular expression

```
I sent you a che(que|ck) for the gr[ae]y-colou?red alumini?um.
```

matches sentences that use either American or British spelling conventions.

Regular expressions are supported by most modern languages, though some take them more to heart than others. They're also used by UNIX commands such as **grep** and **vi**. They are so common that the name is usually shortened to "regex." Entire books have been written about how to harness their power; you can find citations for two of them at the end of this chapter.

The filename matching and expansion performed by the shell when it interprets command lines such as **wc -l *.pl** *is not* a form of regular expression matching. It's a different system called "shell globbing," and it uses a different and simpler syntax.

Regular expressions are not themselves a scripting language, but they're so useful that they merit featured coverage in any discussion of scripting; hence, this section.

The matching process

Code that evaluates a regular expression attempts to match a single given text string to a single given pattern. The “text string” to match can be very long and can contain embedded newlines. It’s sometimes convenient to use a regex to match the contents of an entire file or document.

For the matcher to declare success, the entire search pattern must match a contiguous section of the search text. However, the pattern can match at any position. After a successful match, the evaluator returns the text of the match along with a list of matches for any specially delimited subsections of the pattern.

Literal characters

In general, characters in a regular expression match themselves. So the pattern

```
I am the walrus
```

matches the string “I am the walrus” and that string only. Since it can match anywhere in the search text, the pattern can be successfully matched to the string

> I am the egg man. I am the walrus. Koo koo ka-choo!

However, the actual match is limited to the “I am the walrus” portion. Matching is case sensitive.

Special characters

Table 7.5 shows the meanings of some common special symbols that can appear in regular expressions. These are just the basics—there are many more.

Table 7.5: Special characters in regular expressions (common ones)

Symbol	What it matches or does
.	Matches any character
[*chars*]	Matches any character from a given set
[^*chars*]	Matches any character not in a given set
^	Matches the beginning of a line
$	Matches the end of a line
\w	Matches any "word" character (same as [A-Za-z0-9_])
\s	Matches any whitespace character (same as [\f\t\n\r])[a]
\d	Matches any digit (same as [0-9])
|	Matches either the element to its left or the one to its right
(*expr*)	Limits scope, groups elements, allows matches to be captured
?	Allows zero or one match of the preceding element
*	Allows zero, one, or many matches of the preceding element
+	Allows one or more matches of the preceding element
{ *n* }	Matches exactly *n* instances of the preceding element
{ *min,* }	Matches at least *min* instances (note the comma)
{ *min,max* }	Matches any number of instances from *min* to *max*

a. That is, a space, a form feed, a tab, a newline, or a return

Many special constructs, such as + and |, affect the matching of the "thing" to their left or right. In general, a "thing" is a single character, a subpattern enclosed in parentheses, or a character class enclosed in square brackets. For the | character, however, thingness extends indefinitely to both left and right. If you want to limit the scope of the vertical bar, enclose the bar and both things in their own set of parentheses. For example,

```
I am the (walrus|egg man)\.
```

matches either "I am the walrus." or "I am the egg man.". This example also demonstrates escaping of special characters (here, the dot). The pattern

```
(I am the (walrus|egg man)\. ?){1,2}
```

matches any of the following:

- I am the walrus.

- I am the egg man.
- I am the walrus. I am the egg man.
- I am the egg man. I am the walrus.
- I am the egg man. I am the egg man.
- I am the walrus. I am the walrus.

It also matches “I am the walrus. I am the egg man. I am the walrus.”, even though the number of repetitions is explicitly capped at two. That’s because the pattern need not match the entire search text. Here, the regex matches two sentences and terminates, declaring success. It doesn’t care that another repetition is available.

It is a common error to confuse the regular expression metacharacter `*` (the zero-or-more quantifier) with the shell’s `*` globbing character. The regex version of the star needs something to modify; otherwise, it won’t do what you expect. Use `.*` if any sequence of characters (including no characters at all) is an acceptable match.

Example regular expressions

In the United States, postal ("zip") codes have either five digits or five digits followed by a dash and four more digits. To match a regular zip code, you must match a five-digit number. The following regular expression fits the bill:

```
^\d{5}$
```

The `^` and `$` match the beginning and end of the search text but do not actually correspond to characters in the text; they are "zero-width assertions." These characters ensure that only texts consisting of exactly five digits match the regular expression—the regex will not match five digits within a larger string. The `\d` escape matches a digit, and the quantifier `{5}` says that there must be exactly five one-digit matches.

To accommodate either a five-digit zip code or an extended zip+4, add an optional dash and four additional digits:

```
^\d{5}(-\d{4})?$
```

The parentheses group the dash and extra digits together so that they are considered one optional unit. For example, the regex won't match a five-digit zip code followed by a dash. If the dash is present, the four-digit extension must be present as well or there is no match.

A classic demonstration of regex matching is the following expression,

```
M[ou]'?am+[ae]r ([AEae]l[- ])?[GKQ]h?[aeu]+([dtz][dhz]?){1,2}af[iy]
```

which matches most of the variant spellings of the name of former Libyan head of state Moammar Gadhafi, including

- Muammar al-Kaddafi (BBC)
- Moammar Gadhafi (Associated Press)
- Muammar al-Qadhafi (Al-Jazeera)
- Mu'ammar Al-Qadhafi (U.S. Department of State)

Do you see how each of these would match the pattern? Note that this regular expression is designed to be liberal in what it matches. Many patterns that aren't legitimate spellings also match: for example, "Mo'ammer el Qhuuuzzthaf".

This regular expression also illustrates how quickly the limits of legibility can be reached. Most regex systems support an `x` option that ignores literal whitespace in the pattern and enables comments, allowing the pattern to be spaced out and split over multiple lines. You can then use whitespace to separate logical groups and clarify relationships, just as you would in a procedural language. For example, here's a more readable version of that same Moammar Gadhafi regex:

```
M [ou] '? a m+ [ae] r    # First name: Mu'ammar, Moamar, etc.
\s                       # Whitespace; can't use a literal space here
(                        # Group for optional last name prefix
    [AEae] l             #   Al, El, al, or el
    [-\s]                #   Followed by either a dash or whitespace
)?
[GKQ] h? [aeu]+          # Initial syllable of last name: Kha, Qua, etc.
(                        # Group for consonants at start of 2nd syllable
    [dtz] [dhz]?         #   dd, dh, etc.
){1,2}                   # Group might occur twice, as in Quadhdhafi
af [iy]                  # Final afi or afy
```

This helps a bit, but it’s still pretty easy to torture later readers of your code. So be kind: if you can, use hierarchical matching and multiple small matches instead of trying to cover every possible situation in one large regular expression.

Captures

When a match succeeds, every set of parentheses becomes a “capture group” that records the actual text that it matched. The exact manner in which these pieces are made available to you depends on the implementation and context. In most cases, you can access the results as a list, array, or sequence of numbered variables.

Since parentheses can nest, how do you know which match is which? Easy: the matches arrive in the same order as the opening parentheses. There are as many captures as there are opening parentheses, regardless of the role (or lack of role) that each parenthesized group played in the actual matching. When a parenthesized group is not used (e.g., `Mu(')?ammar` when matched against “Muammar”), its corresponding capture is empty.

If a group is matched more than once, the contents of only the last match are returned. For example, with the pattern

```
(I am the (walrus|egg man)\. ?){1,2}
```

matching the text

```
I am the egg man. I am the walrus.
```

there are two results, one for each set of parentheses:

```
I am the walrus.
walrus
```

Both capture groups actually matched twice. However, only the last text to match each set of parentheses is actually captured.

Greediness, laziness, and catastrophic backtracking

Regular expressions match from left to right. Each component of the pattern matches the longest possible string before yielding to the next component, a characteristic known as greediness.

If the regex evaluator reaches a state from which a match cannot be completed, it unwinds a bit of the candidate match and makes one of the greedy atoms give up some of its text. For example, consider the regex `a*aa` being matched against the input text "aaaaaa".

At first, the regex evaluator assigns the entire input to the `a*` portion of the regex because the `a*` is greedy. When there are no more a's to match, the evaluator goes on to try to match the next part of the regex. But oops, it's an `a`, and there is no more input text that can match an `a`; time to backtrack. The `a*` has to give up one of the a's it has matched.

Now the evaluator can match `a*a`, but it still cannot match the last `a` in the pattern. So it backtracks again and takes away a second a from the `a*`. Now the second and third `a`'s in the pattern both have a's to pair with, and the match is complete.

This simple example illustrates some important general points. First, greedy matching plus backtracking makes it expensive to match apparently simple patterns such as `<img.*></tr>` when processing entire files. The `.*` portion starts by matching everything from the first <img to the end of the input, and only through repeated backtracking does it contract to fit the local tags.

Furthermore, the ></tr> that this pattern binds to is the *last possible* valid match in the input, which is probably not what you want. More likely, you meant to match an <img> tag followed immediately by a </tr> tag. A better way to write this pattern is `<img[^>]*>\s*</tr>`, which allows the initial wild card match to expand only to the end of the current tag, because it cannot cross a right-angle-bracket boundary.

You can also use lazy (as opposed to greedy) wild card operators: `*?` instead of `*`, and `+?` instead of `+`. These versions match as few characters of the input as they can. If that fails, they match more. In many situations, these operators are more efficient and closer to what you want than the greedy versions.

Note, however, that they can produce matches different from those of the greedy operators; the difference is more than just one of implementation. In our HTML example, the lazy pattern would be `<img.*?></tr>`. But even here, the `.*?` could eventually grow to include unwanted >'s because the next tag after an <img> might not be a </tr>. Again, probably not what you want.

Patterns with multiple wild card sections can cause exponential behavior in the regex evaluator, especially if portions of the text can match several of the wild card expressions and especially if the search text does not match the pattern. This situation is not as unusual as it might sound, especially when pattern matching with HTML. Often, you'll want to match certain tags followed by other tags, possibly separated by even more tags, a recipe that might require the regex evaluator to try many possible combinations.

Regex guru Jan Goyvaerts calls this phenomenon “catastrophic backtracking” and writes about it in his blog; see regular-expressions.info/catastrophic.html for details and some good solutions.

(Although this section shows HTML excerpts as examples of text to be matched, regular expressions are not really the right tool for this job. Our external reviewers were uniformly aghast. Ruby and Python both have excellent add-ons that parse HTML documents the proper way. You can then access the portions you’re interested in with XPath or CSS selectors. See the Wikipedia page for XPath and the respective languages’ module repositories for details.)

A couple of take-home points from all this:

- If you can do pattern matching line-by-line rather than file-at-a-time, there is much less risk of poor performance.
- Even though regex notation makes greedy operators the default, they probably shouldn’t be. Use lazy operators.
- All uses of `.*` are inherently suspicious and should be scrutinized.

7.5 PYTHON PROGRAMMING

Python and Ruby are interpreted languages with a pronounced object-oriented inflection. Both are widely used as general-purpose scripting languages and have extensive libraries of third party modules. We discuss Ruby in more detail starting here.

Python offers a straightforward syntax that's usually pretty easy to follow, even when reading other people's code.

We recommend that all sysadmins become fluent in Python. It's the modern era's go-to language for both system administration and general-purpose scripting. It's also widely supported as a glue language for use within other systems (e.g., the PostgreSQL database and Apple's Xcode development environment). It interfaces cleanly with REST APIs and has well-developed libraries for machine learning, data analysis, and numeric computation.

The passion of Python 3

Python was already well on its way to becoming the world's default scripting language when Python 3 was released in 2008. For this release, the developers chose to forgo backward compatibility with Python 2 so that a group of modest but fundamental changes and corrections could be made to the language, particularly in the area of internationalized text processing. The exact list of changes in Python 3 isn't relevant to this brief discussion, but you can find a summary at docs.python.org/3.0/whatsnew/3.0.html.

Unfortunately, the rollout of Python 3 proved to be something of a debacle. The language updates are entirely sensible, but they're not must-haves for the average Python programmer with an existing code base to maintain. For a long time, scripters avoided Python 3 because their favorite libraries didn't support it, and library authors didn't support Python 3 because their clients were still using Python 2.

Even in the best of circumstances, it's difficult to push a large and interdependent user community past this sort of discontinuity. In the case of Python 3, early entrenchments persisted for the better part of a decade. However, as of 2017, that situation finally seems to be changing.

Compatibility libraries that allow the same Python code to run under either version of the language have helped ease the transition, to some extent. But even now, Python 3 remains less common in the wild than Python 2.

As of this writing, py3readiness.org reports that only 17 of the top 360 Python libraries remain incompatible with Python 3. But the long tail of unported software is more sobering: only a tad more than 25% of the libraries warehoused at pypi.python.org (the Python Package Index, aka PyPI) run under Python 3. Of course, many of these projects are older and no longer maintained, but 25% is still a concerningly low number. See caniusepython3.com for up-to-date statistics.

Python 2 or Python 3?

The world's solution to the slowly unfolding Python transition has been to treat Pythons 2 and 3 as separate languages. You needn't consecrate your systems to one or the other; you can run both simultaneously without conflict.

All our example systems ship Python 2 by default, usually as **/usr/bin/python2** with a symbolic link from **/usr/bin/python**. Python 3 can typically be installed as a separate package; the binary is called **python3**.

RHEL  Although the Fedora project is working to make Python 3 its system default, Red Hat and CentOS are far behind and do not even define a prebuilt package for Python 3. However, you can pick one up from Fedora's EPEL (Extra Packages for Enterprise Linux) repository. See the FAQ at fedoraproject.org/wiki/EPEL for instructions on accessing this repository. It's easy to set up, but the exact commands are version-dependent.

For new scripting work or for those new to Python altogether, it makes sense to jump directly to Python 3. That's the syntax we show in this chapter, though in fact it's only the `print` lines that vary between Python 2 and Python 3 in our simple examples.

For existing software, use whichever version of Python the software prefers. If your choice is more complicated than simply new vs. old code, consult the Python wiki at wiki.python.org/moin/Python2orPython3 for an excellent collection of issues, solutions, and recommendations.

Python quick start

For a more thorough introduction to Python than we can give here, Mark Pilgrim's *Dive Into Python 3* is a great place to start. It's available for reading or for download (without charge) at diveintopython3.net, or as a printed book from Apress. A complete citation can be found here.

To start, here's a quick "Hello, world!" script:

```
#!/usr/bin/python3
print("Hello, world!")
```

To get it running, set the execute bit or invoke the **python3** interpreter directly:

```
$ chmod +x helloworld
$ ./helloworld
Hello, world!
```

Python's most notable break with tradition is that indentation is logically significant. Python does not use braces, brackets, or begin and end to delineate blocks. Statements at the same level of indentation automatically form blocks. The exact indentation style (spaces or tabs, depth of indentation) does not matter.

Python blocking is best shown by example. Consider this simple if-then-else statement:

```
import sys

a = sys.argv[1]

if a == "1":
    print('a is one')
    print('This is still the then clause of the if statement.')
else:
    print('a is', a)
    print('This is still the else clause of the if statement.')

print('This is after the if statement.')
```

The first line imports the sys module, which contains the argv array. The two paths through the if statement both have two lines, each indented to the same level. (Colons at the end of a line are normally a clue that the line introduces and is associated with an indented block that follows it.) The final print statement lies outside the context of the if statement.

```
$ python3 blockexample 1
a is one
This is still the then clause of the if statement.
This is after the if statement.

$ python3 blockexample 2
a is 2
This is still the else clause of the if statement.
This is after the if statement.
```

Python's indentation convention is less flexibile for the formatting of code, but it does reduce clutter in the form of braces and semicolons. It's an adjustment for those accustomed to traditional delimiters, but most people ultimately find that they like it.

Python's print function accepts an arbitrary number of arguments. It inserts a space between each pair of arguments and automatically supplies a newline. You can suppress or modify these characters by adding end= or sep= options to the end of the argument list.

For example, the line

```
print("one", "two", "three", sep="-", end="!\n")
```

produces the output

```
one-two-three!
```

Comments are introduced with a sharp (#) and last until the end of the line, just as in **sh**, Perl, and Ruby.

You can split long lines by backslashing the end of line breaks. When you do this, the indentation of only the first line is significant. You can indent the continuation lines however you like. Lines with unbalanced parentheses, square brackets, or curly braces automatically signal continuation even in the absence of backslashes, but you can include the backslashes if doing so clarifies the structure of the code.

Some cut-and-paste operations convert tabs to spaces, and unless you know what you're looking for, this can drive you nuts. The golden rule is never to mix tabs and spaces; use one or the other for indentation. A lot of software makes the traditional assumption that tabs fall at 8-space intervals, which is too much indentation for readable code. Most in the Python community seem to prefer spaces and 4-character indentation.

However you decide to attack the indentation problem, most editors have options that can help save your sanity, either by outlawing tabs in favor of spaces or by displaying spaces and tabs differently. As a last resort, you can translate tabs to spaces with the **expand** command.

Objects, strings, numbers, lists, dictionaries, tuples, and files

All data types in Python are objects, and this gives them more power and flexibility than they have in most languages.

In Python, lists are enclosed in square brackets and indexed from zero. They are essentially similar to arrays, but can hold objects of any type. (A homogeneous and more efficient array type is implemented in the `array` module, but for most purposes, stick with lists.)

Python also has "tuples," which are essentially immutable lists. Tuples are faster than lists and are helpful for representing constant data. The syntax for tuples is the same as for lists, except that the delimiters are parentheses instead of square brackets. Because `(thing)` looks like a simple algebraic expression, tuples that contain only a single element need a marker comma to disambiguate them: `(thing, )`.

Here's some basic variable and data type wrangling in Python:

```
name = 'Gwen'
rating = 10
characters = [ 'SpongeBob', 'Patrick', 'Squidward' ]
elements = ( 'lithium', 'carbon', 'boron' )

print("name:\t%s\nrating:\t%d" % (name, rating))
print("characters:\t%s" % characters)
print("hero:\t%s" % characters[0])
print("elements:\t%s" % (elements, ))
```

This example produces the following output:

```
$ python3 objects
name:       Gwen
rating:     10
characters: ['SpongeBob', 'Patrick', 'Squidward']
hero:       SpongeBob
elements:   ('lithium', 'carbon', 'boron')
```

Note that the default string conversion for list and tuple types represents them as they would be found in source code.

Variables in Python are not syntactically marked or declared by type, but the objects to which they refer do have an underlying type. In most cases, Python does not automatically convert types for you, but individual functions or operators may do so. For example, you cannot concatenate a string and a number (with the `+` operator) without explicitly converting the number to its string representation. However, formatting operators and statements coerce everything to string form.

Every object has a string representation, as can be seen in the output above. Dictionaries, lists, and tuples compose their string representations recursively by stringifying their constituent elements and combining these strings with the appropriate punctuation.

The string formatting operator `%` is a lot like the `sprintf` function from C, but it can be used anywhere a string can appear. It's a binary operator that takes the string on its left and the values to be inserted on its right. If more than one value is to be inserted, the values must be presented as a tuple.

A Python dictionary (also known as a hash or an associative array) represents a set of key/value pairs. You can think of a hash as an array whose subscripts (keys) are arbitrary values; they do not have to be numbers. But in practice, numbers and strings are common keys.

Dictionary literals are enclosed in curly braces, with each key/value pair being separated by a colon. In use, dictionaries work much like lists, except that the subscripts (keys) can be objects other than integers.

```
ordinal = { 1 : 'first', 2 : 'second', 3 : 'third' }
print("The ordinal dictionary contains", ordinal)
print("The ordinal of 1 is", ordinal[1])
```

```
$ python3 dictionary
The ordinal array contains {1: 'first', 2: 'second', 3: 'third'}
The ordinal of 1 is first
```

Python handles open files as objects with associated methods. True to its name, the `readline` method reads a single line, so the example below reads and prints two lines from the **/etc/passwd** file.

```
f = open('/etc/passwd', 'r')
print(f.readline(), end="")
print(f.readline(), end="")
f.close()
```

```
$ python3 fileio
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
```

The newlines at the end of the `print` calls are suppressed with `end=""` because each line already includes a newline character from the original file. Python does not automatically strip these.

Input validation example

Our scriptlet below shows a general scheme for input validation in Python. It also demonstrates the definition of functions and the use of command-line arguments, along with a couple of other Pythonisms.

```
import sys
import os

def show_usage(message, code = 1):
    print(message)
    print("%s: source_dir dest_dir" % sys.argv[0])
    sys.exit(code)

if len(sys.argv) != 3:
    show_usage("2 args required; you supplied %d" % (len(sys.argv) - 1))
elif not os.path.isdir(sys.argv[1]):
    show_usage("Invalid source directory")
elif not os.path.isdir(sys.argv[2]):
    show_usage("Invalid destination directory")

source, dest = sys.argv[1:3]

print("Source directory is", source)
print("Destination directory is", dest)
```

In addition to importing the `sys` module, we also import the `os` module to gain access to the `os.path.isdir` routine. Note that `import` doesn't shortcut your access to any symbols defined by modules; you must use fully qualified names that start with the module name.

The definition of the `show_usage` routine supplies a default value for the exit code in case the caller does not specify this argument explicitly. Since all data types are objects, function arguments are effectively passed by reference.

The `sys.argv` list contains the script name in the first position, so its length is one greater than the number of command-line arguments that were actually supplied. The form `sys.argv[1:3]` is a list slice. Curiously, slices do not include the element at the far end of the specified range, so this slice includes only `sys.argv[1]` and `sys.argv[2]`. You could simply say `sys.argv[1:]` to include the second and subsequent arguments.

Like **sh**, Python has a dedicated "else if" condition; the keyword is `elif`. There is no explicit case or switch statement.

The parallel assignment of the `source` and `dest` variables is a bit different from some languages in that the variables themselves are not in a list. Python allows parallel assignments in either form.

Python uses the same comparison operators for numeric and string values. The "not equal" comparison operator is `!=`, but there is no unary `!` operator; use `not` for this. The Boolean operators

and and or are also spelled out.

Loops

The fragment below uses a for…in construct to iterate through the range 1 to 10.

```
for counter in range(1, 10):
    print(counter, end=" ")
print()                       # Add final newline
```

As with the array slice in the previous example, the right endpoint of the range is not actually included. The output includes only the numbers 1 through 9:

```
1 2 3 4 5 6 7 8 9
```

This is Python's only type of for loop, but it's a powerhouse. Python's for has several features that distinguish it from for in other languages:

- Nothing is special about numeric ranges. Any object can support Python's iteration model, and most common objects do. You can iterate through a string (by character), a list, a file (by character, line, or block), a list slice, etc.
- Iterators can yield multiple values, and you can have multiple loop variables. The assignment at the top of each iteration acts just like Python's regular multiple assignments. This feature is particularly nice for iterating through dictionaries.
- Both for and while loops can have else clauses at the end. The else clause is executed only if the loop terminates normally, as opposed to exiting through a break statement. This feature may initially seem counterintuitive, but it handles certain use cases quite elegantly.

The example script below accepts a regular expression on the command line and matches it against a list of Snow White's dwarves and the colors of their dwarf suits. The first match is printed, with the portions that match the regex surrounded by underscores.

```
import sys
import re

suits = {
    'Bashful':'yellow', 'Sneezy':'brown', 'Doc':'orange', 'Grumpy':'red',
    'Dopey':'green', 'Happy':'blue', 'Sleepy':'taupe'
}
pattern = re.compile("(%s)" % sys.argv[1])

for dwarf, color in suits.items():
    if pattern.search(dwarf) or pattern.search(color):
        print("%s's dwarf suit is %s." %
            (pattern.sub(r"_\1_", dwarf), pattern.sub(r"_\1_", color)))
        break
else:
    print("No dwarves or dwarf suits matched the pattern.")
```

Here's some sample output:

```
$ python3 dwarfsearch '[aeiou]{2}'
Sl_ee_py's dwarf suit is t_au_pe.

$ python3 dwarfsearch 'ga|gu'
No dwarves or dwarf suits matched the pattern.
```

The assignment to `suits` demonstrates Python's syntax for encoding literal dictionaries. The `suits.items()` method is an iterator for key/value pairs—note that we're extracting both a dwarf name and a suit color on each iteration. If you wanted to iterate through only the keys, you could just say `for dwarf in suits`.

Python implements regular expression handling through its `re` module. No regex features are built into the language itself, so regex-wrangling with Python is a bit clunkier than with, say, Perl. Here, the regex `pattern` is initially compiled from the first command-line argument surrounded by parentheses (to form a capture group). Strings are then tested and modified with the `search` and `sub` methods of the regex object. You can also call `re.search` et al. directly as functions, supplying the regex to use as the first argument.

The `\1` in the substitution string is a back-reference to the contents of the first capture group. The strange-looking `r` prefix that precedes the substitution string (`r"_\1_"`) suppresses the normal substitution of escape sequences in string constants (`r` stands for "raw"). Without this, the replacement pattern would consist of two underscores surrounding a character with numeric code 1.

One thing to note about dictionaries is that they have no defined iteration order. If you run the dwarf search a second time, you may well receive a different answer:

```
$ python3 dwarfsearch '[aeiou]{2}'
Dopey's dwarf suit is gr_ee_n.
```

7.6 RUBY PROGRAMMING

Ruby, designed and maintained by Japanese developer Yukihiro “Matz” Matsumoto, shares many features with Python, including a pervasive “everything’s an object” approach. Although initially released in the mid-1990s, Ruby did not gain prominence until a decade later with the release of the Rails web development platform.

Ruby is still closely associated with the web in many people’s minds, but there’s nothing web-specific about the language itself. It works well for general-purpose scripting. However, Python is probably a better choice for a primary scripting language, if only because of its wider popularity.

Although Ruby is roughly equivalent to Python in many ways, it is philosophically more permissive. Ruby classes remain open for modification by other software, for example, and the Rubyist community attaches little or no shame to extensions that modify the standard library.

Ruby appeals to those with a taste for syntactic sugar, features that don’t really change the basic language but that permit code to be expressed more concisely and clearly. In the Rails environment, for example, the line

```
due_date = 7.days.from_now
```

creates a Time object without referencing the names of any time-related classes or doing any explicit date-and-time arithmetic. Rails defines `days` as an extension to Fixnum, the Ruby class that represents integers. This method returns a Duration object that acts like a number; used as a value, it’s equivalent to 604,800, the number of seconds in seven days. Inspected in the debugger, it describes itself as “7 days.” (This form of polymorphism is common to both Ruby and Python. It’s often called “duck typing”; if an object walks like a duck and quacks like a duck, you needn’t worry about whether it’s actually a duck.)

See Chapter 23 for more information about Chef and Puppet.

Ruby makes it easy for developers to create “domain-specific languages” (aka DSLs), mini-languages that are in fact Ruby but that read like specialized configuration systems. Ruby DSLs are used to configure both Chef and Puppet, for example.

Installation

Some systems have Ruby installed by default and some do not. However, it's always available as a package, often in several versions.

To date (version 2.3), Ruby has maintained relatively good compatibility with old code. In the absence of specific warnings, it's generally best to install the most recent version.

See this page for more about RVM.

Unfortunately, most systems' packages lag several releases behind the Ruby trunk. If your package library doesn't include the current release (check ruby-lang.org to determine what that is), install the freshest version through RVM; don't try to do it yourself.

Ruby quick start

Since Ruby is so similar to Python, here a perhaps-eerily-familiar look at some Ruby snippets modeled on those from the Python section earlier in this chapter.

```
#!/usr/bin/env ruby

print "Hello, world!\n\n"

name = 'Gwen'
rating = 10
characters = [ 'SpongeBob', 'Patrick', 'Squidward' ]
elements = { 3 => 'lithium', 7 => 'carbon', 5 => 'boron' }

print "Name:\t", name, "\nRating:\t", rating, "\n"
print "Characters:\t#{characters}\n"
print "Elements:\t#{elements}\n\n"

element_names = elements.values.sort!.map(&:upcase).join(', ')
print "Element names:\t", element_names, "\n\n"

elements.each do |key, value|
    print "Atomic number #{key} is #{value}.\n"
end
```

The output is as follows:

```
Hello, world!

Name:           Gwen
Rating:         10
Characters:     ["SpongeBob", "Patrick", "Squidward"]
Elements:       {3=>"lithium", 7=>"carbon", 5=>"boron"}

Element names:  BORON, CARBON, LITHIUM

Atomic number 3 is lithium.
Atomic number 7 is carbon.
Atomic number 5 is boron.
```

Like Python, Ruby uses brackets to delimit arrays and curly braces to delimit dictionary literals. (Ruby calls them “hashes.”) The `=>` operator separates each hash key from its corresponding value, and the key/value pairs are separated from each other by commas. Ruby does not have tuples.

Ruby’s `print` is a function (or more accurately, a global method), just like that of Python 3. However, if you want newlines, you must specify them explicitly. There’s also a `puts` function that adds newlines for you, but it’s perhaps a bit too smart. If you try to add an extra newline of your own, `puts` won’t insert its own newline.

The parentheses normally seen around the arguments of function calls are optional in Ruby. Developers don't normally include them unless they help to clarify or disambiguate the code. (Note that some of these calls to `print` do include multiple arguments separated by commas.)

In several cases, we've used `#{}` brackets to interpolate the values of variables into double-quoted strings. Such brackets can contain arbitrary Ruby code; whatever value the code produces is automatically converted to string type and inserted into the outer string. You can also concatenate strings with the `+` operator, but interpolation is typically more efficient.

The line that calculates `element_names` illustrates several more Ruby tropes:

```
element_names = elements.values.sort!.map(&:upcase).join(', ')
```

This is a series of method calls, each of which operates on the result returned by the previous method, much like a series of pipes in the shell. For example, `elements`' `values` method produces an array of strings, which `sort!` then orders alphabetically. This array's `map` method calls the `upcase` method on each element, then reassembles all the results back into a new array. Finally, `join` concatenates the elements of that array, interspersed with commas, to produce a string.

The bang at the end of `sort!` warns you that there's something to be wary of when using this method. It isn't significant to Ruby; it's just part of the method's name. In this case, the issue of note is that `sort!` sorts the array in place. There's also a `sort` method (without the `!`) that returns the elements in a new, sorted array.

Blocks

In the previous code example, the text between `do` and `end` is a block, also commonly known in other languages as a lambda function, a closure, or an anonymous function:

```
elements.each do |key, value|
    print "Atomic number #{key} is #{value}.\n"
end
```

Ruby actually has three entities of this general type, known as blocks, procs, and lambdas. The differences among them are subtle and not important for this overview. The block above takes two arguments, which it calls `key` and `value`. It prints the values of both.

`each` looks like it might be a language feature, but it's just a method defined by hashes. `each` accepts the block as an argument and calls it once for each key/value pair the hash contains. This type of iteration function used in combination with a block is highly characteristic of Ruby code. `each` is the standard name for generic iterators, but many classes define more specific versions such as `each_line` or `each_character`.

Ruby has a second syntax for blocks that uses curly braces instead of `do...end` as delimiters. It means exactly the same thing, but it looks more at home as part of an expression. For example,

```
characters.map {|c| c.reverse}  # ["boBegnopS", "kcirtaP", "drawdiuqS"]
```

This form is functionally identical to `characters.map(&:reverse)`, but instead of just telling `map` what method to call, we included an explicit block that calls the `reverse` method.

The value of a block is the value of the last expression it evaluates before completing. Conveniently, pretty much everything in Ruby is an expression (meaning "a piece of code that can be evaluated to produce a value"), including control structures such as `case` (analogous to what most languages call `switch`) and `if-else`. The values of these expressions mirror the value produced by whichever case or branch was activated.

Blocks have many uses other than iteration. They let a single function perform both setup and takedown procedures on behalf of another section of code, so they often represent multi-step operations such as database transactions or filesystem operations.

For example, the following code opens the **/etc/passwd** file and prints out the line that defines the root account:

```
open '/etc/passwd', 'r' do |file|
    file.each_line do |line|
        print line if line.start_with? 'root:'
    end
end
```

The open function opens the file and passes its IO object to the outer block. Once the block has finished running, open automatically closes the file. There's no need for a separate close operation (although it does exist if you want to use it), and the file is closed no matter how the outer block terminates.

The postfix if construct used here might be familiar to those who have used Perl. It's a nice way to express simple conditionals without obscuring the primary action. Here, it's clear at a glance that the inner block is a loop that prints out some of the lines.

In case the structure of that print line is not clear, here it is again with the optional parentheses included. The if has the lowest precedence, and it has a single method call on either side:

```
print(line) if line.start_with?('root:')
```

As with the sort! method we saw on this page, the question mark is just a naming convention for methods that return Boolean values.

The syntax for defining a named function is slightly different from that for a block:

```
def show_usage(msg = nil)
    STDERR.puts msg if msg
    STDERR.puts "Usage: #{$0} filename ..."
    exit 1
end
```

The parentheses are still optional, but in practice, they are always shown in this context unless the function takes no arguments. Here, the msg argument defaults to nil.

The global variable $0 is magic and contains the name by which the current program was invoked. (Traditionally, this would be the first argument of the argv array. But the Ruby convention is that ARGV contains only actual command-line arguments.)

As in C, you can treat non-Boolean values as if they were Booleans, as illustrated here in the form of if msg. The Ruby mapping for this conversion is a bit unusual, though: everything except nil and false counts as true. In particular, 0 is true. (In practice, this usually ends up being what you want.)

Symbols and option hashes

Ruby makes extensive use of an uncommon data type called a symbol, denoted with a colon, e.g., :example. You can think of symbols as immutable strings. They're commonly used as labels or as well-known hash keys. Internally, Ruby implements them as numbers, so they're fast to hash and compare.

Symbols are so commonly used as hash keys that Ruby 2.0 defined an alternative syntax for hash literals to reduce the amount of punctuation clutter. The standard-form hash

```
h = { :animal => 'cat', :vegetable => 'carrot', :mineral => 'zeolite' }
```

can be written in Ruby 2.0 style as

```
h = { animal: 'cat', vegetable: 'carrot', mineral: 'zeolite' }
```

Outside of this hash literal context, symbols retain their : prefixes wherever they appear in the code. For example, here's how to get specific values back out of a hash:

```
healthy_snack = h[:vegetable]   # 'carrot'
```

Ruby has an idiosyncratic but powerful convention for handling options in function calls. If a called function requests this behavior, Ruby collects trailing function-call arguments that resemble hash pairs into a new hash. It then passes that hash to the function as an argument. For example, in the Rails expression

```
file_field_tag :upload, accept: 'application/pdf', id: 'commentpdf'
```

the file_field_tag receives only two arguments: the :upload symbol, and a hash containing the keys :accept and :id. Because hashes have no inherent order, it doesn't matter in what order the options appear.

This type of flexible argument processing is a Ruby standard in other ways, too. Ruby libraries, including the standard library, generally do their best to accept the broadest possible range of inputs. Scalars, arrays, and hashes are often equally valid arguments, and many functions can be called with or without blocks.

Regular expressions in Ruby

Unlike Python, Ruby has a little bit of language-side sugar to help you deal with regular expressions. Ruby supports the traditional `/.../` notation for regular expression literals, and the contents can include `#{}` escape sequences, much like double-quoted strings.

Ruby also defines the `=~` operator (and its negation, `!~`) to test for a match between a string and a regular expression. It evaluates either to the index of the first match or to `nil` if there is no match.

```
"Hermann Hesse" =~ /H[aeiou]/   # => 0
```

To access the components of a match, explicitly invoke the regular expression's `match` method. It returns either `nil` (if no match) or an object that can be accessed as an array of components.

```
if m = /(^H\w*)\s/.match("Heinrich Hoffmeyer headed this heist")
  puts m[0]   # 'Heinrich'
end
```

Here's a look at a Ruby version of the dwarf-suit example from this page:

```
suits = {
    Bashful: 'yellow', Sneezy: 'brown', Doc: 'orange', Grumpy: 'red',
    Dopey: 'green', Happy: 'blue', Sleepy: 'taupe'
}

abort "Usage: #{$0} pattern" unless ARGV.size == 1
pat = /(#{ARGV[0]})/

matches = suits.lazy.select {|dwarf, color| pat =~ dwarf || pat =~ color}

if matches.any?
    dwarf, color = matches.first
    print "%s\'s dwarf suit is %s.\n" %
        [ dwarf.to_s.sub(pat, '_\1_'), color.sub(pat, '_\1_') ]
else
    print "No dwarves or dwarf suits matched the pattern.\n"
end
```

The `select` method on a collection creates a new collection that includes only the elements for which the supplied block evaluates to true. In this case, `matches` is a new hash that includes only pairs for which either the key or the value matches the search pattern. Since we made the starting hash `lazy`, the filtering won't actually occur until we try to extract values from the result. In fact, this code checks only as many pairs as are needed to find a match.

Did you notice that the `=~` pattern-matching operator was used on the symbols that represent the dwarves' names? It works because `=~` is smart enough to convert the symbols to strings before matching. Unfortunately, we have to perform the conversion explicitly (with the `to_s` method)

when applying the substitution pattern; `sub` is only defined on strings, so we need a real, live string on which to call it.

Note also the parallel assignment of `dwarf` and `color`. `matches.first` returns a two-element array, which Ruby automatically unpacks.

The `%` operator for strings works similarly to the same operator in Python; it's the Ruby version of `sprintf`. Here there are two components to fill in, so we pass in the values as a two-element array.

Ruby as a filter

You can use Ruby without a script by putting isolated expressions on the command line. This is an easy way to do quick text transformations (though truth be told, Perl is still much better at this role).

Use the **-p** and **-e** command-line options to loop through STDIN, run a simple expression on each line (represented as the variable `$_`), and print the result. For example, the following command translates **/etc/passwd** to upper case:

```
$ ruby -pe '$_.tr!("a-z", "A-Z")' /etc/passwd
NOBODY:*:-2:-2:UNPRIVILEGED USER:/VAR/EMPTY:/USR/BIN/FALSE
ROOT:*:0:0:SYSTEM ADMINISTRATOR:/VAR/ROOT:/BIN/SH
...
```

ruby -a turns on autosplit mode, which separates input lines into fields that are stored in the array named `$F`. Whitespace is the default field separator, but you can set another separator pattern with the **-F** option.

Autosplit is handy to use in conjunction with **-p** or its nonautoprinting variant, **-n**. The command below uses **ruby -ane** to produce a version of the **passwd** file that includes only usernames and shells.

```
$ ruby -F: -ane 'print $F[0], ":", $F[-1]' /etc/passwd
nobody:/usr/bin/false
root:/bin/sh
...
```

The truly intrepid can use **-i** in conjunction with **-pe** to edit files in place; Ruby reads in the files, presents their lines for editing, and saves the results out to the original files. You can supply a pattern to **-i** that tells Ruby how to back up the original version of each file. For example, **-i.bak** backs up **passwd** as **passwd.bak**. Beware—if you don't supply a backup pattern, you don't get backups at all. Note that there's no space between the **-i** and the suffix.

7.7 Library and environment management for Python and Ruby

Languages have many of the same packaging and version control issues that operating systems do, and they often resolve them in analogous ways. Python and Ruby are similar in this area, so we discuss them together in this section.

Finding and installing packages

The most basic requirement is some kind of easy and standardized way to discover, obtain, install, update, and distribute add-on software. Both Ruby and Python have centralized warehouses for this purpose, Ruby's at rubygems.org and Python's at pypi.python.org.

In the Ruby world, packages are called "gems," and the command that wrangles them is called **gem** as well. **gem search** *regex* shows the available gems with matching names, and **gem install** *gem-name* downloads and installs a gem. You can use the **--user-install** option to install a private copy instead of modifying the system's complement of gems.

The Python equivalent is called **pip** (or **pip2** or **pip3**, depending on which Python versions are installed). Not all systems include **pip** by default. Those that don't typically make it available as a separate (OS-level) package. As with **gem**, **pip search** and **pip install** are the mainstay commands. A **--user** option installs packages into your home directory.

Both **gem** and **pip** understand dependencies among packages, at least at a basic level. When you install a package, you're implicitly asking for all the packages it depends on to be installed as well (if they are not already present).

In a basic Ruby or Python environment, only a single version of a package can be installed at once. If you reinstall or upgrade a package, the old version is removed.

You often have the choice to install a gem or **pip** package through the standard language mechanism (**gem** or **pip**) or through an OS-level package that's stocked in your vendor's standard repository. OS packages are more likely to be installed and run without issues, but they are less likely to be up to date. Neither option is clearly superior.

Creating reproducible environments

Programs, libraries, and languages develop complex webs of dependencies as they evolve together over time. A production-level server might depend on tens or hundreds of these components, each of which has its own expectations about the installation environment. How do you identify which combination of library versions will create a harmonious environment? How do you make sure the configuration you tested in the development lab is the same one that gets deployed to the cloud? More basically, how do you make sure that managing all these parts isn't a big hassle?

Both Python and Ruby have a standardized way for packages to express their dependencies. In both systems, package developers create a text file at the root of the project that lists its dependencies. For Ruby, the file is called **Gemfile**, and for Python, **requirements.txt**. Both formats support flexible version specifications for dependencies, so it's possible for packages to declare that they're compatible with "any release of **simplejson** version 3 or higher" or "Rails 3, but not Rails 4." It's also possible to specify an exact version requirement for any dependency.

Both file formats allow a source to be specified for each package, so dependencies need not be distributed through the language's standard package warehouse. All common sources are supported, from web URLs to local files to GitHub repositories.

You install a batch of Python dependencies with **pip install -r requirements.txt**. Although **pip** does a fine job of resolving individual version specifications, it's unfortunately not able to solve complex dependency relationships among packages on its own. Developers sometimes have to tweak the order in which packages are mentioned in the **requirements.txt** file to achieve a satisfactory result. It's also possible, though uncommon, for new package releases to disturb the version equilibrium.

pip freeze prints out Python's current package inventory in **requirements.txt** format, specifying an exact version for each package. This feature can be helpful for replicating the current environment on a production server.

In the Ruby world, **gem install -g Gemfile** is a fairly direct analog of **pip -r**. In most circumstances, though, it's better to use the Bundler gem to manage dependencies. Run **gem install bundler** to install it (if it's not already on the system), then run **bundle install** from the root directory of the project you're setting up. (Ruby gems can include shell-level commands. They don't typically have man pages, though; run **bundle help** for details, or see bundler.io for complete documentation.)

Bundler has several nice tricks up its sleeve:

- It does true recursive dependency management, so if there's a set of gems that are mutually compatible and that satisfy all constraints, Bundler can find it without help.

- It automatically records the results of version calculations in a file called **Gemfile.lock**. Maintaining this context information lets Bundler handle updates to the **Gemfile** conservatively and efficiently. Bundler modifies only the packages it needs to when migrating to a new version of the **Gemfile**.
- Because **Gemfile.lock** is sticky in this way, running **bundle install** on a deployment server automatically reproduces the package environment found in the development environment. (Or at least, that's the default behavior. It's easy to specify different requirements for development and deployment environments in the **Gemfile** if you need to.)
- In deployment mode (**bundle install --deployment**), Bundler installs missing gems into the local project directory, helping isolate the project from any future changes to the system's package complement. You can then use **bundle exec** to run specific commands within this hybrid gem environment. Some software packages, such as Rails, are Bundler-aware and will use the locally installed packages even without a **bundle exec** command.

Multiple environments

pip and **bundle** handle dependency management for individual Python and Ruby programs, but what if two programs on the same server have conflicting requirements? Ideally, every program in a production environment would have its own library environment that was independent of the system and of all other programs.

__virtualenv__: virtual environments for Python

Python's **virtualenv** package creates virtual environments that live within their own directories. (As with other Python-related commands, there are numeric-suffixed versions of the **virtualenv** command that go with particular Python versions.) After installing the package, just run the **virtualenv** command with a pathname to set up a new environment:

```
$ virtualenv myproject
New python executable in /home/ulsah/myproject/bin/python
Installing setuptools, pip, wheel...done.
```

Each virtual environment has a **bin/** directory that includes binaries for Python and PIP. When you run one of those binaries, you're automatically placed in the corresponding virtual environment. Install packages into the environment as usual by running the virtual environment's copy of **pip**.

To start a virtualized Python program from **cron** or from a system startup script, explicitly specify the path to the proper copy of **python**. (Alternatively, put the path in the script's shebang line.)

When working interactively in the shell, you can **source** a virtual environment's **bin/activate** script to set the virtual environment's versions of **python** and **pip** as the defaults. The script rearranges your shell's PATH variable. Use **deactivate** to leave the virtual environment.

Virtual environments are tied to specific versions of Python. At the time a virtual environment is created, you can set the associated Python binary with **virtualenv**'s **--python** option. The Python binary must already be installed and functioning.

RVM: the Ruby enVironment Manager

Things are similar in the Ruby world, but somewhat more configurable and more complicated. You saw on this page that Bundler can cache local copies of Ruby gems on behalf of a specific application. This is a reasonable approach when moving projects into production, but it isn't so great for interactive use. It also assumes that you want to use the system's installed version of Ruby.

Those who want a more general solution should investigate RVM, a complex and rather unsightly environment virtualizer that uses a bit of shell hackery. To be fair, RVM is an extremely polished example of the "unsightly hack" genus. In practice, it works smoothly.

RVM manages both Ruby versions and multiple gem collections, and it lets you switch among all these on the fly. For example, the command

```
$ rvm ruby-2.3.0@ulsah
```

activates Ruby version 2.3.0 and the gemset called ulsah. References to **ruby** or **gem** now resolve to the specified versions. This magic also works for programs installed by gems, such as **bundle** and **rails**. Best of all, gem management is unchanged; just use **gem** or **bundle** as you normally would, and any newly installed gems automatically end up in the right place.

RVM's installation procedure involves fetching a Bash script from the web and executing it locally. Currently, the commands are

```
$ curl -o /tmp/install -sSL https://get.rvm.io
$ sudo bash /tmp/install stable
```

but check rvm.io for the current version and a cryptographic signature. (Also see this page for some comments on why our example commands don't exactly match RVM's recommendations.) Be sure to install with **sudo** as shown here; if you don't, RVM sets up a private environment in your home directory. (That works fine, but nothing on a production system should refer to your home directory.) You'll also need to add authorized RVM users to the rvm UNIX group.

After the initial RVM installation, don't use **sudo** when installing gems or changing RVM configurations. RVM controls access through membership in the rvm group.

Under the covers, RVM does its magic by manipulating the shell's environment variables and search path. Ergo, it has to be invited into your environment like a vampire by running some shell startup code at login time. When you install RVM at the system level, RVM drops an **rvm.sh** scriptlet with the proper commands into **/etc/profile.d**. Some shells automatically run this stub. Those that don't just need an explicit `source` command, which you can add to your shell's startup files:

```
source /etc/profile.d/rvm.sh
```

RVM doesn't modify the system's original Ruby installation in any way. In particular, scripts that start with a

```
#!/usr/bin/ruby
```

shebang continue to run under the system's default Ruby and to see only system-installed gems. The following variant is more liberal:

```
#!/usr/bin/env ruby
```

It locates the **ruby** command according to the RVM context of the user that runs it.

rvm install installs new versions of Ruby. This RVM feature makes it quite painless to install different versions of Ruby, and it should generally be used in preference to your OS's native Ruby packages, which are seldom up to date. **rvm install** downloads binaries if they are available. If not, it installs the necessary OS packages and then builds Ruby from source code.

Here's how we might set up for deployment a Rails application known to be compatible with Ruby 2.2.1:

```
$ rvm install ruby-2.2.1
Searching for binary rubies, this might take some time.
No binary rubies available for: ubuntu/15.10/x86_64/ruby-2.2.1.
Continuing with compilation. Please read 'rvm help mount' to get more
    information on binary rubies.
Checking requirements for ubuntu.
Installing required packages: gawk, libreadline6-dev, zlib1g-dev,
    libncurses5-dev, automake, libtool, bison, libffi-dev.............
Requirements installation successful.
Installing Ruby from source to: /usr/local/rvm/rubies/ruby-2.2.1, this
    may take a while depending on your cpu(s)...
...
```

If you installed RVM as described above, the Ruby system is installed underneath **/usr/local/rvm** and is accessible to all accounts on the system.

Use **rvm list known** to find out which versions of Ruby RVM knows how to download and build. Rubies shown by **rvm list** have already been installed and are available for use.

```
$ cd myproject.rails
$ rvm ruby-2.2.1@myproject --create --default --ruby-version
ruby-2.2.1 - #gemset created /usr/local/rvm/gems/ruby-2.2.1@myproject
ruby-2.2.1 - #generating myproject wrappers.........
$ gem install bundler
Fetching: bundler-1.11.2.gem (100%)
Successfully installed bundler-1.11.2
1 gem installed
$ bundle
Fetching gem metadata from https://rubygems.org/..........
Fetching version metadata from https://rubygems.org/...
Fetching dependency metadata from https://rubygems.org/..
Resolving dependencies......
...
```

The **ruby-2.2.1@myproject** line specifies both a Ruby version and a gemset. The **--create** flag creates the gemset if it doesn't already exist. **--default** makes this combination your RVM default, and **--ruby-version** writes the names of the Ruby interpreter and gemset to **.ruby-version** and **.ruby-gemset** in the current directory.

If the **.*-version** files exist, RVM automatically reads and honors them when dealing with scripts in that directory. This feature allows each project to specify its own requirements and frees you from the need to remember what goes with what.

To run a package in its requested environment (as documented by **.ruby-version** and **.ruby-gemset**), run the command

```
rvm in /path/to/dir do startup-cmd startup-arg ...
```

This is a handy syntax to use when running jobs out of startup scripts or **cron**. It doesn't depend on the current user having set up RVM or on the current user's RVM configuration.

Alternatively, you can specify an explicit environment for the command, as in

```
rvm ruby-2.2.1@myproject do startup-cmd startup-arg ...
```

Yet a third option is to run a ruby binary from within a wrapper maintained by RVM for this purpose. For example, running

```
/usr/local/rvm/wrappers/ruby-2.2.1@myproject/ruby ...
```

automatically transports you into the Ruby 2.2.1 world with the myproject gemset.

7.8 Revision control with Git

Mistakes are a fact of life. It's important to keep track of configuration and code changes so that when these changes cause problems, you can easily revert to a known-good state. Revision control systems are software tools that track, archive, and grant access to multiple revisions of files.

Revision control systems address several problems. First, they define an organized way to trace the history of modifications to a file such that changes can be understood in context and so that earlier versions can be recovered. Second, they extend the concept of versioning beyond the level of individual files. Related groups of files can be versioned together, taking into account their interdependencies. Finally, revision control systems coordinate the activities of multiple editors so that race conditions cannot cause anyone's changes to be permanently lost and so that incompatible changes from multiple editors do not become active simultaneously.

By far the most popular system in use today is Git, created by the one and only Linus Torvalds. Linus created Git to manage the Linux kernel source code because of his frustration with the version control systems that existed at the time. It is now as ubiquitous and influential as Linux. It's difficult to tell which of Linus's inventions has had a greater impact on the world.

See this page for more information about DevOps.

Most modern software is developed with help from Git, and as result, administrators encounter it daily. You can find, download, and contribute to open source projects on GitHub, GitLab, and other social development sites. You can also use Git to track changes to scripts, configuration management code, templates, and any other text files that need to be tracked over time. We use Git to track the contents of this book. It's well suited to collaboration and sharing, making it an essential tool for sites that embrace DevOps.

Git's shtick is that it has no distinguished central repository. To access a repository, you clone it (including its entire history) and carry it around with you like a hermit crab lugging its shell. Your commits to the repository are local operations, so they're fast and you don't have to worry about communicating with a central server. Git uses an intelligent compression system to reduce the cost of storing the entire history, and in most cases this system is quite effective.

Git is great for developers because they can pile their source code onto a laptop and work without being connected to a network while still reaping all the benefits of revision control. When the time comes to integrate multiple developers' work, their changes can be integrated from one copy of the repository to another in any fashion that suits the organization's workflow. It's always possible to unwind two copies of a repository back to their common ancestor state, no matter how many changes and iterations have occurred after the split.

Git's use of a local repository is a big leap forward in revision control—or perhaps more accurately, it's a big leap backward, but in a good way. Early revision control systems such as RCS and CVS used local repositories but were unable to handle collaboration, change merging, and independent development. Now we've come full circle to a point where putting files under revision control is once again a fast, simple, local operation. At the same time, all Git's advanced collaboration features are available for use in situations that require them.

Git has hundreds of features and can be quite puzzling in advanced use. However, most Git users get by with only a handful of simple commands. Special situations are best handled by searching Google for a description of what you want to do (e.g., "git undo last commit"). The top result is invariably a Stack Overflow discussion that addresses your exact situation. Above all, *don't panic*. Even if it looks like you screwed up the repository and deleted your last few hours of work, Git very likely has a copy stashed away. You just need the reflog fairy to go and fetch it.

Before you start using Git, set your name and email address:

```
$ git config --global user.name "John Q. Ulsah"
$ git config --global user.email "ulsah@admin.com"
```

These commands create the ini-formatted Git config file **~/.gitconfig** if it doesn't already exist. Later **git** commands look in this file for configuration settings. Git power users make extensive customizations here to match their desired workflow.

A simple Git example

We've contrived for you a simple example repository for maintaining some shell scripts. In practice, you can use Git to track configuration management code, infrastructure templates, ad hoc scripts, text documents, static web sites, and anything else you need to work on over time.

The following commands create a new Git repository and populate its baseline:

```
$ pwd
/home/bwhaley
$ mkdir scripts && cd scripts
$ git init
Initialized empty Git repository in /home/bwhaley/scripts/.git/
$ cat > super-script.sh << EOF
> #!/bin/sh
> echo "Hello, world"
> EOF
$ chmod +x super-script.sh
$ git add .
$ git commit -m "Initial commit"
[master (root-commit) 9a4d90c] super-script.sh
 create mode 100755 super-script.sh
 1 file changed, 0 insertions(+), 0 deletions(-)
```

In the sequence above, **git init** creates the repository's infrastructure by creating a **.git** directory in **/home/bwhaley/scripts**. Once you set up an initial "hello, world" script, the command **git add .** copies it to Git's "index," which is a staging area for the upcoming commit.

The index is a not just a list of files to commit; it's a bona fide file tree that's every bit as real as the current working directory and the contents of the repository. Files in the index have contents, and depending on what commands you run, those contents may end up being different from both the repository and the working directory. **git add** really just means "**cp** from the working directory to the index."

git commit enters the contents of the index into the repository. Every commit needs a log message. The **-m** flag lets you include the message on the command line. If you leave it out, **git** starts up an editor for you.

Now make a change and check it into the repository.

```
$ vi super-script.sh
$ git commit super-script.sh -m "Made the script more super"
[master 67514f1] Made the script more super
 1 file changed, 1 insertions(+), 0 deletions(-)
```

Naming the modified files on the **git commit** command line bypasses Git's normal use of the index and creates a revision that includes only changes to the named files. The existing index

remains unchanged, and Git ignores any other files that may have been modified.

If a change involves multiple files, you have a couple of options. If you know exactly which files were changed, you can always list them on the command line as shown above. If you're lazy, you can run **git commit -a** to make Git add all modified files to the index before doing the commit. This last option has a couple of pitfalls, however.

First, there may be modified files that you don't want to include in the commit. For example, if **super-script.sh** had a config file and you had modified that config file for debugging, you might not want to commit the modified file back to the repository.

The second issue is that **git commit -a** picks up only changes to files that are currently under revision control. It does not pick up new files that you may have created in the working directory.

For an overview of Git's state, you can run **git status**. This command informs you of new files, modified files, and staged files all at once. For example, suppose that you added **more-scripts/another-script.sh**. Git might show the following:

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   super-script.sh

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    more-scripts/
    tmpfile

no changes added to commit (use "git add" and/or "git commit -a")
```

another-script.sh is not listed by name because Git doesn't yet see beneath the **more-scripts** directory that contains it. You can see that **super-script.sh** has been modified, and you can also see a spurious **tmpfile** that probably shouldn't be included in the repository. You can run **git diff super-script.sh** to see the changes made to the script. **git** helpfully suggests commands for the next operations you may want to perform.

Suppose you want to track the changes to **super-script.sh** separately from your new **another-script.sh**.

```
$ git commit super-script.sh -m "The most super change yet"
Created commit 6f7853c: The most super change yet
 1 files changed, 1 insertions(+), 0 deletions(-)
```

To eradicate **tmpfile** from Git's universe, create or edit a **.gitignore** file and put the filename inside it. This makes Git ignore the **tmpfile** now and forever. Patterns work, too.

```
$ echo tmpfile >> .gitignore
```

Finally, commit all the outstanding changes:

```
$ git add .
$ sudo git commit -m "Ignore tmpfile; Add another-script.sh to the repo"
Created commit 32978e6: Ignore tmpfile; add another-script.sh to the repo
 2 files changed, 2 insertions(+), 0 deletions(-)
 create mode 100644 .gitignore
 create mode 100755 more-scripts/another-script.sh
```

Note that the **.gitignore** file itself becomes part of the managed set of files, which is usually what you want. It's fine to re-add files that are already under management, so **git add .** is an easy way to say "I want to make the new repository image look like the working directory minus anything listed in **.gitignore**." You couldn't just do a **git commit -a** in this situation because that would pick up neither **another-script.sh** nor **.gitignore**; these files are new to Git and so must be explicitly added.

Git caveats

In an effort to fool you into thinking that it manages files' permissions as well as their contents, Git shows you file modes when adding new files to the repository. It's lying; Git does not track modes, owners, or modification times.

Git *does* track the executable bit. If you commit a script with the executable bit set, any future clones will also be executable. But don't expect Git to track ownership or read-only status. A corollary is that you can't count on using Git to recover complex file hierarchies in situations where ownerships and permissions are important.

Another corollary is that you should never include plain text passwords or other secrets in a Git repository. Not only are they open to inspection by anyone with access to the repository, but they may also be inadvertently unpacked in a form that's accessible to the world.

Social coding with Git

The emergence and rapid growth of social development sites such as GitHub and GitLab is one of the most important trends in recent computing history. Millions of open source software projects are built and managed transparently by huge communities of developers who use every conceivable language. Software has never been easier to create and distribute.

GitHub and GitLab are, in essence, hosted Git repositories with a lot of added features that relate to communication and workflow. Anyone can create a repository. Repositories are accessible both through the **git** command and on the web. The web UI is friendly and offers features to support collaboration and integration.

The social coding experience can be somewhat intimidating for neophytes, but in fact it isn't complicated once some basic terms and methodology are understood.

- "master" is the default name assigned to the first branch in a new repository. Most software projects use this default as their main line of development, although some may not have a master branch at all. The master branch is usually managed to contain current but functional code; bleeding-edge development happens elsewhere. The latest commit is known as the tip or head of the master branch.
- On GitHub, a fork is a snapshot of a repository at a specific point in time. Forks happen when a user doesn't have permission to modify the main repository but wants to make changes, either for future integration with the primary project or to create an entirely separate development path.
- A pull request is a request to merge changes from one branch or fork to another. They're read by the maintainers of the target project and can be accepted to incorporate code from other users and developers. Every pull request is also a discussion thread, so both principals and kibitzers can comment on prospective code updates.
- A committer or maintainer is an individual who has write access to a repository. For large open source projects, this highly coveted status is given only to trusted developers who have a long history of contributions.

You'll often land in a GitHub or GitLab repository when trying to locate or update a piece of software. Make sure you're looking at the trunk repository and not some random person's fork. Looked for a "forked from" indication and follow it.

Be cautious when evaluating new software from these sites. Below are a few questions to ponder before rolling out a random piece of new software at your site:

- How many contributors have participated in development?

- Does the commit history indicate recent, regular development?
- What is the license, and is it compatible with your organization's needs?
- What language is the software written in, and do you know how to manage it?
- Is the documentation complete enough for effective use of the software?

Most projects have a particular branching strategy that they rely on to track changes to the software. Some maintainers insist on rigorous enforcement of their chosen strategy, and others are more lenient. One of the most widely used is the Git Flow model developed by Vincent Driessen; see goo.gl/GDaF for details. Before contributing to a project, familiarize yourself with its development practices to help out the maintainers.

Above all, remember that open source developers are often unpaid. They appreciate your patience and courtesy when engaging through code contributions or opening support issues.

7.9 Recommended reading

Brooks, Frederick P., Jr. *The Mythical Man-Month: Essays on Software Engineering.* Reading, MA: Addison-Wesley, 1995.

Chacon, Scott, and Straub, Ben. *Pro Git, 2nd edition. 2014.* git-scm.com/book/en/v2
The complete Pro Git book, released for free under a Creative Commons license.

Shells and shell scripting

ROBBINS, ARNOLD, AND NELSON H. F. BEEBE. *Classic Shell Scripting.* Sebastopol, CA: O'Reilly Media, 2005. This book addresses the traditional (and portable) Bourne shell dialect. It also includes quite a bit of good info on **sed** and **awk**.

POWERS, SHELLEY, JERRY PEEK, TIM O'REILLY, AND MIKE LOUKIDES. *Unix Power Tools, (3rd Edition)*, Sebastopol, CA: O'Reilly Media, 2002. This classic UNIX book covers a lot of ground, including **sh** scripting and various feats of command-line-fu. Some sections are not aging gracefully, but the shell-related material remains relevant.

SOBELL, MARK G. *A Practical Guide to Linux Commands, Editors, and Shell Programming.* Upper Saddle River, NJ: Prentice Hall, 2012. This book is notable for its inclusion of **tcsh** as well as **bash**.

SHOTTS, WILLIAM E., JR. *The Linux Command Line: A Complete Introduction.* San Francisco, CA: No Starch Press, 2012. This book is specific to **bash**, but it's a nice combination of interactive and programming material, with some extras thrown in. Most of the material is relevant to UNIX as well as Linux.

BLUM, RICHARD, AND CHRISTINE BRESNAHAN. *Linux Command Line and Shell Scripting Bible (3rd Edition).* Indianapolis, IN: John Wiley & Sons, Inc. 2015. This book focuses a bit more specifically on the shell than does the Shotts book, though it's also **bash**-specific.

COOPER, MENDEL. *Advanced Bash-Scripting Guide.* www.tldp.org/LDP/abs/html. A free and very good on-line book. Despite the title, it's safe and appropriate for those new to **bash** as well. Includes lots of good example scripts.

Regular expressions

FRIEDL, JEFFREY. *Mastering Regular Expressions (3rd Edition)*, Sebastopol, CA: O'Reilly Media, 2006.

GOYVAERTS, JAN, AND STEVEN LEVITHAN. *Regular Expressions Cookbook*. Sebastopol, CA: O'Reilly Media, 2012.

GOYVAERTS, JAN. regular-expressions.info. A detailed on-line source of information about regular expressions in all their various dialects.

KRUMINS, PETERIS. *Perl One-Liners: 130 Programs That Get Things Done.* San Francisco, CA: No Starch Press, 2013.

Python

Sweigart, Al. *Automate the Boring Stuff with Python: Practical Programming for Total Beginners.* San Francisco, CA: No Starch Press, 2015. This is an approachable introductory text for Python 3 and programming generally. Examples include common administrative tasks.

Pilgrim, Mark. *Dive Into Python.* Berkeley, CA: Apress, 2004. This classic book on Python 2 is also available for free on the web at diveintopython.net.

Pilgrim, Mark. *Dive Into Python 3*. Berkeley, CA: Apress, 2009. *Dive Into Python* updated for Python 3. Also available to read free on the web at diveintopython3.net.

Ramalho, Luciano. *Fluent Python.* Sebastopol, CA: O'Reilly Media, 2015. Advanced, idiomatic Python 3.

Beazley, David, and Brian K. Jones. *Python Cookbook (3rd Edition)*, Sebastopol, CA: O'Reilly Media, 2013. Covers Python 3.

Gift, Noah, and Jeremy M. Jones. *Python for Unix and Linux System Administrators*, Sebastopol, CA: O'Reilly Media, 2008.

Ruby

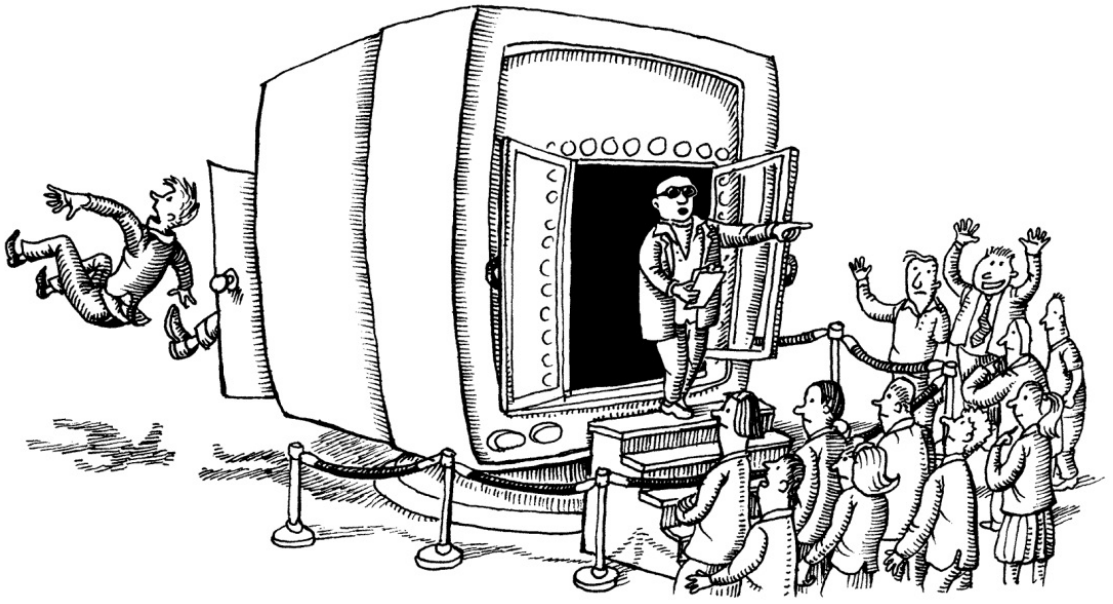
FLANAGAN, DAVID, AND YUKIHIRO MATSUMOTO. *The Ruby Programming Language.* Sebastopol, CA: O'Reilly Media, 2008. This classic, concise, and well-written summary of Ruby comes straight from the horse's mouth. It's relatively matter-of-fact and does not cover Ruby 2.0 and beyond; however, the language differences are minor.

BLACK, DAVID A. *The Well-Grounded Rubyist (2nd Edition).* Shelter Island, NY: Manning Publications, 2014. Don't let the title scare you off if you don't have prior Ruby experience; this is a good, all-around introduction to Ruby 2.1.

THOMAS, DAVE. *Programming Ruby 1.9 & 2.0: The Pragmatic Programmer's Guide (4th Edition)*. Pragmatic Bookshelf, 2013. Classic and frequently updated.

FULTON, HAL. *The Ruby Way: Solutions and Techniques in Ruby Programming (3rd Edition).* Upper Saddle River, NJ: Addison-Wesley, 2015. Another classic and up-to-date guide to Ruby, with a philosophical bent.

8 User Management



Modern computing environments span physical hardware, cloud systems, and virtual hosts. Along with the flexibility of this hybrid infrastructure comes an increasing need for centralized and structured account management. System administrators must understand both the traditional account model used by UNIX and Linux and the ways in which this model has been extended to integrate with directory services such as LDAP and Microsoft's Active Directory.

Account hygiene is a key determinant of system security. Infrequently used accounts are prime targets for attackers, as are accounts with easily guessed passwords. Even if you use your system's automated tools to add and remove users, it's important to understand the changes the tools are making. For this reason, we start our discussion of account management with the flat files you would modify to add users to a stand-alone machine. In later sections, we examine the higher-level user management commands that come with our example operating systems and the configuration files that control their behavior.

Most systems also have simple GUI tools for adding and removing users, but these tools don't usually support advanced features such as a batch mode or advanced localization. The GUI tools are simple enough that we don't think it's helpful to review their operation in detail, so in this chapter we stick to the command line.

This chapter focuses fairly narrowly on adding and removing users. Many topics associated with user management actually live in other chapters and are referenced here only indirectly. For example,

- Pluggable authentication modules (PAM) for password encryption and the enforcement of strong passwords are covered in Chapter 17, *Single Sign-On*. See the material starting on this page.
- Password vaults for managing passwords are described in Chapter 27, *Security* (see this page).
- Directory services such as OpenLDAP and Active Directory are outlined in Chapter 17, *Single Sign-On*, starting here.
- Policy and regulatory issues are major topics of Chapter 31, *Methodology, Policy, and Politics*.

8.1 ACCOUNT MECHANICS

A user is really nothing more than a number. Specifically, an unsigned 32-bit integer known as the user ID or UID. Almost everything related to user account management revolves around this number.

The system defines an API (through standard C library routines) that maps UID numbers back and forth into more complete sets of information about users. For example, **getpwuid()** accepts a UID as an argument and returns a corresponding record that includes information such as the associated login name and home directory. Likewise, **getpwnam()** looks up this same information by login name.

Traditionally, these library calls obtained their information directly from a text file, **/etc/passwd**. As time went on, they began to support additional sources of information such as network information databases (e.g., LDAP) and read-protected files in which encrypted passwords could be stored more securely.

---

*See this page for more details regarding the **nsswitch.conf** file.*

These layers of abstraction (which are often configured in the **nsswitch.conf** file) enable higher-level processes to function without direct knowledge of the underlying account management method in use. For example, when you log in as “dotty”, the logging-in process (window server, **login**, **getty**, or whatever) does a **getpwnam()** on dotty and then validates the password you supply against the encrypted passwd record returned by the library, regardless of its actual origin.

We start with the **/etc/passwd** file approach, which is still supported everywhere. The other options emulate this model in spirit if not in form.

8.2 THE /ETC/PASSWD FILE

/etc/passwd is a list of users recognized by the system. It can be extended or replaced by one or more directory services, so it's complete and authoritative only on stand-alone systems.

Historically, each user's encrypted password was also stored in the **/etc/passwd** file, which is world-readable. However, the onset of more powerful processors made it increasingly feasible to crack these exposed passwords. In response, UNIX and Linux moved the passwords to a separate file (**/etc/master.passwd** on FreeBSD and **/etc/shadow** on Linux) that is not world-readable. These days, the **passwd** file itself contains only a pro-forma entry to mark the former location of the password field (x on Linux and * on FreeBSD).

The system consults **/etc/passwd** at login time to determine a user's UID and home directory, among other things. Each line in the file represents one user and contains seven fields separated by colons:

- Login name
- Encrypted password placeholder (see this page)
- UID (user ID) number
- Default GID (group ID) number
- Optional "GECOS" information: full name, office, extension, home phone
- Home directory
- Login shell

For example, the following lines are all valid **/etc/passwd** entries:

```
root:x:0:0:The System,,x6096,:/:/bin/sh
jl:!:100:0:Jim Lane,ECOT8-3,,:/staff/jl:/bin/sh
dotty:x:101:20:: /home/dotty:/bin/tcsh
```

---

*See this page for more information about the **nsswitch.conf** file.*

If user accounts are shared through a directory service such as LDAP, you might see special entries in the **passwd** file that begin with + or -. These entries tell the system how to integrate the directory service's data with the contents of the **passwd** file. This integration can also be set up in the **/etc/nsswitch.conf** file.

The following sections discuss the **/etc/passwd** fields in more detail.

Login name

Login names (also known as usernames) must be unique and, depending on the operating system, may have character set restrictions. All UNIX and Linux flavors currently limit logins to 32 characters.

Login names can never contain colons or newlines, because these characters are used as field separators and entry separators in the **passwd** file, respectively. Depending on the system, other character restrictions may also be in place. Ubuntu is perhaps the most lax, as it allows logins starting with—or consisting entirely of—numbers and other special characters. For reasons too numerous to list, we recommend sticking with alphanumeric characters for logins, using lower case, and starting login names with a letter.

Login names are case sensitive. We are not aware of any problems caused by mixed-case login names, but lowercase names are traditional and also easier to type. Confusion could ensue if the login names john and John were different people.

Login names should be easy to remember, so random sequences of letters do not make good login names. Since login names are often used as email addresses, it's useful to establish a standard way of forming them. It should be possible for users to make educated guesses about each other's login names. First names, last names, initials, or some combination of these make reasonable naming schemes.

Keep in mind that some email systems treat addresses as being case insensitive, which is yet another good reason to standardize on lowercase login names. RFC5321 requires that the local portion of an address (that is, the part before the @ sign) be treated as case sensitive. The remainder of the address is handled according to the standards of DNS, which is case insensitive. Unfortunately, this distinction is subtle, and it is not universally implemented. Remember also that many legacy email systems predate the authority of the IETF.

Any fixed scheme for choosing login names eventually results in duplicate names, so you sometimes have to make exceptions. Choose a standard way of dealing with conflicts, such as adding a number to the end.

It's common for large sites to implement a full-name email addressing scheme (e.g., John.Q.Public@mysite.com) that hides login names from the outside world. This is a good idea, but it doesn't obviate any of the naming advice given above. If for no other reason than the sanity of administrators, it's best if login names have a clear and predictable correspondence to users' actual names.

Finally, a user should have the same login name on every machine. This rule is mostly for convenience, both yours and the user's.

Encrypted password

Historically, systems encrypted users' passwords with DES. As computing power increased, those passwords became trivial to crack. Systems then moved to hidden passwords and to MD5-based cryptography. Now that significant weaknesses have been discovered in MD5, salted SHA-512-based password hashes have become the current standard. See the *Guide to Cryptography* document at owasp.org for up-to-date guidance.

Our example systems support a variety of encryption algorithms, but they all default to SHA-512. You shouldn't need to update the algorithm choice unless you are upgrading systems from much older releases.


On FreeBSD, the default algorithm can be modified through the **/etc/login.conf** file.


On Debian and Ubuntu, the default was formerly managed through **/etc/login.defs**, but this practice has since been obsoleted by Pluggable Authentication Modules (PAM). Default password policies, including the hashing algorithm to use, can be found in **/etc/pam.d/common-passwd**.

RHEL 
On Red Hat and CentOS, the password algorithm can still be set in **/etc/login.defs** or through the **authconfig** command, as shown here:

```
$ sudo authconfig --passalgo=sha512 --update
```

Changing the password algorithm does not update existing passwords, so users must manually update their passwords before the new algorithm can take effect. To invalidate a user's password and force an update, use

```
$ chage -d 0 username
```

Password quality is another important issue. In theory, longer passwords are more secure, as are passwords that include a range of different character types (e.g., uppercase letters, punctuation marks, and numbers).

Most systems let you impose password construction standards on your users, but keep in mind that users can be adept at skirting these requirements if they find them excessive or burdensome. Table 8.1 shows the default standards used by our example systems.

Table 8.1: Password quality standards

System	Default requirements	Where set
Red Hat CentOS	8+ characters, complexity enforced	**/etc/login.defs** **/etc/security/pwquality.conf** **/etc/pam.d/system-auth**
Debian Ubuntu	6+ characters, complexity enforced	**/etc/login.defs** **/etc/pam.d/common-password**
FreeBSD	No constraints	**/etc/login.conf**

See this page for more comments on password selection.

Password quality requirements are a matter of debate, but we recommend that you prioritize length over complexity (see xkcd.com/comics/password_strength.png). Twelve characters is the minimal length for a future-proof password; note that this is significantly longer than any system's default. Your site may also have organization-wide standards for password quality. If it does, defer to those settings.

If you choose to bypass your system's tools for adding users and instead modify **/etc/passwd** by hand (by running the **vipw** command—see this page) to create a new account, put a * (FreeBSD) or an x (Linux) in the encrypted password field. This measure prevents unauthorized use of the account until you or the user has set a real password.

Encrypted passwords are of constant length (86 characters for SHA-512, 34 characters for MD5, and 13 characters for DES) regardless of the length of the unencrypted password. Passwords are encrypted in combination with a random "salt" so that a given password can correspond to many different encrypted forms. If two users happen to select the same password, this fact usually cannot be discovered by inspection of the encrypted passwords.

MD5-encrypted password fields in the shadow password file always start with 1 or $md5$. Blowfish passwords start with 2, SHA-256 passwords with 5, and SHA-512 passwords with 6.

UID (user ID) number

See this page for a description of the root account.

By definition, root has UID 0. Most systems also define pseudo-users such as bin and daemon to be the owners of commands or configuration files. It's customary to put such fake logins at the beginning of the **/etc/passwd** file and to give them low UIDs and a fake shell (e.g., **/bin/false**) to prevent anyone from logging in as those users.

To allow plenty of room for nonhuman users you might want to add in the future, we recommend that you assign UIDs to real users starting at 1000 or higher. (The desired range for new UIDs can be specified in the configuration files for **useradd**.) By default, our Linux reference systems start UIDs at 1000 and go up from there. FreeBSD starts the first user at UID 1001 and then adds one for each additional user.

Do not recycle UIDs, even when users leave your organization and you delete their accounts. This precaution prevents confusion if files are later restored from backups, where users may be identified by UID rather than by login name.

UIDs should be kept unique across your entire organization. That is, a particular UID should refer to the same login name and the same person on every machine that person is authorized to use. Failure to maintain distinct UIDs can result in security problems with systems such as NFS and can also result in confusion when a user moves from one workgroup to another.

It can be hard to maintain unique UIDs when groups of machines are administered by different people or organizations. The problems are both technical and political. The best solution is to have a central database or directory server that contains a record for each user and enforces uniqueness.

A simpler scheme is to assign each group within an organization its own range of UIDs and to let each group manage its own range. This solution keeps the UID spaces separate but does not address the parallel issue of unique login names. Regardless of your scheme, consistency of approach is the primary goal. If consistency isn't feasible, UID uniqueness is the second-best target.

The Lightweight Directory Access Protocol (LDAP) is a popular system for managing and distributing account information and works well for large sites. It is briefly outlined in this chapter starting here and is covered more thoroughly in Chapter 17, *Single Sign-On*, starting here.

Default GID (group ID) number

Like a UID, a group ID number is a 32-bit integer. GID 0 is reserved for the group called root, system, or wheel. As with UIDs, the system uses several predefined groups for its own housekeeping. Alas, there is no consistency among vendors. For example, the group “bin” has GID 1 on Red Hat and CentOS, GID 2 on Ubuntu and Debian, and GID 7 on FreeBSD.

In ancient times, when computing power was expensive, groups were used for accounting purposes so that the right department could be charged for your seconds of CPU time, minutes of login time, and kilobytes of disk used. Today, groups are used primarily to share access to files.

See this page for more information about setgid directories.

The **/etc/group** file defines the groups, with the GID field in **/etc/passwd** providing a default (or “effective”) GID at login time. The default GID is not treated specially when access is determined; it is relevant only to the creation of new files and directories. New files are normally owned by your effective group; to share files with others in a project group, you must manually change the files’ group owner.

To facilitate collaboration, you can set the setgid bit (02000) on a directory or mount filesystems with the **grpid** option. Both of these measures make newly created files default to the group of their parent directory.

GECOS field

The GECOS field is sometimes used to record personal information about each user. The field is a relic from a much earlier time when some early UNIX systems used General Electric Comprehensive Operating Systems for various services. It has no well-defined syntax. Although you can use any formatting conventions you like, conventionally, comma-separated GECOS entries are placed in the following order:

- Full name (often the only field used)
- Office number and building
- Office telephone extension
- Home phone number

See this page for more information about LDAP.

The **chfn** command lets users change their own GECOS information. **chfn** is useful for keeping things like phone numbers up to date, but it can be misused. For example, a user can change the information to be obscene or incorrect. Some systems can be configured to restrict which fields **chfn** can modify; most college campuses disable it entirely. On most systems, **chfn** understands only the **passwd** file, so if you use LDAP or some other directory service for login information, **chfn** may not work at all.

Home directory

A user's home directory is his or her default directory at login time. Home directories are where login shells look for account-specific customizations such as shell aliases and environment variables, as well as SSH keys, server fingerprints, and other program state.

Be aware that if home directories are mounted over a network filesystem, they may be unavailable in the event of server or network problems. If the home directory is missing at login time, the system might print a message such as "no home directory" and put the user in /.

Such a message appears when you log in on the console or on a terminal, but not when you log in through a display manager such as **xdm**, **gdm**, or **kdm**. Not only will you not see the message, but you will generally be logged out immediately because of the display manager's inability to write to the proper directory (e.g., **~/.gnome**).

Alternatively, the system might disallow home-directory-less logins entirely, depending on the configuration.

Home directories are covered in more detail here.

Login shell

See this page for more information about shells.

The login shell is normally a command interpreter, but it can be any program. A Bourne-shell compatible **sh** is the default for FreeBSD, and **bash** (the GNU “Bourne again” shell) is the default for Linux.

Some systems permit users to change their shell with the **chsh** command, but as with **chfn**, this command might not work if you are using LDAP or some other directory service to manage login information. If you use the **/etc/passwd** file, a sysadmin can always change a user’s shell by editing the **passwd** file with **vipw**.

8.3 The Linux /etc/shadow file

On Linux, the shadow password file is readable only by the superuser and serves to keep encrypted passwords safe from prying eyes and password cracking programs. It also includes some additional account information that wasn't provided for in the original **/etc/passwd** format. These days, shadow passwords are the default on all systems.

The **shadow** file is not a superset of the **passwd** file, and the **passwd** file is not generated from it. You must maintain both files or use tools such as **useradd** that maintain both files on your behalf. Like **/etc/passwd**, **/etc/shadow** contains one line for each user. Each line contains nine fields, separated by colons:

- Login name
- Encrypted password
- Date of last password change
- Minimum number of days between password changes
- Maximum number of days between password changes
- Number of days in advance to warn users about password expiration
- Days after password expiration that account is disabled
- Account expiration date
- A field reserved for future use which is currently always empty

Only the values for the username and password are required. Absolute date fields in **/etc/shadow** are specified in terms of days (*not* seconds) since Jan 1, 1970, which is not a standard way of reckoning time on UNIX or Linux systems. To convert between the date in seconds and in days, run **expr `date+%s` / 86400**.

A typical **shadow** entry looks like this:

```
millert:$6$iTEFbMTM$CXmxPwErbEef9RUBvf1zv8EgXQdaZg2eOd5uXyvt4sFzi
    6G4lIqavLilTQgniAHm3Czw/LoaGzoFzaMm.YwOl/:16971:0:180:14:::
```

Here is a more complete description of each field:

- The login name is the same as in **/etc/passwd**. This field connects a user's **passwd** and **shadow** entries.
- The encrypted password is identical in concept and execution to the one previously stored in **/etc/passwd**.

- The last change field records the time at which the user's password was last changed. This field is filled in by the **passwd** command.
- The fourth field sets the number of days that must elapse between password changes. The idea is to force authentic changes by preventing users from immediately reverting to a familiar password after a required change. However, this feature can be somewhat dangerous in the aftermath of a security intrusion. We suggest setting this field to 0.
- The fifth field sets the maximum number of days allowed between password changes. This feature allows the administrator to enforce password aging; see this page for more information. Under Linux, the actual enforced maximum number of days is the sum of this field and the seventh (grace period) field.
- The sixth field sets the number of days before password expiration when **login** should begin to warn the user of the impending expiration.
- The eighth field specifies the day (in days since Jan 1, 1970) on which the user's account will expire. The user cannot log in after this date until the field has been reset by an administrator. If the field is left blank, the account never expires.

  You can use **usermod** to set the expiration field. It accepts dates in the format *yyyy-mm-dd*.
- The ninth field is reserved for future use (or at this rate, may never be used).

Let's look again at our example **shadow** line:

```
millert:$6$iTEFbMTM$CXmxPwErbEef9RUBvf1zv8EgXQdaZg2eOd5uXyvt4sFzi
    6G4lIqavLilTQgniAHm3Czw/LoaGzoFzaMm.YwOl/:17336:0:180:14:::
```

In this example, the user millert last changed his password on June 19, 2017. The password must be changed again within 180 days, and millert will receive warnings that the password needs to be changed for the last two weeks of this period. The account does not have an expiration date.

Use the **pwconv** utility to reconcile the contents of the **shadow** file and those of the **passwd** file, picking up any new additions and deleting users that are no longer listed in **passwd**.

8.4 FreeBSD's /etc/master.passwd and /etc/login.conf files



The adoption of PAM and the availability of similar user management commands on FreeBSD and Linux have made account administration relatively consistent across platforms, at least at the topmost layer. However, a few differences do exist in the underlying implementation.

The /etc/master.passwd file

On FreeBSD, the "real" password file is **/etc/master.passwd**, which is readable only by root. The **/etc/passwd** file exists for backward compatibility and does not contain any passwords (instead, it has * characters as placeholders).

To edit the password file, run the **vipw** command. This command invokes your editor on a copy of **/etc/master.passwd**, then installs the new version and regenerates the **/etc/passwd** file to reflect any changes. (**vipw** is standard on all UNIX and Linux systems, but it's particularly important to use on FreeBSD because the dual password files need to stay synchronized. See this page.)

In addition to containing all the fields of the **passwd** file, the **master.passwd** file contains three bonus fields. Unfortunately, they're squeezed in between the default GID field and the GECOS field, so the file formats are not directly compatible. The extra three fields are

- Login class
- Password change time
- Expiration time

The login class (if one is specified) refers to an entry in the **/etc/login.conf** file. The class determines resource consumption limits and controls a variety of other settings. See the next section for specifics.

The password change time field implements password aging. It contains the time in seconds since the UNIX epoch after which the user will be forced to change his or her password. You can leave this field blank, indicating that the password never expires.

The account expiration time gives the time and date (in seconds, as for password expiration) at which the user's account will expire. The user cannot log in after this date unless the field is reset by an administrator. If this field is left blank, the account will not expire.

The /etc/login.conf file

FreeBSD's **/etc/login.conf** file sets account-related parameters for users and groups of users. Its format consists of colon-delimited key/value pairs and Boolean flags.

When a user logs in, the login class field of **/etc/master.passwd** determines which entry in **/etc/login.conf** to apply. If the user's **master.passwd** entry does not specify a login class, the default class is used.

A **login.conf** entry can set any of the following:

- Resource limits (maximum process size, maximum file size, number of open files, etc.)
- Session accounting limits (when logins are allowed, and for how long)
- Default environment variables
- Default paths (PATH, MANPATH, etc.)
- Location of the "message of the day" file
- Host and TTY-based access control
- Default **umask**
- Account controls (mostly superseded by the PAM module pam_passwdqc)

The following example overrides several of the default values. It's intended for assignment to system administrators.

```
sysadmin:\
    :ignorenologin:\
    :requirehome@:\
    :maxproc=unlimited:\
    :openfiles=unlimited:\
    :tc=default:
```

Users in the sysadmin login class are allowed to log in even when **/var/run/nologin** exists, and they need not have a working home directory (this option permits logins when NFS is not working). Sysadmin users can start any number of processes and open any number of files (no artificial limit is imposed). The last line pulls in the contents of the default entry.

Although FreeBSD has reasonable defaults, you might be interested in updating the **/etc/login.conf** file to set idle timeout and password expiration warnings. For example, to set the idle timeout to 15 minutes and enable warnings seven days before passwords expire, you would add the following clauses to the definition of default:

```
    :warnpassword=7d:\
    :idletime=15m:\
```

When you modify the **/etc/login.conf** file, you must also run the following command to compile your changes into the hashed version of the file that the system actually refers to in daily operation:

```
$ cap_mkdb /etc/login.conf
```

8.5 THE /ETC/GROUP FILE

The **/etc/group** file contains the names of UNIX groups and a list of each group's members. Here's a portion of the **group** file from a FreeBSD system:

```
wheel:*:0:root
sys:*:3:root,bin
operator:*:5:root
bin:*:7:root
ftp:*:14:dan
staff:*:20:dan,ben,trent
nobody:*:65534:lpd
```

Each line represents one group and contains four fields:

- Group name
- Encrypted password or a placeholder
- GID number
- List of members, separated by commas (be careful not to add spaces)

As in **/etc/passwd**, fields are separated by colons. Group names should be limited to eight characters for compatibility, although many systems do not actually require this.

It's possible to set a group password that allows arbitrary users to enter the group with the **newgrp** command. However, this feature is rarely used. The group password can be set with **gpasswd**, which under Linux stores the encrypted password in the **/etc/gshadow** file.

As with usernames and UIDs, group names and GIDs should be kept consistent among machines that share files through a network filesystem. Consistency can be hard to maintain in a heterogeneous environment because different operating systems use different GIDs for standard system groups.

If a user defaults to a particular group in **/etc/passwd** but does not appear to be in that group according to **/etc/group**, **/etc/passwd** wins the argument. The group memberships granted at login time are the union of those found in the **passwd** and **group** files.

Some older systems limit the number of groups a user can belong to. There is no real limit on current Linux and FreeBSD kernels.

Much as with UIDs, we recommend minimizing the potential for GID collisions by starting local groups at GID 1000 or higher.

The UNIX tradition was originally to add new users to a group that represented their general category such as "students" or "finance." However, this convention increases the likelihood that

users will be able to read one another's files because of slipshod permission settings, even if that is not really the intention of the files' owner.

To avoid this problem, system utilities such as **useradd** and **adduser** now default to putting each user in his or her own personal group (that is, a group named after the user and which includes only that user). This convention is much easier to maintain if personal groups' GIDs match their corresponding users' UIDs.

To let users share files by way of the group mechanism, create separate groups for that purpose. The idea behind personal groups is not to discourage the use of groups per se—it's simply to establish a more restrictive *default* group for each user so that files are not inadvertently shared. You can also limit access to newly created files and directories by setting your user's default **umask** in a default startup file such as **/etc/profile** or **/etc/bashrc** (see this page).

---

*See this page for more information about **sudo**.*

Group membership can also serve as a marker for other contexts or privileges. For example, rather than entering the username of each system administrator into the **sudoers** file, you can configure **sudo** so that everyone in the "admin" group automatically has **sudo** privileges.


Linux supplies the **groupadd**, **groupmod**, and **groupdel** commands to create, modify, and delete groups.


FreeBSD uses the **pw** command to perform all these functions. To add the user "dan" to the group "staff" and then verify that the change was properly implemented, you would run the following commands:

```
$ sudo pw groupmod staff -m dan
$ pw groupshow staff
staff:*:20:dan,evi,garth,trent,ben
```

8.6 MANUAL STEPS FOR ADDING USERS

Before you create an account for a new user at a corporate, government, or educational site, it's important that the user sign and date a copy of your local user agreement and policy statement. (What?! You don't have a user agreement and policy statement? See this page for more information about why you need one and what to put in it.)

Users have no particular reason to want to sign a policy agreement, so it's to your advantage to secure their signatures while you still have some leverage. We find that it takes extra effort to secure a signed agreement after an account has been released. If your process allows for it, have the paperwork precede the creation of the account.

Mechanically, the process of adding a new user consists of several steps required by the system and a few more that establish a useful environment for the new user and incorporate the user into your local administrative system.

Required:

- Edit the **passwd** and **shadow** files (or the **master.passwd** file on FreeBSD) to define the user's account.
- Add the user to the **/etc/group** file (not really necessary, but nice).
- Set an initial password.
- Create, **chown**, and **chmod** the user's home directory.
- Configure roles and permissions (if you use RBAC; see this page).

For the user:

- Copy default startup files to the user's home directory.

For you:

- Have the new user sign your policy agreement.
- Verify that the account is set up correctly.
- Document the user's contact information and account status.

This list cries out for a script or tool, and fortunately, each of our example systems includes at least a partial off-the-shelf solution in the form of an **adduser** or **useradd** command. We take a look at these tools starting on this page.

Editing the passwd and group files

Manual maintenance of the **passwd** and **group** files is error prone and inefficient, so we recommend the slightly higher-level tools such as **useradd**, **adduser**, **usermod**, **pw**, and **chsh** as daily drivers.

If you do have to make manual changes, use the **vipw** command to edit the **passwd** and **shadow** files (or on FreeBSD, the **master.passwd** file). Although it sounds **vi**-centric, it actually invokes your favorite editor as defined in the EDITOR environment variable. More importantly, it locks the files so that editing sessions (or your editing and a user's password change) cannot collide.



After you run **vipw**, our Linux reference systems remind you to edit the **shadow** file after you have edited the **passwd** file. Use **vipw -s** to do so.



Under FreeBSD, **vipw** edits the **master.passwd** file instead of **/etc/passwd**. After installing your changes, **vipw** runs **pwd_mkdb** to generate the derived **passwd** file and two hashed versions of **master.passwd** (one that contains the encrypted passwords and is readable only by root, and another that lacks the passwords and is world-readable).

For example, running **vipw** and adding the following line would define an account called whitney:

```
whitney:*:1003:1003::0:0:Whitney Sather, AMATH 3-27, x7919,:
    /home/staff/whitney:/bin/sh
```

Note the star in the encrypted password field. This prevents use of the account until a real password is set with the **passwd** command (see the next section).

Next, edit **/etc/group** by running **vigr**. Add a line for the new personal group if your site uses them, and add the user's login name to each of the groups in which the user should have membership.

As with **vipw**, using **vigr** ensures that the changes made to the **/etc/group** file are sane and atomic. After an edit session, **vigr** should prompt you to run **vigr -s** to edit the group shadow (**gshadow**) file as well. Unless you want to set a password for the group—which is unusual—you can skip this step.



On FreeBSD, use **pw groupmod** to make changes to the **/etc/group** file.

Setting a password

Set a password for a new user with

```
$ sudo passwd newusername
```

You'll be prompted for the actual password.

Some automated systems for adding new users do not require you to set an initial password. Instead, they force the user to set a password on first login. Although this feature is convenient, it's a giant security hole: anyone who can guess new login names (or look them up in **/etc/passwd**) can swoop down and hijack accounts before the intended users have had a chance to log in.



Among many other functions, FreeBSD's **pw** command can also generate and set random user passwords:

```
$ sudo pw usermod raphael -w random
    Password for 'raphael' is: 1n3tcYu1s
```

See this page for tips on selecting good passwords.

We're generally not fans of random passwords for ongoing use. However, they are a good option for transitional passwords that are only intended to last until the account is actually used.

Creating the home directory and installing startup files

useradd and **adduser** create new users' home directories for you, but you'll likely want to double-check the permissions and startup files for new accounts.

There's nothing magical about home directories. If you neglected to include a home directory when setting up a new user, you can create it with a simple **mkdir**. You need to set ownerships and permissions on the new directory as well, but this is most efficiently done after you've installed any local startup files.

Startup files traditionally begin with a dot and end with the letters **rc**, short for "run command," a relic of the CTSS operating system. The initial dot causes **ls** to hide these "uninteresting" files from directory listings unless the **-a** option is used.

We recommend that you include default startup files for each shell that is popular on your systems so that users continue to have a reasonable default environment even if they change shells. Table 8.2 lists a variety of common startup files.

Table 8.2: Common startup files and their uses

Target	Filename	Typical uses
all shells	**.login_conf**	Sets user-specific login defaults (FreeBSD)
sh	**.profile**	Sets search path, terminal type, and environment
bash[a]	**.bashrc**	Sets the terminal type (if needed) Sets **biff** and **mesg** switches
	.bash_profile	Sets up environment variables Sets command aliases Sets the search path Sets the **umask** value to control permissions Sets CDPATH for filename searches Sets the PS1 (prompt) and HISTCONTROL variables
csh/tcsh	**.login**	Read by "login" instances of **csh**
	.cshrc	Read by all instances of **csh**
vi/vim	**.vimrc/.viminfo**	Sets **vi/vim** editor options
emacs	**.emacs**	Sets **emacs** editor options and key bindings
git	**.gitconfig**	Sets user, editor, color, and alias options for Git
GNOME	**.gconf**	GNOME user configuration via **gconf**
	.gconfpath	Path for additional user configuration via **gconf**
KDE	**.kde/**	Directory of configuration files

a. **bash** also reads **.profile** or **/etc/profile** in emulation of **sh**. The **.bash_profile** file is read by login shells, and the **.bashrc** file is read by interactive, non-login shells.

Sample startup files are traditionally kept in **/etc/skel**. If you customize your systems' startup file examples, **/usr/local/etc/skel** is a reasonable place to put the modified copies.

The entries in Table 8.2 for the GNOME and KDE window environments are really just the beginning. In particular, take a look at **gconf**, which is the tool that stores application preferences for GNOME programs in a manner analogous to the Windows registry.

Make sure that the default shell files you give to new users set a reasonable default value for **umask**; we suggest 077, 027, or 022, depending on the friendliness and size of your site. If you do not assign new users to individual groups, we recommend **umask** 077, which gives the owner full access but the group and the rest of the world no access.

---

*See this page for details on **umask**.*

Depending on the user's shell, **/etc** may contain system-wide startup files that are processed before the user's own startup files. For example, **bash** and **sh** read **/etc/profile** before processing **~/.profile** and **~/.bash_profile**. These files are a good place in which to put site-wide defaults, but bear in mind that users can override your settings in their own startup files. For details on other shells, see the man page for the shell in question.



By convention, Linux also keeps fragments of startup files in the **/etc/profile.d** directory. Although the directory name derives from **sh** conventions, **/etc/profile.d** can actually include fragments for several different shells. The specific shells being targeted are distinguished by filename suffixes (***.sh**, ***.csh**, etc.). There's no magic **profile.d** support built into the shells themselves; the fragments are simply executed by the default startup script in **/etc** (e.g., **/etc/profile** in the case of **sh** or **bash**).

Separating the default startup files into fragments facilitates modularity and allows software packages to include their own shell-level defaults. For example, the **colorls.*** fragments coach shells on how to properly color the output of **ls** so as to make it unreadable on dark backgrounds.

Setting home directory permissions and ownerships

Once you've created a user's home directory and copied in a reasonable default environment, turn the directory over to the user and make sure that the permissions on it are appropriate. The command

```
$ sudo chown -R newuser:newgroup ~newuser
```

sets ownerships properly. Note that you cannot use

```
$ sudo chown newuser:newgroup ~newuser/.*
```

to **chown** the dot files because *newuser* would then own not only his or her own files but also the parent directory ".." as well. This is a common and dangerous mistake.

Configuring roles and administrative privileges

Role-based access control (RBAC) allows system privileges to be tailored for individual users and is available on many of our example systems. RBAC is not a traditional part of the UNIX or Linux access control model, but if your site uses it, role configuration must be a part of the process of adding users. RBAC is covered in detail starting on this page in the *Access Control and Rootly Powers* chapter.

See Chapter 31 for more information about SOX and GLBA

Legislation such as the Sarbanes-Oxley Act, the Health Insurance Portability and Accountability Act (HIPAA), and the Gramm-Leach-Bliley Act in the United States has complicated many aspects of system administration in the corporate arena, including user management. Roles might be your only viable option for fulfilling some of the SOX, HIPAA, and GLBA requirements.

Finishing up

To verify that a new account has been properly configured, first log out, then log in as the new user and execute the following commands:

```
$ pwd       # To verify the home directory
$ ls -la    # To check owner/group of startup files
```

You need to notify new users of their login names and initial passwords. Many sites send this information by email, but that's generally not a secure choice. Better options are to do it in person, over the phone, or through a text message. (If you are adding 500 new freshmen to the campus's CS-1 machines, punt the notification problem to the instructor!) If you must distribute account passwords by email, make sure the passwords expire in a couple of days if they are not used and changed.

See this page for more information about written user contracts.

If your site requires users to sign a written policy agreement or appropriate use policy, be sure this step has been completed before you release a new account. This check prevents oversights and strengthens the legal basis of any sanctions you might later need to impose. This is also the time to point users toward additional documentation on local customs.

Remind new users to change their passwords immediately. You can enforce this by setting the password to expire within a short time. Another option is to have a script check up on new users and be sure their encrypted passwords have changed. Because the same password can have many encrypted representations, this method verifies only that the user has reset the password, not that it has actually been changed to a *different* password.

In environments where you know users personally, it's relatively easy to keep track of who's using a system and why. But if you manage a large and dynamic user base, you need a more formal way to keep track of accounts. Maintaining a database of contact information and account statuses helps you figure out, once the act of creating the account has faded from memory, who people are and why they have an account.

8.7 Scripts for adding users: useradd, adduser, and newusers

Our example systems all come with a **useradd** or **adduser** script that implements the basic procedure outlined above. However, these scripts are configurable, and you will probably want to customize them to fit your environment. Unfortunately, each system has its own idea of what you should customize, where you should implement the customizations, and what the default behavior should be. Accordingly, we cover these details in vendor-specific sections.

Table 8.3 is a handy summary of commands and configuration files related to managing users.

Table 8.3: Commands and configuration files for user management

System	Commands	Configuration files
All Linux	**useradd**, **usermod**, **userdel**	**/etc/login.defs** **/etc/default/useradd**
Debian/Ubuntu[a]	**adduser**, **deluser**	**/etc/adduser.conf** **/etc/deluser.conf**
FreeBSD	**adduser**, **rmuser**	**/etc/login.conf**

a. This suite wraps the standard Linux version and includes a few more features.

useradd on Linux

Most Linux distributions include a basic **useradd** suite that draws its configuration parameters from both **/etc/login.defs** and **/etc/default/useradd**.



The **login.defs** file addresses issues such as password aging, choice of encryption algorithms, location of mail spool files, and the preferred ranges of UIDs and GIDs. You maintain the **login.defs** file by hand. The comments do a good job of explaining the various parameters.

Parameters stored in the **/etc/default/useradd** file include the location of home directories and the default shell for new users. You set these defaults through the **useradd** command itself. **useradd -D** prints the current values, and **-D** in combination with various other flags sets the values of specific options. For example,

```
$ sudo useradd -D -s /bin/bash
```

sets **bash** as the default shell.

Typical defaults are to put new users in individual groups, to use SHA-512 encryption for passwords, and to populate new users' home directories with startup files from **/etc/skel**.

The basic form of the **useradd** command accepts the name of the new account on the command line:

```
$ sudo useradd hilbert
```

This command creates an entry similar to this one in **/etc/passwd**, along with a corresponding entry in the **shadow** file:

```
hilbert:x:1005:20::/home/hilbert:/bin/sh
```

useradd disables the new account by default. You must assign a real password to make the account usable.

A more realistic example is shown below. We specify that hilbert's primary group should be "hilbert" and that he should also be added to the "faculty" group. We override the default home directory location and shell and ask **useradd** to create the home directory if it does not already exist:

```
$ sudo useradd -c "David Hilbert" -d /home/math/hilbert -g hilbert
    -G faculty -m -s /bin/tcsh hilbert
```

This command creates the following **passwd** entry:

```
hilbert:x:1005:30:David Hilbert:/home/math/hilbert:/bin/tcsh
```

The assigned UID is one higher than the highest UID on the system, and the corresponding **shadow** entry is

```
hilbert:!:14322:0:99999:7:0::
```

The password placeholder character(s) in the **passwd** and **shadow** file vary depending on the operating system. **useradd** also adds hilbert to the appropriate groups in **/etc/group**, creates the directory **/home/math/hilbert** with proper ownerships, and populates it from the **/etc/skel** directory.

adduser on Debian and Ubuntu

  In addition to the **useradd** family of commands, the Debian lineage also supplies somewhat higher-level wrappers for these commands in the form of **adduser** and **deluser**. These add-on commands are configured in **/etc/adduser.conf**, where you can specify options such as these:

- Rules for locating home directories: by group, by username, etc.
- Permission settings for new home directories
- UID and GID ranges for system users and general users
- An option to create individual groups for each user
- Disk quotas (Boolean only, unfortunately)
- Regex-based matching of user names and group names

Other typical **useradd** parameters, such as rules for passwords, are set as parameters to the PAM module that does regular password authentication. (See this page for a discussion of PAM, aka Pluggable Authentication Modules.) **adduser** and **deluser** have twin cousins **addgroup** and **delgroup**.

adduser on FreeBSD



FreeBSD comes with **adduser** and **rmuser** shell scripts that you can either use as supplied or modify to fit your needs. The scripts are built on top of the facilities provided by the **pw** command.

adduser can be used interactively if you prefer. By default, it creates user and group entries and a home directory. You can point the script at a file containing a list of accounts to create with the **-f** flag, or enter in each user interactively.

For example, the process for creating a new user “raphael” looks like this:

```
$ sudo adduser
Username: raphael
Full name: Raphael Dobbins
Uid (Leave empty for default): <return>
Login group [raphael]: <return>
Login group is raphael. Invite raphael into other groups? []: <return>
Login class [default]: <return>
Shell (sh csh tcsh bash rbash nologin) [sh]: bash
Home directory [/home/raphael]: <return>
Home directory permissions (Leave empty for default): <return>
Use password-based authentication? [yes]: <return>
Use an empty password? (yes/no) [no]: <return>
Use a random password? (yes/no) [no]: yes
Lock out the account after creation? [no]: <return>
Username    : raphael
Password    : <random>
Full Name   : Raphael Dobbins
Uid         : 1004
Class       :
Groups      : raphael
Home        : /home/raphael
Home Mode   :
Shell       : /usr/local/bin/bash
Locked      : no
OK? (yes/no): yes
adduser: INFO: Successfully added (raphael) to the user database.
adduser: INFO: Password for (raphael) is: RSCAds5fy0vxOt
Add another user? (yes/no): no
Goodbye!
```

newusers on Linux: adding in bulk

Linux's **newusers** command creates multiple accounts at one time from the contents of a text file. It's pretty gimpy, but it can be handy when you need to add a lot of users at once, such as when creating class-specific accounts. **newusers** expects an input file of lines just like the **/etc/passwd** file, except that the password field contains the initial password in clear text. Oops… better protect that file.



newusers honors the password aging parameters set in the **/etc/login.defs** file, but it does not copy in the default startup files as does **useradd**. The only startup file it copies in is **.xauth**.

At a university, what's really needed is a batch **adduser** script that can use a list of students from enrollment or registration data to generate the input for **newusers**, with usernames formed according to local rules and guaranteed to be locally unique, with strong passwords randomly generated, and with UIDs and GIDs increasing for each user. You're probably better off writing your own wrapper for **useradd** in Python than trying to get **newusers** to do what you need.

8.8 SAFE REMOVAL OF A USER'S ACCOUNT AND FILES

When a user leaves your organization, that user's login account and files must be removed from the system. If possible, don't do that chore by hand; instead, let **userdel** or **rmuser** handle it. These tools ensure the removal of all references to the login name that were added by you or your **useradd** program. Once you've removed the remnants, use the following checklist to verify that all residual user data has been removed:

- Remove the user from any local user databases or phone lists.
- Remove the user from the mail aliases database, or add a forwarding address.
- Remove the user's crontab file and any pending **at** jobs or print jobs.
- Kill any of the user's processes that are still running.
- Remove the user from the **passwd**, **shadow**, **group**, and **gshadow** files.
- Remove the user's home directory.
- Remove the user's mail spool (if mail is stored locally).
- Clean up entries on shared calendars, room reservation systems, etc.
- Delete or transfer ownership of any mailing lists run by the deleted user.

Before you remove someone's home directory, be sure to relocate any files that are needed by other users. You usually can't be sure which files those might be, so it's always a good idea to make an extra backup of the user's home directory before deleting it.

Once you have removed all traces of a user, you may want to verify that the user's old UID no longer owns files on the system. To find the paths of orphaned files, you can use the **find** command with the **-nouser** argument. Because **find** has a way of "escaping" onto network servers if you're not careful, it's usually best to check filesystems individually with **-xdev**:

```
$ sudo find filesystem -xdev -nouser
```

If your organization assigns individual workstations to users, it's generally simplest and most efficient to re-image the entire system from a master template before turning the system over to a new user. Before you do the reinstallation, however, it's a good idea to back up any local files on the system's hard disk in case they are needed in the future. (Think license keys!)

Although all our example systems come with commands that automate the process of removing user presence, they probably do not do as thorough a job as you might like unless you have religiously extended them as you expanded the number of places in which user-related information is stored.

  Debian and Ubuntu's **deluser** is a Perl script that calls the usual **userdel**; it undoes all the things **adduser** does. It runs the script **/usr/local/sbin/deluser.local**, if it exists, to facilitate easy localization. The configuration file **/etc/deluser.conf** lets you set options such as these:

- Whether to remove the user's home directory and mail spool
- Whether to back up the user's files, and where to put the backup
- Whether to remove all files on the system owned by the user

RHEL  Whether to delete a group if it now has no members

Red Hat supports a **userdel.local** script but no pre- and post-execution scripts to automate sequence-sensitive operations such as backing up an about-to-be-removed user's files.

 FreeBSD's **rmuser** script does a good job of removing instances of the user's files and processes, a task that other vendors' **userdel** programs do not even attempt.

8.9 USER LOGIN LOCKOUT

On occasion, a user's login must be temporarily disabled. A straightforward way to do this is to put a star or some other character in front of the user's encrypted password in the **/etc/shadow** or **/etc/master.passwd** file. This measure prevents most types of password-regulated access because the password no longer decrypts to anything sensible.

 FreeBSD lets you lock accounts with the **pw** command. A simple

```
$ sudo pw lock someuser
```

puts the string *LOCKED* at the start of the password hash, making the account unusable. Unlock the account by running

```
$ sudo pw unlock someuser
```

 On all our Linux distributions, the **usermod -L** *user* and **usermod -U** *user* commands define an easy way to lock and unlock passwords. They are just shortcuts for the password twiddling described above: the **-L** puts an ! in front of the encrypted password in the **/etc/shadow** file, and the **-U** removes it.

Unfortunately, modifying a user's password simply makes logins fail. It does not notify the user of the account suspension or explain why the account no longer works. In addition, commands such as **ssh** that do not necessarily check the system password may continue to function.

An alternative way to disable logins is to replace the user's shell with a program that prints an explanatory message and supplies instructions for rectifying the situation. The program then exits, terminating the login session.

This approach has both advantages and disadvantages. Any forms of access that check the password but do not pay attention to the shell will not be disabled. To facilitate the "disabled shell" trick, many daemons that afford nonlogin access to the system (e.g., **ftpd**) check to see if a user's login shell is listed in **/etc/shells** and deny access if it is not. This is the behavior you want. Unfortunately, it's not universal, so you may have to do some fairly comprehensive testing if you decide to use shell modification as a way of disabling accounts.

Another issue is that your carefully written explanation of the suspended account might never be seen if the user tries to log in through a window system or through a terminal emulator that does not leave output visible after a logout.

8.10 Risk reduction with PAM

Pluggable Authentication Modules (PAM) is covered in the *Single Sign-On* chapter starting here. PAM centralizes the management of the system's authentication facilities through standard library routines. That way, programs like **login**, **sudo**, **passwd**, and **su** need not supply their own tricky authentication code. An organization can easily expand its authentication methods beyond passwords to options such as Kerberos, one-time passwords, ID dongles, or fingerprint readers. PAM reduces the risk inherent in writing secured software, allows administrators to set site-wide security policies, and defines an easy way to add new authentication methods to the system.

Adding and removing users doesn't involve tweaking the PAM configuration, but the tools involved operate under PAM's rules and constraints. In addition, many of the PAM configuration parameters are similar to those used by **useradd** or **usermod**. If you change a parameter as described in this chapter and **useradd** doesn't seem to be paying attention to it, check to be sure the system's PAM configuration isn't overriding your new value.

8.11 Centralized account management

Some form of centralized account management is essential for medium-to-large enterprises of all types, be they corporate, academic, or governmental. Users need the convenience and security of a single login name, UID, and password across the site. Administrators need a centralized system that allows changes (such as account revocations) to be instantly propagated everywhere.

Such centralization can be achieved in a variety of ways, most of which (including Microsoft's Active Directory system) involve LDAP, the Lightweight Directory Access Protocol, in some capacity. Options range from bare-bones LDAP installations based on open source software to elaborate commercial identity management systems that come with a hefty price tag.

LDAP and Active Directory

LDAP is a generalized, database-like repository that can store user management data as well as other types of data. It uses a hierarchical client/server model that supports multiple servers as well as multiple simultaneous clients. One of LDAP's big advantages as a site-wide repository for login data is that it can enforce unique UIDs and GIDs across systems. It also plays well with Windows, although the reverse is only marginally true.

See the section starting on this page for more information about LDAP and LDAP implementations.

Microsoft's Active Directory uses LDAP and Kerberos and can manage many kinds of data, including user information. It's a bit egotistical and wants to be the boss if it is interacting with UNIX or Linux LDAP repositories. If you need a single authentication system for a site that includes Windows desktops as well as UNIX and Linux systems, it is probably easiest to let Active Directory be in control and to use your UNIX LDAP databases as secondary servers.

See Chapter 17, *Single Sign-On*, for more information on integrating UNIX or Linux with LDAP, Kerberos, and Active Directory.

Application-level single sign-on systems

Application-level single sign-on systems balance user convenience with security. The idea is that a user can sign on once (to a login prompt, web page, or Windows box) and be authenticated at that time. The user then obtains authentication credentials (usually implicitly, so that no active management is required) which can be used to access other applications. The user only has to remember one login and password sequence instead of many.

This scheme allows credentials to be more complex since the user does not need to remember or even deal with them. That theoretically increases security. However, the impact of a compromised account is greater because one login gives an attacker access to multiple applications. These systems make your walking away from a desktop machine while still logged in a significant vulnerability. In addition, the authentication server becomes a critical bottleneck. If it's down, all useful work grinds to a halt across the enterprise.

Although application-level SSO is a simple idea, it implies a lot of back-end complexity because the various applications and machines that a user might want to access must understand the authentication process and SSO credentials.

Several open source SSO systems exist:

- JOSSO, an open source SSO server written in Java
- CAS, the Central Authentication Service, from Yale (also Java)
- Shibboleth, an open source SSO distributed under the Apache 2 license

A host of commercial systems are also available, most of them integrated with identity management suites, which are covered in the next section.

Identity management systems

"Identity management" (sometimes referred to as IAM, for "identity and access management") is a common buzzword in user management. In plain language, it means identifying the users of your systems, authenticating their identities, and granting privileges according to those authenticated identities. The standardization efforts in this realm are led by the World Wide Web Consortium and by The Open Group.

Commercial identity management systems combine several key UNIX concepts into a warm and fuzzy GUI replete with marketing jargon. Fundamental to all such systems is a database of user authentication and authorization data, often stored in LDAP format. Control is achieved with concepts such as UNIX groups, and limited administrative privileges are enforced through tools such as **sudo**. Most such systems have been designed with an eye toward regulations that mandate accountability, tracking, and audit trails.

There are many commercial systems in this space: Oracle's Identity Management, Courion, Avatier Identity Management Suite (AIMS), VMware Identity Manager, and SailPoint's IdentityIQ, to name a few. In evaluating identity management systems, look for capabilities in the following areas:

Oversight:

- Implement a secure web interface for management that's accessible both inside and outside the enterprise.
- Support an interface through which hiring managers can request that accounts be provisioned according to role.
- Coordinate with a personnel database to automatically remove access for employees who are terminated or laid off.

Account management:

- Generate globally unique user IDs.
- Create, change, and delete user accounts across the enterprise, on all types of hardware and operating systems.
- Support a workflow engine; for example, tiered approvals before a user is given certain privileges.
- Make it easy to display all users who have a certain set of privileges. Ditto for the privileges granted to a particular user.
- Support role-based access control, including user account provisioning by role. Allow exceptions to role-based provisioning, including a workflow for the approval of

exceptions.

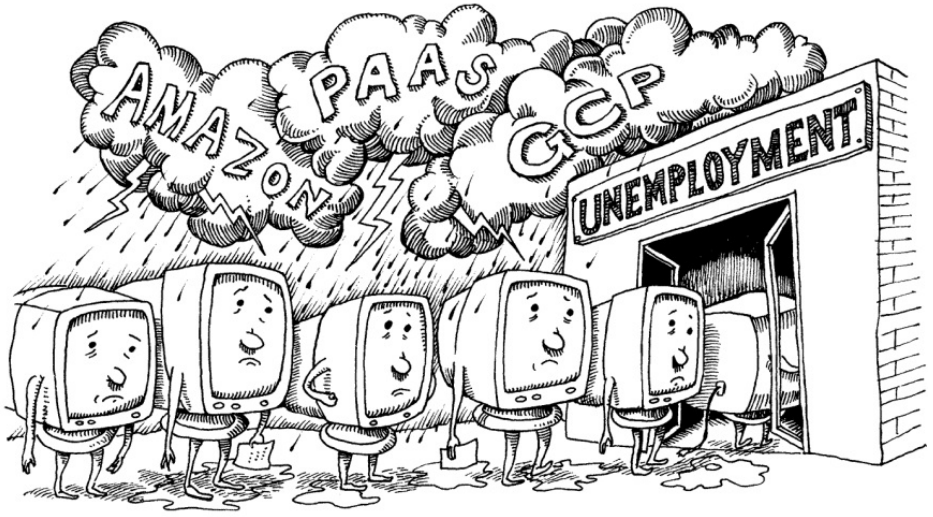
- Configure logging of all changes and administrative actions. Similarly, configure reports generated from logging data (by user, by day, etc.).

Ease of use:

- Let users change (and reset) their own passwords, with enforcement of rules for picking strong passwords.
- Enable users to change their passwords globally in one operation.

Consider also how the system is implemented at the point where authorizations and authentications actually take place. Does the system require a custom agent to be installed everywhere, or does it conform itself to the underlying systems?

9 Cloud Computing



Cloud computing is the practice of leasing computer resources from a pool of shared capacity. Users of cloud services provision resources on demand and pay a metered rate for whatever they consume. Businesses that embrace the cloud enjoy faster time to market, greater flexibility, and lower capital and operating expenses than businesses that run traditional data centers.

The cloud is the realization of "utility computing," first conceived by the late computer scientist John McCarthy, who described the idea in a talk at MIT in 1961. Many technological advances since McCarthy's prescient remarks have helped to bring the idea to fruition. To name just a few:

- Virtualization software reliably allocates CPU, memory, storage, and network resources on demand.
- Robust layers of security isolate users and virtual machines from each other, even as they share underlying hardware.
- Standardized hardware components enable the construction of data centers with vast power, storage, and cooling capacities.
- A reliable global network connects everything.

Cloud providers capitalize on these innovations and many others. They offer myriad services ranging from hosted private servers to fully managed applications. The leading cloud vendors are competitive, highly profitable, and growing rapidly.

This chapter introduces the motivations for moving to the cloud, fills in some background on a few major cloud providers, introduces some of the most important cloud services, and offers tips for controlling costs. As an even briefer introduction, the section *Clouds: VPS quick start by platform*, shows how to create cloud servers from the command line.

Several other chapters in this book include sections that relate to the management of cloud servers. Table 9.1 lists some pointers.

Table 9.1: Cloud topics covered elsewhere in this book

Heading
Recovery of cloud systems (bootstrapping-related issues for the cloud)
Cloud networking (TCP/IP networking for cloud platforms)
Web hosting in the cloud
*Packer (*using Packer to build OS images for the cloud)
Container clustering and management (especially the section on AWS ECS)
CI/CD in practice (a CI/CD pipeline example that uses cloud services)
*Commercial application monitoring tools (*monitoring tools for the cloud)

In addition, Chapter 23, *Configuration Management*, is broadly applicable to the management of cloud systems.

9.1 The cloud in context

The transition from servers in private data centers to the now-ubiquitous cloud has been rapid and dramatic. Let's take a look at the reasons for this stampede.

Cloud providers create technically advanced infrastructure that most businesses cannot hope to match. They locate their data centers in areas with inexpensive electric power and copious networking cross-connects. They design custom server chassis that maximize energy efficiency and minimize maintenance. They use purpose-built network infrastructure with custom hardware and software fine-tuned to their internal networks. They automate aggressively to allow rapid expansion and reduce the likelihood of human error.

Because of all this engineering effort (not to mention the normal economies of scale), the cost of running distributed computing services is much lower for a cloud provider than for a typical business with a small data center. Cost savings are reflected both in the price of cloud services and in the providers' profits.

Layered on top of this hardware foundation are management features that simplify and facilitate the configuration of infrastructure. Cloud providers offer both APIs and user-facing tools that control the provisioning and releasing of resources. As a result, the entire life cycle of a system—or group of systems distributed on a virtual network—can be automated. This concept goes by the name "infrastructure as code," and it contrasts starkly with the manual server procurement and provisioning processes of times past.

Elasticity is another major driver of cloud adoption. Because cloud systems can be programmatically requested and released, any business that has cyclic demand can optimize operating costs by adding more resources during periods of peak usage and removing extra capacity when it is no longer needed. The built-in autoscaling features available on some cloud platforms streamline this process.

Cloud providers have a global presence. With some planning and engineering effort, businesses can reach new markets by releasing services in multiple geographic areas. In addition, disaster recovery is easier to implement in the cloud because redundant systems can be run in separate physical locations.

See this page for more information about DevOps.

All these characteristics pair well with the DevOps approach to system administration, which emphasizes agility and repeatability. In the cloud, you're no longer restricted by slow procurement or provisioning processes, and nearly everything can be automated.

Still, a certain mental leap is required when you don't control your own hardware. One industry metaphor captures the sentiment neatly: servers should be treated as cattle, not as pets. A pet is

named, loved, and cared for. When the pet is sick, it's taken to a veterinarian and nursed back to health. Conversely, cattle are commodities that are herded, traded, and managed in large quantities. Sick cattle are shot.

A cloud server is just one member of a herd, and to treat it otherwise is to ignore a basic fact of cloud computing: cloud systems are ephemeral, and they can fail at any time. Plan for that failure and you'll be more successful at running a resilient infrastructure.

Despite all its advantages, the cloud is not a panacea for quickly reducing costs or improving performance. Directly migrating an existing enterprise application from a data center to a cloud provider (a so-called "lift and shift") is unlikely to be successful without careful planning. Operational processes for the cloud are different, and they entail training and testing. Furthermore, most enterprise software is designed for static environments, but individual systems in the cloud should be treated as short-lived and unreliable. A system is said to be cloud native if it is reliable even in the face of unanticipated events.

9.2 Cloud platform choices

Multiple factors influence a site's choice of cloud provider. Cost, past experience, compatibility with existing technology, security, or compliance requirements, and internal politics are all likely to play a role. The selection process can also be swayed by reputation, provider size, features, and of course, marketing.

Fortunately, there are a lot of cloud providers out there. We've chosen to focus on just three of the major public cloud providers: Amazon Web Services (AWS), Google Cloud Platform (GCP), and DigitalOcean (DO). In this section we mention a few additional options for you to consider. Table 9.2 enumerates the major players in this space.

Table 9.2: The most widely used cloud platforms

Provider	Notable qualities
Amazon Web Services	900lb gorilla. Rapid innovation. Can be expensive. Complex.
DigitalOcean	Simple and reliable. Lovable API. Good for development.
Google Cloud Platform	Technically sophisticated and improving quickly. Emphasizes performance. Comprehensive big-data services.
IBM Softlayer	More like hosting than cloud. Has a global private network.
Microsoft Azure	A distant second in size. Has a history of outages. Possibly worth consideration for Microsoft shops.
OpenStack	Modular DIY open source platform for building private clouds. AWS-compatible APIs.
Rackspace	Public and private clouds running OpenStack. Offers managed services for AWS and Azure. Fanatical support.
VMware vCloud Air	Buzzword-laden service for public, private, and hybrid clouds. Uses VMware technology. Probably doomed.

Public, private, and hybrid clouds

In a public cloud, the vendor controls all the physical hardware and affords access to systems over the Internet. This setup relieves users of the burden of installing and maintaining hardware, but at the expense of less control over the features and characteristics of the platform. AWS, GCP, and DO are all public cloud providers.

Private cloud platforms are similar, but are hosted within an organization's own data center or managed by a vendor on behalf of a single customer. Servers in a private cloud are single-tenant, not shared with other customers as in a public cloud.

Private clouds offer flexibility and programmatic control, just as public clouds do. They appeal to organizations that already have significant capital invested in hardware and engineers, especially those that value full control of their environment.

OpenStack is the leading open source system used to create private clouds. It receives financial and engineering support from enterprises such as AT&T, IBM, and Intel. Rackspace itself is one of the largest contributors to OpenStack.

A combination of public and private clouds is called a hybrid cloud. Hybrids can be useful when an enterprise is first migrating from local servers to a public cloud, for adding temporary capacity to handle peak loads, and for a variety of other organization-specific scenarios. Administrators beware: operating two distinct cloud presences in tandem increases complexity more than proportionally.

VMware's vSphere Air cloud, based on vSphere virtualization technology, is a seamless hybrid cloud for customers that already use VMware virtualization in their on-premises data center. Those users can move applications to and from vCloud Air infrastructure quite transparently.

The term "public cloud" is a bit unfortunate, connoting as it does the security and hygiene standards of a public toilet. In fact, customers of public clouds are isolated from each other by multiple layers of hardware and software virtualization. A private cloud offers little or no practical security benefit over a public cloud.

In addition, operating a private cloud is an intricate and expensive prospect that should not be undertaken lightly. Only the largest and most committed organizations have the engineering resources and wallet needed to implement a robust, secure private cloud. And once implemented, a private cloud's features usually fall short of those offered by commercial public clouds.

For most organizations, we recommend the public cloud over the private or hybrid options. Public clouds offer the highest value and easiest administration. For the remainder of this book, our cloud coverage is limited to public options The next few sections present a quick overview of each of our example platforms.

Amazon Web Services

AWS offers scores of services, ranging from virtual servers (EC2) to managed databases and data warehouses (RDS and Redshift) to serverless functions that execute in response to events (Lambda). AWS releases hundreds of updates and new features each year. It has the largest and most active community of users. AWS is by far the largest cloud computing business.

From the standpoint of most users, AWS has essentially unlimited capacity. However, new accounts come with limits that control how much compute power you can requisition. These restrictions protect both Amazon and you, since costs can quickly spiral out of control if services aren't properly managed. To increase your limits, you complete a form on the AWS support site. The service limit documentation itemizes the constraints associated with each service.

The on-line AWS documentation located at aws.amazon.com/documentation is authoritative, comprehensive, and well organized. It should be the first place you look when researching a particular service. The white papers that discuss security, migration paths, and architecture are invaluable for those interested in constructing robust cloud environments.

Google Cloud Platform

If AWS is the reigning champion of the cloud, Google is the would-be usurper. It competes for customers through nefarious tricks such as lowering prices and directly addressing customers' AWS pain points.

The demand for engineers is so fierce that Google has been known to poach employees from AWS. In they past, they've hosted parties in conjunction with the AWS re:Invent conference in Las Vegas in an attempt to lure both talent and users. As the cloud wars unfold, customers ultimately benefit from this competition in the form of lower costs and improved features.

Google runs the most advanced global network in the world, a strength that benefits its cloud platform. Google data centers are technological marvels that feature many innovations to improve energy efficiency and reduce operational costs. Google is relatively transparent about its operations, and their open source contributions help advance the cloud industry. See google.com/about/datacenters for photos and facts about how Google's data centers operate.

Despite its technical savvy, in some ways Google is a follower in the public cloud, not a leader. Google had released other cloud products as early as 2008, including App Engine, the first platform-as-a-service product. But Google's strategy and the GCP brand were not well developed until 2012. At that point, GCP was already somewhat late to the game.

GCP's services have many of the same features (and often the same names) as their AWS equivalents. If you're familiar with AWS, you'll find the GCP web interface to be somewhat different on the surface. However, the functionality underneath is strikingly similar.

We anticipate that GCP will gain market share in the years to come as it improves its products and builds customer trust. It has hired some of the brightest minds in the industry, and they're bound to develop some innovative technologies. As consumers, we all stand to benefit.

DigitalOcean

DigitalOcean is a different breed of public cloud. Whereas AWS and GCP compete to serve the large enterprises and growth-focused startups, DigitalOcean courts small customers with simpler needs. Minimalism is the name of the game. We like DigitalOcean for experiments and proof-of-concept projects.

DigitalOcean offers data centers in North America, Europe, and Asia. There are several centers in each of these regions, but they are not directly connected and so cannot be considered availability zones (see this page). As a result, it's considerably more difficult to build global, highly available production services on DigitalOcean than on AWS or Google.

DigitalOcean servers are called droplets. They are simple to provision from the command line or web console, and they boot quickly. DigitalOcean supplies images for all our example operating systems except Red Hat. It also has a handful of images for popular open source applications such as Cassandra, Drupal, Django, and GitLab.

DigitalOcean also has load balancer and block storage services. In Chapter 26, *Continuous Integration and Delivery*, we include an example of provisioning a DigitalOcean load balancer with two droplets using HashiCorp's Terraform infrastructure provisioning tool.

9.3 Cloud service fundamentals

Cloud services are loosely grouped into three categories:

- Infrastructure-as-a-Service (IaaS), in which users request raw compute, memory, network, and storage resources. These are typically delivered in the form of virtual private servers, aka VPSs. Under IaaS, users are responsible for managing everything above the hardware: operating systems, networking, storage systems, and their own software.
- Platform-as-a-Service (PaaS), in which developers submit their custom applications packaged in a format specified by the vendor. The vendor then runs the code on the user's behalf. In this model, users are responsible for their own code, while the vendor manages the OS and network.
- Software-as-a-Service (SaaS), the broadest category, in which the vendor hosts and manages software and users pay some form of subscription fee for access. Users maintain neither the operating system nor the application. Almost any hosted web application (think WordPress) falls into the SaaS category.

Table 9.3 shows how each of these abstract models breaks down in terms of the layers involved in a complete deployment.

Table 9.3: Which layers are you responsible for managing?

Layer	Local[a]	IaaS	PaaS	SaaS
Application	✔	✔	✔	
Databases	✔	✔	✔	
Application runtime	✔	✔	✔	
Operating system	✔	✔		
Virtual network, storage, and servers	✔	✔		
Virtualization platform	✔			
Physical servers	✔			
Storage systems	✔			
Physical network	✔			
Power, space, and cooling	✔			

a. Local: local servers and network
IaaS: Infrastructure-as-a-Service (virtual servers)
PaaS: Platform-as-a-Service (e.g., Google App Engine)
SaaS: Software-as-a-Service (e.g., most web-based services)

Of these options, IaaS is the most pertinent to system administration. In addition to defining virtual computers, IaaS providers virtualize the hardware elements that are typically connected to

them, such as disks (now described more generally as “block storage devices”) and networks. Virtual servers can inhabit virtual networks for which you specify the topology, routes, addressing, and other characteristics. In most cases, these networks are private to your organization.

IaaS can also include other core services such as such as databases, queues, key/value stores, and compute clusters. These features combine to create a complete replacement for (and in many cases, an improvement over) the traditional data center.

PaaS is an area of great promise that is not yet fully realized. Current offerings such as AWS Elastic Beanstalk, Google App Engine, and Heroku come with environmental constraints or nuances that render them impractical (or incomplete) for use in busy production environments. Time and again we’ve seen business outgrow these services. However, new services in this area are receiving a lot of attention. We anticipate dramatic improvements in the coming years.

Cloud providers differ widely in terms of their exact features and implementation details, but conceptually, many services are quite similar. The following sections describe cloud services generally, but because AWS is the front-runner in this space, we sometimes adopt its nomenclature and conventions as defaults.

Access to the cloud

Most cloud providers' primary interface is some kind of web-based GUI. New system administrators should use this web console to create an account and to configure their first few resources.

Cloud providers also define APIs that access the same underlying functionality as that of the web console. In most cases, they also have a standard command-line wrapper, portable to most systems, for those APIs.

Even veteran administrators make frequent use of web GUIs. However, it's also important to get friendly with the command-line tools because they lend themselves more readily to automation and repeatability. Use scripts to avoid the tedious and sluggish process of requesting everything through a browser.

Cloud vendors also maintain software development kits (SDKs) for many popular languages to help developers use their APIs. Third party tools use the SDKs to simplify or automate specific sets of tasks. You'll no doubt encounter these SDKs if you write your own tools.

You normally use SSH with public key authentication to access UNIX and Linux systems running in the cloud. See *SSH, the Secure SHell* for more information about the effective use of SSH.

Some cloud providers let you access a console session through a web browser, which can be especially helpful if you mistakenly lock yourself out with a firewall rule or broken SSH configuration. It's not a representation of the system's actual console, though, so you can't use this feature to debug bootstrapping or BIOS issues.

Regions and availability zones

Cloud providers maintain data centers around the world. A few standard terms describe geography-related features.

A “region” is a location in which a cloud provider maintains data centers. In most cases, regions are named after the territory of intended service even though the data centers themselves are more concentrated. For example, Amazon’s us-east-1 region is served by data centers in north Virginia.

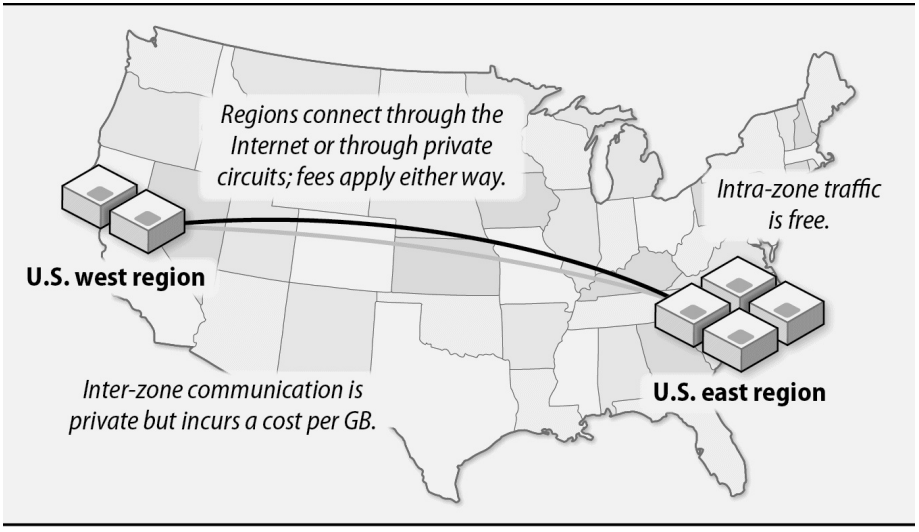
It takes about 5ms for a fiber optic signal to travel 1,000km, so regions the size of the U.S. east coast are fine from a performance standpoint. The network connectivity available to a data center is more important than its exact location.

Some providers also have “availability zones” (or simply “zones”) which are collections of data centers within a region. Zones within a region are peered through high-bandwidth, low-latency, redundant circuits, so inter-zone communication is fast, though not necessarily cheap. Anecdotally, we’ve experienced inter-zone latency of less than 1ms.

Zones are typically designed to be independent of one another in terms of power and cooling, and they’re geographically dispersed so that a natural disaster that affects one zone has a low probability of affecting others in the same region.

Regions and zones are fundamental to building highly available network services. Depending on availability requirements, you can deploy in multiple zones and regions to minimize the impact of a failure within a data center or geographic area. Availability zone outages can occur, but are rare; regional outages are rarer still. Most services from cloud vendors are aware of zones and use them to achieve built-in redundancy.

Exhibit A: Servers distributed among multiple regions and zones



Multiregion deployments are more complex because of the physical distances between regions and the associated higher latency. Some cloud vendors have faster and more reliable inter-region networks than others. If your site serves users around the world, the quality of your cloud vendor's network is paramount.

Choose regions according to geographic proximity to your user base. For scenarios in which the developers and users are in different geographic regions, consider running your development systems close to the developers and production systems closer to the users.

For sites that deliver services to a global user base, running in multiple regions can substantially improve performance for end users. Requests can be routed to each client's regional servers by exploitation of geographic DNS resolution, which determines clients' locations by their source IP addresses.

Most cloud platforms have regions for North America, South America, Europe, and the Asia Pacific countries. Only AWS and Azure have a direct presence in China. Some platforms, notably AWS and vCloud, have regions compatible with strict U.S. federal ITAR requirements.

Virtual private servers

The flagship service of the cloud is the virtual private server, a virtual machine that runs on the provider's hardware. Virtual private servers are sometimes called instances. You can create as many instances as you need, running your preferred operating system and applications, then shut the instances down when they're no longer needed. You pay only for what you use, and there's typically no up-front cost.

Because instances are virtual machines, their CPU power, memory, disk size, and network settings can be customized when the instance is created and even adjusted after the fact. Public cloud platforms define preset configurations called instance types. They range from single-CPU nodes with 512MiB of memory to large systems with many CPU cores and multiple TiB of memory. Some instance types are balanced for general use, and others are specialized for CPU-, memory-, disk-, or network-intensive applications. Instance configurations are one area in which cloud vendors compete vigorously to match market needs.

Instances are created from "images," the saved state of an operating system that contains (at minimum) a root filesystem and a boot loader. An image might also include disk volumes for additional filesystems and other custom settings. You can easily create custom images with your own software and settings.

All our example operating systems are widely used, so cloud platforms typically supply official images for them. (Currently, you must build your own FreeBSD image if you use Google Compute Engine.) Many third party software vendors also maintain cloud images that have their software preinstalled to facilitate adoption by customers. It's also easy to create your own custom images. Learn more about how to create virtual machine images in *Packer*.

Networking

Cloud providers let you create virtual networks with custom topologies that isolate your systems from each other and from the Internet. On platforms that offer this feature, you can set the address ranges of your networks, define subnets, configure routes, set firewall rules, and construct VPNs to connect to external networks. Expect some network-related operational overhead and maintenance when building larger, more complex cloud deployments.

See this page for more information about VPNs.

See this page for more information about RFC1918 private addresses.

You can make your servers accessible to the Internet by leasing publicly routable addresses from your provider—all providers have a large pool of such addresses from which users can draw. Alternatively, servers can be given only a private RFC1918 address within the address space you selected for your network, rendering them publicly inaccessible.

Systems without public addresses are not directly accessible from the Internet, even for administrative attention. You can access such hosts through a jump server or bastion host that is open to the Internet, or through a VPN that connects to your cloud network. For security, the smaller the external-facing footprint of your virtual empire, the better.

Although this all sounds promising, you have even less control over virtual networks than you do over traditional networks, and you're subject to the whims and vagaries of the feature set made available by your chosen provider. It's particularly maddening when new features launch but can't interact with your private network. (We're looking at you, Amazon!)

Skip to this page for the details on TCP/IP networking in the cloud.

Storage

Data storage is a major part of cloud computing. Cloud providers have the largest and most advanced storage systems on the planet, so you'll be hard pressed to match their capacity and capabilities in a private data center. The cloud vendors bill by the amount of data you store. They are highly motivated to give you as many ways as possible to ingest your data. (Case in point: AWS offers on-site visits from the AWS Snowmobile, a 45-foot long shipping container towed by a semi truck that can transfer 100 PiB from your data center to the cloud.)

Here are a few of the most important ways to store data in the cloud:

- "Object stores" contain collections of discrete objects (files, essentially) in a flat namespace. Object stores can accommodate a virtually unlimited amount of data with exceptionally high reliability but relatively slow performance. They are designed for a read-mostly access pattern. Files in an object store are accessed over the network through HTTPS. Examples include AWS S3 and Google Cloud Storage.
- Block storage devices are virtualized hard disks. They can be requisitioned at your choice of capacities and attached to a virtual server, much like SAN volumes on a traditional network. You can move volumes among nodes and customize their I/O profiles. Examples include AWS EBS and Google persistent disks.
- Ephemeral storage is local disk space on a VPS that is created from disk drives on the host server. These are normally fast and capacious, but the data is lost when you delete the VPS. Therefore, ephemeral storage is best used for temporary files. Examples include instance store volumes on AWS and local SSDs on GCP.

In addition to these raw storage services, cloud providers usually offer a variety of freestanding database services that you can access over the network. Relational databases such as MySQL, PostgreSQL, and Oracle run as services on the AWS Relational Database Service. They offer built-in multizone redundancy and encryption for data at rest.

Distributed analytics databases such as AWS Redshift and GCP BigQuery offer incredible ROI; both are worth a second look before you build your own expensive data warehouse. Cloud vendors also offer the usual assortment of in-memory and NoSQL databases such as Redis and **memcached**.

Identity and authorization

Administrators, developers, and other technical staff all need to manage cloud services. Ideally, access controls should conform to the principle of least privilege: each principal can access only the entities that are relevant to it, and nothing more. Depending on the context, such access control specifications can become quite elaborate.

AWS is exceptionally strong in this area. Their service, called Identity and Access Management (IAM), defines not only users and groups but also roles for systems. A server can be assigned policies, for example, to allow its software to start and stop other servers, store and retrieve data in an object store, or interact with queues—all with automatic key rotation. IAM also has an API for key management to help you store secrets safely.

Other cloud platforms have fewer authorization features. Unsurprisingly, Azure's service is based on Microsoft's Active Directory. It pairs well with sites that have an existing directory to integrate with. Google's access control service, also called IAM, is relatively coarse-grained and incomplete in comparison with Amazon's.

Automation

The APIs and CLI tools created by cloud vendors are the basic building blocks of custom automation, but they're often clumsy and impractical for orchestrating larger collections of resources. For example, what if you need to create a new network, launch several VPS instances, provision a database, configure a firewall, and finally, connect all these components? Written in terms of a raw cloud API, that would make for a complex script.

AWS CloudFormation was the first service to address this problem. It accepts a template in JSON or YAML format that describes the desired resources and their associated configuration details. You submit the template to CloudFormation, which checks it for errors, sorts out dependencies among resources, and creates or updates the cloud configuration according to your specifications.

CloudFormation templates are powerful but error prone in human hands because of their strict syntax requirements. A complete template is unbearably verbose and a challenge for humans to even read. Instead of writing these templates by hand, we prefer to automatically render them with a Python library called Troposphere from Mark Peek (see github.com/cloudtools/troposphere).

Third party services also target this problem. Terraform, from the open source company HashiCorp, is a cloud-agnostic tool for constructing and changing infrastructure. As with CloudFormation, you describe resources in a custom template and then let Terraform make the proper API calls to implement your configuration. You can then check your configuration file into version control and manage the infrastructure over time.

Serverless functions

One of the most innovative features in the cloud since its emergence are the cloud function services, sometimes called functions-as-a-service, also referred to as “serverless” features. Cloud functions are a model of code execution that do not require any long-lived infrastructure. Functions execute in response to an event, such as the arrival of a new HTTP request or an object being uploaded to a storage location.

For example, consider a traditional web server. HTTP requests are forwarded by the networking stack of the operating system to a web server, which routes them appropriately. When the response completes, the web server continues to wait for requests.

Contrast this with the serverless model. An HTTP request arrives, and it triggers the cloud function to handle the response. When complete, the cloud function terminates. The owner pays for the period of time that the function executes. There is no server to maintain and no operating system to manage.

AWS introduced Lambda, their cloud function service, at a conference in 2014. Google followed shortly with a Cloud Functions service. Several cloud function implementations exist for projects like OpenStack, Mesos, and Kubernetes.

Serverless functions hold great promise for the industry. A massive ecosystem of tools is emerging to support simpler and more powerful use of the cloud. We’ve found many uses for these short-lived, serverless functions in our day-to-day administrative duties. We anticipate rapid advances in this area in the coming years.

9.4 CLOUDS: VPS QUICK START BY PLATFORM

The cloud is an excellent sandbox in which to learn UNIX and Linux. This short section helps you get up and running with virtual servers on AWS, GCP, or DigitalOcean. As system administrators, we rely extensively on the command line (as opposed to web GUIs) for interacting with the cloud, so we illustrate the use of those tools here.

Amazon Web Services

To use AWS, first set up an account at aws.amazon.com. Once you create the account, immediately follow the guidance in the AWS Trusted Advisor to configure your account according to the suggested best practices. You can then navigate to the individual service consoles for EC2, VPC, etc.

Each AWS service has a dedicated user interface. When you log in to the web console, you'll see the list of services at the top. Within Amazon, each service is managed by an independent team, and the UI unfortunately reflects this fact. Although this decoupling has helped AWS services grow, it does lead to a somewhat fragmented user experience. Some interfaces are more refined and intuitive than others.

To protect your account, enable multifactor authentication (MFA) for the root user, then create a privileged IAM user for day-to-day use. We also generally configure an alias so that users can access the web console without entering an account number. This option is found on the landing page for IAM.

In the next section we introduce the official **aws** CLI tool written in Python. New users might also benefit from Amazon's Lightsail quick start service, which aims to start an EC2 instance with minimum fuss.

***aws**: control AWS subsystems*

---

*See this page for more information about **pip**.*

---

aws is a unified command-line interface to AWS services. It manages instances, provisions storage, edits DNS records, and performs most of the other tasks shown in the web console. The tool relies on the exceptional Boto library, a Python SDK for the AWS API, and it runs on any system with a working Python interpreter. Install it with **pip**:

```
$ pip install awscli
```

To use **aws**, first authenticate it to the AWS API by using a pair of random strings called the "access key ID" and the "secret access key." You generate these credentials in the IAM web console and then copy-and-paste them locally.

Running **aws configure** prompts you to set your API credentials and default region:

```
$ aws configure
AWS Access Key ID: AKIAIOSFODNN7EXAMPLE
AWS Secret Access Key: wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
Default region name [us-east-1]: <return>
Default output format [None]: <return>
```

These settings are saved to **~/.aws/config**. As long as you're setting up your environment, we also recommend that you configure the **bash** shell's autocompletion feature so that subcommands are easier to discover. See the AWS CLI docs for more information.

The first argument to **aws** names the specific service you want to manipulate; for example, **ec2** for actions that control the Elastic Compute Cloud. You can add the keyword **help** at the end of any command to see instructions. For example, **aws help**, **aws ec2 help**, and **aws ec2 describe-instances help** all produce useful man pages.

Creating an EC2 instance

Use **aws ec2 run-instances** to create and launch EC2 instances. Although you can create multiple instances with a single command (by using the **--count** option), the instances must all share the same configuration. Here's a minimal example of a complete command:

```
$ aws ec2 run-instances --image-id ami-d440a6e7
    --instance-type t2.nano --associate-public-ip-address
    --key-name admin-key
# output shown below
```

This example specifies the following configuration details:

- The base system image is an Amazon-supplied version of CentOS 7 named ami-d440a6e7. (AWS calls their images AMIs, for Amazon Machine Images.) Like other AWS objects, the image names are unfortunately not mnemonic; you must look up IDs in the EC2 web console or on the command line (**aws ec2 describe-images**) to decode them.
- The instance type is t2.nano, which is currently the smallest instance type. It has one CPU core and 512MiB of RAM. Details about the available instance types can be found in the EC2 web console.
- A preconfigured key pair is also assigned to control SSH access. You can generate a key pair with the **ssh-keygen** command (see this page), then upload the public key to the AWS EC2 console.

The output of that **aws ec2 run-instances** command is shown below. It's JSON, so it's easily consumed by other software. For example, after launching an instance, a script could extract the instance's IP address and configure DNS, update an inventory system, or coordinate the launch of multiple servers.

```
$ aws ec2 run-instances ...  # Same command as above
{
    "OwnerId": "188238000000",
    "ReservationId": "r-83a02346",
    "Instances": [
        ...
        "PrivateIpAddress": "10.0.0.27",
        "InstanceId": "i-c4f60303",
        "ImageId": "ami-d440a6e7",
        "PrivateDnsName": "ip-10-0-0-27.us-west-2.compute.internal",
        "KeyName": "admin-key",
        "SecurityGroups": [
            {
                "GroupName": "default",
                "GroupId": "sg-9eb477fb"
            }
        ],
        "SubnetId": "subnet-ef67938a",
        "InstanceType": "t2.nano",
    ...
}
```

By default, EC2 instances in VPC subnets do not have public IP addresses attached, rendering them accessible only from other systems within the same VPC. To reach instances directly from the Internet, use the **--associate-public-ip-address** option, as shown in our example command. You can discover the assigned IP address after the fact with **aws ec2 describe-instances** or by finding the instance in the web console.

Firewalls in EC2 are known as "security groups." Because we didn't specify a security group here, AWS assumes the "default" group, which allows no access. To connect to the instance, adjust the security group to permit SSH from your IP address. In real-world scenarios, security group structure should be carefully planned during network design. We discuss security groups in *Security groups and NACLs*.

aws configure sets a default region, so you need not specify a region for the instance unless you want something other than the default. The AMI, key pair, and subnet are all region-specific, and **aws** complains if they don't exist in the region you specify. (In this particular case, the AMI, key pair, and subnet are from the us-east-1 region.)

Take note of the InstanceId field in the output, which is a unique identifier for the new instance. You can use **aws ec2 describe-instances --instance-id** *id* to show details about an existing instance, or just use **aws ec2 describe-instances** to dump all instances in the default region.

Once the instance is running and the default security group has been adjusted to pass traffic on TCP port 22, you can use SSH to log in. Most AMIs are configured with a nonroot account that has **sudo** privileges. For Ubuntu the username is ubuntu; for CentOS, centos. FreeBSD and

Amazon Linux both use ec2-user. The documentation for your chosen AMI should specify the username if it's not one of these.

See Chapter 8 for more information about user management.

Properly configured images allow only public keys for SSH authentication, not passwords. Once you've logged in with the SSH private key, you'll have full **sudo** access with no password required. We recommend disabling the default user after the first boot and creating personal, named accounts.

Viewing the console log

Debugging low-level problems such as startup issues and disk errors can be challenging without access to the instance's console. EC2 lets you retrieve the console output of an instance, which can be useful if the instance is in an error state or appears to be hung. You can do this through the web interface or with **aws ec2 get-console-output**, as shown:

```
$ aws ec2 get-console-output --instance-id i-c4f60303
{
    "InstanceId": "i-c4f60303",
    "Timestamp": "2015-12-21T00:01:45.000Z",
    "Output": "[  0.000000] Initializing cgroup subsys cpuset\r\n[
        0.000000] Initializing cgroup subsys cpu\r\n[  0.000000]
        Initializing cgroup subsys cpuacct\r\n[  0.000000] Linux version
        4.1.7-15.23.amzn1.x86_64 (mockbuild@gobi-build-60006)
        (gcc version 4.8.3 20140911 (Red Hat 4.8.3-9)) #1 SMP Mon Sep
        14 23:20:33 UTC 2015\r\n
    ...
}
```

The full log is of course much longer than this snippet. In the JSON dump, the contents of the log are unhelpfully concatenated as a single line. For better readability, clean it up with **sed**:

```
$ aws ec2 get-console-output --instance-id i-c4f60303 | sed 's/\\r\\n/\\n/g'
{
    "InstanceId": "i-c4f60303",
    "Timestamp": "2015-12-21T00:01:45.000Z",
    "Output": "[  0.000000] Initializing cgroup subsys cpuset
        [  0.000000] Initializing cgroup subsys cpu
        [  0.000000] Initializing cgroup subsys cpuacct
        [  0.000000] Linux version 4.1.7-15.23.amzn1.x86_64
            (mockbuild@gobi-build-60006) (gcc version 4.8.3 20140911
            (Red Hat 4.8.3-9)) #1 SMP Mon Sep 14 23:20:33 UTC 2015
    ...
}
```

This log output comes directly from the Linux boot process. The example above shows a few lines from the moment the instance was first initialized. In most cases, you'll find the most interesting information near the end of the log.

Stopping and terminating instances

When you're finished with an instance, you can "stop" it to shut the instance down but retain it for later use, or "terminate" it to delete the instance entirely. By default, termination also releases the instance's root disk into the ether. Once terminated, an instance can never be resurrected, even by AWS.

```
$ aws ec2 stop-instances --instance-id i-c4f60303
{
    "StoppingInstances": [
        {
            "InstanceId": "i-c4f60303",
            "CurrentState": {
                "Code": 64,
                "Name": "stopping"
            },
            "PreviousState": {
                "Code": 16,
                "Name": "running"
            }
        }
    ]
}
```

Note that virtual machines don't change state instantly; it takes a minute for the hamsters to reset. Hence the presence of transitional states such as "starting" and "stopping." Be sure to account for them in any instance-wrangling scripts you might write.

Google Cloud Platform

To get started with GCP, establish an account at cloud.google.com. If you already have a Google identity, you can sign up using the same account.

GCP services operate within a compartment known as a project. Each project has separate users, billing details, and API credentials, so you can achieve complete separation between disparate applications or areas of business. Once you create your account, create a project and enable individual GCP services according to your needs. Google Compute Engine, the VPS service, is one of the first services you might want to enable.

*Setting up **gcloud***

gcloud, a Python application, is the CLI tool for GCP. It's a component of the Google Cloud SDK, which contains a variety of libraries and tools for interfacing with GCP. To install it, follow the installation instructions at cloud.google.com/sdk.

Your first action should be to set up your environment by running **gcloud init**. This command starts a small, local web server and then opens a browser link to display the Google UI for authentication. After you authenticate yourself through the web browser, **gcloud** asks you (back in the shell) to select a project profile, a default zone, and other defaults. The settings are saved under **~/.config/gcloud/**.

Run **gcloud help** for general information or **gcloud -h** for a quick usage summary. Per-subcommand help is also available; for example, **gcloud help compute** shows a man page for the Compute Engine service.

Running an instance on GCE

Unlike **aws** commands, which return immediately, **gcloud compute** operates synchronously. When you run the **create** command to provision a new instance, for example, **gcloud** makes the necessary API call, then waits until the instance is actually up and running before it returns. This convention avoids the need to poll for the state of an instance after you create it. (See **aws ec2 wait** for information on polling for events or states within AWS EC2.)

To create an instance, first obtain the name or alias of the image you want to boot:

```
$ gcloud compute images list --regexp 'debian.*'
NAME                        PROJECT       ALIAS     DEPRECATED  STATUS
debian-7-wheezy-v20160119   debian-cloud  debian-7              READY
debian-8-jessie-v20160119   debian-cloud  debian-8              READY
```

Then create and boot the instance, specifying its name and the image you want:

```
$ gcloud compute instances create ulsah --image debian-8
# waits for instance to launch...
NAME   ZONE          MACHINE_TYPE   INTERNAL_IP  EXTERNAL_IP     STATUS
ulsah  us-central1-f  n1-standard-1  10.100.0.4   104.197.65.218  RUNNING
```

The output normally has a column that shows whether the instance is “preemptible,” but in this case it was blank and we removed it to make the output fit on the page. Preemptible instances are less expensive than standard instances, but they can run for only 24 hours and can be terminated at any time if Google needs the resources for another purpose. They’re meant for long-lived operations that can tolerate interruptions, such as batch processing jobs.

Preemptible instances are similar in concept to EC2’s “spot instances” in that you pay a discounted rate for otherwise-spare capacity. However, we’ve found Google’s preemptible instances to be more sensible and simpler to manage than AWS’s spot instances. Long-lived standard instances remain the most appropriate choice for most tasks, however.

gcloud initializes the instance with a public and private IP address. You can use the public IP with SSH, but **gcloud** has a helpful wrapper to simplify SSH logins:

```
$ gcloud compute ssh ulsah
Last login: Mon Jan 25 03:33:48 2016
ulsah:~$
```

Cha-ching!

DigitalOcean

With advertised boot times of 55 seconds, DigitalOcean's virtual servers ("droplets") are the fastest route to a root shell. The entry level cost is $5 per month, so they won't break the bank, either.

See this page for more details on setting up Ruby gems.

Once you create an account, you can manage your droplets through DigitalOcean's web site. However, we find it more convenient to use **tugboat**, a command-line tool written in Ruby that uses DigitalOcean's published API. Assuming that you have Ruby and its library manager, **gem**, installed on your local system, just run **gem install tugboat** to install **tugboat**.

See this page for more about SSH.

A couple of one-time setup steps are required. First, generate a pair of cryptographic keys that you can use to control access to your droplets:

```
$ ssh-keygen -t rsa -b 2048 -f ~/.ssh/id_rsa_do
Generating public/private rsa key pair.
Enter passphrase (empty for no passphrase): <return>
Enter same passphrase again: <return>
Your identification has been saved in /Users/ben/.ssh/id_rsa_do.
Your public key has been saved in /Users/ben/.ssh/id_rsa_do.pub.
```

Copy the contents of the public key file and paste them into DigitalOcean's web console (currently under Settings → Security). As part of that process, assign a short name to the public key.

Next, connect **tugboat** to DigitalOcean's API by entering an access token that you obtain from the web site. **tugboat** saves the token for future use in **~/.tugboat**.

```
$ tugboat authorize
Note: You can get your Access Token from https://cloud.digitalocean.
    com/settings/tokens/new
Enter your access token: e9dff1a9a7ffdd8faf3...f37b015b3d459c2795b64
Enter your SSH key path (defaults to ~/.ssh/id_rsa): ~/.ssh/id_rsa_do
Enter your SSH user (optional, defaults to root):
Enter your SSH port number (optional, defaults to 22):
Enter your default region (optional, defaults to nyc1): sfo1
...
Authentication with DigitalOcean was successful.
```

To create and start a droplet, first identify the name of the system image you want to use as a baseline. For example:

```
$ tugboat images | grep -i ubuntu
16.04.1 x64 (slug: , id: 21669205, distro: Ubuntu)
16.04.1 x64 (slug: , id: 22601368, distro: Ubuntu)
16.04.2 x64 (slug: ubuntu-16-04-x64, id: 23754420, distro: Ubuntu)
16.04.2 x32 (slug: ubuntu-16-04-x32, id: 23754441, distro: Ubuntu)
...
```

You also need DigitalOcean's numeric ID for the SSH key you pasted into the web console:

```
$ tugboat keys
SSH Keys:
Name: id_rsa_do, (id: 1587367), fingerprint:
    bc:32:3f:4d:7d:b0:34:ac:2e:3f:01:f1:e1:ea:2e:da
```

This output shows that the numeric ID for the key named id_rsa_do is 1587367. Create and start a droplet like this:

```
$ tugboat create -i ubuntu-16-04-x64 -k 1587367 ulsah-ubuntu
queueing creation of droplet 'ulsah-ubuntu'...Droplet created!
```

Here, the argument to **-k** is the SSH key ID, and the last argument is a short name for the droplet that you can assign as you wish.

Once the droplet has had time to boot, you can log in with **tugboat ssh**:

```
$ tugboat ssh ulsah-ubuntu
Droplet fuzzy name provided. Finding droplet ID...done, 23754420
    (ubuntu-16-04-x64)
Executing SSH on Droplet (ubuntu-16-04-x64)...
This droplet has a private IP, checking if you asked to use the
    Private IP...
You didn't! Using public IP for ssh...
Attempting SSH: root@45.55.1.165
Welcome to Ubuntu 16.04 ((GNU/Linux 4.4.0-28-generic x86_64)
root@ulsah-ubuntu:~#
```

You can create as many droplets as you need, but keep in mind that you'll be billed for each one, even if it's powered down. To inactivate a droplet, power it down, use **tugboat snapshot** *droplet-name snapshot-name* to memorialize the state of the system, and run **tugboat destroy** *droplet-name* to decommission the droplet. You can later recreate the droplet by using the snapshot as a source image.

9.5 Cost control

Cloud newcomers often naïvely anticipate that large-scale systems will be dramatically cheaper to run in the cloud than in a data center. This expectation might stem from the inverse sticker shock engendered by cloud platforms' low, low price per instance-hour. Or perhaps the idea is implanted by the siren songs of cloud marketers, whose case studies always show massive savings.

Regardless of their source, it's our duty to stamp out hope and optimism wherever they are found. In our experience, new cloud customers are often surprised when costs climb quickly.

Cloud tariffs generally consist of several components:

- The compute resources of virtual private servers, load balancers, and everything else that consumes CPU cycles to run your services. Pricing is per hour of use.
- Internet data transfer (both ingress and egress), as well as traffic among zones and regions. Pricing is per GiB or TiB transferred.
- Storage of all types: block storage volumes, object storage, disk snapshots, and in some cases, I/O to and from the various persistence stores. Pricing is per GiB or TiB stored per month.

For compute resources, the pay-as-you-go model, also known as "on-demand pricing," is the most expensive. On AWS and DigitalOcean, the minimum billing increment is one hour, and on GCP it's a minute. Prices range from fractions of a cent per hour (DigitalOcean's smallest droplet type with 512MiB and one CPU core, or AWS t2.nano instances) to several dollars per hour (an i2.8xlarge instance on AWS with 32 cores, 104GiB RAM, and 8 × 800GB local SSDs).

You can realize substantial savings on virtual servers by paying up front for longer terms. On AWS, this is called "reserved instance pricing." Unfortunately, it's unbearably cumbersome and time-consuming to determine precisely what to purchase. Reserved EC2 instances are tied to a specific instance family. If you decide later that you need something different, your investment is lost. On the upside, if you reserve an instance, you are guaranteed that it will be available for your use. With on-demand instances, your desired type might not even be available when you go to provision it, depending on current capacity and demand. AWS continues to tweak its pricing structure, so with luck the current system might be simplified in the future.

For number crunching workloads that can tolerate interruptions, AWS offers spot pricing. The spot market is an auction. If your bid exceeds the current spot price, you'll be granted use of the instance type you requested until the price exceeds your maximum bid, at which point your instance is terminated. The prices can be deeply discounted compared to the EC2 on-demand and reserved prices, but the use cases are limited.

Google Compute Engine pricing is refreshingly simple by comparison. Discounts are automatically applied for sustained use, and you never pay up front. You pay the full base price for the first week of the month, and the incremental price drops each week by 20% of the base rate, to a maximum discount of 60%. The net discount on a full month of use is 30%. That's roughly comparable to the discount on a one-year reserved EC2 instance, but you can change instances at any time. (For the persnickety and the thrifty: because the discount scheme is linked to your billing cycle, the timing of transitions makes a difference. You can switch instance types at the start or end of a cycle with no penalty. The worst case is to switch halfway through a billing cycle, which incurs a penalty of about 20% of an instance's monthly base rate.)

Network traffic can be even more difficult to predict reliably. The culprits commonly found to be responsible for high data-transfer costs include

- Web sites that ingest and serve large media files (videos, images, PDFs, and other large documents) directly from the cloud, rather than offloading them to a CDN (see this page)
- Inter-zone or inter-region traffic for database clusters that replicate for fault tolerance; for example, software such as Cassandra, MongoDB, and Riak
- MapReduce or data warehouse clusters that span multiple zones
- Disk images and volume snapshots transferred between zones or regions for backup (or by some other automated process)

In situations where replication among multiple zones is important for availability, you'll save on transfer expenses by limiting clusters to two zones rather than using three or more. Some software offers tweaks such as compression that can reduce the amount of replicated data.

One substantial source of expense on AWS is provisioned IOPS for EBS volumes. Pricing for EBS is per GiB-month and IOPS-month. The price of a 200GiB EBS volume with 5,000 IOPS is a few hundred dollars per month. A cluster of these just might break the bank.

The best defense against high bills is to measure, monitor, and avoid overprovisioning. Use autoscaling features to remove capacity when it isn't needed, lowering costs at times of low demand. Use more, smaller instances for more fine-grained control. Watch usage patterns carefully before spending a bundle on reserved instances or high-bandwidth volumes. The cloud is flexible, and you can make changes to your infrastructure as needed.

As environments grow, identifying where money is being spent can be a challenge. Larger cloud accounts might benefit from third party services that analyze use and offer tracking and reporting features. The two that we've used are Cloudability and CloudHealth. Both tap in to the billing features of AWS to break down reports by user-defined tag, service, or geographic location.

9.6 Recommended Reading

Wittig, Andreas, and Michael Wittig. *Amazon Web Services In Action*. Manning Publications, 2015.

Google. cloudplatform.googleblog.com. The official blog for the Google Cloud Platform.

Barr, Jeff, and others at Amazon Web Services. aws.amazon.com/blogs/aws. The official blog of Amazon Web Services.

DigitalOcean. digitalocean.com/company/blog. Technical and product blog from DigitalOcean.

Vogels, Werner. *All Things Distributed*. allthingsdistributed.com. The blog of Werner Vogels, CTO at Amazon.

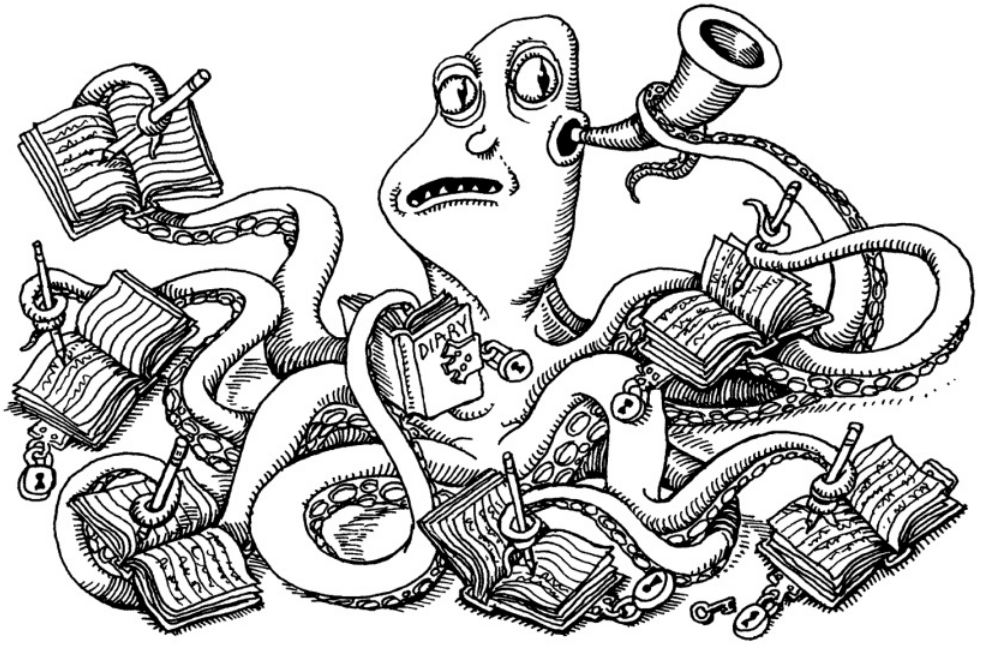
Wardley, Simon. *Bits or pieces?* blog.gardeviance.org. The blog of researcher and cloud trendsetter Simon Wardley. Analysis of cloud industry trends along with occasional rants.

Bias, Randy. cloudscaling.com/blog. Randy Bias is a director at OpenStack and has insightful info on the private cloud industry and its future.

Cantrill, Bryan. *The Observation Deck*. dtrace.org/blogs/bmc. Interesting views and technical thoughts on general computing from the CTO of Joyent, a niche but interesting cloud platform.

Amazon. youtube.com/AmazonWebServices. Conference talks and other video content from AWS.

10 Logging



System daemons, the kernel, and custom applications all emit operational data that is logged and eventually ends up on your finite-sized disks. This data has a limited useful life and may need to be summarized, filtered, searched, analyzed, compressed, and archived before it is eventually discarded. Access and audit logs may need to be managed closely according to regulatory retention rules or site security policies.

A log message is usually a line of text with a few properties attached, including a time stamp, the type and severity of the event, and a process name and ID (PID). The message itself can range from an innocuous note about a new process starting up to a critical error condition or stack trace. It's the responsibility of system administrators to glean useful, actionable information from this ongoing torrent of messages.

This task is known generically as log management, and it can be divided into a few major subtasks:

- Collecting logs from a variety of sources
- Providing a structured interface for querying, analyzing, filtering, and monitoring messages

- Managing the retention and expiration of messages so that information is kept as long as it is potentially useful or legally required, but not indefinitely

UNIX has historically managed logs through an integrated but somewhat rudimentary system, known as syslog, that presents applications with a standardized interface for submitting log messages. Syslog sorts messages and saves them to files or forwards them to another host over the network. Unfortunately, syslog tackles only the first of the logging chores listed above (message collection), and its stock configuration differs widely among operating systems.

Perhaps because of syslog's shortcomings, many applications, network daemons, startup scripts, and other logging vigilantes bypass syslog entirely and write to their own ad hoc log files. This lawlessness has resulted in a complement of logs that varies significantly among flavors of UNIX and even among Linux distributions.

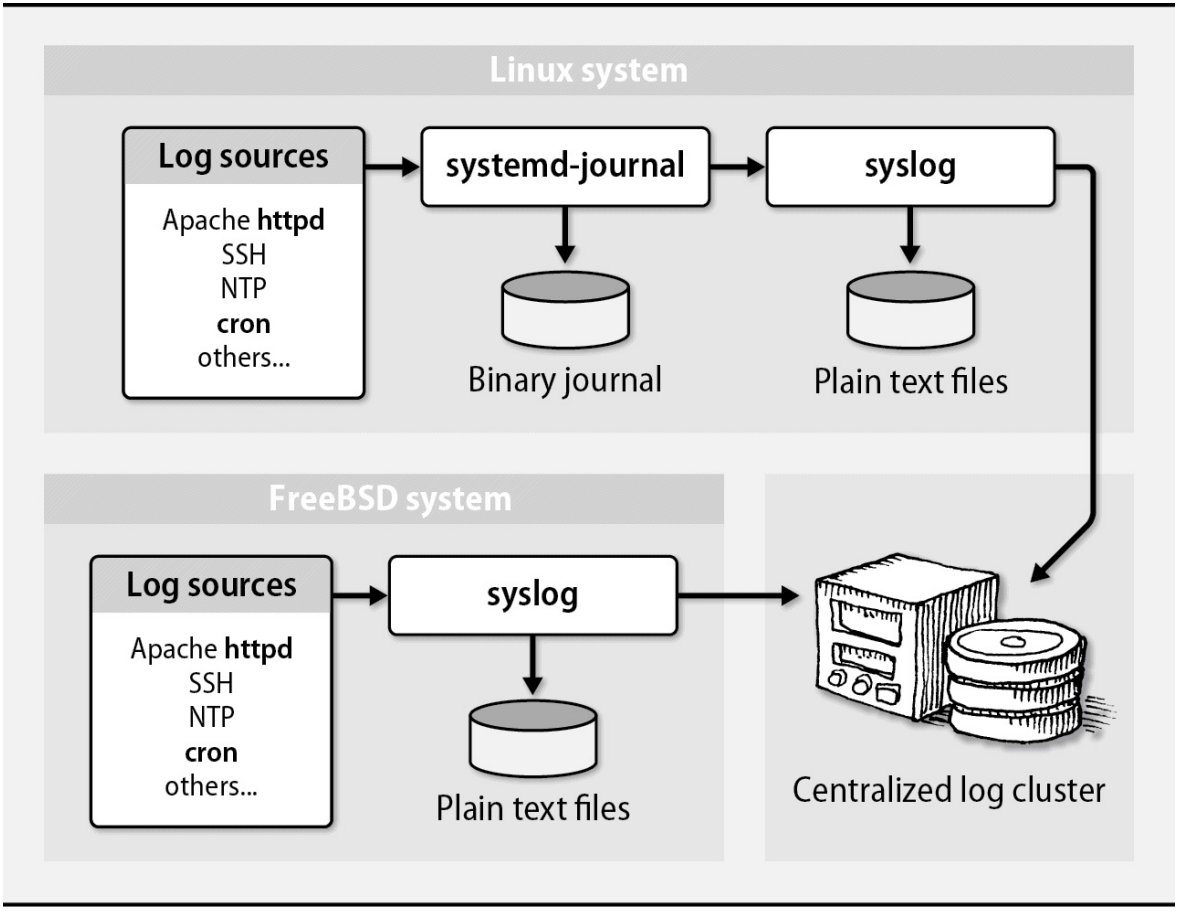


Linux's **systemd** journal represents a second attempt to bring sanity to the logging madness. The journal collects messages, stores them in an indexed and compressed binary format, and furnishes a command-line interface for viewing and filtering logs. The journal can stand alone, or it can coexist with the syslog daemon with varying degrees of integration, depending on the configuration.

A variety of third party tools (both proprietary and open source) address the more complex problem of curating messages that originate from a large network of systems. These tools feature such aids as graphical interfaces, query languages, data visualization, alerting, and automated anomaly detection. They can scale to handle message volumes on the order of terabytes per day. You can subscribe to these products as a cloud service or host them yourself on a private network.

Exhibit A depicts the architecture of a site that uses all the log management services mentioned above. Administrators and other interested parties can run a GUI against the centralized log cluster to review log messages from systems across the network. Administrators can also log in to individual nodes and access messages through the **systemd** journal or the plain text files written by syslog. If this diagram raises more questions than answers for you, you're reading the right chapter.

Exhibit A: Logging architecture for a site with centralized logging



When debugging problems and errors, experienced administrators turn to the logs sooner rather than later. Log files often contain important hints that point toward the source of vexing configuration errors, software bugs, and security issues. Logs are the first place you should look when a daemon crashes or refuses to start, or when a chronic error plagues a system that is trying to boot.

The importance of having a well-defined, site-wide logging strategy has grown along with the adoption of formal IT standards such as PCI DSS, COBIT, and ISO 27001, as well as with the maturing of regulations for individual industries. Today, these external standards may require you to maintain a centralized, hardened, enterprise-wide repository for log activity, with time stamps validated by NTP and with a strictly defined retention schedule. However, even sites without regulatory or compliance requirements can benefit from centralized logging.

This chapter covers the native log management software used on Linux and FreeBSD, including syslog, the **systemd** journal, and **logrotate**. We also introduce some additional tools for centralizing and analyzing logs across the network. The chapter closes with some general advice for setting up a sensible site-wide log management policy.

10.1 LOG LOCATIONS

UNIX is often criticized for being inconsistent, and indeed it is. Just take a look at a directory of log files and you're sure to find some with names like **maillog**, some like **cron.log**, and some that use various distribution- and daemon-specific naming conventions. By default, most of these files are found in **/var/log**, but some renegade applications write their log files elsewhere on the filesystem.

Table 10.1 compiles information about some of the more common log files on our example systems. The table lists the following:

- The log files to archive, summarize, or truncate
- The program that creates each
- An indication of how each filename is specified
- The frequency of cleanup that we consider reasonable
- The systems (among our examples) that use the log file
- A description of the file's contents

Filenames in Table 10.1 are relative to **/var/log** unless otherwise noted. Syslog maintains many of the listed files, but others are written directly by applications.

Table 10.1: Log files on parade

File	Program	Where[a]	Freq[a]	Systems[a]	Contents
apache2/*	**httpd**	F	D	D	Apache HTTP server logs (v2)
apt*	APT	F	M	D	Aptitude package installations
auth.log	**sudo**, etc.[b]	S	M	DF	Authorizations
boot.log	**rc** scripts	F	M	R	Output from system startup scripts
cloud-init.log	**cloud-init**	F	–	–	Output from cloud init scripts
cron, cron/log	**cron**	S	W	RF	**cron** executions and errors
daemon.log	various	S	W	D*	All daemon facility messages
debug*	various	S	D	F,D*	Debugging output
dmesg	kernel	H	–	all	Dump of kernel message buffer
dpkg.log	**dpkg**	F	M	D	Package management log
faillog[c]	**login**	H	W	D*	Failed login attempts
httpd/*	**httpd**	F	D	R	Apache HTTP server logs
kern.log	kernel	S	W	D	All kern facility messages
lastlog	**login**	H	–	R	Last login time per user (binary)
mail*	mail-related	S	W	RF	All mail facility messages
messages	various	S	W	R	The main system log file
samba/*	**smbd**, etc.	F	W	–	Samba (Windows/SMB file sharing)
secure	**sshd**, etc.[b]	S	M	R	Private authorization messages
syslog*	various	S	W	D	The main system log file
wtmp	**login**	H	M	RD	Login records (binary)
xen/*	Xen	F	1m	RD	Xen virtual machine information
Xorg.*n*.log	**Xorg**	F	W	R	X Windows server errors
yum.log	**yum**	F	M	R	Package management log

a. Where: F = Configuration file, H = Hardwired, S = Syslog
Frequency: D = Daily, M = Monthly, NNm = Size-based (in MB, e.g., 1m), W = Weekly
Systems: D = Debian and Ubuntu (D* = Debian only), R = Red Hat and CentOS, F = FreeBSD
b. **passwd**, **sshd**, **login**, and **shutdown** also write to the authorization log.
c. Binary file that must be read with the **faillog** utility

Log files are generally owned by root, although conventions for the ownership and mode of log files vary. In some cases, a less privileged process such as **httpd** may require write access to the log, in which case the ownership and mode should be set appropriately. You might need to use **sudo** to view log files that have tight permissions.

See this page for an introduction to disk partitioning.

Log files can grow quickly, especially the ones for busy services such as web, database, and DNS servers. An out-of-control log file can fill up the disk and bring the system to its knees. For this reason, it's often helpful to define **/var/log** as a separate disk partition or filesystem. (Note that

this advice is just as relevant to cloud-based instances and private virtual machines as it is to physical servers.)

Files not to manage

Most logs are text files to which lines are written as interesting events occur. But a few of the logs listed in Table 10.1 have a rather different context.

wtmp (sometimes **wtmpx**) contains a record of users' logins and logouts as well as entries that record when the system was rebooted or shut down. It's a fairly generic log file in that new entries are simply added to the end of the file. However, the **wtmp** file is maintained in a binary format. Use the **last** command to decode the information.

lastlog contains information similar to that in **wtmp**, but it records only the time of last login for each user. It is a sparse, binary file that's indexed by UID. It will stay smaller if your UIDs are assigned in some kind of numeric sequence, although this is certainly nothing to lose sleep over in the real world. **lastlog** doesn't need to be rotated because its size stays constant unless new users log in.

Finally, some applications (notably, databases) create binary transaction logs. Don't attempt to manage these files. Don't attempt to view them, either, or you'll be treated to a broken terminal window.

How to view logs in the systemd journal

---

*See the sections starting on this page for more information about **systemd** and **systemd** units.*

For Linux distributions running **systemd**, the quickest and easiest way to view logs is to use the **journalctl** command, which prints messages from the **systemd** journal. You can view all messages in the journal, or pass the **-u** flag to view the logs for a specific service unit. You can also filter on other constraints such as time window, process ID, or even the path to a specific executable.

For example, the following output shows journal logs from the SSH daemon:

```
$ journalctl -u ssh
-- Logs begin at Sat 2016-08-27 23:18:17 UTC, end at Sat 2016-08-27
    23:33:20 UTC. --
Aug 27 23:18:24 uxenial sshd[2230]: Server listening on 0.0.0.0 port 22.
Aug 27 23:18:24 uxenial sshd[2230]: Server listening on :: port 22.
Aug 27 23:18:24 uxenial systemd[1]: Starting Secure Shell server...
Aug 27 23:18:24 uxenial systemd[1]: Started OpenBSD Secure Shell server.
Aug 27 23:18:28 uxenial sshd[2326]: Accepted publickey for bwhaley from
    10.0.2.2 port 60341 ssh2: RSA SHA256:aaRfGdl0untn758+UCpxL7gkSwcs
    zkAYe/wukrdBATc
Aug 27 23:18:28 uxenial sshd[2326]: pam_unix(sshd:session): session
    opened for user bwhaley by (uid=0)
Aug 27 23:18:34 uxenial sshd[2480]: Did not receive identification string
    from 10.0.2.2
```

Use **journalctl -f** to print new messages as they arrive. This is the **systemd** equivalent of the much-beloved **tail -f** for following plain text files as they are being appended to.

The next section covers the **systemd-journald** daemon and its configuration.

10.2 THE SYSTEMD JOURNAL



In accordance with its mission to replace all other Linux subsystems, **systemd** includes a logging daemon called **systemd-journald**. It duplicates most of syslog's functions but can also run peacefully in tandem with syslog, depending on how you or the system have configured it. If you're leery of switching to **systemd** because syslog has always "just worked" for you, spend some time to get to know **systemd**. After a little practice, you may be pleasantly surprised.

Unlike syslog, which typically saves log messages to plain text files, the **systemd** journal stores messages in a binary format. All message attributes are indexed automatically, which makes the log easier and faster to search. As discussed above, you can use the **journalctl** command to review messages stored in the journal.

The journal collects and indexes messages from several sources:

- The **/dev/log** socket, to harvest messages from software that submits messages according to syslog conventions
- The device file **/dev/kmsg**, to collect messages from the Linux kernel. The **systemd** journal daemon replaces the traditional **klogd** process that previously listened on this channel and formerly forwarded the kernel messages to syslog.
- The UNIX socket **/run/systemd/journal/stdout**, to service software that writes log messages to standard output
- The UNIX socket **/run/systemd/journal/socket**, to service software that submits messages through the **systemd** journal API
- Audit messages from the kernel's **auditd** daemon

Intrepid administrators can use the **systemd-journal-remote** utility (and its relatives, **systemd-journal-gateway** and **systemd-journal-upload**,) to stream serialized journal messages over the network to a remote journal. Unfortunately, this feature does not come preinstalled on vanilla distributions. As of this writing, packages are available for Debian and Ubuntu but not for Red Hat or CentOS. We expect this lapse to be rectified soon; in the meantime, we recommend sticking with syslog if you need to forward log messages among systems.

Configuring the systemd journal

The default journal configuration file is **/etc/systemd/journald.conf**; however, this file is not intended to be edited directly. Instead, add your customized configurations to the **/etc/systemd/journald.conf.d** directory. Any files placed there with a **.conf** extension are automatically incorporated into the configuration. To set your own options, create a new **.conf** file in this directory and include the options you want.

The default **journald.conf** includes a commented-out version of every possible option, along with each option's default value, so you can see at a glance which options are available. They include the maximum size of journal, the retention period for messages, and various rate-limiting settings.

The Storage option controls whether to save the journal to disk. The possible values are somewhat confusing:

- volatile stores the journal in memory only.
- persistent saves the journal in **/var/log/journal/**, creating the directory if it doesn't already exist.
- auto saves the journal in **/var/log/journal/** but does not create the directory. This is the default value.
- none discards all log data.

Most Linux distributions (including all our examples) default to the value auto and do not come with a **/var/log/journal** directory. Hence, the journal is not saved between reboots by default, which is unfortunate.

You can modify this behavior either by creating the **/var/log/journal** directory or by updating the journal to use persistent storage and restarting **systemd-journald**:

```
# mkdir /etc/systemd/journald.conf.d/
# cat << END > /etc/systemd/journald.conf.d/storage.conf
[Journal]
Storage=persistent
END
# systemctl restart systemd-journald
```

This series of commands creates the custom configuration directory **journald.conf.d**, creates a configuration file to set the Storage option to persistent, and restarts the journal so that the new settings take effect. **systemd-journald** will now create the directory and retain the journal. We recommend this change for all systems; it's a real handicap to lose all log data every time the system reboots.

One of the niftiest journal options is Seal, which enables Forward Secure Sealing (FSS) to increase the integrity of log messages. With FSS enabled, messages submitted to the journal cannot be altered without access to a cryptographic key pair. You generate the key pair itself by running **journalctl --setup-keys**. Refer to the man pages for **journald.conf** and **journalctl** for the full scoop on this option.

Adding more filtering options for journalctl

We showed a quick example of a basic **journalctl** log search here. In this section, we show some additional ways to use **journalctl** to filter messages and gather information about the journal.

To allow normal users to read from the journal without needing **sudo** permissions, add them to the systemd-journal UNIX group.

The **--disk-usage** option shows the size of the journal on disk:

```
# journalctl --disk-usage
Journals take up 4.0M on disk.
```

The **--list-boots** option shows a sequential list of system boots with numerical identifiers. The most recent boot is always 0. The dates at the end of the line show the time stamps of the first and last messages generated during that boot.

```
# journalctl --list-boots
-1 ce0... Sun 2016-11-13 18:54:42 UTC—Mon 2016-11-14 00:09:31
 0 844... Mon 2016-11-14 00:09:38 UTC—Mon 2016-11-14 00:12:56
```

You can use the **-b** option to restrict the log display to a particular boot session. For example, to view logs generated by SSH during the current session:

```
# journalctl -b 0 -u ssh
```

To show all the messages from yesterday at midnight until now:

```
# journalctl --since=yesterday --until=now
```

To show the most recent 100 journal entries from a specific binary:

```
# journalctl -n 100 /usr/sbin/sshd
```

You can use **journalctl --help** as a quick reference for these arguments.

Coexisting with syslog

Both syslog and the **systemd** journal are active by default on each of our example Linux systems. Both packages collect and store log messages. Why would you want both of them running, and how does that even work?

Unfortunately, the journal is missing many of the features that are available in syslog. As the discussion starting on this page demonstrates, rsyslog can receive messages from a variety of input plug-ins and forward them to a diverse set of outputs according to filters and rules, none of which is possible when the **systemd** journal is used. The **systemd** universe does include a remote streaming tool, **systemd-journal-remote**, but it's relatively new and untested in comparison with syslog. Administrators may also find it convenient to keep certain log files in plain text, as syslog does, instead of in the journal's binary format.

We anticipate that over time, new features in the journal will usurp syslog's responsibilities. But for now, Linux distributions still need to run both systems to achieve full functionality.

The mechanics of the interaction between the **systemd** journal and syslog are somewhat convoluted. To begin with, **systemd-journald** takes over responsibility for collecting log messages from **/dev/log**, the logging socket that was historically controlled by syslog. (More specifically, the journal links **/dev/log** to **/run/systemd/journal/dev-log**.) For syslog to get in on the logging action, it must now access the message stream through **systemd**. Syslog can retrieve log messages from the journal in two ways:

- The **systemd** journal can forward messages to another socket (typically **/run/systemd/journal/syslog**), from which the syslog daemon can read them. In this mode of operation, **systemd-journald** simulates the original message submitters and conforms to the standard syslog API. Therefore, only the basic message parameters are forwarded; some **systemd**-specific metadata is lost.

- Alternatively, syslog can consume messages directly from the journal API, in the same manner as the **journalctl** command. This method requires explicit support for cooperation on the part of **syslogd**, but it's a more complete form of integration that preserves the metadata for each message. (See **man systemd.journal-fields** for a rundown of the available metadata.)

Debian and Ubuntu default to the former method, but Red Hat and CentOS use the latter. To determine which type of integration has been configured on your system, inspect the `ForwardToSyslog` option in **/etc/systemd/journald.conf**. If its value is `yes`, socket-forwarding is in use.

10.3 SYSLOG

Syslog, originally written by Eric Allman, is a comprehensive logging system and IETF-standard logging protocol. RFC5424 is the latest version of the syslog specification, but the previous version, RFC3164, may better reflect the real-world installed base.

Syslog has two important functions: to liberate programmers from the tedious mechanics of writing log files, and to give administrators control of logging. Before syslog, every program was free to make up its own logging policy. System administrators had no consistent control over what information was kept or where it was stored.

Syslog is flexible. It lets administrators sort messages by source (“facility”) and importance (“severity level”) and route them to a variety of destinations: log files, users’ terminals, or even other machines. It can accept messages from a wide variety of sources, examine the attributes of the messages, and even modify their contents. Its ability to centralize the logging for a network is one of its most valuable features.

On Linux systems, the original syslog daemon (**syslogd**) has been replaced with a newer implementation called rsyslog (**rsyslogd**). Rsyslog is an open source project that extends the capabilities of the original syslog but maintains backward API compatibility. It is the most reasonable choice for administrators working on modern UNIX and Linux systems and is the only version of syslog we cover in this chapter.



Rsyslog is available for FreeBSD, and we recommend that you adopt it in preference to the standard FreeBSD syslog unless you have simple needs. For instructions on converting a FreeBSD system to use rsyslog, see wiki.rsyslog.com/index.php/FreeBSD. If you decide to stick with FreeBSD’s traditional syslog, jump to this page for configuration information.

Reading syslog messages

You can read plaintext messages from syslog with normal UNIX and Linux text processing tools such as **grep**, **less**, **cat**, and **awk**. The snippet below shows typical events in **/var/log/syslog** from a Debian host:

```
jessie# cat /var/log/syslog
Jul 16 19:43:01 jessie networking[244]: bound to 10.0.2.15 -- renewal
    in 42093 seconds.
Jul 16 19:43:01 jessie rpcbind[397]: Starting rpcbind daemon....
Jul 16 19:43:01 jessie nfs-common[412]: Starting NFS common utilities:
    statd idmapd.
Jul 16 19:43:01 jessie cron[436]: (CRON) INFO (pidfile fd = 3)
Jul 16 19:43:01 jessie cron[436]: (CRON) INFO (Running @reboot jobs)
Jul 16 19:43:01 jessie acpid: starting up with netlink and the input layer
Jul 16 19:43:01 jessie docker[486]: time="2016-07-
    16T19:43:01.972678480Z" level=info msg="Daemon has completed
    initialization"
Jul 16 19:43:01 jessie docker[486]: time="2016-07-
    16T19:43:01.972896608Z" level=info msg="Docker daemon"
    commit=c3959b1 execdriver=native-0.2 graphdriver=aufs
    version=1.10.2
Jul 16 19:43:01 jessie docker[486]: time="2016-07-
    16T19:43:01.979505644Z" level=info msg="API listen on /var/run/
    docker.sock"
```

The example contains entries from several different daemons and subsystems: networking, NFS, **cron**, Docker, and the power management daemon, **acpid**. Each message contains the following space-separated fields:

- Time stamp
- System’s hostname, in this case jessie
- Name of the process and its PID in square brackets
- Message payload

Some daemons encode the payload to add metadata about the message. In the output above, the **docker** process includes its own time stamp, a log level, and information about the configuration of the daemon itself. This additional information is entirely up to the sending process to generate and format.

Rsyslog architecture

Think about log messages as a stream of events and rsyslog as an event-stream processing engine. Log message “events” are submitted as inputs, processed by filters, and forwarded to output destinations. In rsyslog, each of these stages is configurable and modular. By default, rsyslog is configured in **/etc/rsyslog.conf**.

The **rsyslogd** process typically starts at boot and runs continuously. Programs that are syslog aware write log entries to the special file **/dev/log**, a UNIX domain socket. In a stock configuration for systems without **systemd**, **rsyslogd** reads messages from this socket directly, consults its configuration file for guidance on how to route them, and dispatches each message to an appropriate destination. It’s also possible (and common) to configure **rsyslogd** to listen for messages on a network socket.

See this page for more information about signals.

If you modify **/etc/rsyslog.conf** or any of its included files, you must restart the **rsyslogd** daemon to make your changes take effect. A TERM signal makes the daemon exit. A HUP signal causes **rsyslogd** to close all open log files, which is useful for rotating (renaming and restarting) logs.

By longstanding convention, **rsyslogd** writes its process ID to **/var/run/syslogd.pid**, so it’s easy to send signals to **rsyslogd** from a script. (On modern Linux systems, **/var/run** is a symbolic link to **/run**.) For example, the following command sends a hangup signal:

```
$ sudo kill -HUP `/bin/cat /var/run/syslogd.pid`
```

Trying to compress or rotate a log file that **rsyslogd** has open for writing is not healthy and has unpredictable results, so be sure to send a HUP signal before you do this. Refer to this page for information on sane log rotation with the **logrotate** utility.

Rsyslog versions

Red Hat and CentOS use rsyslog version 7, but Debian and Ubuntu have updated to version 8. FreeBSD users installing from ports can choose either version 7 or version 8. As you might expect, the rsyslog project recommends using the most recent version, and we defer to their advice. That said, it won't make or break your logging experience if your operating system of choice is a version behind the latest and greatest.

Rsyslog 8 is a major rewrite of the core engine, and although a lot has changed under the hood for module developers, the user-facing aspects remain mostly unchanged. With a few exceptions, the configurations in the following sections are valid for both versions.

Rsyslog configuration

rsyslogd's behavior is controlled by the settings in **/etc/rsyslog.conf**. All our example Linux distributions include a simple configuration with sensible defaults that suit most sites. Blank lines and lines beginning with a # are ignored. Lines in an rsyslog configuration are processed in order from beginning to end, and order is significant.

At the top of the configuration file are global properties that configure the daemon itself. These lines specify which input modules to load, the default format of messages, ownerships and permissions of files, the working directory in which to maintain rsyslog's state, and other settings. The following example configuration is adapted from the default **rsyslog.conf** on Debian Jessie:

```
# Support local system logging
$ModLoad imuxsock

# Support kernel logging
$ModLoad imklog

# Write messages in the traditional time stamp format
$ActionFileDefaultTemplate RSYSLOG_TraditionalFileFormat

# New log files are owned by root:adm
$FileOwner root
$FileGroup adm

# Default permissions for new files and directories
$FileCreateMode 0640
$DirCreateMode 0755
$Umask 0022

# Location in which to store rsyslog working files
$WorkDirectory /var/spool/rsyslog
```

Most distributions use the $IncludeConfig legacy directive to include additional files from a configuration directory, typically **/etc/rsyslog.d/*.conf**. Because order is important, distributions organize files by preceding file names with numbers. For example, the default Ubuntu configuration includes the following files:

20-ufw.conf
21-cloudinit.conf
50-default.conf

rsyslogd interpolates these files into **/etc/rsyslog.conf** in lexicographic order to form its final configuration.

Filters, sometimes called "selectors," constitute the bulk of an rsyslog configuration. They define how rsyslog sorts and processes messages. Filters are formed from *expressions* that select specific message criteria and *actions* that route selected messages to a desired destination.

Rsyslog understands three configuration syntaxes:

- Lines that use the format of the original syslog configuration file. This format is now known as "**sysklogd** format," after the kernel logging daemon **sysklogd**. It's simple and effective but has some limitations. Use it to construct simple filters.
- Legacy rsyslog directives, which always begin with a `$` sign. The syntax comes from ancient versions of rsyslog and really ought to be obsolete. However, not all options have been converted to the newer syntax, and so this syntax remains authoritative for certain features.
- RainerScript, named for Rainer Gerhards, the lead author of rsyslog. This is a scripting syntax that supports expressions and functions. You can use it to configure most—but not all—aspects of rsyslog.

Many real-world configurations include a mix of all three formats, sometimes to confusing effect. Although it has been around since 2008, RainerScript remains slightly less common than the others. Fortunately, none of the dialects are particularly complex. In addition, many sites will have no need to do major surgery on the vanilla configurations included with their stock distributions.

To migrate from a traditional syslog configuration, simply start with your existing **syslog.conf** file and add options for the rsyslog features you want to activate.

Modules

Rsyslog modules extend the capabilities of the core processing engine. All inputs (sources) and outputs (destinations) are configured through modules, and modules can even parse and mutate messages. Although most modules were written by Rainer Gerhards, some were contributed by third parties. If you're a C programmer, you can write your own.

Module names follow a predictable prefix pattern. Those beginning with `im` are input modules; `om*` are output modules, `mm*` are message modifiers, and so on. Most modules have additional configuration options that customize their behavior. The rsyslog module documentation is the complete reference.

The following list briefly describes some of the more common (or interesting) input and output modules, along with a few nuggets of exotica:

- `imjournal` integrates with the **systemd** journal, as described in *Coexisting with syslog*.
- `imuxsock` reads messages from a UNIX domain socket. This is the default when **systemd** is not present.
- `imklog` understands how to read kernel messages on Linux and BSD.

- imfile converts a plain text file to syslog message format. It's useful for importing log files generated by software that doesn't have native syslog support. Two modes exist: polling mode, which checks the file for updates at a configurable interval, and notification mode (inotify), which uses the Linux filesystem event interface. This module is smart enough to resume where it left off whenever **rsyslogd** is restarted.

- *See this page for more information about TLS.*

---

  imtcp and imudp accept network messages over TCP and UDP, respectively. They allow you to centralize logging on a network. In combination with rsyslog's network stream drivers, the TCP module can also accept mutually authenticated syslog messages through TLS. For Linux sites with extremely high volume, see also the imptcp module.

- If the immark module is present, rsyslog produces time stamp messages at regular intervals. These time stamps can help you figure out that your machine crashed between 3:00 and 3:20 a.m., not just "sometime last night." This information is also a big help when you are debugging problems that seem to occur regularly. Use the MarkMessagePeriod option to configure the mark interval.

- omfile writes messages to a file. This is the most commonly used output module, and the only one configured in a default installation.

- omfwd forwards messages to a remote syslog server over TCP or UDP. This is the module you're looking for if your site needs centralized logging.

- omkafka is a producer implementation for the Apache Kafka data streaming engine. Users at high-volume sites may benefit from being able to process messages that have many potential consumers.

- Similarly to omkafka, omelasticsearch writes directly to an Elasticsearch cluster. See this page for more information about the ELK log management stack, which includes Elasticsearch as one of its components.

- ommysql sends messages to a MySQL database. The rsyslog source distribution includes an example schema. Combine this module with the $MainMsgQueueSize legacy directive for better reliability.

Modules can be loaded and configured through either the legacy or RainerScript configuration formats. We show some examples in the format-specific sections below.

***sysklogd** syntax*

The **sysklogd** syntax is the traditional syslog configuration format. If you encounter a standard **syslogd**, such as the version installed on stock FreeBSD, this is likely all you'll need to

understand. (But note that the configuration file for the traditional **syslogd** is **/etc/syslog.conf**, not **/etc/rsyslog.conf**.)

This format is primarily intended for routing messages of a particular type to a desired destination file or network address. The basic format is

```
selector        action
```

The selector is separated from the action by one or more spaces or tabs. For example, the line

```
auth.*          /var/log/auth.log
```

causes messages related to authentication to be saved in **/var/log/auth.log**.

Selectors identify the source program ("facility") that is sending a log message and the message's priority level ("severity") with the syntax

```
facility.severity
```

Both facility names and severity levels must be chosen from a short list of defined values; programs can't make up their own. Facilities are defined for the kernel, for common groups of utilities, and for locally written programs. Everything else is classified under the generic facility "user."

Selectors can contain the special keywords `*` and `none`, meaning all or nothing, respectively. A selector can include multiple facilities separated by commas. Multiple selectors can be combined with semicolons.

In general, selectors are ORed together: a message matching any selector is subject to the line's *action*. However, a selector with a level of `none` excludes the listed facilities regardless of what other selectors on the same line might say.

Here are some examples of ways to format and combine selectors:

```
# Apply action to everything from facility.level
facility.level                  action

# Everything from facility1.level and facility2.level
facility1,facility2.level       action

# Only facility1.level1 and facility2.level2
facility1.level1;facility2.level2 action

# All facilities with severity level
*.level                         action

# All facilities except badfacility
*.level;badfacility.none        action
```

Table 10.2 lists the valid facility names. They are defined in **syslog.h** in the standard library.

Table 10.2: Syslog facility names

Facility	Programs that use it
*	All facilities except "mark"
auth	Security- and authorization-related commands
authpriv	Sensitive/private authorization messages
cron	The **cron** daemon
daemon	System daemons
ftp	The FTP daemon, **ftpd** (obsolete)
kern	The kernel
local0-7	Eight flavors of local message
lpr	The line printer spooling system
mail	**sendmail, postfix**, and other mail-related software
mark	Time stamps generated at regular intervals
news	The Usenet news system (obsolete)
syslog	**syslogd** internal messages
user	User processes (the default if not specified)

Don't take the distinction between auth and authpriv too seriously. *All* authorization-related messages are sensitive, and none should be world-readable. **sudo** logs use authpriv.

Table 10.3 lists the valid severity levels in order of descending importance.

Table 10.3: Syslog severity levels (descending severity)

Level	Approximate meaning
emerg	Panic situations; system is unusable
alert	Urgent situations; immediate action required
crit	Critical conditions
err	Other error conditions
warning	Warning messages
notice	Things that might merit investigation
info	Informational messages
debug	For debugging only

The severity level of a message specifies its importance. The distinctions between the various levels are sometimes fuzzy. There's a clear difference between notice and warning and between warning and err, but the exact shade of meaning expressed by alert as opposed to crit is a matter of conjecture.

Levels indicate the *minimum* importance that a message must have to be logged. For example, a message from SSH at level warning would match the selector `auth.warning` as well as the selectors

auth.info, auth.notice, auth.debug, *.warning, *.notice, *.info, and *.debug. If the configuration directs auth.info messages to a particular file, auth.warning messages will go there also.

The format also allows the characters = and ! to be prefixed to priority levels to indicate "this priority only" and "except this priority and higher," respectively. Table 10.4 shows examples.

Table 10.4: Examples of priority level qualifiers

Selector	Meaning
`auth.info`	Auth-related messages of info priority and higher
`auth.=info`	Only messages at info priority
`auth.info;auth.!err`	Only priorities info, notice, and warning
`auth.debug;auth.!=warning`	All priorities except warning

The *action* field tells what to do with each message. Table 10.5 lists the options.

Table 10.5: Common actions

Action	Meaning
filename	Appends the message to a file on the local machine
@hostname	Forwards the message to the **rsyslogd** on *hostname*
@ipaddress	Forwards the message to *ipaddress* on UDP port 514
@@ipaddress	Forwards the message to *ipaddress* on TCP port 514
| *fifoname*	Writes the message to the named pipe *fifoname*[a]
user1,user2,…	Writes the message to the screens of *users* if they are logged in
*	Writes the message to all users who are currently logged in
~	Discards the message
^*program;template*	Formats the message according to the *template* specification and sends it to *program* as the first argument[b]

a. See **man mkfifo** for more information.
b. See **man 5 rsyslog.conf** for further details on templates.

If a *filename* (or *fifoname*) action is specified, the name should be an absolute path. If you specify a nonexistent filename, **rsyslogd** will create the file when a message is first directed to it. The ownership and permissions of the file are specified in the global configuration directives as shown on this page.

Here are a few configuration examples that use the traditional syntax:

```
# Kernel messages to kern.log
kern.*                          -/var/log/kern.log

# Cron messages to cron.log
cron.*                          /var/log/cron.log

# Auth messages to auth.log
auth,authpriv.*                 /var/log/auth.log

# All other messages to syslog
*.*;auth,authpriv,cron,kern.none  -/var/log/syslog
```

You can preface a *filename* action with a dash to indicate that the filesystem should not be **sync**ed after each log entry is written. **sync**ing helps preserve as much logging information as possible in the event of a crash, but for busy log files it can be devastating in terms of I/O performance. We recommend including the dashes (and thereby inhibiting **sync**ing) as a matter of course. Remove the dashes only temporarily when investigating a problem that is causing kernel panics.

Legacy directives

Although rsyslog calls these “legacy” options, they remain in widespread use, and you will find them in the majority of rsyslog configurations. Legacy directives can configure all aspects of rsyslog, including global daemon options, modules, filtering, and rules.

In practice, however, these directives are most commonly used to configure modules and the **rsyslogd** daemon itself. Even the rsyslog documentation warns against using the legacy format for message-processing rules, claiming that it is “extremely hard to get right.” Stick with the **sysklogd** or RainerScript formats for actually filtering and processing messages.

Daemon options and modules are straightforward. For example, the options below enable logging over UDP and TCP on the standard syslog port (514). They also permit keep-alive packets to be sent to clients to keep TCP connections open; this option reduces the cost of reconstructing connections that have timed out.

```
$ModLoad imudp
$UDPServerRun 514
$ModLoad imtcp
$InputTCPServerRun 514
$InputTCPServerKeepAlive on
```

To put these options into effect, you could add the lines to a new file to be included in the main configuration such as **/etc/rsyslog.d/10-network-inputs.conf**. Then restart **rsyslogd**. Any options that modify a module’s behavior must appear after the module has been loaded.

Table 10.6 describes a few of the more common legacy directives.

Table 10.6: Rsyslog legacy configuration options

Option	Purpose
`$MainMsgQueueSize`	Size of memory buffer between received and sent messages[a]
`$MaxMessageSize`	Defaults to 8kB; must precede loading of any input modules
`$LocalHostName`	Overrides the local hostname
`$WorkDirectory`	Specifies where to save rsyslog working files
`$ModLoad`	Loads a module
`$MaxOpenFiles`	Modifies the defaults system `nofile` limit for **rsyslogd**
`$IncludeConfig`	Includes additional configuration files
`$UMASK`	Sets the umask for new files created by **rsyslogd**

a. This option is useful for slow outputs such as database inserts.

RainerScript

The RainerScript syntax is an event-stream-processing language with filtering and control-flow capabilities. In theory, you can also set basic **rsyslogd** options through RainerScript. But since some legacy options still don't have RainerScript equivalents, why confuse things by using multiple option syntaxes?

RainerScript is more expressive and human-readable than **rsyslogd**'s legacy directives, but it has an unusual syntax that's unlike any other configuration system we've seen. In practice, it feels somewhat cumbersome. Nonetheless, we recommend it for filtering and rule development if you need those features. In this section we discuss only a subset of its functionality.


Of our example distributions, only Ubuntu uses RainerScript in its default configuration files. However, you can use RainerScript format on any system running rsyslog version 7 or newer.

You can set global daemon parameters by using the global() configuration object. For example:

```
global(
   workDirectory="/var/spool/rsyslog"
   maxMessageSize="8192"
)
```

Most legacy directives have identically named RainerScript counterparts, such as workDirectory and maxMessageSize in the lines above. The equivalent legacy syntax for this configuration would be:

```
$WorkDirectory /var/spool/rsyslog
$MaxMessageSize 8192
```

You can also load modules and set their operating parameters through RainerScript. For example, to load the UDP and TCP modules and apply the same configuration demonstrated on this page, you'd use the following RainerScript:

```
module(load="imudp")
input(type="imudp" port="514")
module(load="imtcp" KeepAlive="on")
input(type="imtcp" port="514")
```

In RainerScript, modules have both “module parameters” and “input parameters.” A module is loaded only once, and a module parameter (e.g., the KeepAlive option in the imtcp module above) applies to the module globally. By contrast, input parameters can be applied to the same module multiple times. For example, we could instruct rsyslog to listen on both TCP ports 514 and 1514:

```
module(load="imtcp" KeepAlive="on")
input(type="imtcp" port="514")
input(type="imtcp" port="1514")
```

Most of the benefits of RainerScript relate to its filtering capabilities. You can use expressions to select messages that match a certain set of characteristics, then apply a particular action to the matching messages. For example, the following lines route authentication-related messages to **/var/log/auth.log**:

```
if $syslogfacility-text == 'auth' then {
    action(type="omfile" file="/var/log/auth.log")
}
```

In this example, $syslogfacility-text is a message property—that is, a part of the message’s metadata. Properties are prefixed by a dollar sign to indicate to rsyslog that they are variables. In this case, the action is to use the omfile output module to write matching messages to **auth.log**.

Table 10.7 lists some of the most frequently used properties.

Table 10.7: Commonly used rsyslog message properties

Property	Meaning
$msg	The text of the message, without metadata
$rawmsg	The full message as received, including metadata
$hostname	The hostname from the message
$syslogfacility	Syslog facility in numerical form; see RFC3164
$syslogfacility-text	Syslog facility in text form
$syslogseverity	Syslog severity in numeric form; see RFC3164
$syslogseverity-text	Syslog severity in text form
$timegenerated	Time at which the message was received by **rsyslogd**
$timereported	Time stamp from the message itself

A given filter can include multiple filters and multiple actions. The following fragment targets kernel messages of critical severity. It logs the messages to a file and sends email to alert an

administrator of the problem.

```
module(load="ommail")

if $syslogseverity-text == 'crit' and $syslogfacility-text == 'kern' then {
    action(type="omfile" file="/var/log/kern-crit.log")
    action(type="ommail"
        server="smtp.admin.com"
        port="25"
        mailfrom="rsyslog@admin.com"
        mailto="ben@admin.com"
        subject.text="Critical kernel error"
        action.execonlyonceeveryinterval="3600"
    )
}
```

Here, we've specified that we don't want more than one email message generated per hour (3,600 seconds).

Filter expressions support regular expressions, functions, and other sophisticated techniques. Refer to the RainerScript documentation for complete details.

Config file examples

In this section we show three sample configurations for rsyslog. The first is a basic but complete configuration that writes log messages to files. The second example is a logging client that forwards syslog messages and **httpd** access and error logs to a central log server. The final example is the corresponding log server that accepts log messages from a variety of logging clients.

These examples rely heavily on RainerScript because it's the suggested syntax for the latest versions of rsyslog. A few of the options are valid only in rsyslog version 8 and include Linux-specific settings such as inotify.

Basic rsyslog configuration

The following file can serve as a generic RainerScript **rsyslog.conf** for any Linux system:

```
module(load="imuxsock")             # Local system logging
module(load="imklog")               # Kernel logging
module(load="immark" interval="3600")  # Hourly mark messages

# Set global rsyslogd parameters
global(
    workDirectory = "/var/spool/rsyslog"
    maxMessageSize = "8192"
)

# The output file module does not need to be explicitly loaded,
# but we can load it ourselves to override default parameter values.

module(load="builtin:omfile"
    # Use traditional timestamp format
    template="RSYSLOG_TraditionalFileFormat"

    # Set the default permissions for all log files.
    fileOwner="root"
    fileGroup="adm"
    dirOwner="root"
$IncludeConfig /etc/rsyslog.d/*.conf

# Include files from /etc/rsyslog.d; there's no RainerScript equivalent

)
    dirCreateMode="0755"
    fileCreateMode="0640"
    dirGroup="adm"
```

This example begins with a few default log collection options for **rsyslogd**. The default file permissions of 0640 for new log files is more restrictive than the omfile default of 0644.

Network logging client

This logging client forwards system logs and the Apache access and error logs to a remote server over TCP.

```
# Send all syslog messages to the server; this is sysklogd syntax
*.*             @@logs.admin.com

# imfile reads messages from a file
# inotify is more efficient than polling
# It's the default, but noted here for illustration
module(load="imfile" mode="inotify")

# Import Apache logs through the imfile module
input(type="imfile"
    Tag="apache-access"
    File="/var/log/apache2/access.log"
    Severity="info"
)
input(type="imfile"
    Tag="apache-error"
    File="/var/log/apache2/error.log"
    Severity="info"
)

# Send Apache logs to the central log host
if $programname contains 'apache' then {
    action(type="omfwd"
        Target="logs.admin.com"
        Port="514"
        Protocol="tcp"
    )
}
```

Apache **httpd** does not write messages to syslog by default, so the access and error logs are read from text files with imfile. The messages are tagged for later use in a filter expression. (**httpd** can log directly to syslog with mod_syslog, but we use imfile here for illustration.)

At the end of the file, the if statement is a filter expression that searches for Apache messages and forwards those to logs.admin.com, the central log server. Logs are sent over TCP, which although more reliable than UDP still can potentially drop messages. You can use RELP (the Reliable Event Logging Protocol), a nonstandard output module, to guarantee log delivery.

See Chapter 23 for more about configuration management.

In a real-world scenario, you might render the Apache-related portion of this configuration to **/etc/rsyslog.d/55-apache.conf** as part of the configuration management setup for the server.

Central logging host

The configuration of the corresponding central log server is straightforward: listen for incoming logs on TCP port 514, filter by log type, and write to files in the site-wide logging directory.

```
# Load the TCP input module and listen on port 514
# Do not accept more than 500 simultaneous clients
module(load="imtcp" MaxSessions="500")
input(type="imtcp" port="514")

# Save to different files based on the type of message
if $programname == 'apache-access' then {
    action(type="omfile" file="/var/log/site/apache/access.log")
} else if $programname == 'apache-error' then {
    action(type="omfile" file="/var/log/site/apache/error.log")
} else {
    # Everything else goes to a site-wide syslog file
    action(type="omfile" file="/var/log/site/syslog")
}
```

The central logging host generates a time stamp for each message as it writes out the message. Apache messages include a separate time stamp that was generated when **httpd** logged the message. You'll find both of these time stamps in the site-wide log files.

Syslog message security

Rsyslog can send and receive log messages over TLS, a layer of encryption and authentication that runs on top of TCP. See this page for general information about TLS.

The example below assumes that the certificate authority, public certificates, and keys have already been generated. See this page for details on public key infrastructure and certificate generation.

This configuration introduces a new option: the network stream driver, a module that operates at a layer between the network and rsyslog. It typically implements features that enhance basic network capabilities. TLS is enabled by the gtls netstream driver.

The following example enables the gtls driver for a log server. The gtls driver requires a CA certificate, a public certificate, and the server's private key. The imtcp module then enables the gtls stream driver.

```
global(
    defaultNetstreamDriver="gtls"
    defaultNetstreamDriverCAFile="/etc/ssl/ca/admin.com.pem"
    defaultNetstreamDriverCertFile="/etc/ssl/certs/server.com.pem"
    defaultNetstreamDriverKeyFile="/etc/ssl/private/server.com.key"
)
module(
    load="imtcp"
    streamDriver.name="gtls"
    streamDriver.mode="1"
    streamDriver.authMode="x509/name"
)
input(type="imtcp" port="6514")
```

The log server listens on the TLS version of the standard syslog port, 6514. The authMode option tells syslog what type of validation to perform. x509/name, the default, checks that the certificate is signed by a trusted authority and also validates the subject name that binds a certificate to a specific client through DNS.

Configuration for the client side of the TLS connection is similar. Use the client certificate and private key, and use the gtls netstream driver for the log forwarding output module.

```
global(
    defaultNetstreamDriver="gtls"
    defaultNetstreamDriverCAFile="/etc/ssl/ca/admin.com.pem"
    defaultNetstreamDriverCertFile="/etc/ssl/certs/client.com.pem"
    defaultNetstreamDriverKeyFile="/etc/ssl/private/client.com.key"
)

*.*     action(type="omfwd"
            Protocol="tcp"
            Target="logs.admin.com"
            Port="6514"
            StreamDriverMode="1"
            StreamDriver="gtls"
            StreamDriverAuthMode="x509/name"
        )
```

In this case, we forward all log messages with a sort of Frankenstein version of the **sysklogd** syntax: the action component is a RainerScript form instead of one of the standard **sysklogd**-native options. If you need to be pickier about which messages to forward (or you need to send different classes of message to different destinations), you can use RainerScript filter expressions, as demonstrated in several of the examples earlier in this chapter.

Syslog configuration debugging

The **logger** command is useful for submitting log entries from shell scripts or the command line. You can also use it to test changes to rsyslog's configuration. For example, if you have just added the line

```
local5.warning      /tmp/evi.log
```

and want to verify that it is working, run the command

```
$ logger -p local5.warning "test message"
```

A line containing "test message" should be written to **/tmp/evi.log**. If this doesn't happen, perhaps you've forgotten to restart **rsyslogd**?

10.4 KERNEL AND BOOT-TIME LOGGING

The kernel and the system startup scripts present some special challenges in the domain of logging. In the case of the kernel, the problem is to create a permanent record of the boot process and kernel operation without building in dependencies on any particular filesystem or filesystem organization. For startup scripts, the challenge is to capture a coherent and accurate narrative of the startup process without permanently tying any system daemons to a startup log file, interfering with any
programs' own logging, or gooping up the startup scripts with glue that serves only to capture boot-time messages.

For kernel logging at boot time, kernel log entries are stored in an internal buffer of limited size. The buffer is large enough to accommodate messages about all the kernel's boot-time activities. When the system is up and running, a user process accesses the kernel's log buffer and disposes of its contents.


On Linux systems, **systemd-journald** reads kernel messages from the kernel buffer by reading the device file **/dev/kmsg**. You can view these messages by running **journalctl -k** or its alias, **journalctl --dmesg**. You can also use the traditional **dmesg** command.


On FreeBSD and older Linux systems, the **dmesg** command is the best way to view the kernel buffer; the output even contains messages that were generated before **init** started.

Another issue related to kernel logging is the appropriate management of the system console. As the system is booting, it's important for all output to come to the console. However, once the system is up and running, console messages may be more an annoyance than a help, especially if the console is used for logins.

Under Linux, **dmesg** lets you set the kernel's console logging level with a command-line flag. For example,

```
ubuntu$ sudo dmesg -n 2
```

Level 7 is the most verbose and includes debugging information. Level 1 includes only panic messages; the lower-numbered levels are the most severe. All kernel messages continue to go to the central buffer (and thence, to syslog) regardless of whether they are forwarded to the console.

10.5 Management and rotation of log files

Erik Troan's **logrotate** utility implements a variety of log management policies and is standard on all our example Linux distributions. It also runs on FreeBSD, but you'll have to install it from the ports collection. By default, FreeBSD uses a different log rotation package, called **newsyslog**; see this page for details.

logrotate: cross-platform log management

A **logrotate** configuration consists of a series of specifications for groups of log files to be managed. Options that appear outside the context of a log file specification (such as errors, rotate, and weekly in the following example) apply to all subsequent specifications. They can be overridden within the specification for a particular file and can also be respecified later in the file to modify the defaults.

Here's a somewhat contrived example that handles several different log files:

```
# Global options
errors errors@book.admin.com
rotate 5
weekly

# Logfile rotation definitions and options
/var/log/messages {
    postrotate
        /bin/kill -HUP `cat /var/run/syslogd.pid`
    endscript
}

/var/log/samba/*.log {
    notifempty
    copytruncate
    sharedscripts
    postrotate
        /bin/kill -HUP `cat /var/lock/samba/*.pid`
    endscript
}
```

This configuration rotates **/var/log/messages** every week. It keeps five versions of the file and notifies rsyslog each time the file is reset. Samba log files (there might be several) are also rotated weekly, but instead of being moved aside and restarted, they are copied and then truncated. The Samba daemons are sent HUP signals only after all log files have been rotated.

Table 10.8 lists the most useful **logrotate.conf** options.

Table 10.8: logrotate options

Option	Meaning
`compress`	Compresses all noncurrent versions of the log file
`daily`, `weekly`, `monthly`	Rotates log files on the specified schedule
`delaycompress`	Compresses all versions but current and next-most-recent
`endscript`	Marks the end of a `prerotate` or `postrotate` script
`errors` *emailaddr*	Emails error notifications to the specified *emailaddr*
`missingok`	Doesn't complain if the log file does not exist
`notifempty`	Doesn't rotate the log file if it is empty
`olddir` *dir*	Specifies that older versions of the log file be placed in *dir*
`postrotate`	Introduces a script to run after the log has been rotated
`prerotate`	Introduces a script to run before any changes are made
`rotate` *n*	Includes *n* versions of the log in the rotation scheme
`sharedscripts`	Runs scripts only once for the entire log group
`size` *logsize*	Rotates if log file size > *logsize* (e.g., `100K`, `4M`)

logrotate is normally run out of **cron** once a day. Its standard configuration file is **/etc/logrotate.conf**, but multiple configuration files (or directories containing configuration files) can appear on **logrotate**'s command line.

This feature is used by Linux distributions, which define the **/etc/logrotate.d** directory as a standard place for **logrotate** config files. **logrotate**-aware software packages (there are many) can drop in log management instructions as part of their installation procedure, thus greatly simplifying administration.

The delaycompress option is worthy of further explanation. Some applications continue to write to the previous log file for a bit after it has been rotated. Use delaycompress to defer compression for one additional rotation cycle. This option results in three types of log files lying around: the active log file, the previously rotated but not yet compressed file, and compressed, rotated files.



In addition to **logrotate**, Ubuntu has a simpler program called **savelog** that manages rotation for individual files. It's more straightforward than **logrotate** and doesn't use (or need) a config file. Some packages prefer to use their own **savelog** configurations rather than **logrotate**.

newsyslog: log management on FreeBSD

The misleadingly named **newsyslog**—so named because it was originally intended to rotate files managed by syslog—is the FreeBSD equivalent of **logrotate**. Its syntax and implementation are entirely different from those of **logrotate,** but aside from its peculiar date formatting, the syntax of a **newsyslog** configuration is actually somewhat simpler.

The primary configuration file is **/etc/newsyslog.conf**. See **man newsyslog** for the format and syntax. The default **/etc/newsyslog.conf** has examples for standard log files.

Like **logrotate**, **newsyslog** runs from **cron**. In a vanilla FreeBSD configuration, **/etc/crontab** includes a line that runs **newsyslog** once per hour.

10.6 Management of logs at scale

It’s one thing to capture log messages, store them on disk, and forward them to a central server. It’s another thing entirely to handle logging data from hundreds or thousands of servers. The message volumes are simply too high to be managed effectively without tools designed to function at this scale. Fortunately, multiple commercial and open source tools are available to address this need.

The ELK stack

The clear leader in the open source space—and indeed, one of the better software suites we've had the pleasure of working with—is the formidable "ELK" stack consisting of Elasticsearch, Logstash, and Kibana. This combination of tools helps you sort, search, analyze, and visualize large volumes of log data generated by a global network of logging clients. ELK is built by Elastic (elastic.co), which also offers training, support, and enterprise add-ons for ELK.

Elasticsearch is a scalable database and search engine with a RESTful API for querying data. It's written in Java. Elasticsearch installations can range from a single node that handles a low volume of data to several dozen nodes in a cluster that indexes many thousands of events each second. Searching and analyzing log data is one of the most popular applications for Elasticsearch.

If Elasticsearch is the hero of the ELK stack, Logstash is its sidekick and trusted partner. Logstash accepts data from many sources, including queueing systems such as RabbitMQ and AWS SQS. It can also read data directly from TCP or UDP sockets and from the traditional logging stalwart, syslog. Logstash can parse messages to add additional structured fields and can filter out unwanted or nonconformant data. Once messages have been ingested, Logstash can write them to a wide variety of destinations, including, of course, Elasticsearch.

You can send log entries to Logstash in a variety of ways. You can configure a syslog input for Logstash and use the rsyslog `omfwd` output module, as described in *Rsyslog configuration*. You can also use a dedicated log shipper. Elastic's own version is called Filebeat and can ship logs either to Logstash or directly to Elasticsearch.

The final ELK component, Kibana, is a graphical front end for Elasticsearch. It gives you a search interface through which to find the entries you need among all the data that has been indexed by Elasticsearch. Kibana can create graphs and visualizations that help to generate new insights about your applications. It's possible, for example, to plot log events on a map to see geographically what's happening with your systems. Other plug-ins add alerting and system monitoring interfaces.

Of course, ELK doesn't come without operational burden. Building a large scale ELK stack with a custom configuration is no simple task, and managing it takes time and expertise. Most administrators we know (present company included!) have accidentally lost data because of bugs in the software or operational errors. If you choose to deploy ELK, be aware that you're signing up for substantial administrative overhead.

We are aware of at least one service, logz.io, that offers production-grade ELK-as-a-service. You can send log messages from your network over encrypted channels to an endpoint that logz.io provides. There, the messages are ingested, indexed, and made available through Kibana. This is not a low-cost solution, but it's worth evaluating. As with many cloud services, you may find that it's ultimately more expensive to replicate the service locally.

Graylog

Graylog is the spunky underdog to ELK's pack leader. It resembles the ELK stack in several ways: it keeps data in Elasticsearch, and it can accept log messages either directly or through Logstash, just as in the ELK stack. The real differentiator is the Graylog UI, which many users proclaim to be superior and easier to use.

See this page for more information about RBAC and this page for more information about LDAP.

Some of the enterprise (read: paid) features of ELK are included in the Graylog open source product, including support for role-based access control and LDAP integration. Graylog is certainly worthy of inclusion in a bake-off when you're choosing a new logging infrastructure.

Logging as a service

Several commercial log management offerings are available. Splunk is the most mature and trusted; both hosted and on-premises versions are available. Some of the largest corporate networks rely on Splunk, not only as a log manager but also as a business analytics system. But if you choose Splunk, be prepared to pay dearly for the privilege.

Alternative SaaS options include Sumo Logic, Loggly, and Papertrail, all of which have native syslog integration and a reasonable search interface. If you use AWS, Amazon's CloudWatch Logs service can collect log data both from AWS services and from your own applications.

10.7 LOGGING POLICIES

Over the years, log management has emerged from the realm of system administration minutiae to become a formidable enterprise management challenge in its own right. IT standards, legislative edicts, and provisions for security-incident handling may all impose requirements on the handling of log data. A majority of sites will eventually need to adopt a holistic and structured approach to the management of this data.

Log data from a single system has a relatively inconsequential effect on storage, but a centralized event register that covers hundreds of servers and dozens of applications is a different story entirely. Thanks in large part to the mission-critical nature of web services, application and daemon logs have become as important as those generated by the operating system.

Keep these questions in mind when designing your logging strategy:

- How many systems and applications will be included?
- What type of storage infrastructure is available?
- How long must logs be retained?
- What types of events are important?

The answers to these questions depend on business requirements and on any applicable standards or regulations. For example, one standard from the Payment Card Industry Security Standards Council requires that logs be retained on easy-access media (e.g., a locally mounted hard disk) for three months and archived to long-term storage for at least one year. The same standard also includes guidance about the types of data that must be included.

Of course, as one of our reviewers mentioned, you can’t be subpoenaed for log data you do not possess. Some sites do not collect (or intentionally destroy) sensitive log data for this reason. You might or might not be able get away with this kind of approach, depending on the compliance requirements that apply to you.

However you answer the questions above, be sure to gather input from your information security and compliance departments if your organization has them.

For most applications, consider capturing at least the following information:

- Username or user ID
- Event success or failure
- Source address for network events
- Date and time (from an authoritative source, such as NTP)

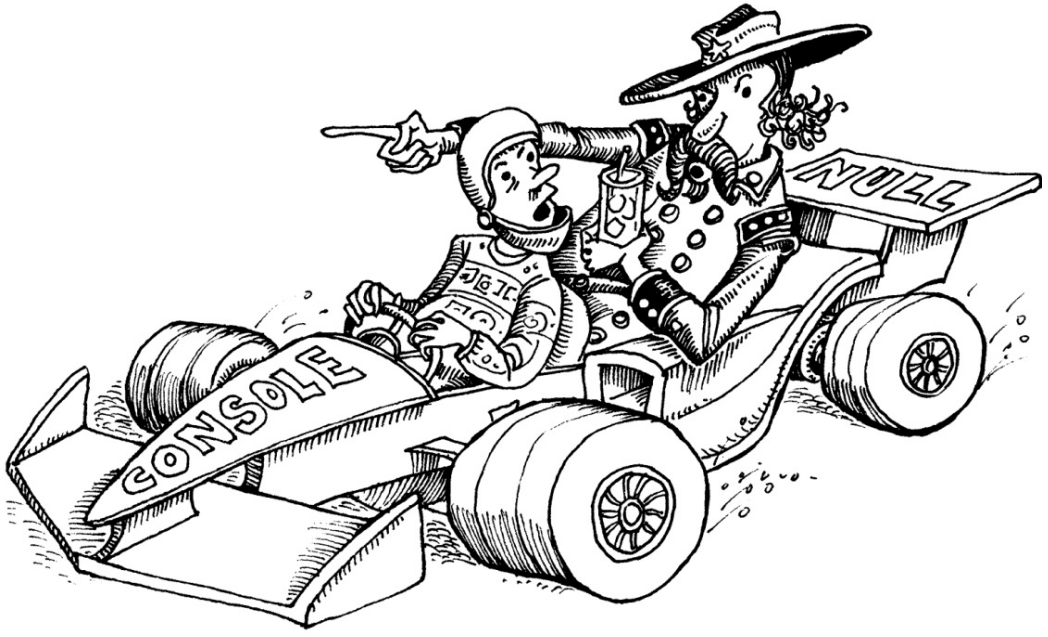
- Sensitive data added, altered, or removed
- Event details

A log server should have a carefully considered storage strategy. For example, a cloud-based system might offer immediate access to 90 days of data, with a year of older data being rolled over to an object storage service and three additional years being saved to an archival storage solution. Storage requirements evolve over time, so a successful implementation must adapt easily to changing conditions.

Limit shell access to centralized log servers to trusted system administrators and personnel involved in addressing compliance and security issues. These log warehouse systems have no real role in the organization's daily business beyond satisfying auditability requirements, so application administrators, end users, and the help desk have no business accessing them. Access to log files on the central servers should itself be logged.

Centralization takes work, and at smaller sites it may not represent a net benefit. We suggest twenty servers as a reasonable threshold for considering centralization. Below that size, just ensure that logs are rotated properly and are archived frequently enough to avoid filling up a disk. Include log files in a monitoring solution that alerts you if a log file stops growing.

11 Drivers and the Kernel



The kernel is the central government of a UNIX or Linux system. It's responsible for enforcing rules, sharing resources, and providing the core services that user processes rely on.

We don't usually think too much about what the kernel is doing. That's fortunate, because even a simple command such as **cat /etc/passwd** entails a complex series of underlying actions. If the system were an airliner, we'd want to think in terms of commands such as "increase altitude to 35,000 feet" rather than having to worry about the thousands of tiny internal steps that were needed to manage the airplane's control surfaces.

The kernel hides the details of the system's hardware underneath an abstract, high-level interface. It's akin to an API for application programmers: a well-defined interface that provides useful facilities for interacting with the system. This interface provides five basic features:

- Management and abstraction of hardware devices
- Processes and threads (and ways to communicate among them)
- Management of memory (virtual memory and memory-space protection)
- I/O facilities (filesystems, network interfaces, serial interfaces, etc.)
- Housekeeping functions (startup, shutdown, timers, multitasking, etc.)

Only device drivers are aware of the specific capabilities and communication protocols of the system's hardware. User programs and the rest of the kernel are largely independent of that knowledge. For example, a filesystem on disk is very different from a network filesystem, but the kernel's VFS layer makes them look the same to user processes and to other parts of the kernel. You don't need to know whether the data you're writing is headed to block 3,829 of disk device #8 or whether it's headed for Ethernet interface e1000e wrapped in a TCP packet. All you need to know is that it will go to the file descriptor you specified.

Processes (and threads, their lightweight cousins) are the mechanisms through which the kernel implements CPU time sharing and memory protection. The kernel fluidly switches among the system's processes, giving each runnable thread a small slice of time in which to get work done. The kernel prevents processes from reading and writing each other's memory spaces unless they have explicit permission to do so.

The memory management system defines an address space for each process and creates the illusion that the process owns an essentially unlimited region of contiguous memory. In reality, different processes' memory pages are jumbled together in the system's physical memory. Only the kernel's bookkeeping and memory protection schemes keep them sorted out.

Layered on top of the hardware device drivers, but below most other parts of the kernel, are the I/O facilities. These consist of filesystem services, the networking subsystem, and various other services that are used to get data into and out from the system.

11.1 Kernel chores for system administrators

Nearly all of the kernel's multilayered functionality is written in C, with a few dabs of assembly language code thrown in to give access to CPU features that are not accessible through C compiler directives (e.g., the atomic read-modify-write instructions defined by many CPUs). Fortunately, you can be a perfectly effective system administrator without being a C programmer and without ever touching kernel code.

That said, it's inevitable that at some point you'll need to make some tweaks. These can take several forms.

Many of the kernel's behaviors (such as network-packet forwarding) are controlled or influenced by tuning parameters that are accessible from user space. Setting these values appropriately for your environment and workload is a common administrative task.

Another common kernel-related task is the installation of new device drivers. New models and types of hardware (video cards, wireless devices, specialized audio cards, etc.) appear on the market constantly, and vendor-distributed kernels aren't always equipped to take advantage of them.

In some cases, you may even need to build a new version of the kernel from source code. Sysadmins don't have to build kernels as frequently as they used to, but it still makes sense in some situations. It's easier than it sounds.

Kernels are tricky. It's surprisingly easy to destabilize the kernel even through minor changes. Even if the kernel boots, it may not run as well as it should. What's worse, you may not even realize that you've hurt performance unless you have a structured plan for assessing the results of your work. Be conservative with kernel changes, especially on production systems, and always have a backup plan for reverting to a known-good configuration.

11.2 Kernel version numbering

Before we dive into the depths of kernel wrangling, it's worth spending a few words to discuss kernel versions and their relationship to distributions.

The Linux and FreeBSD kernels are under continuous active development. Over time, defects are fixed, new features added, and obsolete features removed.

Some older kernels continue to be supported for an extended period of time. Likewise, some distributions choose to emphasize stability and so run the older, more tested kernels. Other distributions try to offer the most recent device support and features but might be a bit less stable as a result. It's incumbent upon you as an administrator to select among these options in a manner that accommodates your users' needs. No single solution is appropriate for every context.

Linux kernel versions


The Linux kernel and the distributions based on it are developed separately from one other, so the kernel has its own versioning scheme. Some kernel releases do achieve a sort of iconic popularity, so it's not unusual to find that several independent distributions are all using the same kernel. You can check with **uname -r** to see what kernel a given system is running.

See semver.org for more information about semantic versioning.

Linux kernels are named according to the rules of so-called semantic versioning, that is, they include three components: a major version, a minor version, and a patch level. At present, there is no predictable relationship between a version number and its intended status as a stable or development kernel; kernels are blessed as stable when the developers decide that they're stable. In addition, the kernel's major version number has historically been incremented somewhat capriciously.

Many stable versions of the Linux kernel can be under long-term maintenance at one time. The kernels shipped by major Linux distributions often lag the latest releases by a substantial margin. Some distributions even ship kernels that are formally out of date.

You can install newer kernels by compiling and installing them from the kernel source tree. However, we don't recommend that you do this. Different distributions have different goals, and they select kernel versions appropriate to those goals. You never know when a distribution has avoided a newer kernel because of some subtle but specific concern. If you need a more recent kernel, install a distribution that's designed around that kernel rather than trying to shoehorn the new kernel into an existing system.

FreeBSD kernel versions



FreeBSD takes a fairly straightforward approach to versions and releases. The project maintains two major production versions, which as of this writing are versions 10 and 11. The kernel has no separate versioning scheme; it's released as part of the complete operating system and shares its version number.

The older of the two major releases (in this case, FreeBSD 10) can be thought of as a maintenance version. It doesn't receive sweeping new features, and it's maintained with a focus on stability and security updates.

The more recent version (FreeBSD 11, right now) is where active development occurs. Stable releases intended for general use are issued from this tree as well. However, the kernel code is always going to be newer and somewhat less battle-tested than that of the previous major version.

In general, dot releases occur about every four months. Major releases are explicitly supported for five years, and the dot releases within them are supported for three months after the next dot release comes out. That's not an extensive lifetime for old dot releases; FreeBSD expects you to stay current with patches.

11.3 Devices and their drivers

A device driver is an abstraction layer that manages the system's interaction with a particular type of hardware so that the rest of the kernel doesn't need to know its specifics. The driver translates between the hardware commands understood by the device and a stylized programming interface defined (and used) by the kernel. The driver layer helps keep the majority of the kernel device-independent.

Given the remarkable pace at which new hardware is developed, it is practically impossible to keep main-line OS distributions up to date with the latest hardware. Hence, you will occasionally need to add a device driver to your system to support a new piece of hardware.

Device drivers are system-specific, and they are often specific to a particular range of kernel revisions as well. Drivers for other operating systems (e.g., Windows) do not work on UNIX and Linux, so keep this in mind when you purchase new hardware. In addition, devices vary in their degree of compatibility and functionality when used with various Linux distributions, so it's wise to pay some attention to the experiences that other sites have had with any hardware you are considering.

Hardware vendors are attracted to the FreeBSD and Linux markets and often publish appropriate drivers for their products. In the optimal case, your vendor furnishes you with both a driver and installation instructions. Occasionally, you might find the driver you need only on some sketchy-looking and uncommented web page. Caveat emptor.

Device files and device numbers

In most cases, device drivers are part of the kernel; they are not user processes. However, a driver can be accessed both from within the kernel and from user space, usually through “device files” that live in the **/dev** directory. The kernel maps operations on these files into calls to the code of the driver.

Most non-network devices have one or more corresponding files in **/dev.** Complex servers may support hundreds of devices. By virtue of being device files, the files in **/dev** each have a major and minor device number associated with them. The kernel uses these numbers to map device-file references to the corresponding driver.

The major device number identifies the driver with which the file is associated (in other words, the type of device). The minor device number usually identifies which particular instance of a given device type is to be addressed. The minor device number is sometimes called the unit number.

You can see the major and minor number of a device file with **ls -l**:

```
linux$ ls -l /dev/sda
brw-rw----  1 root disk 8, 0 Jul 13 01:38 /dev/sda
```

This example shows the first SCSI/SATA/SAS disk on a Linux system. It has a major number of 8 and a minor number of 0.

The minor device number is sometimes used by the driver to select or enable certain characteristics particular to that device. For example, a tape drive can have one file in **/dev** that rewinds the drive automatically when it’s closed and another file that does not. The driver is free to interpret the minor device number in whatever way it likes. Look up the man page for the driver to determine what convention it is using.

There are actually two types of device files: block device files and character device files. A block device is read or written one block (a group of bytes, usually a multiple of 512) at a time; a character device can be read or written one byte at a time. The character `b` at the start of the **ls** output above indicates that **/dev/sda** is a block device; **ls** would show this character as a `c` if it were a character device.

Traditionally, certain devices could act as either block or character devices, and separate device files existed to make them accessible in either mode. Disks and tapes led dual lives, but most other devices did not. However, this parallel access system is not used anymore. FreeBSD represents all formerly dual-mode devices as character devices, and Linux represents them as block devices.

It is sometimes convenient to implement an abstraction as a device driver even when it controls no actual device. Such phantom devices are known as pseudo-devices. For example, a user who

logs in over the network is assigned a pseudo-TTY (PTY) that looks, feels, and smells like a serial port from the perspective of high-level software. This trick allows programs written in the days when everyone used a physical terminal to continue to function in the world of windows and networks. **/dev/zero**, **/dev/null**, and **/dev/urandom** are some other examples of pseudo-devices.

When a program performs an operation on a device file, the kernel intercepts the reference, looks up the appropriate function name in a table, and transfers control to the appropriate part of the driver.

To perform an operation that doesn't have a direct analog in the filesystem model (ejecting a DVD, for example), a program traditionally uses the **ioctl** system call to pass a message directly from user space into the driver. Standard **ioctl** request types are registered by a central authority in a manner similar to the way that network protocol numbers are maintained by IANA.

FreeBSD continues to use the traditional **ioctl** system. Traditional Linux devices also use **ioctl**, but modern networking code uses the more flexible Netlink sockets system described in RFC3549. These sockets provide a more flexible messaging system than **ioctl** without the need for a central authority.

Challenges of device file management

Device files have been a tricky problem for many years. When systems supported only a few types of devices, manual maintenance of device files was manageable. As the number of available devices grew, however, the **/dev** filesystem became cluttered, often with files irrelevant to the current system. Red Hat Enterprise Linux version 3 included more than 18,000 device files, one for every possible device that could be attached to the system! The creation of static device files quickly became a crushing problem and an evolutionary dead end.

USB, FireWire, Thunderbolt, and other device interfaces introduce additional wrinkles. Ideally, a drive that is initially recognized as **/dev/sda** would remain available as **/dev/sda** despite intermittent disconnections and regardless of the activity of other devices and buses. The presence of other transient devices such as cameras, printers, and scanners (not to mention other types of removable media) muddies the waters and makes the persistent identity problem even worse.

Network interfaces have this same problem; they are devices but do not have device files to represent them in **/dev.** For these devices, the modern approach is to use the relatively simple Predictable Network Interface Names system, which assigns interface names that are stable across reboots, changes in hardware, and changes in drivers. Modern systems now have analogous methods for dealing with the names of other devices, too.

Manual creation of device files

Modern systems manage their device files automatically. However, a few rare corner cases may still require you to create devices manually with the **mknod** command. So here's how to do it:

mknod *filename type major minor*

Here, *filename* is the device file to be created, *type* is **c** for a character device or **b** for a block device, and *major* and *minor* are the major and minor device numbers. If you are creating a device file that refers to a driver that's already present in your kernel, check the documentation for the driver to find the appropriate major and minor device numbers.

Modern device file management

Linux and FreeBSD both automate the management of device files. In classic UNIX fashion, the systems are more or less the same in concept but entirely separate in their implementations and in the formats of their configuration files. Let a thousand flowers bloom!

When a new device is detected, both systems automatically create the device's corresponding device files. When a device goes away (e.g., a USB thumb drive is unplugged), its device files are removed. For architectural reasons, both Linux and FreeBSD isolate the "creating device files" part of this equation.

In FreeBSD, devices are created by the kernel in a dedicated filesystem type (devfs) that's mounted on **/dev**. In Linux, a daemon running in user space called **udev** is responsible for this activity. Both systems listen to an underlying stream of kernel-generated events that report the arrival and departure of devices.

However, there's a lot more we might want to do with a newly discovered device than just create a device file for it. If it represents a piece of removable storage media, for example, we might want to automount it as a filesystem. If it's a hub or a communications device, we might want to get it set up with the appropriate kernel subsystem.

Both Linux and FreeBSD leave such advanced procedures to a user-space daemon: **udevd** in the case of Linux, and **devd** in the case of FreeBSD. The main conceptual distinction between the two platforms is that Linux concentrates most intelligence in **udevd**, whereas FreeBSD's devfs filesystem is itself slightly configurable.

Table 11.1 summarizes the components of the device file management systems on both platforms.

Table 11.1: Outline of automatic device management

Component	Linux	FreeBSD
/dev filesystem	udev / devtmpfs	devfs
/dev FS configuration files	–	**/etc/devfs.conf** **/etc/devfs.rules**
Device manager daemon	**udevd**	**devd**
Daemon configuration files	**/etc/udev/udev.conf** **/etc/udev/rules.d** **/lib/udev/rules.d**	**/etc/devd.conf**
FIlesystem automounts	**udevd**	autofs

Linux device management



Linux administrators should understand how **udevd**'s rule system works and should know how to use the **udevadm** command. Before peering into those details, however, let's first review the underlying technology of sysfs, the device information repository from which **udevd** gets its raw data.

Sysfs: a window into the souls of devices

Sysfs was added to the Linux kernel at version 2.6. It is a virtual, in-memory filesystem implemented by the kernel to provide detailed and well-organized information about the system's available devices, their configurations, and their state. Sysfs device information is accessible both from within the kernel and from user space.

You can explore the **/sys** directory, where sysfs is typically mounted, to find out everything from what IRQ a device is using to how many blocks have been queued for writing to a disk controller. One of the guiding principles of sysfs is that each file in **/sys** should represent only one attribute of the underlying device. This convention imposes a certain amount of structure on an otherwise chaotic data set.

Table 11.2 shows the directories within **/sys**, each of which is a subsystem that has been registered with sysfs. The exact directories vary slightly by distribution.

Table 11.2: Subdirectories of /sys

Directory	What it contains
block	Information about block devices such as hard disks
bus	Buses known to the kernel: PCI-E, SCSI, USB, and others
class	A tree organized by functional types of devices[a]
dev	Device information split between character and block devices
devices	An ancestrally correct representation of all discovered devices
firmware	Interfaces to platform-specific subsystems such as ACPI
fs	A directory for some, but not all, filesystems known to the kernel
kernel	Kernel internals such as cache and virtual memory status
module	Dynamic modules loaded by the kernel
power	A few details about the system's power state; mostly unused

a. For example, sound and graphic cards, input devices, and network interfaces

Device configuration information was formerly found in the **/proc** filesystem, if it was available at all. **/proc** was inherited from System V UNIX and grew organically and somewhat randomly over time. It ended up collecting all manner of unrelated information, including many elements unrelated to processes. Although extra junk in **/proc** is still supported for backward

compatibility, **/sys** is a more predictable and organized way of reflecting the kernel's internal data structures. We anticipate that all device-specific information will move to **/sys** over time.

udevadm*: explore devices*

The **udevadm** command queries device information, triggers events, controls the **udevd** daemon, and monitors udev and kernel events. Its primary use for administrators is to build and test rules, which are covered in the next section.

udevadm expects one of six commands as its first argument: **info**, **trigger**, **settle**, **control**, **monitor**, or **test**. Of particular interest to system administrators are **info**, which prints device-specific information, and **control**, which starts and stops **udevd** or forces it to reload its rules files. The **monitor** command displays events as they occur.

The following command shows all udev attributes for the device sdb. The output is truncated here, but in reality it goes on to list all parent devices—such as the USB bus—that are ancestors of sdb in the device tree.

```
linux$ udevadm info -a -n sdb
...
looking at device '/devices/pci0000:00/0000:00:11.0/0000:02:03.0/
    usb1/1-1/1-1:1.0/host6/target6:0:0/6:0:0:0/block/sdb':
    KERNEL=="sdb"
    SUBSYSTEM=="block"
    DRIVER==""
    ATTR{range}=="16"
    ATTR{ext_range}=="256"
    ATTR{removable}=="1"
    ATTR{ro}=="0"
    ATTR{size}=="1974271"
    ATTR{capability}=="53"
    ATTR{stat}==" 71 986 1561 860 1 0 1 12 0 592 872"
...
```

All paths in **udevadm** output (such as **/devices/pci0000:00/…**) are relative to **/sys**, even though they may appear to be absolute pathnames.

The output is formatted so that you can feed it back to udev when constructing rules. For example, if the ATTR{size}=="1974271" clause were unique to this device, you could copy that snippet into a rule as the identifying criterion.

Refer to the man page on **udevadm** for additional options and syntax.

Rules and persistent names

udevd relies on a set of rules to guide its management of devices. The default rules live in the **/lib/udev/rules.d** directory, but local rules belong in **/etc/udev/rules.d**. You need never edit or

delete the default rules; you can ignore or override a file of default rules by creating a new file with the same name in the custom rules directory.

The master configuration file for **udevd** is **/etc/udev/udev.conf**; however, the default behaviors are reasonable. The **udev.conf** files on our example distributions contain only comments, with the exception of one line that enables error logging.

Sadly, because of political bickering among distributors and developers, there is little rule synergy among distributions. Many of the filenames in the default rules directory are the same from distribution to distribution, but the contents of the files differ significantly.

Rule files are named according to the pattern *nn-description*.**rules**, where nn is usually a two-digit number. Files are processed in lexical order, so lower numbers are processed first. Files from the two rules directories are combined before the udev daemon, **udevd**, parses them. The **.rules** suffix is mandatory; files without it are ignored.

Rules are of the form

```
match_clause, [match_clause, ...] assign_clause, [ assign_clause, ... ]
```

The match clauses define the situations in which the rule is to be applied, and the assignment clauses tell **udevd** what to do when a device is consistent with all the rule's match clauses. Each clause consists of a key, an operator, and a value. For example, the match clause `ATTR{size}=="1974271"` was referred to above as a potential component of a rule; it selects all devices whose size attribute is exactly 1,974,271.

Most match keys refer to device properties (which **udevd** obtains from the **/sys** filesystem), but some refer to other context-dependent attributes, such as the operation being handled (e.g., device addition or removal). All match clauses must match in order for a rule to be activated.

Table 11.3 shows the match keys understood by **udevd**.

Table 11.3: udevd match keys

Match key	Function
ACTION	Matches the event type, e.g., add or remove
ATTR{*filename*}	Matches a device's sysfs values[a]
DEVPATH	Matches a specific device path
DRIVER	Matches the driver used by a device
ENV{*key*}	Matches the value of an environment variable
KERNEL	Matches the kernel's name for the device
PROGRAM	Runs an external command; matches if the return code is 0
RESULT	Matches the output of the last call through PROGRAM
SUBSYSTEM	Matches a specific subsystem
TEST{*omask*}	Tests whether a file exists; the *omask* is optional

a. The *filename* is a leaf in the sysfs tree that corresponds to a specific attribute.

The assignment clauses specify actions **udevd** should take to handle any matching events. Their format is similar to that for match clauses.

The most important assignment key is NAME, which indicates how **udevd** should name a new device. The optional SYMLINK assignment key creates a symbolic link to the device through its desired path in **/dev.**

Here, we put these components together with an example configuration for a USB flash drive. Suppose we want to make the drive's device name persist across insertions and we want the drive to be mounted and unmounted automatically.

To start with, we insert the flash drive and check to see how the kernel identifies it. This task can be approached in a couple of ways. By running the **lsusb** command, we can inspect the USB bus directly:

```
ubuntu$ lsusb
Bus 001 Device 007: ID 1307:0163 Transcend, Inc. USB Flash Drive
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 002 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
```

Alternatively, we can check for kernel log entries by running **dmesg** or **journalctl**. In our case, the attachment leaves an extensive audit trail:

```
Aug  9 19:50:03 ubuntu kernel: [42689.253554] scsi 8:0:0:0:
    Direct-Access    Ut163   USB2FlashStorage 0.00 PQ: 0 ANSI: 2
Aug  9 19:50:03 ubuntu kernel: [42689.292226] sd 8:0:0:0: [sdb]
    1974271 512-byte hardware sectors: (1.01 GB/963 MiB)
...
Aug  9 19:50:03 ubuntu kernel: [42689.304749] sd 8:0:0:0: [sdb]
    1974271 512-byte hardware sectors: (1.01 GB/963 MiB)
Aug  9 19:50:03 ubuntu kernel: [42689.307182]  sdb: sdb1
Aug  9 19:50:03 ubuntu kernel: [42689.427785] sd 8:0:0:0: [sdb]
    Attached SCSI removable disk
Aug  9 19:50:03 ubuntu kernel: [42689.428405] sd 8:0:0:0: Attached
    scsi generic sg3 type 0
```

The log messages above indicate that the drive was recognized as sdb, which gives us an easy way to identify the device in **/sys.** We can now examine the **/sys** filesystem with **udevadm** in search of some rule snippets that are characteristic of the device and so might be useful to incorporate in udev rules.

```
ubuntu$ udevadm info -a -p /block/sdb/sdb1
looking at device '/devices/pci0000:00/0000:00:11.0/0000:02:03.0/
    usb1/1-1/1-1:1.0/host30/target30:0:0/30:0:0:0/block/sdb/sdb1':
    KERNEL=="sdb1"
    SUBSYSTEM=="block"
    DRIVER==""
    ATTR{partition}=="1"
    ATTR{start}=="63"
    ATTR{size}=="1974208"
    ATTR{stat}=="  71  792  1857  808  0  0  0  0  0  512  808"

  looking at parent device '/devices/pci0000:00/0000:00:11.0/0000:02:03
    .0/usb1/1-1/1-1:1.0/host30/target30:0:0/30:0:0:0/block/sdb':
    KERNELS=="sdb"
    SUBSYSTEMS=="block"
    DRIVERS==""
    ATTRS{scsi_level}=="3"
    ATTRS{vendor}=="Ut163   "
    ATTRS{model}=="USB2FlashStorage"
...
```

The output from **udevadm** shows several opportunities for matching. One possibility is the size field, which is likely to be unique to this device. However, if the size of the partition were to change, the device would not be recognized. Instead, we can use a combination of two values: the kernel's naming convention of sd plus an additional letter, and the contents of the model attribute, USB2FlashStorage. For creating rules specific to this particular flash drive, another good choice would be the device's serial number (which we've omitted from the output here).

We next put our rules for this device in the file **/etc/udev/rules.d/10-local.rules**. Because we have multiple objectives in mind, we need a series of rules.

First, we take care of creating device symlinks in **/dev.** The following rule uses our knowledge of the ATTRS and KERNEL match keys, gleaned from **udevadm**, to identify the device:

```
ATTRS{model}=="USB2FlashStorage", KERNEL=="sd[a-z]1",
    SYMLINK+="ate-flash%n"
```

(The rule has been folded here to fit the page; in the original file, it's all one line.)

When the rule triggers, **udevd** sets up **/dev/ate-flash***N* as a symbolic link to the device (where *N* is the next integer in sequence, starting at 0). We don't really expect more than one of these devices to appear on the system. If more copies do appear, they receive unique names in **/dev**, but the exact names will depend on the insertion order of the devices.

Next, we use the ACTION key to run some commands whenever the device appears on the USB bus. The RUN assignment key lets us create an appropriate mount point directory and mount the device there.

```
ACTION=="add", ATTRS{model}=="USB2FlashStorage", KERNEL=="sd[a-z]1",
    RUN+="/bin/mkdir -p /mnt/ate-flash%n"
ACTION=="add", ATTRS{model}=="USB2FlashStorage", KERNEL=="sd[a-z]1",
    PROGRAM=="/lib/udev/vol_id -t %N", RESULT=="vfat",
    RUN+="/bin/mount vfat /dev/%k /mnt/ate-flash%n"
```

The PROGRAM and RUN keys look similar, but PROGRAM is a match key that's active during the rule selection phase, whereas RUN is an assignment key that's part of the rule's actions once triggered. The second rule above verifies that the flash drive contains a Windows filesystem before mounting it with the **-t vfat** option to the **mount** command.

Similar rules clean up when the device is removed:

```
ACTION=="remove", ATTRS{model}=="USB2FlashStorage",
    KERNEL=="sd[a-z]1", RUN+="/bin/umount -l /mnt/ate-flash%n"
ACTION=="remove", ATTRS{model}=="USB2FlashStorage",
    KERNEL=="sd[a-z]1", RUN+="/bin/rmdir /mnt/ate-flash%n"
```

Now that our rules are in place, we must notify **udevd** of our changes. **udevadm**'s **control** command is one of the few that require root privileges:

```
ubuntu$ sudo udevadm control --reload-rules
```

Typos are silently ignored after a reload, even with the **--debug** flag, so be sure to double-check the rules' syntax.

That's it! Now when the flash drive is plugged into a USB port, **udevd** creates a symbolic link called **/dev/ate-flash1** and mounts the drive as **/mnt/ate-flash1**.

```
ubuntu$ ls -l /dev/ate*
lrwxrwxrwx 1 root root 4 2009-08-09 21:22 /dev/ate-flash1 -> sdb1

ubuntu$ mount | grep ate
/dev/sdb1 on /mnt/ate-flash1 type vfat (rw)
```

FreeBSD device management


As we saw in the brief overview on this page, FreeBSD's implementation of the self-managing **/dev** filesystem is called devfs, and its user-level device management daemon is called **devd**.

Devfs: automatic device file configuration

Unlike Linux's udev filesystem, devfs itself is somewhat configurable. However, the configuration system is both peculiar and rather impotent. It's split into boot-time (**/etc/devfs.conf**) and dynamic (**/etc/devfs.rules**) portions. The two configuration files have different syntaxes and somewhat different capabilities.

Devfs for static (nonremovable) devices is configured in **/etc/devfs.conf**. Each line is a rule that starts with an action. The possible actions are link, own, and perm. The link action sets up symbolic links for specific devices. The own and perm actions change the ownerships and permissions of device files, respectively.

Each action accepts two parameters, the interpretation of which depends on the specific action. For example, suppose we want our DVD-ROM drive **/dev/cd0** to also be accessible by the name **/dev/dvd**. The following line would do the trick:

```
link cd0 dvd
```

We could set the ownerships and permissions on the device with the following lines.

```
own cd0 root:sysadmin
perm cd0 0660
```

Just as **/etc/devfs.conf** specifies actions to take for built-in devices, **/etc/devfs.rules** contains rules for removable devices. Rules in **devfs.rules** also have the option to make devices hidden or inaccessible, which can be useful for **jail**(8) environments.

***devd**: higher-level device management*

The **devd** daemon runs in the background, watching for kernel events related to devices and acting on the rules defined in **/etc/devd.conf**. The configuration of **devd** is detailed in the **devd.conf** man page, but the default **devd.conf** file includes many useful examples and enlightening comments.

The format of **/etc/devd.conf** is conceptually simple, consisting of "statements" containing groups of "substatements". Statements are essentially rules, and substatements provide details about the rule. Table 11.4 lists the available statement types.

Table 11.4: Statement types in /etc/devd.conf

Statement	What it specifies
`attach`	What to do when a device is inserted
`detach`	What to do when a device is removed
`nomatch`	What to do if no other statements match a device
`notify`	How to respond to kernel events about a device
`options`	Configuration options for **devd** itself

Despite being conceptually simple, the configuration language for substatements is rich and complex. For this reason, many of the common configuration statements are already included in the standard distribution's configuration file. In many cases, you will never need to modify the default **/etc/devd.conf**.

Automatic mounting of removable media devices such as USB hard disks and thumb drives is now handled by FreeBSD's implementation of autofs, not by **devd**. See this page for general information about autofs. Although autofs is found on most UNIX-like operating systems, FreeBSD is unusual in assigning it this extra task.

11.4 Linux kernel configuration



You can use any of three basic methods to configure a Linux kernel. Chances are that you will have the opportunity to try all of them eventually. The methods are

- Modifying tunable (dynamic) kernel configuration parameters
- Building a kernel from scratch (by compiling it from the source code, possibly with modifications and additions)
- Loading new drivers and modules into an existing kernel on the fly

These procedures are used in different situations, so learning which approaches are needed for which tasks is half the battle. Modifying tunable parameters is the easiest and most common kernel tweak, whereas building a kernel from source code is the hardest and least often required. Fortunately, all these approaches become second nature with a little practice.

Tuning Linux kernel parameters

Many modules and drivers in the kernel were designed with the knowledge that one size doesn't fit all. To increase flexibility, special hooks allow parameters such as an internal table's size or the kernel's behavior in a particular circumstance to be adjusted on the fly by the system administrator. These hooks are accessible through an extensive kernel-to-userland interface represented by files in the **/proc** filesystem (aka procfs). In some cases, a large user-level application (especially an infrastructure application such as a database) might require a sysadmin to adjust kernel parameters to accommodate its needs.

You can view and set kernel options at run time through special files in **/proc/sys**. These files mimic standard Linux files, but they are really back doors into the kernel. If a file in **/proc/sys** contains a value you want to change, you can try writing to it. Unfortunately, not all files are writable (regardless of their apparent permissions), and not much documentation is available. If you have the kernel source tree installed, you may be able to read about some of the values and their meanings in the subdirectory **Documentation/sysctl** (or on-line at kernel.org/doc).

For example, to change the maximum number of files the system can have open at once, try something like

```
linux# cat /proc/sys/fs/file-max
34916
linux# echo 32768 > /proc/sys/fs/file-max
```

Once you get used to this unorthodox interface, you'll find it quite useful. However, note that changes are not remembered across reboots.

A more permanent way to modify these same parameters is to use the **sysctl** command. **sysctl** can set individual variables either from the command line or from a list of *variable=value* pairs in a configuration file. By default, **/etc/sysctl.conf** is read at boot time and its contents are used to set the initial values of parameters.

For example, the command

```
linux# sysctl net.ipv4.ip_forward=0
```

turns off IP forwarding. (Alternatively, you can manually edit **/etc/sysctl.conf**.) You form the variable names used by **sysctl** by replacing the slashes in the **/proc/sys** directory structure with dots.

Table 11.5 lists some commonly tuned parameters for Linux kernel version 3.10.0 and higher. Default values vary widely among distributions.

Table 11.5: Files in /proc/sys for some tunable kernel parameters

File	What it does
cdrom/autoclose	Autocloses the CD-ROM when mounted
cdrom/autoeject	Autoejects the CD-ROM when unmounted
fs/file-max	Sets max number of open files
kernel/ctrl-alt-del	Reboots on <Control-Alt-Delete>; may increase security on unsecured consoles
kernel/panic	Sets seconds to wait before rebooting after a kernel panic: 0 = loop or hang indefinitely
kernel/panic_on_oops	Determines the kernel's behavior after encountering an oops or a bug: 1 = always panic
kernel/printk_ratelimit	Sets minimum seconds between kernel messages
kernel/printk_ratelimit_burst	Sets number of messages in succession before the **printk** rate limit is actually enforced
kernel/shmmax	Sets max amount of shared memory
net/ip*/conf/default/rp_filter	Enables IP source route verification[a]
net/ip*/icmp_echo_ignore_all	Ignores ICMP pings when set to 1[b]
net/ip*/ip_forward	Allows IP forwarding when set to 1[c]
net/ip*/ip_local_port_range	Sets local port range used during connection setup[d]
net/ip*/tcp_syncookies	Protects against SYN flood attacks; turn on if you suspect denial-of-service (DoS) attacks
tcp_fin_timeout	Sets seconds to wait for a final TCP FIN packet[e]
vm/overcommit_memory	Controls memory overcommit behavior, i.e., how the kernel reacts when physical memory is insufficient to handle a VM allocation request
vm/overcommit_ratio	Defines how much physical memory (as a percentage) will be used when overcommitting

a. This antispoofing mechanism makes the kernel drop packets received from "impossible" paths.
b. The related variable **icmp_echo_ignore_broadcasts** ignores broadcast ICMP pings. It's almost always a good idea to set this value to 1.
c. Only set this value to 1 if you explicitly intend to use your Linux box as a network router.
d. Increase this range to 1024–65000 on servers that initiate many outbound connections.
e. Try setting this value lower (~20) on high-traffic servers to increase performance.

Note that there are two IP networking subdirectories of **/proc/sys/net**: **ipv4** and **ipv6**. In the past, administrators only had to worry about IPv4 behaviors because that was the only game in town. But as of this writing (2017), the IPv4 address blocks have all been assigned, and IPv6 is deployed and in use almost everywhere, even within smaller organizations.

In general, when you change a parameter for IPv4, you should also change that parameter for IPv6, if you are supporting both protocols. It's all too easy to modify one version of IP and not the other, then run into problems several months or years later when a user reports strange network behavior.

Building a custom kernel

Because Linux evolves rapidly, you'll likely be faced with the need to build a custom kernel at some point or another. The steady flow of kernel patches, device drivers, and new features that arrive on the scene is something of a mixed blessing. On one hand, it's a privilege to live at the center of an active and vibrant software ecosystem. On the other hand, just keeping abreast of the constant flow of new material can be a job of its own.

If it ain't broke, don't fix it

Carefully weigh your site's needs and risks when planning kernel upgrades and patches. A new release may be the latest and greatest, but is it as stable as the current version? Could the upgrade or patch be delayed and installed with another group of patches at the end of the month? Resist the temptation to let keeping up with the Joneses (in this case, the kernel-hacking community) dominate the best interests of your user community.

A good rule of thumb is to upgrade or apply patches only when the productivity gains you expect to obtain (usually measured in terms of reliability and performance) exceed the effort and lost time required for the installation. If you're having trouble quantifying the specific gain, that's a good sign that the patch can wait for another day. (Of course, security-related patches should be installed promptly.)

Setting up to build the Linux kernel

It's less likely that you'll need to build a kernel on your own if you're running a distribution that uses a "stable" kernel to begin with. It used to be that the second part of the version number indicated whether the kernel was stable (even numbers) or in development (odd numbers). But these days, the kernel developers no longer follow that system. Check kernel.org for the official word on any particular kernel version. The kernel.org site is also the best source for Linux kernel source code if you are not relying on a particular distribution or vendor to provide you with a kernel.

Each distribution has a specific way to configure and build custom kernels. However, distributions also support the traditional way of doing things, which is what we describe here. It's generally safest to use your distributor's recommended procedure.

Configuring kernel options

Most distributions install kernel source files in versioned subdirectories under **/usr/src/kernels**. In all cases, you need to install the kernel source package before you can build a kernel on your system. See Chapter 6, *Software Installation and Management*, for tips on package installation.

Kernel configuration revolves around the **.config** file at the root of the kernel source directory. All the kernel configuration information is specified in this file, but its format is somewhat cryptic. Use the decoding guide in

kernel_src_dir/**Documentation/Configure.help**

to find out what the various options mean. It's usually inadvisable to edit the **.config** file by hand because the effect of changing options is not always obvious. Options are frequently interdependent, so turning on an option might not be a simple matter of changing an `n` to a `y`.

To save folks from having to edit the **.config** file directly, Linux has several **make** targets that help you configure the kernel through a user interface. If you are running KDE, the prettiest configuration interface is provided by **make xconfig**. Likewise, if you're running GNOME, **make gconfig** is probably the best option. These commands bring up a graphical configuration screen in which you can pick the devices to add to your kernel (or to compile as loadable modules).

If you are not running KDE or GNOME, you can use a terminal-based alternative invoked with **make menuconfig**. Finally, the bare-bones **make config** prompts you to respond to every single configuration option that's available, which results in a lot of questions—and if you change your mind, you have to start over. We recommend **make xconfig** or **make gconfig** if your environment supports them; otherwise, use **make menuconfig**. Avoid **make config**, the least flexible and most painful option.

If you're migrating an existing kernel configuration to a new kernel version (or tree), you can use the **make oldconfig** target to read in the previous config file and ask only the questions that are new to this edition of the kernel.

These tools are straightforward as far as the options you can turn on. Unfortunately, they are painful to use if you want to maintain multiple versions of the kernel to accompany multiple architectures or hardware configurations found in your environment.

All the various configuration interfaces described above generate a **.config** file that looks something like this:

```
# Automatically generated make config: don't edit
# Code maturity level options

CONFIG_EXPERIMENTAL=y

# Processor type and features
# CONFIG_M386 is not set
# CONFIG_M486 is not set
# CONFIG_M586 is not set
# CONFIG_M586TSC is not set
CONFIG_M686=y
CONFIG_X86_WP_WORKS_OK=y
CONFIG_X86_INVLPG=y
CONFIG_X86_BSWAP=y
CONFIG_X86_POPAD_OK=y
CONFIG_X86_TSC=y
CONFIG_X86_GOOD_APIC=y
...
```

As you can see, the contents are cryptic and do not attempt to describe what the various CONFIG tags mean. Each line refers to a specific kernel configuration option. The value `y` compiles the option into the kernel, and the value `m` enables the option as a loadable module.

Some options can be configured as modules and some can't. You just have to know which is which; it will not be clear from the **.config** file. Nor are the CONFIG tags easily mapped to meaningful information.

The option hierarchy is extensive, so set aside many hours if you plan to scrutinize every possibility.

Building the kernel binary

Setting up an appropriate **.config** file is the most important part of the Linux kernel configuration process, but you must jump through several more hoops to turn that file into a finished kernel.

Here's an outline of the entire process:

1. Change directory (**cd**) to the top level of the kernel source directory.
2. Run **make xconfig**, **make gconfig**, or **make menuconfig**.
3. Run **make clean**.
4. Run **make**.
5. Run **make modules_install**.
6. Run **make install**.

See this page for more information about GRUB.

You might also have to update, configure, and install the GRUB boot loader's configuration file if this was not performed by the **make install** step. The GRUB updater scans the boot directory to see which kernels are available and automatically includes them in the boot menu.

The **make clean** step is not strictly necessary, but it's generally a good idea to start with a clean build environment. In practice, many problems can be traced back to this step having been skipped.

Adding a Linux device driver

On Linux systems, device drivers are typically distributed in one of three forms:

- A patch against a specific kernel version
- A loadable kernel module
- An installation script or package that installs the driver

The most common form is the installation script or package. If you're lucky enough to have one of these for your new device, you should be able to follow the standard procedure for installing new software.

In situations where you have a patch against a specific kernel version, you can in most cases install the patch with the following procedure:

```
linux# cd kernel_src_dir ; patch -p1 < patch_file
```

11.5 FreeBSD kernel configuration



FreeBSD supports the same three methods of changing kernel parameters as Linux: dynamically tuning the running kernel, building a new kernel from source, and loading dynamic modules.

Tuning FreeBSD kernel parameters

Many FreeBSD kernel parameters can be changed dynamically with the **sysctl** command, as is done on Linux. You can set values automatically at boot time by adding them to **/etc/sysctl.conf**. Many, many parameters can be changed this way; type **sysctl -a** to see them all. Not everything that shows up in the output of that command can be changed; many parameters are read-only.

The following paragraphs outline a few of the more commonly modified or interesting parameters that you might want to adjust.

net.inet.ip.forwarding and net.inet6.ip6.forwarding control IP packet forwarding for IPv4 and IPv6, respectively.

kern.maxfiles sets the maximum number of file descriptors that the system can open. You may need to increase this on systems such as database or web servers.

net.inet.tcp.mssdflt sets the default TCP maximum segment size, which is the size of the TCP packet payload carried over IPv4. Certain payload sizes are too large for long-haul network links, and hence might be dropped by their routers. Changing this parameter can be useful when debugging long-haul connectivity issues.

net.inet.udp.blackhole controls whether an ICMP "port unreachable" packet is returned when a packet arrives for a closed UDP port. Enabling this option (that is, *disabling* "port unreachable" packets) might slow down port scanners and potential attackers.

net.inet.tcp.blackhole is similar in concept to the udp.blackhole parameter. TCP normally sends an RST (connection reset) response when packets arrive for closed ports. Setting this parameter to 1 prevents any SYN (connection setup) arriving on a closed port from generating an RST. Setting it to 2 prevents RST responses to any segment at all that arrives on a closed port.

kern.ipc.nmbclusters controls the number of mbuf clusters available to the system. Mbufs are the internal storage structure for network packets, and mbuf clusters can be thought of as the mbuf "payload." For servers that experience heavy network loads, this value may need to be increased from the default (currently 253,052 on FreeBSD 10).

kern.maxvnodes sets the maximum number of vnodes, which are kernel data structures that track files. Increasing the number of available vnodes can improve disk throughput on a busy server. Examine the value of vfs.numvnodes on servers that are experiencing poor performance; if its value is close to the value of kern.maxvnodes, increase the latter.

Building a FreeBSD kernel

Kernel source comes from the FreeBSD servers in the form of a compressed tarball. Just download and unpack to install. Once the kernel source tree has been installed, the process for configuring and building the kernel is similar to that of Linux. However, the kernel source always lives in **/usr/src/sys**. Under that directory is a set of subdirectories, one for each architecture that is supported. Inside each architecture directory, a **conf** subdirectory includes a configuration file named **GENERIC** for the so-called "generic kernel," which supports every possible device and option.

The configuration file is analogous to the Linux **.config** file. The first step in making a custom kernel is to copy the **GENERIC** file to a new, distinct name in the same directory, e.g., **MYCUSTOM**. The second step is to edit the config file and modify its parameters by commenting out functions and devices that you don't need. The final step is to build and install the kernel. That final step must be performed in the top-level **/usr/src** directory.

FreeBSD kernel configuration files must be edited by hand. There are no dedicated user interfaces for this task as there are in the Linux world. Information on the general format is available from the **config**(5) man page, and information about how the config file is used can be found in the **config**(8) man page.

The configuration file contains some internal comments that describe what each option does. However, you do still need some background knowledge on a wide variety of technologies to make informed decisions about what to leave in. In general, you'll want to leave all the options from the **GENERIC** configuration enabled and modify only the device-specific lines lower in the configuration file. It's best to leave options enabled unless you're absolutely certain you don't need them.

For the final build step, FreeBSD has a single, highly automated **make buildkernel** target that combines parsing the configuration file, creating the build directories, copying the relevant source files, and compiling those files. This target accepts the custom configuration filename in the form of a build variable, KERNCONF. An analogous install target, **make installkernel**, installs the kernel and boot loader.

Here is a summary of the process:

1. Change directory (**cd**) to **/usr/src/sys/***arch*/**conf** for your architecture.
2. Copy the generic configuration: **cp GENERIC MYCUSTOM**.
3. Edit your **MYCUSTOM** configuration file.
4. Change directory to **/usr/src**.
5. Run **make buildkernel KERNCONF=MYCUSTOM**.

6. Run **make installkernel KERNCONF=MYCUSTOM**.

Note that these steps are not cross-compilation-enabled! That is, if your build machine has an AMD64 architecture, you cannot **cd** to **/usr/src/sys/sparc/conf**, follow the normals steps, and end up with a SPARC-ready kernel.

11.6 Loadable kernel modules

Loadable kernel modules (LKMs) are available in both Linux and FreeBSD. LKM support allows a device driver—or any other kernel component—to be linked into and removed from the kernel while the kernel is running. This capability facilitates the installation of drivers because it avoids the need to update the kernel binary. It also allows the kernel to be smaller because drivers are not loaded unless they are needed.

Although loadable drivers are convenient, they are not 100% safe. Any time you load or unload a module, you risk causing a kernel panic. So don’t try out an untested module when you are not willing to crash the machine.

Like other aspects of device and driver management, the implementation of loadable modules is OS-dependent.

Loadable kernel modules in Linux



Under Linux, almost anything can be built as a loadable kernel module. The exceptions are the root filesystem type (whatever that might be on a given system) and the PS/2 mouse driver.

Loadable kernel modules are conventionally stored under **/lib/modules/***version*, where *version* is the version of your Linux kernel as returned by **uname -r**.

You can inspect the currently loaded modules with the **lsmod** command:

```
redhat$ lsmod
Module              Size    Used by
ipmi_devintf        13064   2
ipmi_si             36648   1
ipmi_msghandler     31848   2 ipmi_devintf,ipmi_si
iptable_filter      6721    0
ip_tables           21441   1 iptable_filter
...
```

Loaded on this machine are the Intelligent Platform Management Interface modules and the **iptables** firewall, among other modules.

As an example of manually loading a kernel module, here's how we would insert a module that implements sound output to USB devices:

```
redhat$ sudo modprobe snd-usb-audio
```

We can also pass parameters to modules as they are loaded; for example,

```
redhat$ sudo modprobe snd-usb-audio nrpacks=8 async_unlink=1
```

modprobe is a semi-automatic wrapper around a more primitive command, **insmod**. **modprobe** understands dependencies, options, and installation and removal procedures. It also checks the version number of the running kernel and selects an appropriate version of the module from within **/lib/modules**. It consults the file **/etc/modprobe.conf** to figure out how to handle each individual module.

Once a loadable kernel module has been manually inserted into the kernel, it remains active until you explicitly request its removal or reboot the system. You could use **modprobe -r snd-usb-audio** to remove the audio module loaded above. Removal works only if the number of current references to the module (listed in the "Used by" column of **lsmod**'s output) is 0.

You can dynamically generate an **/etc/modprobe.conf** file that corresponds to all your currently installed modules by running **modprobe -c**. This command generates a long file that looks like this:

```
#This file was generated by: modprobe -c
path[pcmcia]=/lib/modules/preferred
path[pcmcia]=/lib/modules/default
path[pcmcia]=/lib/modules/2.6.6
path[misc]=/lib/modules/2.6.6
...
# Aliases
alias block-major-1 rd
alias block-major-2 floppy
...
alias char-major-4 serial
alias char-major-5 serial
alias char-major-6 lp
...
alias dos msdos
alias plip0 plip
alias ppp0 ppp
options ne io=x0340 irq=9
```

The path statements tell where a particular module can be found. You can modify or add entries of this type to keep your modules in a nonstandard location.

The alias statements map between module names and block-major device numbers, character-major device numbers, filesystems, network devices, and network protocols.

The options lines are not dynamically generated but must be manually added by an administrator. They specify options that should be passed to a module when it is loaded. For example, you could use the following line to pass in additional options to the USB sound module:

```
options snd-usb-audio nrpacks=8 async_unlink=1
```

modprobe also understands the statements install and remove. These statements allow commands to be executed when a specific module is inserted into or removed from the running kernel.

Loadable kernel modules in FreeBSD



Kernel modules in FreeBSD live in **/boot/kernel** (for standard modules that are part of the distribution) or **/boot/modules** (for ported, proprietary, and custom modules). Each kernel module uses the **.ko** filename extension, but it is not necessary to specify that extension when loading, unloading, or viewing the status of the module.

For example, to load a module named **foo.ko**, run **kldload foo** in the appropriate directory. To unload the module, run **kldunload foo** from any location. To view the module's status, run **kldstat -m foo** from any location. Running **kldstat** without any parameters displays the status of all currently loaded modules.

Modules listed in either of the files **/boot/defaults/loader.conf** (system defaults) or **/boot/loader.conf** are loaded automatically at boot time. To add a new entry to **/boot/loader.conf**, use a line of the form

```
zfs_load="YES"
```

The appropriate variable name is just the module basename with `_load` appended to it. The line above ensures that the module **/boot/kernel/zfs.ko** will be loaded at boot; it implements the ZFS filesystem.

11.7 Booting

Now that we have covered kernel basics, it's time to learn what actually happens when a kernel loads and initializes at startup. You've no doubt seen countless boot messages, but do you know what all of those messages actually mean?

The following messages and annotations come from some key phases of the boot process. They almost certainly won't be an exact match for what you see on your own systems and kernels. However, they should give you a notion of some of the major themes in booting and a feeling for how the Linux and FreeBSD kernels start up.

Linux boot messages



The first boot log we examine is from a CentOS 7 machine running a 3.10.0 kernel.

```
Feb 14 17:18:57 localhost kernel: Initializing cgroup subsys cpuset
Feb 14 17:18:57 localhost kernel: Initializing cgroup subsys cpu
Feb 14 17:18:57 localhost kernel: Initializing cgroup subsys cpuacct
Feb 14 17:18:57 localhost kernel: Linux version 3.10.0-327.el7.x86_64
    (builder@kbuilder.dev.centos.org) (gcc version 4.8.3 20140911 (Red
    Hat 4.8.3-9) (GCC) ) #1 SMP Thu Nov 19 22:10:57 UTC 2015
Feb 14 17:18:57 localhost kernel: Command line: BOOT_IMAGE=/
    vmlinuz-3.10.0-327.el7.x86_64 root=/dev/mapper/centos-root ro
    crashkernel=auto rd.lvm.lv=centos/root rd.lvm.lv=centos/swap rhgb
    quiet LANG=en_US.UTF-8
```

These initial messages tell us that the top-level control groups (cgroups) are starting up on a Linux 3.10.0 kernel. The messages tell us who built the kernel and where, and which compiler they used (**gcc**). Note that although this log comes from a CentOS system, CentOS is a clone of Red Hat, and the boot messages remind us of that fact.

The parameters set in the GRUB boot configuration and passed from there into the kernel are listed above as the “command line.”

```
Feb 14 17:18:57 localhost kernel: e820: BIOS-provided physical RAM map:
Feb 14 17:18:57 localhost kernel: BIOS-e820: [mem
    0x0000000000000000-0x000000000009fbff] usable
Feb 14 17:18:57 localhost kernel: BIOS-e820: [mem 0x000000000009fc00-
    0x000000000009ffff] reserved
...
Feb 14 17:18:57 localhost kernel: Hypervisor detected: KVM
Feb 14 17:18:57 localhost kernel: AGP: No AGP bridge found
Feb 14 17:18:57 localhost kernel: x86 PAT enabled: cpu 0, old
    0x7040600070406, new 0x7010600070106
Feb 14 17:18:57 localhost kernel: CPU MTRRs all blank - virtualized
    system.
Feb 14 17:18:57 localhost kernel: e820: last_pfn = 0xdfff0 max_arch_pfn
    = 0x400000000
Feb 14 17:18:57 localhost kernel: found SMP MP-table at [mem
    0x0009fff0-0x0009ffff] mapped at [ffff88000009fff0]
Feb 14 17:18:57 localhost kernel: init_memory_mapping: [mem
    0x00000000-0x000fffff]
...
```

These messages describe the processor that the kernel has detected and show how the RAM is mapped. Note that the kernel is aware that it’s booting within a hypervisor and is not actually running on bare hardware.

```
Feb 14 17:18:57 localhost kernel: ACPI: bus type PCI registered
Feb 14 17:18:57 localhost kernel: acpiphp: ACPI Hot Plug PCI
    Controller Driver version: 0.5
...
Feb 14 17:18:57 localhost kernel: PCI host bridge to bus 0000:00
Feb 14 17:18:57 localhost kernel: pci_bus 0000:00: root bus resource
    [bus 00-ff]
Feb 14 17:18:57 localhost kernel: pci_bus 0000:00: root bus resource
    [io  0x0000-0xffff]
Feb 14 17:18:57 localhost kernel: pci_bus 0000:00: root bus resource
    [mem 0x00000000-0xffffffff]
...
Feb 14 17:18:57 localhost kernel: SCSI subsystem initialized
Feb 14 17:18:57 localhost kernel: ACPI: bus type USB registered
Feb 14 17:18:57 localhost kernel: usbcore: registered new interface
    driver usbfs
Feb 14 17:18:57 localhost kernel: PCI: Using ACPI for IRQ routing
```

Here the kernel initializes the system's various data buses, including the PCI bus and the USB subsystem.

```
Feb 14 17:18:57 localhost kernel: Non-volatile memory driver v1.3
Feb 14 17:18:57 localhost kernel: Linux agpgart interface v0.103
Feb 14 17:18:57 localhost kernel: crash memory driver: version 1.1
Feb 14 17:18:57 localhost kernel: rdac: device handler registered
Feb 14 17:18:57 localhost kernel: hp_sw: device handler registered
Feb 14 17:18:57 localhost kernel: emc: device handler registered
Feb 14 17:18:57 localhost kernel: alua: device handler registered
Feb 14 17:18:57 localhost kernel: libphy: Fixed MDIO Bus: probed
...
Feb 14 17:18:57 localhost kernel: usbserial: USB Serial support
    registered for generic
Feb 14 17:18:57 localhost kernel: i8042: PNP: PS/2 Controller
    [PNP0303:PS2K,PNP0f03:PS2M] at 0x60,0x64 irq 1,12
Feb 14 17:18:57 localhost kernel: serio: i8042 KBD port 0x60,0x64 irq 1
Feb 14 17:18:57 localhost kernel: serio: i8042 AUX port 0x60,0x64 irq 12
Feb 14 17:18:57 localhost kernel: mousedev: PS/2 mouse device common
    for all mice
Feb 14 17:18:57 localhost kernel: input: AT Translated Set 2 keyboard as
    /devices/platform/i8042/serio0/input/input2
Feb 14 17:18:57 localhost kernel: rtc_cmos rtc_cmos: rtc core: registered
    rtc_cmos as rtc0
Feb 14 17:18:57 localhost kernel: cpuidle: using governor menu
Feb 14 17:18:57 localhost kernel: usbhid: USB HID core driver
Feb 14 17:18:57 localhost kernel: rtc_cmos rtc_cmos: alarms up to one
    day, 114 bytes nvram
```

These messages document the kernel's discovery of various devices, including the power button, a USB hub, a mouse, and a real-time clock (RTC) chip. Some of the "devices" are metadevices

rather than actual hardware; these constructs manage groups of real, related hardware devices. For example, the usbhid (USB Human Interface Device) driver manages keyboards, mice, tablets, game controllers, and other types of input devices that follow USB reporting standards.

```
Feb 14 17:18:57 localhost kernel: drop_monitor: Initializing network
    drop monitor service
Feb 14 17:18:57 localhost kernel: TCP: cubic registered
Feb 14 17:18:57 localhost kernel: Initializing XFRM netlink socket
Feb 14 17:18:57 localhost kernel: NET: Registered protocol family 10
Feb 14 17:18:57 localhost kernel: NET: Registered protocol family 17
```

In this phase, the kernel initializes a variety of network drivers and facilities.

The drop monitor is a Red Hat kernel subsystem that implements comprehensive monitoring of network packet loss. “TCP cubic” is a congestion-control algorithm optimized for high-latency, high-bandwidth connections, so-called long fat pipes.

As mentioned here, Netlink sockets are a modern approach to communication between the kernel and user-level processes. The XFRM Netlink socket is the link between the user-level IPsec process and the kernel’s IPsec routines.

The last two lines document the registration of two additional network protocol families.

```
Feb 14 17:18:57 localhost kernel: Loading compiled-in X.509 certificates
Feb 14 17:18:57 localhost kernel: Loaded X.509 cert 'CentOS Linux kpatch
    signing key: ea0413152cde1d98ebdca3fe6f0230904c9ef717'
Feb 14 17:18:57 localhost kernel: Loaded X.509 cert 'CentOS Linux Driver
    update signing key: 7f421ee0ab69461574bb358861dbe77762a4201b'
Feb 14 17:18:57 localhost kernel: Loaded X.509 cert 'CentOS Linux kernel
    signing key: 79ad886a113ca0223526336c0f825b8a94296ab3'
Feb 14 17:18:57 localhost kernel: registered taskstats version 1
Feb 14 17:18:57 localhost kernel: Key type trusted registered
Feb 14 17:18:57 localhost kernel: Key type encrypted registered
```

Like other OSs, CentOS provides a way to incorporate and validate updates. The validation portion uses X.509 certificates that are installed into the kernel.

```
Feb 14 17:18:57 localhost kernel: IMA: No TPM chip found, activating
    TPM-bypass!
Feb 14 17:18:57 localhost kernel: rtc_cmos rtc_cmos: setting system
    clock to 2017-02-14 22:18:57 UTC (1487110737)
```

Here, the kernel reports that it’s unable to find a Trusted Platform Module (TPM) on the system. TPM chips are cryptographic hardware devices that provide for secure signing operations. When used properly, they can make it much more difficult to hack into a system.

For example, the TPM can be used to sign kernel code and to make the system refuse to execute any portion of the code for which the current signature doesn’t match the TPM signature. This

measure helps avoid the execution of maliciously injected code. An admin who expects to have a working TPM would be unhappy to see this message!

The last message shows the kernel setting the battery-backed real-time clock to the current time of day. This is the same RTC that we saw mentioned earlier when it was identified as a device.

```
Feb 14 17:18:57 localhost kernel: e1000: Intel(R) PRO/1000 Network
    Driver - version 7.3.21-k8-NAPI
Feb 14 17:18:57 localhost kernel: e1000: Copyright (c) 1999-2006
    Intel Corporation.
Feb 14 17:18:58 localhost kernel: e1000 0000:00:03.0 eth0:
    (PCI:33MHz:32-bit) 08:00:27:d0:ae:6f
Feb 14 17:18:58 localhost kernel: e1000 0000:00:03.0 eth0: Intel(R)
    PRO/1000 Network Connection
```

See this page for more information about DHCP.

Now the kernel has found the gigabit Ethernet interface and initialized it. The interface's MAC address (08:00:27:d0:ae:6f) might be of interest to you if you wanted this machine to obtain its IP address through DHCP. Specific IP addresses are often locked to specific MACs in the DHCP server configuration so that servers can have IP address continuity.

```
Feb 14 17:18:58 localhost kernel: scsi host0: ata_piix
Feb 14 17:18:58 localhost kernel: ata1: PATA max UDMA/33 cmd 0x1f0 ctl
    0x3f6 bmdma 0xd000 irq 14
Feb 14 17:18:58 localhost kernel: ahci 0000:00:0d.0: flags: 64bit ncq
    stag only ccc
Feb 14 17:18:58 localhost kernel: scsi host2: ahci
Feb 14 17:18:58 localhost kernel: ata3: SATA max UDMA/133 abar
    m8192@0xf0806000 port 0xf0806100 irq 21
Feb 14 17:18:58 localhost kernel: ata2.00: ATAPI: VBOX CD-ROM, 1.0,
    max UDMA/133
Feb 14 17:18:58 localhost kernel: ata2.00: configured for UDMA/33
Feb 14 17:18:58 localhost kernel: scsi 1:0:0:0: CD-ROM          VBOX
    CD-ROM         1.0  PQ: 0 ANSI: 5
Feb 14 17:18:58 localhost kernel: tsc: Refined TSC clocksource
    calibration: 3399.654 MHz
Feb 14 17:18:58 localhost kernel: ata3: SATA link up 3.0 Gbps (SStatus
    123 SControl 300)
Feb 14 17:18:58 localhost kernel: ata3.00: ATA-6: VBOX HARDDISK, 1.0,
    max UDMA/133
Feb 14 17:18:58 localhost kernel: ata3.00: 16777216 sectors, multi 128:
    LBA48 NCQ (depth 31/32)
Feb 14 17:18:58 localhost kernel: ata3.00: configured for UDMA/133
Feb 14 17:18:58 localhost kernel: scsi 2:0:0:0: Direct-Access     ATA
    VBOX HARDDISK    1.0  PQ: 0 ANSI: 5
Feb 14 17:18:58 localhost kernel: sr 1:0:0:0: [sr0] scsi3-mmc drive:
    32x/32x xa/form2 tray
Feb 14 17:18:58 localhost kernel: cdrom: Uniform CD-ROM driver Revision:
    3.20
Feb 14 17:18:58 localhost kernel: sd 2:0:0:0: [sda] 16777216 512-byte
    logical blocks: (8.58 GB/8.00 GiB)
Feb 14 17:18:58 localhost kernel: sd 2:0:0:0: [sda] Attached SCSI disk
Feb 14 17:18:58 localhost kernel: SGI XFS with ACLs, security attributes,
    no debug enabled
Feb 14 17:18:58 localhost kernel: XFS (dm-0): Mounting V4 Filesystem
Feb 14 17:18:59 localhost kernel: XFS (dm-0): Ending clean mount
```

Here the kernel recognizes and initializes various drives and support devices (hard disk drives, a SCSI-based virtual CD-ROM, and an ATA hard disk). It also mounts a filesystem (XFS) that is part of the device-mapper subsystem (the dm-0 filesystem).

As you can see, Linux kernel boot messages are verbose almost to a fault. However, you can rest assured that you'll see everything the kernel is doing as it starts up, a most useful feature if you encounter problems.

FreeBSD boot messages



The log below is from a FreeBSD 10.3-RELEASE system that runs the kernel shipped with the release. Much of the output will look eerily familiar; the sequence of events is quite similar to that found in Linux. One notable difference is that the FreeBSD kernel produces far fewer boot messages than does Linux. Compared to Linux, FreeBSD is downright taciturn.

```
Sep 25 12:48:36 bucephalus kernel: FreeBSD 10.3-RELEASE #0
    r297264: Fri Mar 25 02:10:02 UTC 2016
Sep 25 12:48:36 bucephalus kernel: root@releng1.nyi.freebsd.org:
    /usr/obj/usr/src/sys/GENERIC amd64
Sep 25 12:48:36 bucephalus kernel: FreeBSD clang version 3.4.1
    (tags/RELEASE_34/dot1-final 208032) 20140512
```

The initial messages above tell you the OS release, the time at which the kernel was built from source, the name of the builder, the configuration file that was used, and finally, the compiler that generated the code (Clang version 3.4.1: a compiler front end, really; but let's not quibble).

```
Sep 25 12:48:36 bucephalus kernel: real memory  = 4831838208 (4608 MB)
Sep 25 12:48:36 bucephalus kernel: avail memory = 4116848640 (3926 MB)
```

Above are the system's total amount of memory and the amount that's available to user-space code. The remainder of the memory is reserved for the kernel itself.

Total memory of 4608MB probably looks a bit strange. However, this FreeBSD instance is running under a hypervisor. The amount of "real memory" is an arbitrary value that was set when the virtual machine was configured. On bare-metal systems, the total memory is likely to be a power of 2, since that's how actual RAM chips are manufactured (e.g., 8192MB).

```
Sep 25 12:48:36 bucephalus kernel: vgapci0: <VGA-compatible display>
    mem 0xe0000000-0xe0ffffff irq 18 at device 2.0 on pci0
Sep 25 12:48:36 bucephalus kernel: vgapci0: Boot video device
```

There's the default video display, which was found on the PCI bus. The output shows the memory range to which the frame buffer has been mapped.

```
Sep 25 12:48:36 bucephalus kernel: em0: <Intel(R) PRO/1000 Legacy
    Network Connection 1.1.0> port 0xd010-0xd017 mem 0xf0000000-
    0xf001ffff irq 19 at device 3.0 on pci0
Sep 25 12:48:36 bucephalus kernel: em0: Ethernet address:
    08:00:27:b5:49:fc
```

And above, the Ethernet interface, along with its hardware (MAC) address.

```
Sep 25 12:48:36 bucephalus kernel: usbus0: 12Mbps Full Speed USB v1.0
Sep 25 12:48:36 bucephalus kernel: ugen0.1: <Apple> at usbus0
Sep 25 12:48:36 bucephalus kernel: uhub0: <Apple OHCI root HUB, class
    9/0, rev 1.00/1.00, addr 1> on usbus0
Sep 25 12:48:36 bucephalus kernel: ada0 at ata0 bus 0 scbus0 tgt 0 lun 0
Sep 25 12:48:36 bucephalus kernel: cd0 at ata1 bus 0 scbus1 tgt 0 lun 0
Sep 25 12:48:36 bucephalus kernel: cd0: <VBOX CD-ROM 1.0> Removable
    CD-ROM SCSI device
Sep 25 12:48:36 bucephalus kernel: cd0: Serial Number VB2-01700376
Sep 25 12:48:36 bucephalus kernel: cd0: 33.300MB/s transfers (UDMA2,
    ATAPI 12bytes, PIO 65534bytes)
Sep 25 12:48:36 bucephalus kernel: cd0: Attempt to query device size
    failed: NOT READY, Medium not present
Sep 25 12:48:36 bucephalus kernel: ada0: <VBOX HARDDISK 1.0> ATA-6
    device
Sep 25 12:48:36 bucephalus kernel: ada0: Serial Number
    VBcf309b40-154c5085
Sep 25 12:48:36 bucephalus kernel: ada0: 33.300MB/s transfers (UDMA2,
    PIO 65536bytes)
Sep 25 12:48:36 bucephalus kernel: ada0: 4108MB (8413280 512 byte
    sectors)
Sep 25 12:48:36 bucephalus kernel: ada0: Previously was known as ad0
```

As shown above, the kernel initializes the USB bus, the USB hub, the CD-ROM drive (actually a DVD-ROM drive, but virtualized to look like a CD-ROM), and the ada disk driver.

```
Sep 25 12:48:36 bucephalus kernel: random: unblocking device.
Sep 25 12:48:36 bucephalus kernel: Timecounter "TSC-low" frequency
    1700040409 Hz quality 1000
Sep 25 12:48:36 bucephalus kernel: Root mount waiting for: usbus0
Sep 25 12:48:36 bucephalus kernel: uhub0: 12 ports with 12
    removable, self powered
Sep 25 12:48:36 bucephalus kernel: Trying to mount root from
    ufs:/dev/ada0p2 [rw]...
```

See this page for more comments on the "random" driver.

The final messages in the FreeBSD boot log document a variety of odds and ends. The "random" pseudo-device harvests entropy from the system and generates random numbers. The kernel seeded its number generator and put it in nonblocking mode. A few other devices came up, and the kernel mounted the root filesystem.

At this point, the kernel boot messages end. Once the root filesystem has been mounted, the kernel transitions to multiuser mode and initiates the user-level startup scripts. Those scripts, in turn, start the system services and make the system available for use.

11.8 Booting alternate kernels in the cloud

Cloud instances boot differently from traditional hardware. Most cloud providers sidestep GRUB and use either a modified open source boot loader or some kind of scheme that avoids the use of a boot loader altogether. Therefore, booting an alternate kernel on a cloud instance usually requires that you interact with the cloud provider's web console or API.

This section briefly outlines some of the specifics that relate to booting and kernel selection on our example cloud platforms. For a more general introduction to cloud systems, see Chapter 9, *Cloud Computing*.

On AWS, you'll need to start with a base AMI (Amazon machine image) that uses a boot loader called PV-GRUB. PV-GRUB runs a patched version of legacy GRUB and lets you specify the kernel in your AMI's **menu.lst** file.

After compiling a new kernel, edit **/boot/grub/menu.lst** to add it to the boot list:

```
default 0
fallback 1
timeout 0
hiddenmenu

title My Linux Kernel
root (hd0)
kernel /boot/my-vmlinuz-4.3 root=LABEL=/ console=hvc0
initrd /boot/my-initrd.img-4.3

title Amazon Linux
root (hd0)
kernel /boot/vmlinuz-4.1.10-17.31.amzn1.x86 root=LABEL=/ console=hvc0
initrd /boot/initramfs-4.1.10-17.31.amzn1.x86.img
```

Here, the custom kernel is the default, and the fallback option points to the standard Amazon Linux kernel. Having a fallback helps ensure that your system can boot even if your custom kernel can't be loaded or doesn't work correctly. See Amazon EC2's *User Guide for Linux Instances* for more details on this process.

Historically, DigitalOcean bypassed the boot loader through a QEMU (short for Quick Emulator) feature that allowed a kernel and RAM disk to be loaded directly into a droplet. Thankfully, DigitalOcean now allows droplets to use their own boot loaders. Most modern operating systems are supported, including CoreOS, FreeBSD, Fedora, Ubuntu, Debian, and CentOS. Changes to boot options, including selection of the kernel, are handled by the respective OS boot loaders (GRUB, usually).

Google Cloud Platform (GCP) is the most flexible platform when it comes to boot management. Google lets you upload complete system disk images to your Compute Engine account. Note that

in order for a GCP image to boot properly, it must use the MBR partitioning scheme and include an appropriate (installed) boot loader. UEFI and GPT do not apply here!

The cloud.google.com/compute/docs/creating-custom-image tutorial on building images is incredibly thorough and specifies not only the required kernel options but also the recommended settings for kernel security.

11.9 Kernel errors

Kernel crashes (aka kernel panics) are an unfortunate reality that can happen even on properly configured systems. They have a variety of causes. Bad commands entered by privileged users can certainly crash the system, but a more common cause is faulty hardware. Physical memory failures and hard drive errors (bad sectors on a platter or device) are both notorious for causing kernel panics.

It's also possible for bugs in the implementation of the kernel to result in crashes. However, such crashes are exceedingly rare in kernels anointed as "stable." Device drivers are another matter, however. They come from many different sources and are often of less-than-exemplary code quality.

If hardware is the underlying cause of a crash, keep in mind that the crash may have occurred long after the device failure that sparked it. For example, you can often remove a hot-swappable hard drive without causing immediate problems. The system continues to hum along without (much) complaint until you try to reboot or perform some other operation that depends on that particular drive.

Despite the names "panic" and "crash," kernel panics are usually relatively structured events. User-space programs rely on the kernel to police them for many kinds of misbehavior, but the kernel has to monitor itself. Consequently, kernels include a liberal helping of sanity-checking code that attempts to validate important data structures and invariants in passing. None of those checks should ever fail; if they do, it's sufficient reason to panic and halt the system, and the kernel does so proactively.

Or at least, that's the traditional approach. Linux has liberalized this rule somewhat through the "oops" system; see the next section.

Linux kernel errors

Linux has four varieties of kernel failure: soft lockups, hard lockups, panics, and the infamous Linux “oops.” Each one of these usually provides a complete stack trace, except for certain soft lockups that are recoverable without a panic.



A soft lockup occurs when the system is in kernel mode for more than a few seconds, thus preventing user-level tasks from running. The interval is configurable, but it is usually around 10 seconds, which is a long time for a process to be denied CPU cycles! During a soft lockup, the kernel is the only thing running, but it is still servicing interrupts such as those from network interfaces and keyboards. Data is still flowing in and out of the system, albeit in a potentially crippled fashion.

A hard lockup is the same as a soft lockup, but with the additional complication that most processor interrupts go unserviced. Hard lockups are overtly pathological conditions that are detected relatively quickly, whereas soft lockups can occur even on correctly configured systems that are experiencing some kind of extreme condition, such as a high CPU load.

In both cases, a stack trace and a display of the CPU registers (a “tombstone”) are usually dumped to the console. The trace shows the sequence of function calls that resulted in the lockup. In most cases, this trace tells you quite a bit about the cause of the problem.

A soft or hard lockup is almost always the result of a hardware failure, the most common culprit being bad memory. The second most common reason for a soft lockup is a kernel spinlock that has been held too long; however, this situation normally occurs only with nonstandard kernel modules. If you are running any unusual modules, try unloading them and see if the problem recurs.

When a lockup occurs, the usual behavior is for the system to stay frozen so that the tombstone remains visible on the console. But in some environments, it’s preferable to have the system panic and thus reboot. For example, an automated test rig needs systems to avoid hanging, so these systems are often configured to reboot into a safe kernel after encountering a lockup.

sysctl can configure both soft and hard lockups to panic:

```
linux$ sudo sysctl kernel.softlockup_panic=1
linux$ sudo sysctl kernel.nmi_watchdog=1
```

You can set these parameters at boot by listing them in **/etc/sysctl.conf**, just like any other kernel parameter.

The Linux “oops” system is a generalization of the traditional UNIX “panic after any anomaly” approach to kernel integrity. Oops doesn’t stand for anything; it’s just the English word oops, as in “Oops! I zeroed out your SAN again.” Oopses in the Linux kernel can lead to a panic, but they needn’t always. If the kernel can repair or address an anomaly through a less drastic measure, such as killing an individual process, it might do that instead.

When an oops occurs, the kernel generates a tombstone in the kernel message buffer that's viewable with the **dmesg** command. The cause of the oops is listed at the top. It might be something like "unable to handle kernel paging request at virtual address 0x0000000000000."

You probably won't be debugging your own kernel oopses. However, your chance of attracting the interest of a kernel or module developer is greatly increased if you do a good job of capturing the available context and diagnostic information, including the full tombstone.

The most valuable information is at the beginning of the tombstone. That fact can present a problem after a full-scale kernel panic. On a physical system, you may be able to just go to the console and page up through the history to see the full dump. But on a virtual machine, the console might be a window that becomes frozen when the Linux instance panics; it depends on the hypervisor. If the text of the tombstone has scrolled out of view, you won't be able to see the cause of the crash.

One way to minimize the likelihood of information loss is to increase the resolution of the console screen. We've found that a resolution of 1280 × 1024 is adequate to display the full text of most kernel panics.

You can set the console resolution by modifying **/etc/grub2/grub.cfg** and adding `vga=795` as a kernel startup parameter for the kernel you want to boot. You can also set the resolution by adding this clause to the kernel "command line" from GRUB's boot menu screen. The latter approach lets you test the waters without making any permanent changes.

To make the change permanent, find the menu item with the boot command for the kernel that you wish to boot, and modify it. For example, if the boot command looks like this:

```
linux16 /vmlinuz-3.10.0-229.el7.x86_64 root=/dev/mapper/centos-root
    ro rd.lvm.lv=centos/root rd.lvm.lv=centos/swap crashkernel=auto
    biosdevname=0 net.ifnames=0 LANG=en_US.UTF-8
```

then simply modify it to add the `vga=795` parameter at the end:

```
linux16 /vmlinuz-3.10.0-229.el7.x86_64 root=/dev/mapper/centos-root
    ro rd.lvm.lv=centos/root rd.lvm.lv=centos/swap crashkernel=auto
    biosdevname=0 net.ifnames=0 LANG=en_US.UTF-8 vga=795
```

Other resolutions can be achieved by setting the `vga` boot parameter to other values. Table 11.6 lists the possibilities.

Table 11.6: VGA mode values

	Color depth (bits)			
Geometry	**8**	**15**	**16**	**24**
640 x 480	769	784	785	786
800 x 600	771	787	788	789
1024 x 768	773	790	791	792
1280 x 1024	775	793	884	795
1400 x 1050	834	–	–	–
1600 x 1200	884	–	–	–

FreeBSD kernel panics



FreeBSD does not divulge much information when the kernel panics. If you are running a generic kernel from a production release, the best thing to do if you are encountering regular panics is to instrument the kernel for debugging. Rebuild the generic kernel with `makeoptions DEBUG=-g` enabled in the kernel configuration, and reboot with that new kernel. Once the system panics again, you can use **kgdb** to generate a stack trace from the resulting crash dump in **/var/crash**.

Of course, if you're running unusual kernel modules and the kernel doesn't panic when you don't load them, that's a good indication of where the issue lies.

An important note: crash dumps are the same size as real (physical) memory, so you must ensure that **/var/crash** has at least that much space available before you enable these dumps. There are ways to get around this, though: for more information, see the man pages for **dumpon** and **savecore** and the `dumpdev` variable in **/etc/rc.conf**.

11.10 Recommended reading

You can visit lwn.net for the latest information on what the kernel community is doing. In addition, we recommend the following books.

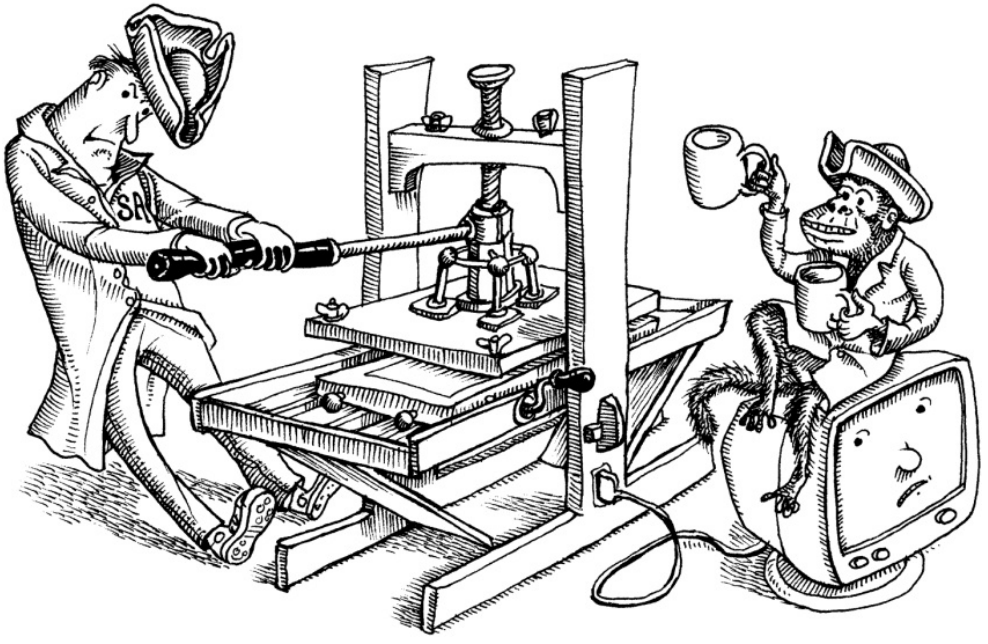
Bovet, Daniel P., and Marco Cesati. *Understanding the Linux Kernel (3rd Edition)*. Sebastopol, CA: O'Reilly Media, 2006.

Love, Robert. *Linux Kernel Development (3rd Edition)*. Upper Saddle River, NJ: Addison-Wesley Professional, 2010.

McKusick, Marshall Kirk, et al. *The Design and Implementation of the FreeBSD Operating System (2nd Edition)*. Upper Saddle River, NJ: Addison-Wesley Professional, 2014.

Rosen, Rami. *Linux Kernel Networking: Implementation and Theory*. Apress, 2014.

12 Printing



Printing is a necessary evil. No one wants to deal with it, but every user wants to print. For better or worse, printing on UNIX and Linux systems typically requires at least some configuration and occasionally some coddling by a system administrator.

Ages ago, there were three common printing systems: BSD, System V, and CUPS (the Common UNIX Printing System). Today, Linux and FreeBSD both use CUPS, an up-to-date, sophisticated, network- and security-aware printing system. CUPS includes a modern, browser-based GUI as well as shell-level commands that allow the printing system to be controlled by scripts.

Before we start, a general point: system administrators often consider printing a lower priority than do users. Administrators are accustomed to reading documents on-line, but users often need hard copy, and they want the printing system to work 100% of the time. Satisfying these desires is one of the easier ways for sysadmins to earn Brownie points with users.

Printing relies on a handful of pieces:

- A print “spooler” that collects and schedules jobs. The word “spool” originated as an acronym for Simultaneous Peripheral Operation On-Line. Now it’s just a generic term.

- User-level utilities (command-line interfaces or GUIs) that talk to the spooler. These utilities send jobs to the spooler, query the system about jobs (both pending and complete), remove or reschedule jobs, and configure the other parts of the system.
- Back ends that talk to the printing devices themselves. (These are normally unseen and hidden under the floorboards.)
- A network protocol that lets spoolers communicate and transfer jobs.

Modern environments often use network-attached printers that minimize the amount of setup and processing that must be done on the UNIX or Linux side.

12.1 CUPS PRINTING

CUPS was created by Michael Sweet and has been adopted as the default printing system for Linux, FreeBSD, and macOS. Michael has been at Apple since 2007, where he continues to develop CUPS and its ecosystem.

Just as newer mail transport systems include a command called **sendmail** that lets older scripts (and older system administrators!) work as they always did back in **sendmail**'s glory days, CUPS supplies traditional commands such as **lp** and **lpr** that are backward compatible with legacy UNIX printing systems.

CUPS servers are also web servers, and CUPS clients are web clients. The clients can be commands such as the CUPS versions of **lpr** and **lpq**, or they can be applications with their own GUIs. Under the covers they're all web apps, even if they're talking only to the CUPS daemon on the local system. CUPS servers can also act as clients of other CUPS servers.

A CUPS server offers a web interface to its full functionality on port 631. For administrators, a web browser is usually the most convenient way to manage the system; just navigate to http://*printhost*:631. If you need secure communication with the daemon (and your system offers it), use https://*printhost*:433 instead. Scripts can use discrete commands to control the system, and users normally access it through a GNOME or KDE interface. These routes are all equivalent.

HTTP is the underlying protocol for all interactions among CUPS servers and their clients. Actually, it's the Internet Printing Protocol, a souped-up version of HTTP. Clients submit jobs with the HTTP/IPP POST operation and request status with HTTP/IPP GET. The CUPS configuration files also look suspiciously similar to Apache web server configuration files.

Interfaces to the printing system

CUPS printing is often done from a GUI, and administration is often done through a web browser. As a sysadmin, though, you (and perhaps some of your hard-core terminal users) might want to use shell-level commands as well. CUPS includes work-alike commands for many of the basic, shell-level printing commands of the legacy BSD and System V printing systems. Unfortunately, CUPS doesn't necessarily emulate all the bells and whistles. Sometimes, it emulates the old interfaces entirely *too* well; instead of giving you a quick usage summary, **lpr --help** and **lp --help** just print error messages.

Here's how you might print the files **foo.pdf** and **/tmp/testprint.ps** to your default printer under CUPS:

```
$ lpr foo.pdf /tmp/testprint.ps
```

The **lpr** command transmits copies of the files to the CUPS server, **cupsd**, which stores them in the print queue. CUPS processes each file in turn as the printer becomes available.

When printing, CUPS examines both the document and the printer's PostScript Printer Description (PPD) file to see what needs to be done to get the document to print properly. (Despite the name, PPD files are used even for non-PostScript printers.)

To prepare a job for printing on a specific printer, CUPS passes it through a series of filters. For example, one filter might reformat the job so that two reduced-size page images print on each physical page (aka "2-up printing"), and another might transform the job from PostScript to PCL. Filters can also do printer-specific processing such as printer initialization. Some filters perform rasterization, turning abstract instructions such as "draw a line across the page" into a bitmap image. Such rasterizers are useful for printers that do not include their own rasterizers or that don't speak the language in which a job was originally submitted.

The final stage of the print pipeline is a back end that transmits the job from the host to the printer through an appropriate protocol such as Ethernet. The back end also communicates status information in the other direction, back to the CUPS server. After transmitting the print job, the CUPS daemon returns to processing its queues and handling requests from clients, and the printer goes off to print the job it was shipped.

The print queue

cupsd's centralized control of the printing system makes it easy to understand what the user-level commands are doing. For example, the **lpq** command requests job status information from the server and reformats it for display. Other CUPS clients ask the server to suspend, cancel, or reprioritize jobs. They can also move jobs from one queue to another.

Most changes require jobs to be identified by their job number, which you can get from **lpq**. For example, to remove a print job, just run **lprm** *jobid*.

lpstat -t summarizes the print server's overall status.

Multiple printers and queues

The CUPS server maintains a separate queue for each printer. Command-line clients accept an option (typically **-P** *printer* or **-p** *printer*) by which you specify the queue to address. You can also set a default printer for yourself by setting the PRINTER environment variable

```
$ export PRINTER=printer_name
```

or by telling CUPS to use a particular default for your account.

```
$ lpoptions -dprinter_name
```

When run as root, **lpoptions** sets system-wide defaults in **/etc/cups/lpoptions**, but it's more typically used by individual, nonroot users. **lpoptions** lets each user define personal printer instances and defaults, which it stores in **~/.cups/lpoptions**. The command **lpoptions -l** lists the current settings.

Printer instances

If you have only one printer but want to use it in several ways—say, both for quick drafts and for final production work—CUPS lets you set up different “printer instances” for these different uses.

For example, if you already have a printer named Phaser_6120, the command

```
$ lpoptions -p Phaser_6120/2up -o number-up=2 -o job-sheets=standard
```

creates an instance named Phaser_6120/2up that performs 2-up printing and adds banner pages. Once the instance has been created, the command

```
$ lpr -P Phaser_6120/2up biglisting.ps
```

prints the PostScript file **biglisting.ps** as a 2-up job with a banner page.

Network printer browsing

From CUPS's perspective, a network of machines isn't very different from an isolated machine. Every computer runs a **cupsd**, and all the CUPS daemons talk to one another.

If you're working on the command line, you configure a CUPS daemon to accept print jobs from remote systems by editing the **/etc/cups/cupsd.conf** file (see this page). By default, servers that are set up this way broadcast information every 30 seconds about the printers they serve. As a result, computers on the local network automatically learn about the printers that are available to them. You can effect the same configuration by clicking a checkbox in the CUPS GUI in your browser.

If someone has plugged in a new printer, if you've brought your laptop into work, or if you've just installed a new workstation, you can tell **cupsd** to redetermine what printing services are available; click the Find New Printers button in the Administration tab of the CUPS GUI.

Because broadcast packets do not cross subnet boundaries, it's a bit tricker to make printers available to multiple subnets. One solution is to designate, on each subnet, a slave server that polls the other subnets' servers for information and then relays that information to machines on the local subnet.

For example, suppose the print servers allie (192.168.1.5) and jj (192.168.2.14) live on different subnets and we want both of them to be accessible to users on a third subnet, 192.168.3. We designate a slave server (say, copeland, 192.168.3.10) and add these lines to its **cupsd.conf** file:

```
BrowsePoll allie
BrowsePoll jj
BrowseRelay 127.0.0.1 192.168.3.255
```

The first two lines tell the slave's **cupsd** to poll the **cupsd**s on allie and jj for information about the printers they serve. The third line tells copeland to relay the information it learns to its own subnet. Simple!

Filters

Rather than using a specialized printing tool for every printer, CUPS uses a chain of filters to convert each printed file into a form the destination printer can understand.

The CUPS filter scheme is elegant. Given a document and a target printer, CUPS uses its **.types** files to figure out the document's MIME type. It consults the printer's PPD file to figure out what MIME types the printer can handle. It then uses **.convs** files to deduce what filter chains could convert one format to the other, and what each prospective chain would cost. Finally, it picks a chain and passes the document through those filters. The final filter in the chain passes the printable format to a back end, which transmits the data to the printer through whatever hardware or protocol the printer understands.

We can flesh out that process a bit. CUPS uses rules in **/usr/share/cups/mime/mime.types** to figure out the incoming data type. For example, the rule

```
application/pdf              pdf string (0,%PDF)
```

means "If a file has a **.pdf** extension or starts with the string `%PDF`, then its MIME type is application/pdf."

CUPS figures out how to convert one data type to another by looking up rules in the file **mime.convs** (usually in **/etc/cups** or **/usr/share/cups/mime**). For example,

```
application/pdf              application/postscript 33 pdftops
```

means "To convert an application/pdf file to an application/postscript file, run the filter **pdftops**." The number 33 is the cost of the conversion. When CUPS finds that several filter chains can convert a file from one type to another, it picks the chain with the lowest total cost. (Costs are chosen by whoever creates the **mime.convs** file—the distribution maintainers, perhaps. If you want to spend time tuning them because you think you can do a better job, you may have too much free time.)

The last component in a CUPS pipeline is a filter that talks directly to the printer. In the PPD of a non-PostScript printer, you might see lines such as

```
*cupsFilter: "application/vnd.cups-postscript 0 foomatic-rip"
```

or even

```
*cupsFilter: "application/vnd.cups-postscript foomatic-rip"
```

The quoted string has the same format as a line in **mime.convs**, but there's only one MIME type instead of two. This line advertises that the **foomatic-rip** filter converts data of type application/vnd.cups-postscript to the printer's native data format. The cost is zero (or omitted)

because there's only one way to do this step, so why pretend there's a cost? (Some PPDs for non-PostScript printers, like those from the Gutenprint project, are slightly different.)

To find the filters available on your system, try running **locate pstops**. **pstops** is a popular filter that massages PostScript jobs in various ways, such as adding a PostScript command to set the number of copies. Wherever you find **pstops**, the other filters won't be far away.

You can ask CUPS for a list of the available back ends by running **lpinfo -v**. If your system lacks a back end for the network protocol you need, it may be available from the web or from your Linux distributor.

12.2 CUPS SERVER ADMINISTRATION

cupsd starts at boot time and runs continuously. All our example systems are set up this way by default.

See this page for details about Apache configuration.

The CUPS configuration file, **cupsd.conf**, is usually found in **/etc/cups**. The file format is similar to that of the Apache configuration file. If you're comfortable with one of these files, you'll be comfortable with the other. You can view and edit **cupsd.conf** with a text editor or, once again, from the CUPS web GUI.

The default config file is well commented. The comments and the **cupsd.conf** man page are good enough that we won't belabor the details of configuration here.

CUPS reads its configuration file only at startup. If you change the contents of **cupsd.conf**, you must restart **cupsd** for changes to take effect. If you make changes through **cupsd**'s web GUI, **cupsd** restarts automatically.

Network print server setup

If you're having trouble printing over the network, review the browser-based CUPS GUI and make sure you've checked all the right boxes. Possible problem areas include an unpublished printer, a CUPS server that isn't broadcasting its printers to the network, or a CUPS server that won't accept network print jobs.

If you're editing the **cupsd.conf** file directly, you'll need to make a couple of changes. First, change

```
<Location />
Order Deny,Allow
Deny From All
Allow From 127.0.0.1
</Location>
```

to

```
<Location />
Order Deny,Allow
Deny From All
Allow From 127.0.0.1
Allow From netaddress
</Location>
```

Replace *netaddress* with the IP address of the network from which you want to accept jobs (e.g., 192.168.0.0).

Then, look for the BrowseAddress keyword and set it to the broadcast address on that network plus the CUPS port; for example,

```
BrowseAddress 192.168.0.255:631
```

These steps tell the server to accept requests from any machine on the designated subnet and to broadcast what it knows about the printers it's serving to every CUPS daemon on that network. That's it! Once you restart **cupsd**, it comes back as a server.

Printer autoconfiguration

You can use CUPS without a printer (e.g., to convert files to PDF or fax format), but its typical role is to manage real printers. In this section, we review the ways in which you can deal with the printers per se.

In some cases, adding a printer is trivial. CUPS autodetects USB printers when they're plugged into the system and figures out what to do with them.

Even if you have to do some configuration work yourself, adding a printer is often no more painful than plugging in the hardware, connecting to the CUPS web interface at localhost:631/admin, and answering a few questions. KDE and GNOME come with their own printer configuration widgets, which you may prefer to the CUPS interface. (We like the CUPS GUI.)

If someone else adds a printer and one or more CUPS servers running on the network know about it, your CUPS server will learn of its existence. You don't have to explicitly add the printer to the local inventory or copy PPDs to your machine. It's magic.

Network printer configuration

Network printers—that is, printers whose primary hardware interface is an Ethernet jack or Wi-Fi radio—need some configuration of their own just to be proper citizens of the TCP/IP network. In particular, they need to know their own IP addresses and netmasks. That information is usually conveyed to them in one of two ways.

Network printers can get this information from a BOOTP or DHCP server, and this method works well in environments that have many such printers. See *DHCP: the Dynamic Host Configuration Protocol* for more information about DHCP.

Alternatively, you can assign the printer a static IP address from its console, which usually consists of a set of buttons on the printer's front panel and a one-line display. Fumble around with the menus until you discover where to set the IP address. (If there is a menu option to print the menus, use it and put the printed version underneath the printer for future reference.)

Once configured, network printers usually have a web console that's accessible from a browser. However, printers must have an IP address and must be up and running on the network before you can access them this way, so this interface is unavailable just when it's most needed.

Printer configuration examples

Below, we add the parallel printer groucho and the network printer fezmo from the command line:

```
$ sudo lpadmin -p groucho -E -v parallel:/dev/lp0 -m pxlcolor.ppd
$ sudo lpadmin -p fezmo -E -v socket://192.168.0.12 -m laserjet.ppd
```

Groucho is attached to port **/dev/lp0** and fezmo is at IP address 192.168.0.12. We specify each device in the form of a universal resource indicator (URI) and choose an appropriate PPD from the ones in **/usr/share/cups/model**.

As long as **cupsd** has been configured as a network server, it immediately makes the new printers available to other clients on the network. No restart is required.

CUPS accepts a wide variety of URIs for printers. Here are a few more examples:

- ipp://zoe.admin.com/ipp
- lpd://riley.admin.com/ps
- serial://dev/ttyS0?baud=9600+parity=even+bits=7
- socket://gillian.admin.com:9100
- usb://XEROX/Phaser%206120?serial=YGG210547

Some types take options (e.g., serial) and others don’t. **lpinfo -v** lists the devices your system can see and the types of URIs that CUPS understands.

Service shutoff

Removing a printer is easily done with **lpadmin -x**:

```
$ sudo lpadmin -x fezmo
```

OK, but what if you just want to disable a printer temporarily for service instead of removing it? You can block the print queue at either end. If you disable the tail (the exit or printer side) of the queue, users can still submit jobs, but the jobs won't print until the outlet is re-enabled. If you disable the head (the entrance) of the queue, jobs that are already in the queue can still print, but the queue rejects attempts to submit new jobs.

The **cupsdisable** and **cupsenable** commands control the exit side of the queue, and the **reject** and **accept** commands control the submission side. For example,

```
$ sudo cupsdisable groucho
$ sudo reject corbet
```

Which to use? It's a bad idea to accept print jobs that have no hope of being printed in the foreseeable future, so use **reject** for extended downtime. For brief interruptions that should be invisible to users (e.g., changing a toner cartridge), use **cupsdisable**.

Administrators occasionally ask for a mnemonic to help them remember which commands control which end of the queue. Consider: if CUPS "rejects" a job, that means you can't "inject" it. Another way to keep the commands straight is to remember that accepting and rejecting are things you can do to print *jobs*, whereas disabling and enabling are things you can do to *printers*. It doesn't make any sense to "accept" a printer or queue.

CUPS itself sometimes temporarily disables a printer that it's having trouble with (e.g., if someone has dislodged a cable). Once you fix the problem, remember to re-**cupsenable** the queue. If you forget, **lpstat** will tell you. (For a complete discussion of this issue and an alternative approach, see linuxprinting.org/beh.html.)

Other configuration tasks

Today's printers are eminently configurable, and CUPS lets you tweak a wide variety of features through its web interface and through the **lpadmin** and **lpoptions** commands. As a rule of thumb, **lpadmin** is for system-wide tasks and **lpoptions** is for per-user tasks.

lpadmin can restrict access to printers and queues. For example, you can set up printing quotas and specify which users can print to which printers.

Table 12.1 lists the commands that come with CUPS and classifies them according to their origin.

Table 12.1: CUPS command-line utilities and their origins

	Command	Function
CUPS	**cups-config**	Prints API, compiler, directory, and link information
	cupsdisable [a]	Stops printing on a printer
	cupsenable [a]	Restarts printing on a printer
	lpinfo	Shows available devices or drivers
	lpoptions	Displays or sets printer options and defaults
	lppasswd	Adds, changes, or deletes digest passwords
System V	**accept**, **reject**	Accepts or rejects queue submissions
	cancel	Cancels print jobs
	lp	Queues jobs for printing
	lpadmin	Configures printers
	lpmove	Moves an existing print job to a new destination
	lpstat	Prints status information
BSD	**lpc**	Acts as a general printer-control program
	lpq	Displays print queues
	lpr	Queues jobs for printing
	lprm	Cancels print jobs

a. These are actually just the **disable** and **enable** commands from System V, renamed.

12.3 Troubleshooting tips

Printers combine all the foibles of a mechanical device with all the communication eccentricities of a foreign operating system. They (and the software that drives them) seem dedicated to creating problems for you and your users. The next sections offer some general tips for dealing with printer adversity.

Print daemon restart

Always remember to restart daemons after changing a configuration file.

You can restart **cupsd** in whatever way your system normally restarts daemons, usually **systemctl restart org.cups.cupsd.service** or a similar incantation. In theory, you can also send **cupsd** a HUP signal. Alternatively, you can use the CUPS GUI.

Log files

CUPS maintains three logs: a page log, an access log, and an error log. The page log lists the pages that CUPS has printed. The other two logs are just like the access log and error log for Apache, which should not be surprising since the CUPS server is a web server.

The **cupsd.conf** file specifies the logging level and the locations of the log files. They're all typically kept underneath **/var/log**.

Here's an excerpt from a log file that corresponds to a single print job:

```
I [21/June/2017:18:59:08] Adding start banner page "none" to job 24.
I [21/June/2017:18:59:08] Adding end banner page "none" to job 24.
I [21/June/2017:18:59:08] Job 24 queued on 'Phaser_6120' by 'jsh'.
I [21/June/2017:18:59:08] Started filter /usr/libexec/cups/filter/pstops
    (PID 19985) for job 24.
I [21/June/2017:18:59:08] Started backend /usr/libexec/cups/backend/usb
    (PID 19986) for job 24.
```

Direct printing connections

Under CUPS, to verify the physical connection to a local printer, you can directly run the printer's back end. For example, here's what we get when we execute the back end for a USB-connected printer:

```
$ /usr/lib/cups/backend/usb
direct usb "Unknown" "USB Printer (usb)"
direct usb://XEROX/Phaser%206120?serial=YGG210547 "XEROX Phaser
    6120" "Phaser 6120"
```

When the USB cable for the Phaser 6120 is disconnected, that printer drops out of the back end's output:

```
$ /usr/lib/cups/backend/usb
direct usb "Unknown" "USB Printer (usb)"
```

Network printing problems

To begin tracking down a network printing problem, first try connecting to the printer daemon. You can connect to **cupsd** with a web browser (*hostname*:631) or with the **telnet** command (**telnet** *hostname* **631**).

If you have problems debugging a network printer connection, keep in mind that there must be a queue for the job on some machine, a way to decide where to send the job, and a method of sending the job to the machine that hosts the print queue. On the print server, there must be a place to queue the job, sufficient permissions to allow the job to be printed, and a way to output to the device.

Any and all of these prerequisites will, at some point, go awry. Be prepared to hunt for problems in many places, including these:

- System log files on the sending machine, for name resolution and permission problems
- System log files on the print server, for permission problems
- Log files on the sending machine, for missing filters, unknown printers, missing directories, etc.
- The print daemon's log files on the print server's machine, for messages about bad device names, incorrect formats, etc.
- The printer log file on the printing machine, for errors in transmitting the job
- The printer log file on the sending machine, for errors about preprocessing or queuing the job

The locations of CUPS log files are specified in **/etc/cups/cupsd.conf**. See Chapter 10, *Logging*, for general information about log management.

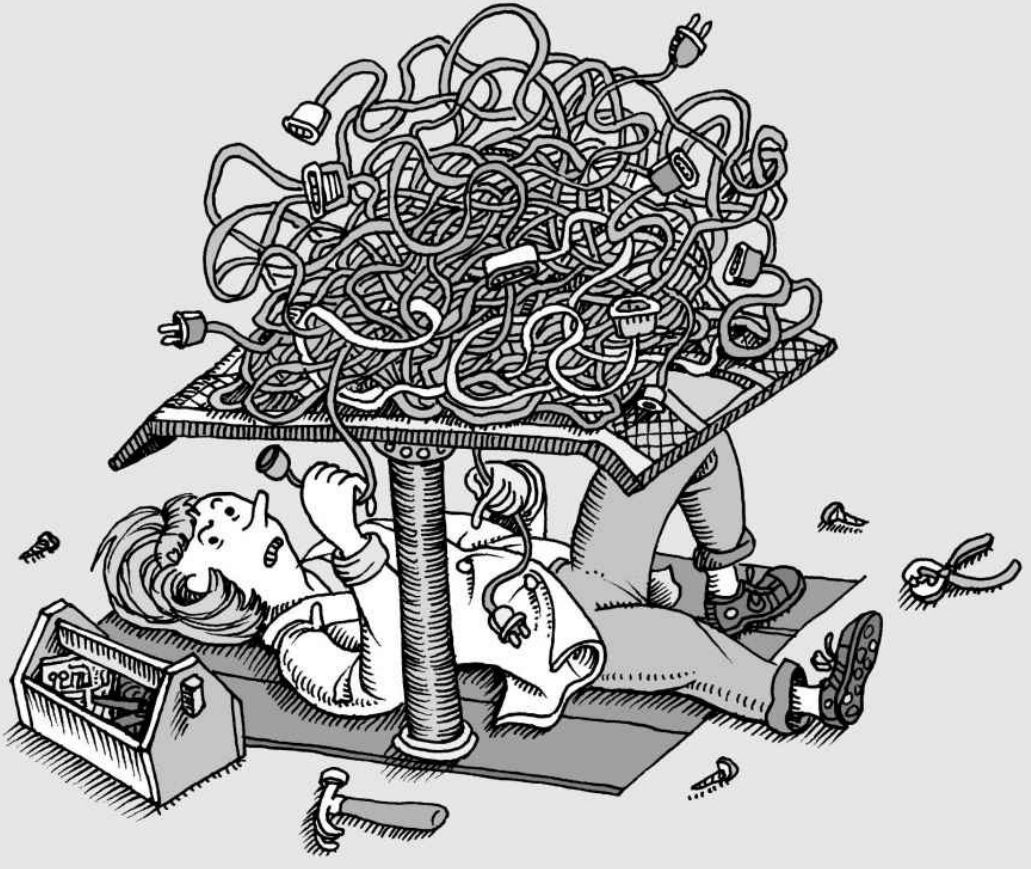
12.4 Recommended reading

CUPS comes with a lot of documentation in HTML format. An excellent way to access it is to connect to a CUPS server and click the link for on-line help. Of course, this isn't any help if you're consulting the documentation to figure out why you can't connect to the CUPS server. On your computer, the documents should be installed in **/usr/share/doc/cups** in both HTML and PDF formats. If they aren't there, ask your distribution's package manager or look on cups.org.

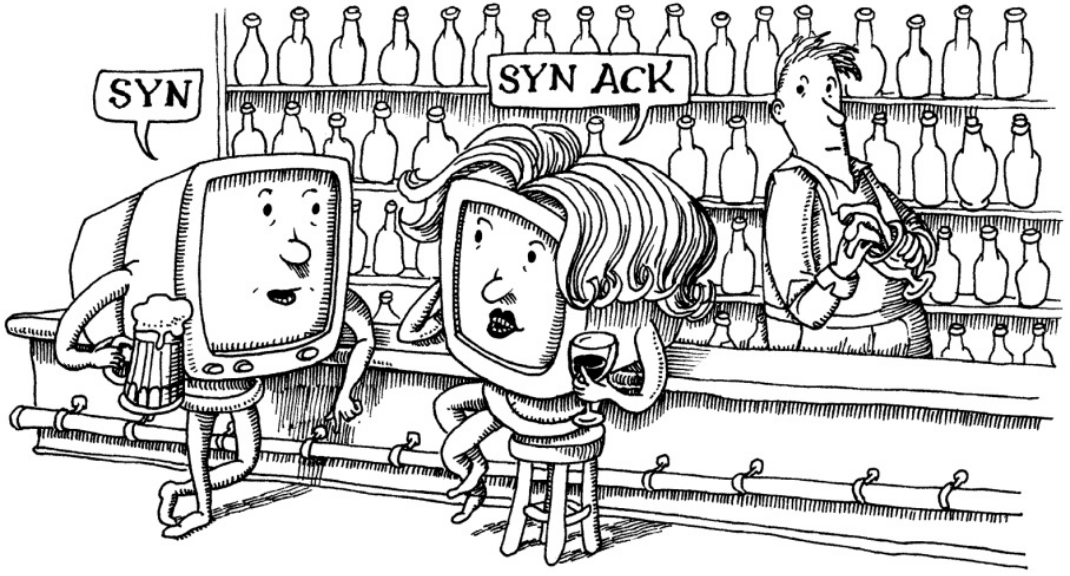
Shah, Ankur. *CUPS Administrative Guide: A practical tutorial to installing, managing, and securing this powerful printing system*. Birmingham, UK: Packt Publishing, 2008.

SECTION TWO

NETWORKING



13 TCP/IP Networking



It would be hard to overstate the importance of networks to modern computing, although that doesn't seem to stop people from trying. At many sites—perhaps even the majority—web and email access are the primary uses of computers. As of 2017, internetworldstats.com estimates the Internet to have more than 3.7 billion users, or just slightly less than half of the world's population. In North America, Internet penetration approaches 90%.

TCP/IP (Transmission Control Protocol/Internet Protocol) is the networking system that underlies the Internet. TCP/IP does not depend on any particular hardware or operating system, so devices that speak TCP/IP can all exchange data ("interoperate") despite their many differences.

TCP/IP works on networks of any size or topology, whether or not they are connected to the outside world. This chapter introduces the TCP/IP protocols in the context of the Internet, but stand-alone networks are quite similar at the TCP/IP level.

13.1 TCP/IP AND ITS RELATIONSHIP TO THE INTERNET

TCP/IP and the Internet share a history that goes back multiple decades. The technical success of the Internet is due largely to the elegant and flexible design of TCP/IP and its open and nonproprietary protocol suite. In turn, the leverage provided by the Internet has helped TCP/IP prevail over several competing protocol suites that were favored at one time or another for political or commercial reasons.

The progenitor of the modern Internet was a research network called ARPANET, established in 1969 by the U.S. Department of Defense. By the end of the 1980s the network was no longer a research project and we transitioned to the commercial Internet. Today's Internet is a collection of private networks owned by Internet service providers (ISPs) that interconnect at many so-called peering points.

Who runs the Internet?

Oversight of the Internet and the Internet protocols has long been a cooperative and open effort, but its exact structure has changed as the Internet has evolved into a public utility and a driving force in the world economy. Current Internet governance is split roughly into administrative, technical, and political wings, but the boundaries between these functions are often vague. The major players are listed below:

- ICANN, the Internet Corporation for Assigned Names and Numbers: if any one group can be said to be in charge of the Internet, this is probably it. It's the only group with any sort of actual enforcement capability. ICANN controls the allocation of Internet addresses and domain names, along with various other snippets such as protocol port numbers. It is organized as a nonprofit corporation headquartered in California. (icann.org)
- ISOC, the Internet Society: ISOC is an open-membership organization that represents Internet users. Although it has educational and policy functions, it's best known as the umbrella organization for the technical development of the Internet. In particular, it is the parent organization of the Internet Engineering Task Force (ietf.org), which oversees most technical work. ISOC is an international nonprofit organization with offices in Washington, D.C. and Geneva. (isoc.org)
- IGF, the Internet Governance Forum: a relative newcomer, the IGF was created by the United Nations in 2006 to establish a home for international and policy-oriented discussions related to the Internet. It's currently structured as a yearly conference series, but its importance is likely to grow over time as governments attempt to exert more control over the operation of the Internet. (intgovforum.org)

Of these groups, ICANN has the toughest job: establishing itself as the authority in charge of the Internet, undoing the mistakes of the past, and foreseeing the future, all while keeping users, governments, and business interests happy.

Network standards and documentation

If your eyes haven't glazed over just from reading the title of this section, you've probably already had several cups of coffee. Nonetheless, accessing the Internet's authoritative technical documentation is a crucial skill for system administrators, and it's more entertaining than it sounds.

The technical activities of the Internet community are summarized in documents known as Requests for Comments or RFCs. Protocol standards, proposed changes, and informational bulletins all usually end up as RFCs. RFCs start their lives as Internet Drafts, and after lots of email wrangling and IETF meetings they either die or are promoted to the RFC series. Anyone who has comments on a draft or proposed RFC is encouraged to reply. In addition to standardizing the Internet protocols, the RFC mechanism sometimes just documents or explains aspects of existing practice.

RFCs are numbered sequentially; currently, there are about 8,200. RFCs also have descriptive titles (e.g., *Algorithms for Synchronizing Network Clocks*), but to forestall ambiguity they are usually cited by number. Once distributed, the contents of an RFC are never changed. Updates are distributed as new RFCs with their own reference numbers. Updates may either extend and clarify existing RFCs or supersede them entirely.

RFCs are available from numerous sources, but rfc-editor.org is dispatch central and will always have the most up-to-date information. Look up the status of an RFC at rfc-editor.org before investing the time to read it; it may no longer be the most current document on that subject.

The Internet standards process itself is detailed in RFC2026. Another useful meta-RFC is RFC5540, *40 Years of RFCs*, which describes some of the cultural and technical context of the RFC system.

Don't be scared away by the wealth of technical detail found in RFCs. Most contain introductions, summaries, and rationales that are useful for system administrators even when the technical details are not. Some RFCs are specifically written as overviews or general introductions. RFCs might not be the gentlest way to learn about a topic, but they are authoritative, concise, and free.

Not all RFCs are full of boring technical details. Here are some of our favorites on the lighter side (usually written on April 1st):

- RFC1149 – *Standard for Transmission of IP Datagrams on Avian Carriers*
- RFC1925 – *The Twelve Networking Truths*
- RFC3251 – *Electricity over IP*
- RFC4041 – *Requirements for Morality Sections in Routing Area Drafts*

- RFC6214 – *Adaptation of RFC1149 for IPv6*
- RFC6921 – *Design Considerations for Faster-Than-Light Communication*
- RFC7511 – *Scenic Routing for IPv6*

In addition to being assigned its own serial number, an RFC can also be assigned an FYI (For Your Information) number, a BCP (Best Current Practice) number, or a STD (Standard) number. FYIs, STDs, and BCPs are subseries of the RFCs that include documents of special interest or importance.

FYIs are introductory or informational documents intended for a broad audience. They can be a good place to start research on an unfamiliar topic if you can find one that's relevant. Unfortunately, this series has languished recently and not many of the FYIs are up to date.

BCPs document recommended procedures for Internet sites. They consist of administrative suggestions and for system administrators are often the most valuable of the RFC subseries.

STDs document Internet protocols that have completed the IETF's review and testing process and have been formally adopted as standards.

RFCs, FYIs, BCPs, and STDs are numbered sequentially within their own series, so a document can bear several different identifying numbers. For example, RFC1713, *Tools for DNS Debugging*, is also known as FYI27.

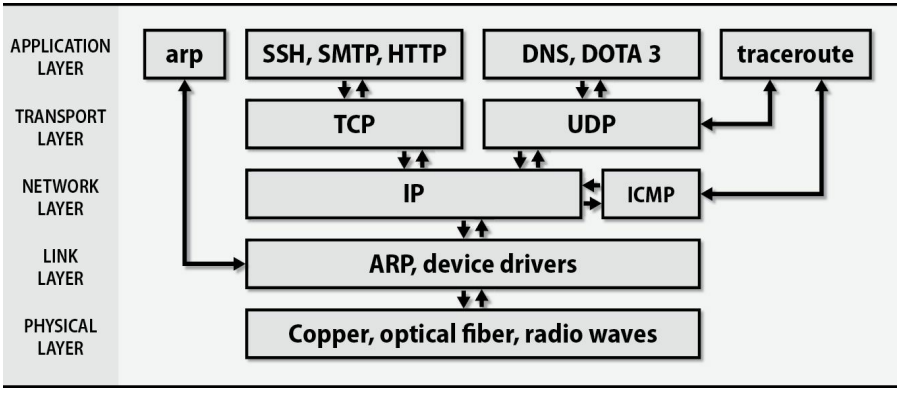
13.2 Networking basics

Now that we've provided a bit of context, let's look at the TCP/IP protocols themselves. TCP/IP is a protocol "suite," a set of network protocols designed to work smoothly together. It includes several components, each defined by a standards-track RFC or series of RFCs:

- IP, the Internet Protocol, which routes data packets from one machine to another (RFC791)
- ICMP, the Internet Control Message Protocol, which defines several kinds of low-level support for IP, including error messages, routing assistance, and debugging help (RFC792)
- ARP, the Address Resolution Protocol, which translates IP addresses to hardware addresses (RFC826; ARP can be used with other protocol suites, but it's an integral part of the way TCP/IP works on most LAN media.)
- UDP, the User Datagram Protocol, which implements unverified, one-way data delivery (RFC768)
- TCP, the Transmission Control Protocol, which implements reliable, full duplex, flow-controlled, error-corrected conversations (RFC793)

These protocols are arranged in a hierarchy or "stack", with the higher-level protocols making use of the protocols beneath them. TCP/IP is conventionally described as a five-layer system (as shown in Exhibit A), but the actual TCP/IP protocols inhabit only three of these layers.

Exhibit A: TCP/IP layering model



IPv4 and IPv6

The version of TCP/IP that has been in widespread use for nearly five decades is protocol revision 4, aka IPv4. It uses 4-byte IP addresses. A modernized version, IPv6, expands the IP address space to 16 bytes and incorporates several other lessons learned from the use of IPv4. It removes several features of IP that experience has shown to be of little value, making the protocol potentially faster and easier to implement. IPv6 also integrates security and authentication into the basic protocol.

Operating systems and network devices have supported IPv6 for a long time. Google reports statistics about its clients' use of IPv6 at google.com/ipv6. As of March 2017, the fraction of peers using IPv6 to contact Google sites has risen to about 14% world-wide. In the United States, it's over 30%.

Those numbers look healthy, but in fact they're perhaps a bit deceptive because most mobile devices default to IPv6 when they're on the carrier's data network, and there are a lot of phones out there. Home and enterprise networks remain overwhelmingly centered on IPv4.

The development and deployment of IPv6 were to a large extent motivated by the concern that the world was running out of 4-byte IPv4 address space. And indeed that concern proved well founded: at this point, only Africa has any remaining IPv4 addresses still available for assignment (see ipv4.potaroo.net for details). The Asia-Pacific region was the first to run out of addresses (on April 19, 2011).

Given that we've already lived through the IPv4 apocalypse and have used up all our IPv4 addresses, how is it that the world continues to rely predominantly on IPv4?

For the most part, we've learned to make more efficient use of the IPv4 addresses that we have. Network Address Translation (NAT; see this page) lets entire networks of machines hide behind a single IPv4 address. Classless Inter-Domain Routing (CIDR; see this page) flexibly subdivides networks and promotes efficient backbone routing. Contention for IPv4 addresses still exists, but like broadcast spectrum, it tends to be reallocated in economic rather than technological ways these days.

The underlying issue that limits IPv6's adoption is that IPv4 support remains mandatory for a device to be a functional citizen of the Internet. For example, here are a few major web sites that as of 2017 are still not reachable through IPv6: Amazon, Reddit, eBay, IMDB, Hotmail, Tumblr, MSN, Apple, The New York Times, Twitter, Pinterest, Bing, WordPress, Dropbox, craigslist, Stack Overflow. We could go on, but you get the drift.

This list consists of sites whose primary web addresses are not associated with any IPv6 addresses (AAAA records) in DNS. Microsoft Bing's presence on the list is particularly interesting given that it's one of a handful of major sites showcased in materials for the World IPv6 Launch marketing campaign of 2012 (tag line: "This time it is for real"). We don't know

the full story behind this situation, but Bing evidently supported IPv6 at one point, then later decided it wasn't worth the trouble. See worldipv6launch.org.

Your choice is not between IPv4 and IPv6; it's between supporting IPv4 alone and supporting both IPv4 and IPv6. When all the services listed above—and scores more in the second tier—have added IPv6 support, then you can reasonably consider adopting IPv6 instead of IPv4. Until then, it doesn't seem unreasonable to ask IPv6 to justify the effort of its implementation by providing better performance, security, or features. Or perhaps, by opening the door to a world of IPv6-only services that simply can't be accessed through IPv4.

Unfortunately, those services don't exist, and IPv6 doesn't actually offer any of those benefits. Yes, it's an elegant and well-designed protocol that improves on IPv4. And yes, it is in some ways easier to administer than IPv4 and requires fewer hacks (e.g., less need for NAT). But in the end, it's just a cleaned-up version of IPv4 with a larger address space. The fact that you must manage it alongside IPv4 eliminates any potential efficiency gain. IPv6's raison d'être remains the millennial fear of IPv4 address exhaustion, and to date, the effects of that exhaustion just haven't been painful enough to motivate widespread migration to IPv6.

We've been publishing this book for a long time, and over the last few editions, IPv6 has always seemed like it was one more update away from meriting coverage as a primary technology. 2017 brings an uncanny sense of deja vu, with IPv6 looming ever brighter on the horizon but still solving no immediate problems and offering few specific incentives to convert. IPv6 is the future of networking, and evidently, it always will be.

The arguments in favor of actually deploying IPv6 inside your network remain largely attitudinal: It will have to be done at some point. IPv6 is superior from an engineering standpoint. You need to develop IPv6 expertise so that you're not caught flat-footed when the IPv6 rapture finally arrives. All the cool kids are doing it.

We say: sure, go ahead, support IPv6 if you feel like it. That's a responsible and forward-thinking path. It's civic-minded, too—your adoption of IPv6 hastens the day when IPv6 is all we have to deal with. But if you don't feel like diving into IPv6, that's fine too. You'll have years of warning before there's any real need to transition.

Of course, none of these comments apply if your organization offers public services on the Internet. In that case, it's your solemn duty to implement IPv6. Don't screw things up for the rest of us by continuing to impede IPv6's adoption. Do you want to be Google, or do you want to be Microsoft Bing?

There's also an argument to be made for IPv6 in data centers where direct connectivity to the outside world of IPv4 is not needed. In these limited environments, you may indeed have the option to migrate to IPv6 and leave IPv4 behind, thereby simplifying your infrastructure. Internet-facing servers can speak IPv4 even as they route all internal and back-end traffic over IPv6.

A couple of points:

- IPv6 has been production-ready for a long time. Implementation bugs aren't a major concern. Expect it to work as reliably as IPv4.
- From a hardware standpoint, IPv6 support should be considered mandatory for all new device acquisitions. It's doubtful that you could find any piece of enterprise-grade networking gear that doesn't support IPv6 these days, but a lot of consumer-grade equipment remains IPv4-only.

In this book, we focus on IPv4 as the mainstream version of TCP/IP. IPv6-specific material is explicitly marked. Fortunately for sysadmins, IPv4 and IPv6 are highly analogous. If you understand IPv4, you already know most of what you need to know about IPv6. The main difference between the versions lies in their addressing schemes. In addition to longer addresses, IPv6 introduces a few additional addressing concepts and some new notation. But that's about it.

Packets and encapsulation

TCP/IP supports a variety of physical networks and transport systems, including Ethernet, token ring, MPLS (Multiprotocol Label Switching), wireless Ethernet, and serial-line-based systems. Hardware is managed within the link layer of the TCP/IP architecture, and higher-level protocols do not know or care about the specific hardware being used.

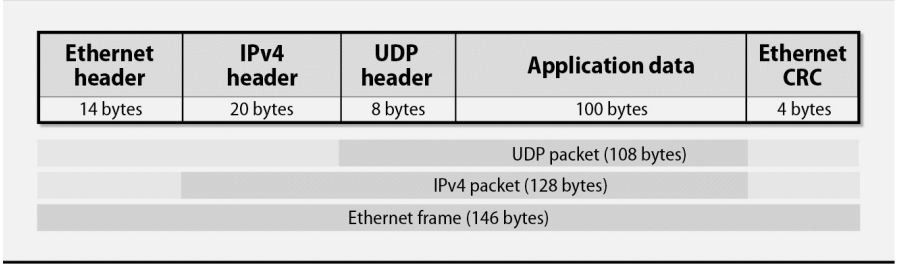
Data travels on a network in the form of packets, bursts of data with a maximum length imposed by the link layer. Each packet consists of a header and a payload. The header tells where the packet came from and where it's going. It can also include checksums, protocol-specific information, or other handling instructions. The payload is the data to be transferred.

The name of the primitive data unit depends on the layer of the protocol. At the link layer it is called a frame, at the IP layer a packet, and at the TCP layer a segment. In this book, we use "packet" as a generic term that encompasses these various cases.

As a packet travels down the protocol stack (from TCP or UDP transport to IP to Ethernet to the physical wire) in preparation for being sent, each protocol adds its own header information. Each protocol's finished packet becomes the payload part of the packet generated by the next protocol. This nesting is known as encapsulation. On the receiving machine, the encapsulation is reversed as the packet travels back up the protocol stack.

For example, a UDP packet being transmitted over Ethernet contains three different wrappers or envelopes. On the Ethernet wire, it is framed with a simple header that lists the source and next-hop destination hardware addresses, the length of the frame, and the frame's checksum (CRC). The Ethernet frame's payload is an IP packet, the IP packet's payload is a UDP packet, and the UDP packet's payload is the data being transmitted. Exhibit B shows the components of such a frame.

Exhibit B: A typical network packet



Ethernet framing

One of the main chores of the link layer is to add headers to packets and to put separators between them. The headers contain each packet's link-layer addressing information and checksums, and the separators ensure that receivers can tell where one packet stops and the next one begins. The process of adding these extra bits is known generically as framing.

The link layer is divided into two parts: MAC, the Media Access Control sublayer, and LLC, the Logical Link Control sublayer. The MAC sublayer deals with the media and transmits packets onto the wire. The LLC sublayer handles the framing.

Today, a single standard for Ethernet framing is in common use: DIX Ethernet II. In the distant past, several slightly different standards based on IEEE 802.2 were also used. You might still run across vestigial references to framing choices in network documentation, but you can now ignore this issue.

Maximum transfer unit

The size of packets on a network can be limited both by hardware specifications and by protocol conventions. For example, the payload of a standard Ethernet frame is traditionally 1,500 bytes. The size limit is associated with the link-layer protocol and is called the maximum transfer unit or MTU. Table 13.1 shows some typical values for the MTU.

Table 13.1: MTUs for various types of network

Network type	Maximum transfer unit
Ethernet	1,500 bytes (1,492 with 802.2 framing)
IPv6 (all hardware)	At least 1,280 bytes at the IP layer
Token ring	Configurable [a]
Point-to-point WAN links (T1, T3)	Configurable, often 1,500 or 4,500 bytes

a. Common values are 552; 1,064; 2,088; 4,508; and 8,232. Sometimes 1,500 to match Ethernet.

IPv4 splits packets to conform to the MTU of a particular network link. If a packet is routed through several networks, one of the intermediate networks might have a smaller MTU than the network of origin. In this case, an IPv4 router that forwards the packet onto the small-MTU network further subdivides the packet in a process called fragmentation.

Fragmentation of in-flight packets is an unwelcome chore for a busy router, so IPv6 largely removes this feature. Packets can still be fragmented, but the originating host must do the work itself. All IPv6 networks are required to support an MTU of at least 1,280 bytes at the IP layer, so IPv6 senders also have the option of limiting themselves to packets of this size.

IPv4 senders can discover the lowest-MTU link through which a packet must pass by setting the packet's "do not fragment" flag. If the packet reaches an intermediate router that cannot forward the packet without fragmenting it, the router returns an ICMP error message to the sender. The ICMP packet includes the MTU of the network that's demanding smaller packets, and this MTU then becomes the governing packet size for communication with that destination.

IPv6 path MTU discovery works similarly, but since intermediate routers are never allowed to fragment IPv6 packets, all IPv6 packets act as if they had a "do not fragment" flag enabled. Any IPv6 packet that's too large to fit into a downstream pipe causes an ICMP message to be returned to the sender.

The TCP protocol automatically does path MTU discovery, even in IPv4. UDP is not so nice and is happy to shunt extra work to the IP layer.

IPv4 fragmentation problems can be insidious. Although path MTU discovery should automatically resolve MTU conflicts, an administrator must occasionally intervene. If you are using a tunneled architecture for a virtual private network, for example, you should look at the

size of the packets that are traversing the tunnel. They are often 1,500 bytes to start with, but once the tunneling header is added, they become 1,540 bytes or so and must be fragmented. Setting the MTU of the link to a smaller value averts fragmentation and increases the overall performance of the tunneled network. Consult the **ifconfig** or **ip-link** man page to see how to set an interface's MTU.

13.3 Packet addressing

Like letters or email messages, network packets must be properly addressed to reach their destinations. Several addressing schemes are used in combination:

- MAC (Media Access Control) addresses for use by hardware
- IPv4 and IPv6 network addresses for use by software
- Hostnames for use by people

Hardware (MAC) addressing

Each of a host's network interfaces usually has one link-layer MAC address that distinguishes it from other machines on the physical network, plus one or more IP addresses that identify the interface on the global Internet. This last part bears repeating: IP addresses identify *network interfaces, not machines*. (To users the distinction is irrelevant, but administrators must know the truth.)

The lowest level of addressing is dictated by network hardware. For example, Ethernet devices are assigned a unique 6-byte hardware address at the time of manufacture. These addresses are traditionally written as a series of 2-digit hex bytes separated by colons; for example, 00:50:8d:9a:3b:df.

Token ring interfaces have a similar address that is also six bytes long. Some point-to-point networks (such as PPP) need no hardware addresses at all; the identity of the destination is specified as the link is established.

A 6-byte Ethernet address is divided into two parts. The first three bytes identify the manufacturer of the hardware, and the last three bytes are a unique serial number that the manufacturer assigns. Sysadmins can sometimes identify the brand of machine that is trashing a network by looking up the 3-byte identifier in a table of vendor IDs. The 3-byte codes are actually IEEE Organizationally Unique Identifiers (OUIs), so you can look up them up directly in the IEEE's database at

standards.ieee.org/regauth/oui

Of course, the relationships among the manufacturers of chipsets, components, and systems are complex, so the vendor ID embedded in a MAC address can be misleading, too.

In theory, Ethernet hardware addresses are permanently assigned and immutable. However, many network interfaces let you override the hardware address and set one of your own choosing. This feature can be handy if you have to replace a broken machine or network card and for some reason must use the old MAC address (e.g., all your switches filter it, or your DHCP server hands out addresses according to MAC addresses, or your MAC address is also a software license key). Spoofable MAC addresses are also helpful if you need to infiltrate a wireless network that uses MAC-based access control. But for simplicity, it's generally advisable to preserve the uniqueness of MAC addresses.

IP addressing

See this page for more details about NAT and private address spaces.

At the next level up from the hardware, Internet addressing (more commonly known as IP addressing) is used. IP addresses are hardware independent. Within any particular network context, an IP address identifies a specific and unique destination. However, it's not quite accurate to say that IP addresses are globally unique because several special cases muddy the water: NAT uses one interface's IP address to handle traffic for multiple machines; IP private address spaces are addresses that multiple sites can use at once, as long as the addresses are not visible to the Internet; anycast addressing shares one IP address among several machines.

See this page for more information about ARP.

The mapping from IP addresses to hardware addresses is implemented at the link layer of the TCP/IP model. On networks such as Ethernet that support broadcasting (that is, networks that allow packets to be addressed to "all hosts on this physical network"), senders use the ARP protocol to discover mappings without assistance from a system administrator. In IPv6, an interface's MAC address is often used as part of the IP address, making the translation between IP and hardware addressing virtually automatic.

Hostname “addressing”

IP addresses are sequences of numbers, so they are hard for people to remember. Operating systems allow one or more hostnames to be associated with an IP address so that users can type rfc-editor.org instead of 4.31.198.49. This mapping can be set up in several ways, ranging from a static file (**/etc/hosts**) to the LDAP database system to DNS, the world-wide Domain Name System. Keep in mind that hostnames are really just a convenient shorthand for IP addresses, and as such, they refer to network interfaces rather than computers.

See Chapter 16 for more information about DNS.

Ports

IP addresses identify a machine's network interfaces, but they are not specific enough to address individual processes or services, many of which might be actively using the network at once. TCP and UDP extend IP addresses with a concept known as a port, a 16-bit number that supplements an IP address to specify a particular communication channel. Valid ports are in the range 1–65,535.

Standard services such as SMTP, SSH, and HTTP associate themselves with "well known" ports defined in **/etc/services**. Here are some typical entries from the **services** file:

```
...
smtp            25/udp                      # Simple Mail Transfer
smtp            25/tcp                      # Simple Mail Transfer
...
domain          53/udp                      # Domain Name Server
domain          53/tcp                      # Domain Name Server
...
http            80/udp      www www-http    # World Wide Web HTTP
http            80/tcp      www www-http    # World Wide Web HTTP
...
kerberos        88/udp                      # Kerberos
kerberos        88/tcp                      # Kerberos
...
```

The **services** file is part of the infrastructure. You should never need to modify it, although you can do so if you want to add a nonstandard service. You can find a full list of assigned ports at iana.org/assignments/port-numbers.

Although both TCP and UDP have ports, and those ports have the same sets of potential values, the port spaces are entirely separate and unrelated. Firewalls must be configured separately for each of these protocols.

To help prevent impersonation of system services, UNIX systems restrict programs from binding to port numbers under 1,024 unless they are run as root or have an appropriate Linux capability. Anyone can communicate with a server running on a low port number; the restriction applies only to the program listening on the port.

Today, the privileged port system is as much a nuisance as it is a bulwark against malfeasance. In many cases, it's more secure to run standard services on unprivileged ports as nonroot users and to forward network traffic to these high-numbered ports through a load balancer or some other type of network appliance. This practice limits the proliferation of unnecessary root privileges and adds an additional layer of abstraction to your infrastructure.

Address types

The IP layer defines several broad types of address, some of which have direct counterparts at the link layer:

- Unicast – addresses that refer to a single network interface
- Multicast – addresses that simultaneously target a group of hosts
- Broadcast – addresses that include all hosts on the local subnet
- Anycast – addresses that resolve to any one of a group of hosts

Multicast addressing facilitates applications such as video conferencing for which the same set of packets must be sent to all participants. The Internet Group Management Protocol (IGMP) constructs and manages sets of hosts that are treated as one multicast destination.

Multicast is largely unused on today's Internet, but it's slightly more mainstream in IPv6. IPv6 broadcast addresses are really just specialized forms of multicast addressing.

Anycast addresses bring load balancing to the network layer by allowing packets to be delivered to whichever of several destinations is closest in terms of network routing. You might expect that they'd be implemented similarly to multicast addresses, but in fact they are more like unicast addresses.

Most of the implementation details for anycast support are handled at the level of routing rather than through IP. The novelty of anycast addressing is really just the relaxation of the traditional requirement that IP addresses identify unique destinations. Anycast addressing is formally described for IPv6, but the same tricks can be applied to IPv4, too—for example, as is done for root DNS name servers.

13.4 IP ADDRESSES: THE GORY DETAILS

With the exception of multicast addresses, Internet addresses consist of a network portion and a host portion. The network portion identifies a logical network to which the address refers, and the host portion identifies a node on that network. In IPv4, addresses are 4 bytes long and the boundary between network and host portions is set administratively. In IPv6, addresses are 16 bytes long and the network portion and host portion are always 8 bytes each.

IPv4 addresses are written as decimal numbers, one for each byte, separated by periods; for example, 209.85.171.147. The leftmost byte is the most significant and is always part of the network portion.

When 127 is the first byte of an address, it denotes the “loopback network,” a fictitious network that has no real hardware interface and only one host. The loopback address 127.0.0.1 always refers to the current host. Its symbolic name is “localhost”. (This is another small violation of IP address uniqueness since every host thinks 127.0.0.1 is a different computer.)

IPv6 addresses and their text-formatted equivalents are a bit more complicated. They’re discussed in the section *IPv6 addressing*.

An interface’s IP address and other parameters are set with the **ip address** (Linux) or **ifconfig** (FreeBSD) command. Details on configuring a network interface start here.

IPv4 address classes

Historically, IP addresses had an inherent “class” that depended on the first bits of the leftmost byte. The class determined which bytes of the address were in the network portion and which were in the host portion. Today, an explicit mask identifies the network portion, and the boundary can fall between any two adjacent bits, not just between bytes. However, the traditional classes are still used as defaults when no explicit division is specified.

Classes A, B, and C denote regular IP addresses. Classes D and E are multicasting and research addresses. Table 13.2 describes the characteristics of each class. The network portion of an address is denoted by N, and the host portion by H.

Table 13.2: Historical IPv4 address classes

Class	1st byte[a]	Format	Comments
A	1-127	N.H.H.H	Very early networks, or reserved for DoD
B	128-191	N.N.H.H	Large sites, usually subnetted, were hard to get
C	192-223	N.N.N.H	Were easy to get, often obtained in sets
D	224-239	–	Multicast addresses, not permanently assigned
E	240-255	–	Experimental addresses

a. The value 0 is special and is not used as the first byte of regular IP addresses. The value 127 is reserved for the loopback address.

It’s unusual for a single physical network to have thousands of computers attached to it, so class A and class B addresses (which allow for 16,777,214 hosts and 65,534 hosts per network, respectively) are really quite wasteful. For example, the 127 class A networks use up half the available 4-byte address space. Who knew that IPv4 address space would become so precious!

IPv4 subnetting

To make better use of these addresses, you can now reassign part of the host portion to the network portion by specifying an explicit 4-byte (32-bit) "subnet mask" or "netmask" in which the 1s correspond to the desired network portion and the 0s correspond to the host portion. The 1s must be leftmost and contiguous. At least eight bits must be allocated to the network part and at least two bits to the host part. Ergo, there are really only 22 possible values for an IPv4 netmask.

For example, the four bytes of a class B address would normally be interpreted as N.N.H.H. The implicit netmask for class B is therefore 255.255.0.0 in decimal notation. With a netmask of 255.255.255.0, however, the address would be interpreted as N.N.N.H. Use of the mask turns a single class B network address into 256 distinct class-C-like networks, each of which can support 254 hosts.

---

*See this page for more information about **ip** and **ifconfig**.*

Netmasks are assigned with the **ip** or **ifconfig** command as each network interface is set up. By default, these commands use the inherent class of an address to figure out which bits are in the network part. When you set an explicit mask, you simply override this behavior.

Netmasks that do not end at a byte boundary can be annoying to decode and are often written as /XX, where XX is the number of bits in the network portion of the address. This is sometimes called CIDR (Classless Inter-Domain Routing; see this page) notation. For example, the network address 128.138.243.0/26 refers to the first of four networks whose first bytes are 128.138.243. The other three networks have 64, 128, and 192 as their fourth bytes. The netmask associated with these networks is 255.255.255.192 or 0xFFFFFFC0; in binary, it's 26 ones followed by 6 zeros. Exhibit C breaks out these numbers in a bit more detail.

Exhibit C: Netmask base conversion

IP address	128		.	138		.	243		.	0		
Decimal netmask	255		.	255		.	255		.	192		
Hex netmask	f	f	.	f	f	.	f	f	.	c	0	
Binary netmask	1111	1111	.	1111	1111	.	1111	1111	.	1100	0000	

A /26 network has 6 bits left (32 – 26 = 6) to number hosts. 2^6 is 64, so the network has 64 potential host addresses. However, it can only accommodate 62 actual hosts because the all-0 and all-1 host addresses are reserved (they are the network and broadcast addresses, respectively).

In our 128.138.243.0/26 example, the extra two bits of network address obtained by subnetting can take on the values 00, 01, 10, and 11. The 128.138.243.0/24 network has thus been divided into four /26 networks:

- 128.138.243.0/26 (0 in decimal is **00**000000 in binary)
- 128.138.243.64/26 (64 in decimal is **01**000000 in binary)
- 128.138.243.128/26 (128 in decimal is **10**000000 in binary)
- 128.138.243.192/26 (192 in decimal is **11**000000 in binary)

The boldfaced bits of the last byte of each address are the bits that belong to the network portion of that byte.

Tricks and tools for subnet arithmetic

It's confusing to do all this bit twiddling in your head, but some tricks can make it simpler. The number of hosts per network and the value of the last byte in the netmask always add up to 256:

last netmask byte = 256 – net size

For example, 256 – 64 = 192, which is the final byte of the netmask in the preceding example. Another arithmetic fact is that the last byte of an actual network address (as opposed to a netmask) must be evenly divisible by the number of hosts per network. We see this in action in the 128.138.243.0/26 example, where the last bytes of the networks are 0, 64, 128, and 192—all divisible by 64.

Given an IP address (say, 128.138.243.100), we cannot tell without the associated netmask what the network address and broadcast address will be. Table 13.3 shows the possibilities for /16 (the default for a class B address), /24 (a plausible value), and /26 (a reasonable value for a small network).

Table 13.3: Example IPv4 address decodings

IP address	Netmask	Network	Broadcast
128.138.243.100/16	255.255.0.0	128.138.0.0	128.138.255.255
128.138.243.100/24	255.255.255.0	128.138.243.0	128.138.243.255
128.138.243.100/26	255.255.255.192	128.138.243.64	128.138.243.127

The network address and broadcast address steal two hosts from each network, so it would seem that the smallest meaningful network would have four possible hosts: two real hosts—usually at either end of a point-to-point link—and the network and broadcast addresses. To have four values for hosts requires two bits in the host portion, so such a network would be a /30 network with netmask 255.255.255.252 or 0xFFFFFFFC. However, a /31 network is treated as a special case (see RFC3021) and has no network or broadcast address; both of its two addresses are used for hosts, and its netmask is 255.255.255.254.

A handy web site called the IP Calculator by Krischan Jodies (jodies.de/ipcalc) helps with binary/hex/mask arithmetic. IP Calculator displays everything you might need to know about a network address and its netmask, broadcast address, hosts, etc.

A command-line version of the tool, **ipcalc**, is also available. It's in the standard repositories for Debian, Ubuntu, and FreeBSD.

RHEL  Red Hat and CentOS include a similar but unrelated program that's also called **ipcalc**. However, it's relatively useless because it only understands default IP address classes.

Here's some sample **ipcalc** output, munged a bit to help with formatting:

```
$ ipcalc 24.8.175.69/28
Address:   24.8.175.69          00011000.00001000.10101111.0100 0101
Netmask:   255.255.255.240 = 28 11111111.11111111.11111111.1111 0000
Wildcard:  0.0.0.15             00000000.00000000.00000000.0000 1111
=>
Network:   24.8.175.64/28       00011000.00001000.10101111.0100 0000
HostMin:   24.8.175.65          00011000.00001000.10101111.0100 0001
HostMax:   24.8.175.78          00011000.00001000.10101111.0100 1110
Broadcast: 24.8.175.79          00011000.00001000.10101111.0100 1111
Hosts/Net: 14                   Class A
```

The output includes both easy-to-understand versions of the addresses and “cut and paste” versions. Very useful.

If a dedicated IP calculator isn’t available, the standard utility **bc** makes a good backup utility since it can do arithmetic in any base. Set the input and output bases with the **ibase** and **obase** directives. Set the **obase** first; otherwise, it’s interpreted relative to the new **ibase**.

CIDR: Classless Inter-Domain Routing

Like subnetting, of which it is a direct extension, CIDR relies on an explicit netmask to define the boundary between the network and host parts of an address. But unlike subnetting, CIDR allows the network portion to be made *smaller* than would be implied by an address's implicit class. A short CIDR mask can have the effect of aggregating several networks for purposes of routing. Hence, CIDR is sometimes referred to as supernetting.

CIDR is defined in RFC4632.

CIDR simplifies routing information and imposes hierarchy on the routing process. Although CIDR was intended as only an interim solution along the road to IPv6, it has proved to be sufficiently powerful to handle the Internet's growth problems for more than two decades.

For example, suppose that a site has been given a block of eight class C addresses numbered 192.144.0.0 through 192.144.7.0 (in CIDR notation, 192.144.0.0/21). Internally, the site could use them as

- 1 network of length /21 with 2,046 hosts, netmask 255.255.248.0
- 8 networks of length /24 with 254 hosts each, netmask 255.255.255.0
- 16 networks of length /25 with 126 hosts each, netmask 255.255.255.128
- 32 networks of length /26 with 62 hosts each, netmask 255.255.255.192

and so on. But from the perspective of the Internet, it's not necessary to have 32, 16, or even 8 routing table entries for these addresses. They all refer to the same organization, and all the packets go to the same ISP. A single routing entry for 192.144.0.0/21 suffices. CIDR makes it easy to sub-allocate portions of addresses and thus increases the number of available addresses manyfold.

Inside your network, you can mix and match regions of different subnet lengths as long as all the pieces fit together without overlaps. This is called variable length subnetting. For example, an ISP with the 192.144.0.0/21 allocation could define some /30 networks for point-to-point customers, some /24s for large customers, and some /27s for smaller folks.

All the hosts on a network must be configured with the same netmask. You can't tell one host that it is a /24 and another host on the same network that it is a /25.

Address allocation

Only network numbers are formally assigned; sites must define their own host numbers to form complete IP addresses. You can subdivide the address space that has been assigned to you into subnets however you like.

Administratively, ICANN (the Internet Corporation for Assigned Names and Numbers) has delegated blocks of addresses to five regional Internet registries, and these regional authorities are responsible for doling out subblocks to ISPs within their regions. These ISPs in turn divide up their blocks and hand out pieces to individual clients. Only large ISPs should ever have to deal directly with one of the ICANN-sponsored address registries.

Table 13.4 lists the regional registration authorities.

Table 13.4: Regional Internet registries

Name	Site	Region covered
ARIN	arin.net	North America, part of the Caribbean
APNIC	apnic.net	Asia/Pacific region, including Australia and New Zealand
AfriNIC	afrinic.net	Africa
LACNIC	lacnic.net	Central and South America, part of the Caribbean
RIPE NCC	ripe.net	Europe and surrounding areas

The delegation from ICANN to regional registries and then to national or regional ISPs has allowed for further aggregation in the backbone routing tables. ISP customers who have been allocated address space within the ISP's block do not need individual routing entries on the backbone. A single entry for the aggregated block that points to the ISP suffices.

Private addresses and network address translation (NAT)

Another factor that has mitigated the effect of the IPv4 address crisis is the use of private IP address spaces, described in RFC1918. These addresses are used by your site internally but are never shown to the Internet (or at least, not intentionally). A border router translates between your private address space and the address space assigned by your ISP.

RFC1918 sets aside 1 class A network, 16 class B networks, and 256 class C networks that will never be globally allocated and can be used internally by any site. Table 13.5 shows the options. (The "CIDR range" column shows each range in the more compact CIDR notation; it does not add additional information.)

Table 13.5: IP addresses reserved for private use

IP class	From	To	CIDR range
Class A	10.0.0.0	10.255.255.255	10.0.0.0/8
Class B	172.16.0.0	172.31.255.255	172.16.0.0/12
Class C	192.168.0.0	192.168.255.255	192.168.0.0/16

The original idea was that sites would choose an address class from among these options to fit the size of their organizations. But now that CIDR and subnetting are universal, it probably makes the most sense to use the class A address (subnetted, of course) for all new private networks.

To allow hosts that use these private addresses to talk to the Internet, the site's border router runs a system called NAT (Network Address Translation). NAT intercepts packets addressed with these internal addresses and rewrites their source addresses, using a valid external IP address and perhaps a different source port number. It also maintains a table of the mappings it has made between internal and external address/port pairs so that the translation can be performed in reverse when answering packets arrive from the Internet.

Many garden-variety "NAT" gateways actually perform Port Address Translation, aka PAT: they use a single external IP address and multiplex connections for many internal clients onto the port space of that single address. For example, this is the default configuration for most of the mass-market routers used with cable modems. In practice, NAT and PAT are similar in terms of their implementation, and both systems are commonly referred to as NAT.

A site that uses NAT must still request a small section of address space from its ISP, but most of the addresses thus obtained are used for NAT mappings and are not assigned to individual hosts. If the site later wants to choose another ISP, only the border router and its NAT configuration need be updated, not the configurations of the individual hosts.

Large organizations that use NAT and RFC1918 addresses must institute some form of central coordination so that all hosts, independently of their department or administrative group, have unique IP addresses. The situation can become complicated when one company that uses RFC1918 address space acquires or merges with another company that's doing the same thing. Parts of the combined organization must often renumber.

It is possible to have a UNIX or Linux box perform the NAT function, but most sites prefer to delegate this task to their routers or network connection devices. See the vendor-specific sections later in this chapter for details. (Of course, many routers now run embedded Linux kernels. Even so, these dedicated systems are still generally more reliable and more secure than general-purpose computers that also forward packets.)

An incorrect NAT configuration can let private-address-space packets escape onto the Internet. The packets might get to their destinations, but answering packets won't be able to get back. CAIDA, an organization that collects operational data from the Internet backbone, finds that 0.1% to 0.2% of the packets on the backbone have either private addresses or bad checksums. This sounds like a tiny percentage, but it represents thousands of packets every minute on a busy circuit. See caida.org for other interesting statistics and network measurement tools.

One issue raised by NAT is that an arbitrary host on the Internet cannot initiate connections to your site's internal machines. To get around this limitation, NAT implementations let you preconfigure externally visible "tunnels" that connect to specific internal hosts and ports. Many routers also support the Universal Plug and Play (UPnP) standards promoted by Microsoft, one feature of which allows interior hosts to set up their own dynamic NAT tunnels. This can be either a godsend or a security risk, depending on your perspective. You can easily disable the feature at the router if you so desire.

Another NAT-related issue is that some applications embed IP addresses in the data portion of packets; these applications are foiled or confused by NAT. Examples include some media streaming systems, routing protocols, and FTP commands. NAT sometimes breaks VPNs, too.

NAT hides interior structure. This secrecy feels like a security win, but the security folks say NAT doesn't really help for security and does not replace the need for a firewall. Unfortunately, NAT also foils attempts to measure the size and topology of the Internet. See RFC4864, *Local Network Protection for IPv6*, for a good discussion of both the real and illusory benefits of NAT in IPv4.

IPv6 addressing

IPv6 addresses are 128 bits long. These long addresses were originally intended to solve the problem of IP address exhaustion. But now that they're here, they are being exploited to help with issues of routing, mobility, and locality of reference.

The boundary between the network portion and the host portion of an IPv6 address is fixed at /64, so there can be no disagreement or confusion about how long an address's network portion "really" is. Stated another way, true subnetting no longer exists in the IPv6 world, although the term "subnet" lives on as a synonym for "local network." Even though network numbers are always 64 bits long, routers needn't pay attention to all 64 bits when making routing decisions. They can route packets by prefix, just as they do under CIDR.

IPv6 address notation

The standard notation for IPv6 addresses divides the 128 bits of an address into 8 groups of 16 bits each, separated by colons. For example,

2607:f8b0:000a:0806:0000:0000:0000:200e

This is a real IPv6 address, so don't use it on your own systems, even for experimentation. RFC3849 suggests that documentation and examples show IPv6 addresses within the prefix block 2001:db8::/32. But we wanted to show a real example that's routed on the Internet backbone.

Each 16-bit group is represented by 4 hexadecimal digits. This is different from IPv4 notation, in which each byte of the address is represented by a decimal (base 10) number.

A couple of notational simplifications help limit the amount of typing needed to represent IPv6 addresses. First, you needn't include leading zeros within a group. 000a in the third group above can be written simply as a, and 0806 in the fourth group can be written as 806. Groups with a value of 0000 should be represented as 0. Application of this rule reduces the address above to the following string:

2607:f8b0:a:806:0:0:0:200e

Second, you can replace any number of contiguous, zero-valued, 16-bit groups with a double colon:

2607:f8b0:a:806::200e

The :: can be used only once within an address. However, it can appear as the first or last component. For example, the IPv6 loopback address (analogous to 127.0.0.1 in IPv4) is ::1, which is equivalent to 0:0:0:0:0:0:0:1.

The original specification for IPv6 addresses, RFC4921, documented these notational simplifications but did not require their use. As a result, there can be multiple RFC491-compliant

ways to write a given IPv6 address, as illustrated by the several versions of the example address above.

This polymorphousness makes searching and matching difficult because addresses must be normalized before they can be compared. That's a problem: we can't expect standard data-wrangling software such as spreadsheets, scripting languages, and databases to know about the details of IPv6 notation.

RFC5952 updates RFC4921 to make the notational simplifications mandatory. It also adds a few more rules to ensure that every address has only a single text representation:

- Hex digits a–f must be represented by lowercase letters.
- The :: element cannot replace a single 16-bit group. (Just use :0:.)
- If there is a choice of groups to replace with ::, the :: must replace the longest possible sequence of zeros.

You will still see RFC5952-noncompliant addresses out in the wild, and nearly all networking software accepts them, too. However, we strongly recommend following the RFC5952 rules in your configurations, recordkeeping, and software.

IPv6 prefixes

IPv4 addresses were not designed to be geographically clustered in the manner of phone numbers or zip codes, but clustering was added after the fact in the form of the CIDR conventions. (Of course, the relevant "geography" is really routing space rather than physical location.) CIDR was so technically successful that hierarchical subassignment of network addresses is now assumed throughout IPv6.

Your IPv6 ISP obtains blocks of IPv6 prefixes from one of the regional registries listed in Table 13.4. The ISP in turn assigns you a prefix that you prepend to the local parts of your addresses, usually at your border router. Organizations are free to set delegation boundaries wherever they wish within the address spaces assigned to them.

Whenever an address prefix is represented in text form, IPv6 adopts CIDR notation to represent the length of the prefix. The general pattern is

IPv6-address/prefix-length-in-decimal

The *IPv6-address* portion is as outlined in the previous section. It must be a full-length 128-bit address. In most cases, the address bits beyond the prefix are set to zero. However, it's sometimes appropriate to specify a complete host address along with a prefix length; the intent and meaning are usually clear from context.

The IPv6 address shown in the previous section leads to a Google server. The 32-bit prefix that's routed on the North American Internet backbone is

2607:f8b0::/32

In this case, the address prefix was assigned by ARIN directly to Google, as you can verify by looking up the prefix at arin.net. There is no intervening ISP. Google is responsible for structuring the remaining 32 bits of the network number as it sees fit. Most likely, several additional layers of prefixing are used within the Google infrastructure.

In this case, the prefix lengths of the ARIN-assigned address block and the backbone routing table entry are the same, but that is not always true. The allocation prefix determines an administrative boundary, whereas the routing prefix relates to route-space locality.

Automatic host numbering

A machine's 64-bit interface identifier (the host portion of the IPv6 address) can be automatically derived from the interface's 48-bit MAC (hardware) address with the algorithm known as "modified EUI-64," documented in RFC4291.

Specifically, the interface identifier is just the MAC address with the two bytes 0xFFFE inserted in the middle and one bit complemented. The bit you will flip is the 7th most significant bit of the first byte; in other words, you XOR the first byte with 0x02. For example, on an interface with MAC address 00:1b:21:30:e9:c7, the autogenerated interface identifier would be 021b:21ff:fe30:e9c7. The underlined digit is 2 instead of 0 because of the flipped bit.

This scheme allows for automatic host numbering, which is a nice feature for sysadmins since only the network portion of addresses need be managed.

That the MAC address can be seen at the IP layer has both good and bad implications. The good part is that host number configuration can be completely automatic. The bad part is that the manufacturer of the interface card is encoded in the first half of the MAC address (see this page), so you inevitably give up some privacy. Prying eyes and hackers with code for a particular architecture will be helped along. The IPv6 standards point out that sites are not *required* to use MAC addresses to derive host IDs; they can use whatever numbering system they want.

Virtual servers have virtual network interfaces. The MAC addresses associated with these interfaces are typically randomized, which all but guarantees uniqueness within a particular local context.

Stateless address autoconfiguration

The autogenerated host numbers described in the previous section combine with a couple of other simple IPv6 features to enable automatic network configuration for IPv6 interfaces. The overall scheme is known as SLAAC, for StateLess Address AutoConfiguration.

SLAAC configuring for an interface begins by assigning an address on the "link-local network," which has the fixed network address fe80::/64. The host portion of the address is set from the MAC address of the interface, as described above.

IPv6 does not have IPv4-style broadcast addresses per se, but the link-local network serves roughly the same purpose: it means "this physical network." Routers never forward packets that were sent to addresses on this network.

Once the link-local address for an interface has been set, the IPv6 protocol stack sends an ICMP Router Solicitation packet to the "all routers" multicast address. Routers respond with ICMP Router Advertisement packets that list the IPv6 network numbers (prefixes, really) in use on the network.

If one of these networks has its "autoconfiguration OK" flag set, the inquiring host assigns an additional address to its interface that combines the network portion advertised by the router with the autogenerated host portion constructed with the modified EUI-64 algorithm. Other fields in the Router Advertisement allow a router to identify itself as an appropriate default gateway and to communicate the network's MTU.

See this page for more information about DHCP.

The end result is that a new host becomes a full citizen of the IPv6 network without the need for any server (other than the router) to be running on the network and without any local configuration. Unfortunately, the system does not address the configuration of higher-level software such as DNS, so you may still want to run a traditional DHCPv6 server, too.

You will sometimes see IPv6 network autoconfiguration associated with the name Neighbor Discovery Protocol. Although RFC4861 is devoted to the Neighbor Discovery Protocol, the term is actually rather vague. It covers the use and interpretation of a variety of ICMPv6 packet types, some of which are only peripherally related to discovering network neighbors. From a technical perspective, the relationship is that the SLAAC procedure described above uses some, but not all, of the ICMPv6 packet types defined in RFC4861. It's clearer to just call it SLAAC or "IPv6 autoconfiguration" and to reserve "neighbor discovery" for the IP-to-MAC mapping process described starting here.

IPv6 tunneling

Various schemes have been proposed to ease the transition from IPv4 to IPv6, mostly focusing on ways to tunnel IPv6 traffic through the IPv4 network to compensate for gaps in IPv6 support. The two tunneling systems in common use are called 6to4 and Teredo; the latter, named after a family of wood-boring shipworms, can be used on systems behind a NAT device.

IPv6 information sources

Here are some useful sources of additional IPv6 information:

- worldipv6launch.com – A variety of IPv6 propaganda
- RFC3587 – *IPv6 Global Unicast Address Format*

- RFC4291 – *IP Version 6 Addressing Architecture*

13.5 Routing

Routing is the process of directing a packet through the maze of networks that stand between its source and its destination. In the TCP/IP system, it is similar to asking for directions in an unfamiliar country. The first person you talk to might point you toward the right city. Once you were a bit closer to your destination, the next person might be able to tell you how to get to the right street. Eventually, you get close enough that someone can identify the building you're looking for.

Routing information takes the form of rules ("routes"), such as "To reach network A, send packets through machine C." There can also be a default route that tells what to do with packets bound for a network to which no explicit route exists.

Routing information is stored in a table in the kernel. Each table entry has several parameters, including a mask for each listed network. To route a packet to a particular address, the kernel picks the most specific of the matching routes—that is, the one with the longest mask. If the kernel finds no relevant route and no default route, then it returns a "network unreachable" ICMP error to the sender.

The word "routing" is commonly used to mean two distinct things:

- Looking up a network address in the routing table as part of the process of forwarding a packet toward its destination
- Building the routing table in the first place

In this section we examine the forwarding function and look at how routes can be manually added to or deleted from the routing table. We defer the more complicated topic of routing protocols that build and maintain the routing table until Chapter 15.

Routing tables

You can examine a machine's routing table with **ip route show** on Linux or **netstat -r** on FreeBSD. Although **netstat** on Linux is on its way out, it still exists and continues to work. We use **netstat** for the examples below just to avoid having to show two different versions of the output. The **ip** version contains similar content, but its format is somewhat different.

Use **netstat -rn** to avoid DNS lookups and present all information numerically, which is generally more useful. Here is a short example of an IPv4 routing table to give you a better idea of what routes look like:

```
redhat$ netstat -rn
Destination      Genmask          Gateway         Fl  MSS   Iface
132.236.227.0    255.255.255.0    132.236.227.93  U   1500  eth0
default          0.0.0.0          132.236.227.1   UG  1500  eth0
132.236.212.0    255.255.255.192  132.236.212.1   U   1500  eth1
132.236.220.64   255.255.255.192  132.236.212.6   UG  1500  eth1
127.0.0.1        255.255.255.255  127.0.0.1       U   3584  lo
```

This host has two network interfaces: 132.236.227.93 (eth0) on the network 132.236.227.0/24 and 132.236.212.1 (eth1) on the network 132.236.212.0/26.

The destination field is usually a network address, although you can also add host-specific routes (their genmask is 255.255.255.255 since all bits are consulted). An entry's gateway field must contain the full IP address of a local network interface or adjacent host; on Linux kernels it can be 0.0.0.0 to invoke the default gateway.

For example, the fourth route in the table above says that to reach the network 132.236.220.64/26, packets must be sent to the gateway 132.236.212.6 through interface eth1. The second entry is a default route; packets not explicitly addressed to any of the three networks listed (or to the machine itself) are sent to the default gateway host, 132.236.227.1.

A host can route packets only to gateway machines that are reachable through a directly connected network. The local host's job is limited to moving packets one hop closer to their destinations, so it is pointless to include information about nonadjacent gateways in the local routing table. Each gateway that a packet visits makes a fresh next-hop routing decision by consulting its own local routing database. (The IPv4 source routing feature is an exception to this rule; see this page.)

Routing tables can be configured statically, dynamically, or with a combination of the two approaches. A static route is one that you enter explicitly with the **ip** (Linux) or **route** (FreeBSD) command. Static routes remain in the routing table as long as the system is up; they are often set up at boot time from one of the system startup scripts. For example, the Linux commands

```
ip route add 132.236.220.64/26 via 132.236.212.6 dev eth1
ip route add default via 132.236.227.1 dev eth0
```

add the fourth and second routes displayed by **netstat -rn** above. (The first and third routes in that display were added automatically when the eth0 and eth1 interfaces were configured.) The equivalent FreeBSD commands are similar:

```
route add –net 132.236.220.64/26 gw 132.236.212.6 eth1
route add default gw 132.236.227.1 eth0
```

The final route is also added at boot time. It configures the loopback interface, which prevents packets sent from the host to itself from going out on the network. Instead, they are transferred directly from the network output queue to the network input queue inside the kernel.

In a stable local network, static routing is an efficient solution. It is easy to manage and reliable. However, it requires that the system administrator know the topology of the network accurately at boot time and that the topology not change often.

Most machines on a local area network have only one way to get out to the rest of the network, so the routing problem is easy. A default route added at boot time suffices to point toward the way out. Hosts that use DHCP (see this page) to get their IP addresses can also obtain a default route with DHCP.

For more complicated network topologies, dynamic routing is required. Dynamic routing is implemented by a daemon process that maintains and modifies the routing table. Routing daemons on different hosts communicate to discover the topology of the network and to figure out how to reach distant destinations. Several routing daemons are available. See Chapter 15, *IP Routing*, for details.

ICMP redirects

Although IP generally does not concern itself with the management of routing information, it does define a naïve damage control feature called an ICMP redirect. When a router forwards a packet to a machine on the same network from which the packet was originally received, something is clearly wrong. Since the sender, the router, and the next-hop router are all on the same network, the packet could have been forwarded in one hop rather than two. The router can conclude that the sender's routing tables are inaccurate or incomplete.

In this situation, the router can notify the sender of its problem by sending an ICMP redirect packet. In effect, a redirect says, "You should not be sending packets for host *xxx* to me; you should send them to host *yyy* instead."

In theory, the recipient of a redirect can adjust its routing table to fix the problem. In practice, redirects contain no authentication information and are therefore untrustworthy. Dedicated routers usually ignore redirects, but most UNIX and Linux systems accept them and act on them by default. You'll need to consider the possible sources of redirects in your network and disable their acceptance if they could pose a problem.



Under Linux, the variable **accept_redirects** in the **/proc** hierarchy controls the acceptance of ICMP redirects. See this page for instructions on examining and resetting this variable.



On FreeBSD the parameters net.inet.icmp.drop_redirect and net.inet6.icmp6.rediraccept control the acceptance of ICMP redirects. Set them to 1 and 0, respectively, in the file **/etc/sysctl.conf** to ignore redirects. (To activate the new settings, reboot or run **sudo /etc/rc.d/sysctl reload**.)

13.6 IPv4 ARP AND IPv6 NEIGHBOR DISCOVERY

Although IP addresses are hardware-independent, hardware addresses must still be used to actually transport data across a network's link layer. (An exception is for point-to-point links, where the identity of the destination is sometimes implicit.) IPv4 and IPv6 use separate but eerily similar protocols to discover the hardware address associated with a particular IP address.

IPv4 uses ARP, the Address Resolution Protocol, defined in RFC826. IPv6 uses parts of the Neighbor Discovery Protocol defined in RFC4861. These protocols can be used on any kind of network that supports broadcasting or all-nodes multicasting, but they are most commonly described in terms of Ethernet.

If host A wants to send a packet to host B on the same Ethernet, it uses ARP or ND to discover B's hardware address. If B is not on the same network as A, host A uses the routing system to determine the next-hop router along the route to B and then uses ARP or ND to find that router's hardware address. These protocols can only be used to find the hardware addresses of machines that are directly connected to the sending host's local networks.

Every machine maintains a table in memory called the ARP or ND cache which contains the results of recent queries. Under normal circumstances, many of the addresses a host needs are discovered soon after booting, so ARP and ND do not account for a lot of network traffic.

These protocols work by broadcasting or multicasting a packet of the form "Does anyone know the hardware address for IP address X?" The machine being searched for recognizes its own IP address and replies, "Yes, that's the IP address assigned to one of my network interfaces, and the corresponding MAC address is 08:00:20:00:fb:6a."

The original query includes the IP and MAC addresses of the requester so that the machine being sought can reply without issuing a query of its own. Thus, the two machines learn each other's address mappings with only one exchange of packets. Other machines that overhear the requester's initial broadcast can record its address mapping, too.


On Linux, the **ip neigh** command examines and manipulates the caches created by ARP and ND, adds or deletes entries, and flushes or prints the table. **ip neigh show** displays the contents of the caches.


On FreeBSD, the **arp** command manipulates the ARP cache and the **ndp** command gives access to the ND cache.

These commands are generally useful only for debugging and for situations that involve special hardware. For example, if two hosts on a network are using the same IP address, one has the right ARP or ND table entry and one is wrong. You can use the cache information to track down the offending machine.

Inaccurate cache entries can be a sign that someone with access to your local network is attempting to hijack network traffic. This type of attack is known generically as ARP spoofing or ARP cache poisoning.

13.7 DHCP: THE DYNAMIC HOST CONFIGURATION PROTOCOL

DHCP is defined in RFCs 2131, 2132, and 3315.

When you plug a device or computer into a network, it usually obtains an IP address for itself on the local network, sets up an appropriate default route, and connects itself to a local DNS server. The Dynamic Host Configuration Protocol (DHCP) is the hidden Svengali that makes this magic happen.

The protocol lets a DHCP client “lease” a variety of network and administrative parameters from a central server that is authorized to distribute them. The leasing paradigm is particularly convenient for PCs that are turned off when not in use and for networks that must support transient guests such as laptops.

Leasable parameters include

- IP addresses and netmasks
- Gateways (default routes)
- DNS name servers
- Syslog hosts
- WINS servers, X font servers, proxy servers, NTP servers
- TFTP servers (for loading a boot image)

There are dozens more—see RFC2132 for IPv4 and RFC3315 for IPv6. Real-world use of the more exotic parameters is rare, however.

Clients must report back to the DHCP server periodically to renew their leases. If a lease is not renewed, it eventually expires. The DHCP server is then free to assign the address (or whatever was being leased) to a different client. The lease period is configurable, but it’s usually quite long (hours or days).

Even if you want each host to have its own permanent IP address, DHCP can save you time and suffering because it concentrates configuration information on the DHCP server rather than requiring it to be distributed to individual hosts. Once the server is up and running, clients can use DHCP to obtain their network configurations at boot time. The clients needn’t know that they’re receiving a static configuration.

DHCP software

ISC, the Internet Systems Consortium, maintains a nice open source reference implementation of DHCP. Major versions 2, 3, and 4 of ISC's software are all in common use, and all these versions work fine for basic service. Version 3 supports backup DHCP servers, and version 4 supports IPv6. Server, client, and relay agents are all available from isc.org.

Vendors all package some version of the ISC software, although you may have to install the server portion explicitly. The server package is called **dhcp** on Red Hat and CentOS, **isc-dhcp-server** on Debian and Ubuntu, and **isc-dhcp43-server** on FreeBSD. Make sure you're installing the software you intend, as many systems package multiple implementations of both the server and client sides of DHCP.

It's best not to tamper with the client side of DHCP, since that part of the code is relatively simple and comes preconfigured and ready to use. Changing the client side of DHCP is not trivial.

However, if you need to run a DHCP *server*, we recommend the ISC package over vendor-specific implementations. In a typical heterogeneous network environment, administration is greatly simplified by standardizing on a single implementation. The ISC software is a reliable, open source solution that builds without incident on most systems.

Another option to consider is Dnsmasq, a server that implements DHCP service in combination with a DNS forwarder. It's a tidy package that runs on pretty much any system. The project home page is thekelleys.org.uk/dnsmasq/doc.html.

DHCP server software is also built into most routers. Configuration is usually more painful than on a UNIX- or Linux-based server, but reliability and availability might be higher.

In the next few sections, we briefly discuss the DHCP protocol, explain how to set up the ISC server that implements it, and review some client configuration issues.

DHCP behavior

DHCP is a backward-compatible extension of BOOTP, a protocol originally devised to help diskless workstations boot. DHCP generalizes the parameters that can be supplied and adds the concept of a lease period for assigned values.

A DHCP client begins its interaction with a DHCP server by broadcasting a “Help! Who am I?” message. IPv4 clients initiate conversations with the DHCP server by using the generic all-1s broadcast address. The clients don’t yet know their subnet masks and therefore can’t use the subnet broadcast address. IPv6 uses multicast addressing instead of broadcasting.

If a DHCP server is present on the local network, it negotiates with the client to supply an IP address and other networking parameters. If there is no DHCP server on the local net, servers on different subnets can receive the initial broadcast message through a separate piece of DHCP software that acts as a relay agent.

When the client’s lease time is half over, it attempts to renew its lease. The server is obliged to keep track of the addresses it has handed out, and this information must persist across reboots. Clients are supposed to keep their lease state across reboots too, although many do not. The goal is to maximize stability in network configuration. In theory, all software should be prepared for network configurations to change at a moment’s notice, but some software still makes unwarranted assumptions about the continuity of the network.

ISC's DHCP software

ISC's server daemon is called **dhcpd**, and its configuration file is **dhcpd.conf**, usually found in **/etc** or **/etc/dhcp3**. The format of the config file is a bit fragile; leave out a semicolon and you may receive a cryptic, unhelpful error message.

When setting up a new DHCP server, you must also make sure that an empty lease database file has been created. Check the summary at the end of the man page for **dhcpd** to find the correct location for the lease file on your system. It's usually somewhere underneath **/var**.

To set up the **dhcpd.conf** file, you need the following information:

- The subnets for which **dhcpd** should manage IP addresses, and the ranges of addresses to dole out
- A list of static IP address assignments you want to make (if any), along with the MAC (hardware) addresses of the recipients
- The initial and maximum lease durations, in seconds
- Any other options the server should pass to DHCP clients: netmask, default route, DNS domain, name servers, etc.

The **dhcpd** man page outlines the configuration process, and the **dhcpd.conf** man page covers the exact syntax of the config file. In addition to setting up your configuration, make sure **dhcpd** is started automatically at boot time. (See Chapter 2, *Booting and System Management Daemons*, for instructions.) It's helpful to make startup of the daemon conditional on the existence of the **dhcpd.conf** file if your system doesn't automatically do this for you.

Below is a sample **dhcpd.conf** file from a Linux box with two interfaces, one internal and one that connects to the Internet. This machine performs NAT translation for the internal network (see this page) and leases out a range of 10 IP addresses on this network as well.

Every subnet must be declared, even if no DHCP service is provided on it, so this **dhcpd.conf** file contains a dummy entry for the external interface. It also includes a `host` entry for one particular machine that needs a fixed address.

```
# global options

option domain-name "synack.net";
option domain-name-servers gw.synack.net;
option subnet-mask 255.255.255.0;
default-lease-time 600;
max-lease-time 7200;

subnet 192.168.1.0 netmask 255.255.255.0 {
    range 192.168.1.51 192.168.1.60;
    option broadcast-address 192.168.1.255;
    option routers gw.synack.net;
}

subnet 209.180.251.0 netmask 255.255.255.0 {
}

host gandalf {
    hardware ethernet 08:00:07:12:34:56;
    fixed-address gandalf.synack.net;
}
```

See Chapter 16 for more information about DNS.

Unless you make static IP address assignments such as the one for gandalf above, you need to consider how your DHCP configuration interacts with DNS. The easy option is to assign a generic name to each dynamically leased address (e.g., dhcp1.synack.net) and allow the names of individual machines to float along with their IP addresses. Alternatively, you can configure **dhcpd** to update the DNS database as it hands out addresses. The dynamic update solution is more complicated, but it has the advantage of preserving each machine's hostname.

ISC's DHCP relay agent is a separate daemon called **dhcrelay**. It's a simple program with no configuration file of its own, although vendors often add a startup harness that feeds it the appropriate command-line arguments for your site. **dhcrelay** listens for DHCP requests on local networks and forwards them to a set of remote DHCP servers that you specify. It's handy both for centralizing the management of DHCP service and for provisioning backup DHCP servers.

ISC's DHCP client is similarly configuration free. It stores status files for each connection in the directory **/var/lib/dhcp** or **/var/lib/dhclient**. The files are named after the interfaces they describe. For example, **dhclient-eth0.leases** would contain all the networking parameters that **dhclient** had set up on behalf of the eth0 interface.

13.8 Security issues

We address the topic of security in a chapter of its own (Chapter 27), but several security issues relevant to IP networking merit discussion here. In this section, we briefly look at a few networking features that have acquired a reputation for causing security problems, and we recommend ways to minimize their impact. The details of our example systems' default behavior on these issues (and the appropriate methods for changing them) vary considerably and are discussed in the system-specific material starting here.

IP forwarding

A UNIX or Linux system that has IP forwarding enabled can act as a router. That is, it can accept third party packets on one network interface, match them to a gateway or destination host on another interface, and retransmit the packets.

Unless your system has multiple network interfaces and is actually supposed to function as a router, it’s best to turn this feature off. Hosts that forward packets can sometimes be coerced into compromising security by making external packets appear to have come from inside your network. This subterfuge can help an intruder’s packets evade network scanners and packet filters.

It is perfectly acceptable for a host to have network interfaces on multiple subnets and to use them for its own traffic without forwarding third party packets.

ICMP redirects

ICMP redirects (see this page) can maliciously reroute traffic and tamper with your routing tables. Most operating systems listen to ICMP redirects and follow their instructions by default. It would be bad if all your traffic were rerouted to a competitor's network for a few hours, especially while backups were running! We recommend that you configure your routers (and hosts acting as routers) to ignore and perhaps log ICMP redirect attempts.

Source routing

IPv4's source routing mechanism lets you specify an explicit series of gateways for a packet to transit on the way to its destination. Source routing bypasses the next-hop routing algorithm that's normally run at each gateway to determine how a packet should be forwarded.

Source routing was part of the original IP specification; it was intended primarily to facilitate testing. It can create security problems because packets are often filtered according to their origin. If someone can cleverly route a packet to make it appear to have originated within your network instead of the Internet, it might slip through your firewall. We recommend that you neither accept nor forward source-routed packets.

Despite the Internet's dim view of IPv4 source routing, it somehow managed to sneak its way into IPv6 as well. However, this IPv6 feature was deprecated by RFC5095 in 2007. Compliant IPv6 implementations are now required to reject source-routed packets and return an error message to the sender. Linux and FreeBSD both follow the RFC5095 behavior, as do commercial routers.

Even so, IPv6 source routing may be poised to stage a mini-comeback in the form of "segment routing," a feature that has now been integrated into the Linux kernel. See lwn.net/Articles/722804 for a discussion of this technology.

Broadcast pings and other directed broadcasts

Ping packets addressed to a network's broadcast address (instead of to a particular host address) are typically delivered to every host on the network. Such packets have been used in denial-of-service attacks; for example, the so-called Smurf attacks. (The "Smurf attacks" Wikipedia article has details.)

Broadcast pings are a form of "directed broadcast" in that they are packets sent to the broadcast address of a distant network. The default handling of such packets has been gradually changing. For example, versions of Cisco's IOS up through 11.x forwarded directed broadcast packets by default, but IOS releases since 12.0 do not. It is usually possible to convince your TCP/IP stack to ignore broadcast packets that come from afar, but since this behavior must be set on each interface, the task can be nontrivial at a large site.

IP spoofing

The source address on an IP packet is normally filled in by the kernel's TCP/IP implementation and is the IP address of the host from which the packet was sent. However, if the software creating the packet uses a raw socket, it can fill in any source address it likes. This is called IP spoofing and is usually associated with some kind of malicious network behavior. The machine identified by the spoofed source IP address (if it is a real address at all) is often the victim in the scheme. Error and return packets can disrupt or flood the victim's network connections. Packet spoofing from a large set of external machines is called a "distributed denial-of-service attack."

Deny IP spoofing at your border router by blocking outgoing packets whose source address is not within your address space. This precaution is especially important if your site is a university where students like to experiment and might be tempted to carry out digital vendettas.

If you are using private address space internally, you can filter at the same time to catch any internal addresses escaping to the Internet. Such packets can never be answered (because they lack a backbone route) and always indicate that your site has some kind of internal configuration error.

In addition to detecting outbound packets with bogus source addresses, you must also protect against an attacker's forging the source address on external packets to fool your firewall into thinking that they originated on your internal network. A heuristic known as "unicast reverse path forwarding" (uRPF) helps to address this problem. It makes IP gateways discard packets that arrive on an interface different from the one on which they would be transmitted if the source address were the destination. It's a quick sanity check that uses the normal IP routing table as a way to validate the origin of network packets. Dedicated routers implement uRPF, but so does the Linux kernel. On Linux, it's enabled by default.

If your site has multiple connections to the Internet, it might be perfectly reasonable for inbound and outbound routes to be different. In this situation, you'll have to turn off uRPF to make your routing work properly. If your site has only one way out to the Internet, then turning on uRPF is usually safe and appropriate.

Host-based firewalls

Traditionally, a network packet filter or firewall connects your local network to the outside world and controls traffic according to a site-wide policy. Unfortunately, Microsoft has warped everyone's perception of how a firewall should work with its notoriously insecure Windows systems. The last few Windows releases all come with their own personal firewalls, and they complain bitterly if you try to turn off the firewall.

Our example systems all include packet filtering software, but you should not infer from this that every UNIX or Linux machine needs its own firewall. The packet filtering features are there primarily to allow these machines to serve as network gateways.

However, we don't recommend using a workstation as a firewall. Even with meticulous hardening, full-fledged operating systems are too complex to be fully trustworthy. Dedicated network equipment is more predictable and more reliable—even if it secretly runs Linux.

Even sophisticated software solutions like those offered by Check Point (whose products run on UNIX, Linux, and Windows hosts) are not as secure as a dedicated device such as Cisco's Adaptive Security Appliance series. The software-only solutions are nearly the same price, to boot.

A more thorough discussion of firewall-related issues begins here.

Virtual private networks

Many organizations that have offices in several locations would like to have all those locations connected to one big private network. Such organizations can use the Internet as if it were a private network by establishing a series of secure, encrypted "tunnels" among their various locations. A network that includes such tunnels is known as a virtual private network or VPN.

VPN facilities are also needed when employees must connect to your private network from their homes or from the field. A VPN system doesn't eliminate every possible security issue relating to such ad hoc connections, but it's secure enough for many purposes.

See this page for more information about IPsec.

Some VPN systems use the IPsec protocol, which was standardized by the IETF in 1998 as a relatively low-level adjunct to IP. Others, such as OpenVPN, implement VPN security on top of TCP by using Transport Layer Security (TLS), the successor to the Secure Sockets Layer (SSL). TLS is also on the IETF's standards track, although it hasn't yet been fully adopted as of this writing (2017).

A variety of proprietary VPN implementations are also available. These systems generally don't interoperate with one another or with the standards-based VPN systems, but that's not necessarily a major drawback if all the endpoints are under your control.

The TLS-based VPN solutions seem to be the marketplace winners at this point. They are just as secure as IPsec and considerably less complicated. Having a free implementation in the form of OpenVPN doesn't hurt either.

For users at home and at large, a common paradigm is for them to download a small Java or executable component through their web browser. This component then implements VPN connectivity back to the enterprise network. The mechanism is convenient for users, but be aware that the browser-based systems differ widely in their implementations: some offer VPN service through a pseudo-network-interface, while others forward only specific ports. Still others are little more than glorified web proxies.

Be sure you understand the underlying technology of the solutions you're considering, and don't expect the impossible. True VPN service (that is, full IP-layer connectivity through a network interface) requires administrative privileges and software installation on the client, whether that client is a Windows system or a Linux laptop. Check browser compatibility too, since the voodoo involved in implementing browser-based VPN solutions often doesn't translate among browsers.

13.9 Basic network configuration

Only a few steps are involved in adding a new machine to an existing local area network, but every system does it slightly differently. Systems with a GUI installed typically include a control panel for network configuration, but these visual tools address only simple scenarios. On a typical server, you just enter the network configuration directly into text files.

Before bringing up a new machine on a network that is connected to the Internet, secure it (Chapter 27, *Security*) so that you are not inadvertently inviting attackers onto your local network.

Adding a new machine to a local network goes like this:

1. Assign a unique IP address and hostname.
2. Configure network interfaces and IP addresses.
3. Set up a default route and perhaps fancier routing.
4. Point to a DNS name server to allow access to the rest of the Internet.

If you rely on DHCP for basic provisioning, most of the configuration chores for a new machine are performed on the DHCP server rather than on the new machine itself. New OS installations typically default to configuration through DHCP, so new machines may require no network configuration at all. Refer to the DHCP section starting here for general information.

After any change that might affect startup, always reboot to verify that the machine comes up correctly. Six months later when the power has failed and the machine refuses to boot, it's hard to remember what change you made that might have caused the problem.

The process of designing and installing a physical network is touched on in Chapter 14, *Physical Networking*. If you are dealing with an existing network and have a general idea of how it is set up, it may not be necessary for you to read too much more about the physical aspects of networking unless you plan to extend the existing network.

In this section, we review the various issues involved in manual network configuration. This material is general enough to apply to any UNIX or Linux system. In the vendor-specific sections starting here, we address the unique twists that separate the various vendors' systems.

As you work through basic network configuration on any machine, you'll find it helpful to test your connectivity with basic tools such as **ping** and **traceroute**. See *Network troubleshooting* for a description of these tools.

Hostname and IP address assignment

See Chapter 16 for more information about DNS.

Administrators have various heartfelt theories about how the mapping from hostnames to IP addresses is best maintained: through the **hosts** file, LDAP, the DNS system, or perhaps some combination of those options. The conflicting goals are scalability, consistency, and maintainability versus a system that is flexible enough to allow machines to boot and function when not all services are available.

Another consideration when you're designing your addressing system is the possible need to renumber your hosts in the future. Unless you are using RFC1918 private addresses (see this page), your site's IP addresses might change when you switch ISPs. Such a transition becomes daunting if you must visit each host on the network to reconfigure its address. To expedite renumbering, you can use hostnames in configuration files and confine address mappings to a few centralized locations such as the DNS database and your DHCP configuration files.

The **/etc/hosts** file is the oldest and simplest way to map names to IP addresses. Each line starts with an IP address and continues with the various symbolic names by which that address is known.

Here is a typical **/etc/hosts** file for the host lollipop:

```
127.0.0.1        localhost
::1              localhost ip6-localhost
ff02::1          ip6-allnodes
ff02::2          ip6-allrouters
192.108.21.48    lollipop.atrust.com lollipop loghost
192.108.21.254   chimchim-gw.atrust.com chimchim-gw
192.108.21.1     ns.atrust.com ns
192.225.33.5     licenses.atrust.com license-server
```

A minimalist version would contain only the first three lines. localhost is commonly the first entry in the **/etc/hosts** file; this entry is unnecessary on many systems, but it doesn't hurt to include it. You can freely intermix IPv4 and IPv6 addresses.

Because **/etc/hosts** contains only local mappings and must be maintained on each client system, it's best reserved for mappings that are needed at boot time (e.g., the host itself, the default gateway, and name servers). Use DNS or LDAP to find mappings for the rest of the local network and the rest of the world. You can also use **/etc/hosts** to specify mappings that you do not want the rest of the world to know about and therefore do not publish in DNS. (You can also use a split DNS configuration to achieve this goal; see this page.)

The **hostname** command assigns a hostname to a machine. **hostname** is typically run at boot time from one of the startup scripts, which obtains the name to be assigned from a configuration file. (Of course, each system does this slightly differently. See the system-specific sections beginning here for details.) The hostname should be fully qualified: that is, it should include both the hostname and the DNS domain name, such as anchor.cs.colorado.edu.

At a small site, you can easily dole out hostnames and IP addresses by hand. But when many networks and many different administrative groups are involved, it helps to have some central coordination to ensure uniqueness. For dynamically assigned networking parameters, DHCP takes care of the uniqueness issues.

Network interface and IP configuration

A network interface is a piece of hardware that can potentially be connected to a network. The actual hardware varies widely. It can be an RJ-45 jack with associated signaling hardware for wired Ethernet, a wireless radio, or even a virtual piece of hardware that connects to a virtual network.

Every system has at least two network interfaces: a virtual loopback interface and at least one real network card or port. On PCs with multiple Ethernet jacks, a separate network interface usually controls each jack. (These interfaces quite often have hardware different from that of each other as well.)

On most systems, you can see all the network interfaces with **ip link show** (Linux) or **ifconfig -a** (FreeBSD), whether or not the interfaces have been configured or are currently running. Here's an example from an Ubuntu system:

```
$ ip link show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state
    UNKNOWN mode DEFAULT group default qlen 1
   link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: enp0s5: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
    pfifo_fast state UP mode DEFAULT group default qlen 1000
   link/ether 00:1c:42:b4:fa:54 brd ff:ff:ff:ff:ff:ff
```

Interface naming conventions vary. Current versions of Linux try to ensure that interface names don't change over time, so the names are somewhat arbitrary (e.g., enp0s5). FreeBSD and older Linux kernels use a more traditional driver + instance number scheme, resulting in names like em0 or eth0.

Network hardware often has configurable options that are specific to its media type and have little to do with TCP/IP per se. One common example of this is modern-day Ethernet, wherein an interface card might support 10, 100, 1000, or even 10000 Mb/s in either half-duplex or full-duplex mode. Most equipment defaults to autonegotiation mode, in which both the card and its upstream connection (usually a switch port) try to guess what the other wants to use.

Historically, autonegotiation worked about as well as a blindfolded cowpoke trying to rope a calf. Modern network devices play better together, but autonegotiation is still a possible source of failure. High packet loss rates (especially for large packets) are a common artifact of failed autonegotiation.

If you manually configure a network link, turn off autonegotiation on both sides. It makes intuitive sense that you might be able to manually configure one side of the link and then let the other side automatically adapt to those settings. But alas, that is not how Ethernet autoconfiguration actually works. All participants must agree that the network is either automatically or manually configured.

The exact method by which hardware options like autonegotiation are set varies widely, so we defer discussion of those details to the system-specific sections starting here.

Above the level of interface hardware, every network protocol has its own configuration. IPv4 and IPv6 are the only protocols you might normally want to configure, but it's important to understand that configurations are defined per interface/protocol pair. In particular, IPv4 and IPv6 are completely separate worlds, each of which has its own configuration.

IP configuration is largely a matter of setting an IP address for the interface. IPv4 also needs to know the subnet mask ("netmask") for the attached network so that it can distinguish the network and host portions of addresses. At the level of network traffic in and out of an interface, IPv6 does not use netmasks; the network and host portions of an IPv6 address are of fixed size.

In IPv4, you can set the broadcast address to any IP address that's valid for the network to which the host is attached. Some sites have chosen weird values for the broadcast address in the hope of avoiding certain types of denial-of-service attacks that use broadcast pings, but this is risky and probably overkill. Failure to properly configure every machine's broadcast address can lead to broadcast storms, in which packets travel from machine to machine until their TTLs expire.

A better way to avoid problems with broadcast pings is to prevent your border routers from forwarding them and to tell individual hosts not to respond to them. IPv6 no longer has broadcasting at all; it's been replaced with various forms of multicasting.

You can assign more than one IP address to an interface. In the past it was sometimes helpful to do this to allow one machine to host several web sites; however, the need for this feature has been superseded by the HTTP Host header and the SNI feature of TLS. See this page for details.

Routing configuration

This book's discussion of routing is divided among several sections in this chapter and Chapter 15, *IP Routing*. Although most of the basic information about routing is found here and in the sections about the **ip route** (Linux) and **route** (FreeBSD) commands, you might find it helpful to read the first few sections of Chapter 15 if you need more information.

Routing is performed at the IP layer. When a packet bound for some other host arrives, the packet's destination IP address is compared with the routes in the kernel's routing table. If the address matches a route in the table, the packet is forwarded to the next-hop gateway IP address associated with that route.

There are two special cases. First, a packet may be destined for some host on a directly connected network. In this case, the "next-hop gateway" address in the routing table is one of the local host's own interfaces, and the packet is sent directly to its destination. This type of route is added to the routing table for you by the **ifconfig** or **ip address** command when you configure a network interface.

Second, it may be that no route matches the destination address. In this case, the default route is invoked if one exists. Otherwise, an ICMP "network unreachable" or "host unreachable" message is returned to the sender.

Many local area networks have only one way out, so all they need is a single default route that points to the exit. On the Internet backbone, the routers do not have default routes. If there is no routing entry for a destination, that destination cannot be reached.

Each **ip route** (Linux) or **route** (FreeBSD) command adds or removes one route. Here are two prototypical commands:

```
linux# ip route add 192.168.45.128/25 via zulu-gw.atrust.net
freebsd# route add -net 192.168.45.128/25 zulu-gw.atrust.net
```

These commands add a route to the 192.168.45.128/25 network through the gateway router zulu-gw.atrust.net, which must be either an adjacent host or one of the local host's own interfaces. Naturally, the hostname zulu-gw.atrust.net must be resolvable to an IP address. Use a numeric IP address if your DNS server is on the other side of the gateway!

Destination networks were traditionally specified with separate IP addresses and netmasks, but all routing-related commands now understand CIDR notation (e.g., 128.138.176.0/20). CIDR notation is clearer and relieves you of the need to fuss over some of the system-specific syntax issues.

Some other tricks:

- To inspect existing routes, use the command **netstat -nr**, or **netstat -r** if you want to see names instead of numbers. Numbers are often better if you are debugging, since

the name lookup may be the thing that is broken. An example of **netstat** output is shown here. On Linux, **ip route show** is the officially blessed command for seeing routes. However, we find its output less clear than **netstat**'s.

- Use the keyword **default** instead of an address or network name to set the system's default route. This mnemonic is identical to 0.0.0.0/0, which matches any address and is less specific than any real routing destination.
- Use **ip route del** (Linux) or **route del** (FreeBSD) to remove entries from the routing table.
- Run **ip route flush** (Linux) or **route flush** (FreeBSD) to initialize the routing table and start over.
- IPv6 routes are set up similarly to IPv4 routes; include the **-6** option to **route** to tell it that you're setting an IPv6 routing entry. **ip route** can normally recognize IPv6 routes on its own (by inspecting the format of the addresses), but it accepts the **-6** argument, too.
- **/etc/networks** maps names to network numbers, much like the **hosts** file maps hostnames to IP addresses. Commands such as **ip** and **route** that expect a network number can accept a name if it is listed in the **networks** file. Network names can also be listed in DNS; see RFC1101.

DNS configuration

To configure a machine as a DNS client, you need only set up the **/etc/resolv.conf** file. DNS service is not strictly required, but it's hard to imagine a situation in which you'd want to eliminate it completely.

The **resolv.conf** file lists the DNS domains that should be searched to resolve names that are incomplete (that is, not fully qualified, such as anchor instead of anchor.cs.colorado.edu) and the IP addresses of the name servers to contact for name lookups. A sample is shown below; for more details, see this page.

```
search cs.colorado.edu colorado.edu
nameserver 128.138.242.1
nameserver 128.138.243.151
nameserver 192.108.21.1
```

/etc/resolv.conf should list the "closest" stable name server first. Servers are contacted in order, and the timeout after which the next server in line is tried can be quite long. You can have up to three nameserver entries. If possible, you should always have more than one.

If the local host obtains the addresses of its DNS servers through DHCP, the DHCP client software stuffs these addresses into the **resolv.conf** file for you when it obtains the leases. Since DHCP configuration is the default for most systems, you generally need not configure the **resolv.conf** file manually if your DHCP server has been set up correctly.

Many sites use Microsoft's Active Directory DNS server implementation. That works fine with the standard UNIX and Linux **resolv.conf**; there's no need to do anything differently.

System-specific network configuration

On early UNIX systems, you configured the network by editing the system startup scripts and directly changing the commands they contained. Modern systems have read-only scripts; they cover a variety of configuration scenarios and choose among them by reusing information from other system files or consulting configuration files of their own.

Although this separation of configuration and implementation is a good idea, every system does it a little bit differently. The format and use of the **/etc/hosts** and **/etc/resolv.conf** files are relatively consistent among UNIX and Linux systems, but that's about all you can count on for sure.

Most systems offer some sort of GUI interface for basic configuration tasks, but the mapping between the visual interface and the configuration files behind the scenes is often unclear. In addition, the GUIs tend to ignore advanced configurations, and they are relatively inconvenient for remote and automated administration.

In the next sections, we pick apart some of the variations among our example systems, describe what's going on under the hood, and cover the details of network configuration for each of our supported operating systems. In particular, we cover

- Basic configuration
- DHCP client configuration
- Dynamic reconfiguration and tuning
- Security, firewalls, filtering, and NAT configuration

Keep in mind that most network configuration happens at boot time, so there's some overlap between the information here and the information presented in Chapter 2, *Booting and System Management Daemons*.

13.10 LINUX NETWORKING

Linux developers love to tinker, and they often implement features and algorithms that aren't yet accepted standards. One example is the Linux kernel's addition of pluggable congestion-control algorithms in release 2.6.13. The several options include variations for lossy networks, high-speed WANs with lots of packet loss, satellite links, and more. The standard TCP "reno" mechanism (slow start, congestion avoidance, fast retransmit, and fast recovery) is still used by default. A variant might be more appropriate for your environment (but probably not). See lwn.net/Articles/701165 for some hints on when to consider the use of alternate congestion control algorithms.



After any change to a file that controls network configuration at boot time, you may need either to reboot or to bring the network interface down and then up again for your change to take effect. You can use **ifdown** *interface* and **ifup** *interface* for this purpose on most Linux systems.

NetworkManager

Linux support for mobile networking was relatively scattershot until the advent of NetworkManager in 2004. It consists of a service that's run continuously, along with a system tray app for configuring individual network interfaces. In addition to various kinds of wired network, NetworkManager also handles transient wireless networks, wireless broadband, and VPNs. It continually assesses the available networks and shifts service to "preferred" networks as they become available. Wired networks are most preferred, followed by familiar wireless networks.

This system represented quite a change for Linux network configuration. In addition to being more fluid than the traditional static configuration, it's also designed to be run and managed by users rather than system administrators. NetworkManager has been widely adopted by Linux distributions, including all our examples, but in an effort to avoid breaking existing scripts and setups, it's usually made available as a sort of "parallel universe" of network configuration in addition to whatever traditional network configuration was used in the past.

Debian and Ubuntu run NetworkManager by default, but keep the statically configured network interfaces out of the NetworkManager domain. Red Hat and CentOS don't run NetworkManager by default at all.

NetworkManager is primarily of use on laptops, since their network environment may change frequently. For servers and desktop systems, NetworkManager isn't necessary and may in fact complicate administration. In these environments, it should be ignored or configured out.

ip: manually configure a network

Linux systems formerly used the same basic commands for network configuration and status checking as traditional UNIX: **ifconfig**, **route**, and **netstat**. These are still available on most distributions, but active development has moved on to the **iproute2** package, which features the commands **ip** (for most everyday network configuration, including routing) and **ss** (for examining the state of network sockets, roughly replacing **netstat**).

If you're accustomed to the traditional commands, it's worth the effort to transition your brain to **ip**. The legacy commands won't be around forever, and although they cover the common configuration scenarios, they don't give access to the full feature set of the Linux networking stack. **ip** is cleaner and more regular.

ip takes a second argument for you to specify what kind of object you want to configure or examine. There are many options, but the common ones are **ip link** for configuring network interfaces, **ip address** for binding network addresses to interfaces, and **ip route** for changing or printing the routing table. You can abbreviate **ip** arguments, so **ip ad** is the same as **ip address**. We show full names for clarity.

Most objects understand **list** or **show** to print out a summary of their current status, so **ip link show** prints a list of network interfaces, **ip route show** dumps the current routing table, and **ip address show** lists all assigned IP addresses.

The man pages for **ip** are divided by subcommand. For example, to see detailed information about interface configuration, run **man ip-link**. You can also run **ip link help** to see a short cheat sheet.

The UNIX **ifconfig** command conflates the concept of interface configuration with the concept of configuring the settings for a particular network protocol. In fact, several protocols can run on a given network interface (the prime example being simultaneous IPv4 and IPv6), and each of those protocols can support multiple addresses, so **ip**'s distinction between **ip link** and **ip address** is actually quite smart. Most of what system administrators traditionally think of as "interface configuration" really has to do with setting up IPv4 and IPv6.

ip accepts a **-4** or **-6** argument to target IPv4 or IPv6 explicitly, but it's rarely necessary to specify these options. **ip** guesses the right mode just by looking at the format of the addresses you provide.

Basic configuration of an interface looks like this:

```
# ip address add 192.168.1.13/26 broadcast 192.168.1.63 dev enp0s5
```

In this case the **broadcast** clause is superfluous because that would be the default value anyway, given the netmask. But this is how you would set it if you needed to.

Of course, in daily life you won’t normally be setting up network addresses by hand. The next sections describe how our example distributions handle static configuration of the network from the perspective of configuration files.

Debian and Ubuntu network configuration



As shown in Table 13.6, Debian and Ubuntu configure the network in **/etc/hostname** and **/etc/network/interfaces**, with a bit of help from the file **/etc/network/options**.

Table 13.6: Ubuntu network configuration files in /etc

File	What's set there
hostname	Hostname
network/interfaces	IP address, netmask, default route

The hostname is set in **/etc/hostname**. The name in this file should be fully qualified; its value is used in a variety of contexts, some of which require qualification.

The IP address, netmask, and default gateway are set in **/etc/network/interfaces**. A line starting with the iface keyword introduces each interface. The iface line can be followed by indented lines that specify additional parameters. For example,

```
auto lo enp0s5
iface lo inet loopback
iface enp0s5 inet static
    address 192.168.1.102
    netmask 255.255.255.0
    gateway 192.168.1.254
```

The **ifup** and **ifdown** commands read this file and bring the interfaces up or down by calling lower-level commands (such as **ip**) with the appropriate parameters. The auto clause specifies the interfaces to be brought up at boot time or whenever **ifup -a** is run.

The inet keyword in the iface line is the address family, IPv4. To configure IPv6 as well, include an inet6 configuration.

The keyword static is called a "method" and specifies that the IP address and netmask for enp0s5 are directly assigned. The address and netmask lines are required for static configurations. The gateway line specifies the address of the default network gateway and is used to install a default route.

To configure interfaces with DHCP, just specify that in the interfaces file:

```
iface enp0s5 inet dhcp
```

Red Hat and CentOS network configuration

RHEL  Red Hat and CentOS's network configuration revolves around **/etc/sysconfig**. Table 13.7 shows the various configuration files.

Table 13.7: Red Hat network configuration files in /etc/sysconfig

File	What's set there
network	Hostname, default route
network-scripts/ifcfg-*ifname*	Per-interface parameters: IP address, netmask, etc.
network-scripts/route-*ifname*	Per-interface routing: arguments to **ip route**

You set the machine's hostname in **/etc/sysconfig/network**, which also contains lines that specify the machine's DNS domain and default gateway. (Essentially, this file is where you specify all interface-independent network settings.)

For example, here is a **network** file for a host with a single Ethernet interface:

```
NETWORKING=yes
NETWORKING_IPV6=no
HOSTNAME=redhat.toadranch.com
DOMAINNAME=toadranch.com        ### optional
GATEWAY=192.168.1.254
```

Interface-specific data is stored in **/etc/sysconfig/network-scripts/ifcfg-***ifname*, where *ifname* is the name of the network interface. These configuration files set the IP address, netmask, network, and broadcast address for each interface. They also include a line that specifies whether the interface should be configured "up" at boot time.

A generic machine has files for an Ethernet interface (eth0) and for the loopback interface (lo). For example,

```
DEVICE=eth0
IPADDR=192.168.1.13
NETMASK=255.255.255.0
NETWORK=192.168.1.0
BROADCAST=192.168.1.255
MTU=1500
ONBOOT=yes
```

and

```
DEVICE=lo
IPADDR=127.0.0.1
NETMASK=255.0.0.0
NETWORK=127.0.0.0
BROADCAST=127.255.255.255
ONBOOT=yes
NAME=loopback
```

are the **ifcfg-eth0** and **ifcfg-lo** files for the machine redhat.toadranch.com described in the **network** file above.

A DHCP-based setup for eth0 is even simpler:

```
DEVICE=eth0
BOOTPROTO=dhcp
ONBOOT=yes
```

After changing configuration information in /**etc/sysconfig**, run **ifdown** *ifname* followed by **ifup** *ifname* for the appropriate interface. If you reconfigure multiple interfaces at once, you can use the command **sysctl restart network** to reset networking.

Lines in **network-scripts/route-***ifname* are passed as arguments to **ip route** when the corresponding interface is configured. For example, the line

```
default via 192.168.0.1
```

sets a default route. This isn't really an interface-specific configuration, but for clarity, it should go in the file that corresponds to the interface on which you expect default-routed packets to be transmitted.

Linux network hardware options

The **ethtool** command queries and sets a network interface's media-specific parameters such as link speed and duplex. It replaces the old **mii-tool** command, but some systems still include both. If **ethtool** is not installed by default, it's usually included in an optional package of its own (also called **ethtool**).

You can query the status of an interface just by naming it. For example, this eth0 interface (a generic NIC on a PC motherboard) has autonegotiation enabled and is currently running at full speed:

```
# ethtool eth0
Settings for eth0:
        Supported ports: [ TP MII ]
        Supported link modes:   10baseT/Half    10baseT/Full
                                100baseT/Half   100baseT/Full
                                1000baseT/Half  1000baseT/Full
        Supports auto-negotiation: Yes
        Advertised link modes:  10baseT/Half    10baseT/Full
                                100baseT/Half   100baseT/Full
                                1000baseT/Half  1000baseT/Full
        Advertised auto-negotiation: Yes
        Speed: 1000Mb/s
        Duplex: Full
        Port: MII
        PHYAD: 0
        Transceiver: internal
        Auto-negotiation: on
        Supports Wake-on: pumbg
        Wake-on: g
        Current message level: 0x00000033 (51)
        Link detected: yes
```

To lock this interface to 100 Mb/s full duplex, use the command

```
# ethtool -s eth0 speed 100 duplex full
```

If you are trying to determine whether autonegotiation is reliable in your environment, you may also find **ethtool -r** helpful. It forces the parameters of the link to be renegotiated immediately.

Another useful option is **-k**, which shows what protocol-related tasks are assigned to the network interface rather than being performed by the kernel. Most interfaces can calculate checksums, and some can assist with segmentation as well. Unless you believe that a network interface is not doing these tasks reliably, it's always better to offload them. You can use **ethtool -K** in combination with various suboptions to force or disable specific types of offloading. (The **-k** option shows current values and the **-K** option sets them.)

Any changes you make with **ethtool** are transient. If you want them to be enforced consistently, make sure that **ethtool** gets run as part of the system's network configuration. It's best to do this as part of the per-interface configuration; if you just arrange to have some **ethtool** commands run at boot time, your configuration will not properly cover cases in which the interfaces are restarted without a reboot of the system.

RHEL  On Red Hat and CentOS systems, you can include an ETHTOOL_OPTS= line in the configuration file for the interface underneath **/etc/sysconfig/network-scripts**. The **ifup** command passes the entire line as arguments to **ethtool**.

  In Debian and Ubuntu, you can run **ethtool** commands directly from the configuration for a particular network in **/etc/network/interfaces**.

Linux TCP/IP options

Linux puts a representation of each tunable kernel variable into the **/proc** virtual filesystem. See *Tuning Linux kernel parameters* for general information about the **/proc** mechanism.

The networking variables are under **/proc/sys/net/ipv4** and **/proc/sys/net/ipv6**. We formerly showed a complete list here, but there are too many to list these days.

The **ipv4** directory includes a lot more parameters than does the **ipv6** directory, but that's mostly because IP-version-independent protocols such as TCP and UDP confine their parameters to the **ipv4** directory. A prefix such as **tcp_** or **udp_** tells you which protocol the parameter relates to.

The **conf** subdirectories within **ipv4** and **ipv6** contain parameters that are set per interface. They include subdirectories **all** and **default** and a subdirectory for each interface (including the loopback). Each subdirectory has the same set of files.

```
ubuntu$ ls -F /proc/sys/net/ipv4/conf/default
accept_local         drop_gratuitous_arp                 proxy_arp
accept_redirects     drop_unicast_in_l2_multicast        proxy_arp_pvlan
accept_source_route  force_igmp_version                  route_localnet
arp_accept           forwarding                          rp_filter
arp_announce         igmpv2_unsolicited_report_interval  secure_redirects
arp_filter           igmpv3_unsolicited_report_interval  send_redirects
arp_ignore           ignore_routes_with_linkdown         shared_media
arp_notify           log_martians                        src_valid_mark
bootp_relay          mc_forwarding                       tag
disable_policy       medium_id
disable_xfrm         promote_secondaries
```

If you change a variable in the **conf/enp0s5** subdirectory, for example, your change applies to that interface only. If you change the value in the **conf/all** directory, you might expect it to set the corresponding value for all existing interfaces, but this is not what happens. Each variable has its own rules for accepting changes via **all**. Some values are ORed with the current values, some are ANDed, and still others are MAXed or MINed. As far as we are aware, there is no documentation for this process except in the kernel source code, so the whole debacle is probably best avoided. Just confine your modifications to individual interfaces.

If you change a variable in the **conf/default** directory, the new value propagates to any interfaces that are later configured. On the other hand, it's nice to keep the defaults unmolested as reference information; they make a nice sanity check if you want to undo other changes.

The **/proc/sys/net/ipv4/neigh** and **/proc/sys/net/ipv6/neigh** directories also contain a subdirectory for each interface. The files in each subdirectory control ARP table management and IPv6 neighbor discovery for that interface. Here is the list of variables; the ones starting with gc (for garbage collection) determine how ARP table entries are timed out and discarded.

```
ubuntu$ ls -F /proc/sys/net/ipv4/neigh/default
anycast_delay           gc_interval    locktime         retrans_time
app_solicit             gc_stale_time  mcast_resolicit  retrans_time_ms
base_reachable_time     gc_thresh1     mcast_solicit    ucast_solicit
base_reachable_time_ms  gc_thresh2     proxy_delay      unres_qlen
delay_first_probe_time  gc_thresh3     proxy_qlen       unres_qlen_bytes
```

To see the value of a variable, use **cat**. To set it, you can use **echo** redirected to the proper filename, but the **sysctl** command (which is just a command interface to the same variables) is often easier.

For example, the command

```
ubuntu$ cat /proc/sys/net/ipv4/icmp_echo_ignore_broadcasts
0
```

shows that this variable's value is 0, meaning that broadcast pings are not ignored. To set it to 1 (and avoid falling prey to Smurf-type denial of service attacks), run

```
ubuntu$ sudo sh -c "echo 1 > icmp_echo_ignore_broadcasts"
```

from the **/proc/sys/net** directory. (If you try this command in the form **sudo echo 1 > icmp_echo_ignore_broadcasts**, you just generate a "permission denied" message—your shell attempts to open the output file before it runs **sudo**. You want the **sudo** to apply to both the **echo** command and the redirection. Ergo, you must create a root subshell in which to execute the entire command.)

You can also use the **sysctl** command to achieve the same configuration:

```
ubuntu$ sysctl net.ipv4.icmp_echo_ignore_broadcasts=1
```

sysctl variable names are pathnames relative to **/proc/sys**. Dots are the traditional separator, but **sysctl** also accepts slashes if you prefer them.

You are typically logged in over the same network you are tweaking as you adjust these variables, so be careful! You can mess things up badly enough to require a reboot from the console to recover, which might be inconvenient if the system happens to be in Point Barrow, Alaska, and it's January. Test-tune these variables on your desktop system before you even think of tweaking a production machine.

To change any of these parameters permanently (or more accurately, to reset them every time the system boots), add the appropriate variables to **/etc/sysctl.conf**, which is read by the **sysctl** command at boot time. For example, the line

```
net.ipv4.ip_forward=0
```

in the **/etc/sysctl.conf** file turns off IP forwarding on this host.

Some of the options under **/proc** are better documented than others. Your best bet is to look at the man page for the protocol in question in section 7 of the manuals. For example, **man 7 icmp** documents six of the eight available options. (You must have man pages for the Linux kernel installed to see man pages about protocols.)

You can also look at the **ip-sysctl.txt** file in the kernel source distribution for some good comments. If you don't have kernel source installed, just Google for ip-sysctl-txt to reach the same document.

Security-related kernel variables

Table 13.8 shows Linux's default behavior with regard to various touchy network issues. For a brief description of the implications of these behaviors, see this page. We recommend that you verify the values of these variables so that you do not answer broadcast pings, do not listen to routing redirects, and do not accept source-routed packets. These should be the defaults on current distributions except for **accept_redirects**.

Table 13.8: Default security-related network behaviors in Linux

Feature	Host	Gateway	Control file (in /proc/sys/net/ipv4)
IP forwarding	off	on	**ip_forward** for the whole system **conf**/*interface*/**forwarding** per interface[a]
ICMP redirects	obeys	ignores	**conf**/*interface*/**accept_redirects**
Source routing	off	off	**conf**/*interface*/**accept_source_route**
Broadcast ping	ignores	ignores	**icmp_echo_ignore_broadcasts**

a. The *interface* can be either a specific interface name or **all**.

13.11 FreeBSD networking



As a direct descendant of the BSD lineage, FreeBSD remains something of a reference implementation of TCP/IP. It lacks many of the elaborations that complicate the Linux networking stack. From the standpoint of system administrators, FreeBSD network configuration is simple and direct.

ifconfig: configure network interfaces

ifconfig enables or disables a network interface, sets its IP address and subnet mask, and sets various other options and parameters. It is usually run at boot time with command-line parameters taken from config files, but you can also run it by hand to make changes on the fly. Be careful if you are making **ifconfig** changes and are logged in remotely—many a sysadmin has been locked out this way and had to drive in to fix things.

An **ifconfig** command most commonly has the form

```
ifconfig interface [family] address options ...
```

For example, the command

```
ifconfig em0 192.168.1.13/26 up
```

sets the IPv4 address and netmask associated with the interface em0 and readies the interface for use.

interface identifies the hardware interface to which the command applies. The loopback interface is named lo0. The names of real interfaces vary according to their hardware drivers. **ifconfig -a** lists the system's network interfaces and summarizes their current settings.

The *family* parameter tells **ifconfig** which network protocol ("address family") you want to configure. You can set up multiple protocols on an interface and use them all simultaneously, but they must be configured separately. The main options here are **inet** for IPv4 and **inet6** for IPv6; **inet** is assumed if you omit the parameter.

The *address* parameter specifies the interface's IP address. A hostname is also acceptable here, but the hostname must be resolvable to an IP address at boot time. For a machine's primary interface, this means that the hostname must appear in the local **hosts** file, since other name resolution methods depend on the network having been initialized.

The keyword **up** turns the interface on; **down** turns it off. When an **ifconfig** command assigns an IP address to an interface, as in the example above, the **up** parameter is implicit and need not be mentioned by name.

For subnetted networks, you can specify a CIDR-style netmask as shown in the example above, or you can include a separate **netmask** argument. The mask can be specified in dotted decimal notation or as a 4-byte hexadecimal number beginning with **0x**.

The **broadcast** option specifies the IP broadcast address for the interface, expressed in either hex or dotted quad notation. The default broadcast address is one in which the host part is set to all 1s. In the **ifconfig** example above, the autoconfigured broadcast address is 192.168.1.61.

FreeBSD network hardware configuration

FreeBSD does not have a dedicated command analogous to Linux's **ethertool**. Instead, **ifconfig** passes configuration information down to the network interface driver through the **media** and **mediaopt** clauses. The legal values for these options vary with the hardware. To find the list, read the man page for the specific driver.

For example, an interface named em0 uses the "em" driver. **man 4 em** shows that this is the driver for certain types of Intel-based wired Ethernet hardware. To force this interface to gigabit mode using four-pair cabling (the typical configuration), the command would be

```
# ifconfig em0 media 1000baseT mediaopt full-duplex
```

You can include these media options along with other configuration clauses for the interface.

FreeBSD boot-time network configuration

FreeBSD's static configuration system is mercifully simple. All the network parameters live in **/etc/rc.conf**, along with other system-wide settings. Here's a typical configuration:

```
hostname="freebeer"
ifconfig_em0="inet 192.168.0.48 netmask 255.255.255.0"
defaultrouter="192.168.0.1"
```

Each network interface has its own ifconfig_* variable. The value of the variable is simply passed to **ifconfig** as a series of command-line arguments. The defaultrouter clause identifies a default network gateway.

To obtain the system's networking configuration from a DHCP server, use the following token:

```
ifconfig_em0="DHCP"
```

This form is magic and is not passed on to **ifconfig**, which wouldn't know how to interpret a **DHCP** argument. Instead, it makes the startup scripts run the command **dhclient em0**. To modify the operational parameters of the DHCP system (timeouts and such), set them in **/etc/dhclient.conf**. The default version of this file is empty except for comments, and you shouldn't normally need to modify it.

If you modify the network configuration, you can run **service netif restart** to repeat the initial configuration procedure. If you changed the defaultrouter parameter, also run **service routing restart**.

FreeBSD TCP/IP configuration

FreeBSD's kernel-level networking options are controlled similarly to those of Linux (see this page), except that there's no **/proc** hierarchy you can go rooting around in. Instead, run **sysctl -ad** to list the available parameters and their one-line descriptions. There are a lot of them (5,495 on FreeBSD 11), so you need to grep for likely suspects such as "redirect" or "^net".

Table 13.9 lists a selection of security-related parameters.

Table 13.9: Default security-related network parameters in FreeBSD

Parameter	Dfl	What it does when set to 1
net.inet.ip.forwarding	0	Acts as a router for IPv4 packets
net.inet6.ip6.forwarding	0	Acts as a router for IPv6 packets
net.inet.tcp.blackhole	0	Disables "unreachable" messages for closed ports
net.inet.udp.blackhole	0	Does not send RST packets for closed TCP ports
net.inet.icmp.drop_redirect	0	Ignores IPv4 ICMP redirects
net.inet6.icmp6.rediraccept	1	Accepts (obeys) IPv6 ICMP redirects
net.inet.ip.accept_sourceroute	0	Allows source-routed IPv4 packets

The blackhole options are potentially useful on systems that you want to shield from port scanners, but they do change the standard behaviors of UDP and TCP. You might also want to disable acceptance of ICMP redirects for both IPv4 and IPv6.

You can set parameters in the running kernel with **sysctl**. For example,

```
$ sudo sysctl net.inet.icmp.drop_redirect=1
```

To have the parameter set at boot time, list it in **/etc/sysctl.conf**.

```
net.inet.icmp.drop_redirect=1
```

13.12 NETWORK TROUBLESHOOTING

Several good tools are available for debugging a network at the TCP/IP layer. Most give low-level information, so you must understand the main ideas of TCP/IP and routing to use them.

In this section, we start with some general troubleshooting strategy. We then cover several essential tools, including **ping**, **traceroute**, **tcpdump**, and Wireshark. We don't discuss the **arp**, **ndp**, **ss**, or **netstat** commands in this chapter, though they, too, are useful debugging tools.

Before you attack your network, consider these principles:

- Make one change at a time. Test each change to make sure that it had the effect you intended. Back out any changes that have an undesired effect.
- Document the situation as it was before you got involved, and document every change you make along the way.
- Start at one end of a system or network and work through the system's critical components until you reach the problem. For example, you might start by looking at the network configuration on a client, work your way up to the physical connections, investigate the network hardware, and finally, check the server's physical connections and software configuration.
- Or, use the layers of the network to negotiate the problem. Start at the "top" or "bottom" and work your way through the protocol stack.

This last point deserves a bit more discussion. As described here, the architecture of TCP/IP defines several layers of abstraction at which components of the network can function. For example, HTTP depends on TCP, TCP depends on IP, IP depends on the Ethernet protocol, and the Ethernet protocol depends on the integrity of the network cable. You can dramatically reduce the amount of time spent debugging a problem if you first figure out which layer is misbehaving.

Ask yourself questions like these as you work up or down the stack:

- Do you have physical connectivity and a link light?
- Is your interface configured properly?
- Do your ARP tables show other hosts?
- Is there a firewall on your local machine?
- Is there a firewall anywhere between you and the destination?
- If firewalls are involved, do they pass ICMP ping packets and responses?

- Can you ping the localhost address (127.0.0.1)?
- Can you ping other local hosts by IP address?
- Is DNS working properly?
- Can you ping other local hosts by hostname?
- Can you ping hosts on another network?
- Do high-level services such as web and SSH servers work?
- Did you really check the firewalls?

If a machine hangs at boot time, boots very slowly, or hangs on inbound SSH connections, DNS should be a prime suspect. Most systems use an approach to name resolution that's configurable in **/etc/nsswitch.conf**. If the system runs **nscd**, the name service caching daemon, that component deserves some suspicion as well. If **nscd** crashes or is misconfigured, name lookups are affected. Use the **getent** command to check whether your resolver and name servers are working properly (e.g., **getent hosts google.com**).

Once you've identified where the problem lies and have a fix in mind, step back to consider the effect that your subsequent tests and prospective fixes will have on other services and hosts.

ping: check to see if a host is alive

The **ping** command and its IPv6 twin **ping6** are embarrassingly simple, but in many situations they are the only commands you need for network debugging. They send an ICMP ECHO_REQUEST packet to a target host and wait to see if the host answers back.

You can use **ping** to check the status of individual hosts and to test segments of the network. Routing tables, physical networks, and gateways are all involved in processing a ping, so the network must be more or less working for **ping** to succeed. If **ping** doesn't work, you can be pretty sure that nothing more sophisticated will work either.

However, this rule does not apply to networks or hosts that block ICMP echo requests with a firewall. (Recent versions of Windows block ping requests by default.) Make sure that a firewall isn't interfering with your debugging before you conclude that the target host is ignoring a ping. You might consider disabling a meddlesome firewall for a short period of time to facilitate debugging.

If your network is in bad shape, chances are that DNS is not working. Simplify the situation by using numeric IP addresses when pinging, and use **ping**'s **-n** option to prevent **ping** from attempting to do reverse lookups on IP addresses—these lookups also trigger DNS requests.

Be aware of the firewall issue if you're using **ping** to check your Internet connectivity, too. Some well-known sites answer **ping** packets and others don't. We've found google.com to be a consistent responder.

Most versions of **ping** run in an infinite loop unless you supply a packet count argument. Once you've had your fill of pinging, type the interrupt character (usually <Control-C>) to get out.

Here's an example:

```
linux$ ping beast
PING beast (10.1.1.46): 56 bytes of data.
64 bytes from beast (10.1.1.46): icmp_seq=0 ttl=54 time=48.3ms
64 bytes from beast (10.1.1.46): icmp_seq=1 ttl=54 time=46.4ms
64 bytes from beast (10.1.1.46): icmp_seq=2 ttl=54 time=88.7ms
^C
--- beast ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2026ms
rtt min/avg/max/mdev = 46.490/61.202/88.731/19.481 ms
```

The output for beast shows the host's IP address, the ICMP sequence number of each response packet, and the round trip travel time. The most obvious thing that the output above tells you is that the server beast is alive and connected to the network.

The ICMP sequence number is a particularly valuable piece of information. Discontinuities in the sequence indicate dropped packets. They're normally accompanied by a message for each

missing packet.

Despite the fact that IP does not guarantee the delivery of packets, a healthy network should drop very few of them. Lost-packet problems are important to track down because they tend to be masked by higher-level protocols. The network may appear to function correctly, but it will be slower than it ought to be, not only because of the retransmitted packets but also because of the protocol overhead needed to detect and manage them.

To track down the cause of disappearing packets, first run **traceroute** (see the next section) to discover the route that packets are taking to the target host. Then ping the intermediate gateways in sequence to discover which link is dropping packets. To pin down the problem, you need to send a fair number of packets. The fault generally lies on the link between the last gateway you can ping without loss of packets and the gateway beyond that.

The round trip time reported by **ping** can afford insight into the overall performance of a path through a network. Moderate variations in round trip time do not usually indicate problems. Packets may occasionally be delayed by tens or hundreds of milliseconds for no apparent reason; that's just the way IP works. You should see a fairly consistent round trip time for the majority of packets, with occasional lapses. Many of today's routers implement rate-limited or low-priority responses to ICMP packets, which means that a router may defer responding to your ping if it is already dealing with a lot of other traffic.

The **ping** program can send echo request packets of any size, so by using a packet larger than the MTU of the network (1,500 bytes for Ethernet), you can force fragmentation. This practice helps you identify media errors or other low-level issues such as problems with a congested network or VPN. You specify the desired packet size in bytes with the **-s** flag:

```
$ ping -s 1500 cuinfo.cornell.edu
```

Note that even a simple command like **ping** can have dramatic effects. In 1998, the so-called Ping of Death attack crashed large numbers of UNIX and Windows systems. It was launched simply by transmission of an overly large ping packet. When the fragmented packet was reassembled, it filled the receiver's memory buffer and crashed the machine. The Ping of Death issue has long since been fixed, but keep in mind several other caveats regarding **ping**.

First, it's hard to distinguish the failure of a network from the failure of a server with only the **ping** command. In an environment where ping tests normally work, a failed ping just tells you that *something* is wrong.

It's also worth noting that a successful ping does not guarantee much about the target machine's state. Echo request packets are handled within the IP protocol stack and do not require a server process to be running on the probed host. A response guarantees only that a machine is powered on and has not experienced a kernel panic. You'll need higher-level methods to verify the availability of individual services such as HTTP and DNS.

traceroute: trace IP packets

traceroute, originally written by Van Jacobson, uncovers the sequence of gateways through which an IP packet travels to reach its destination. All modern operating systems come with some version of **traceroute**. Even Windows has it, but the command is spelled **tracert** (extra history points if you can guess why).

The syntax is simply

```
traceroute hostname
```

Most of the variety of options are not important in daily use. As usual, the *hostname* can be specified as either a DNS name or an IP address. The output is simply a list of hosts, starting with the first gateway and ending at the destination. For example, a **traceroute** from our host jaguar to our host nubark produces the following output:

```
$ traceroute nubark
traceroute to nubark (192.168.2.10), 30 hops max, 38 byte packets
 1  lab-gw (172.16.8.254)  0.840 ms  0.693 ms  0.671 ms
 2  dmz-gw (192.168.1.254)  4.642 ms  4.582 ms  4.674 ms
 3  nubark (192.168.2.10)  7.959 ms  5.949 ms  5.908 ms
```

From this output we can tell that jaguar is three hops away from nubark, and we can see which gateways are involved in the connection. The round trip time for each gateway is also shown—three samples for each hop are measured and displayed. A typical **traceroute** between Internet hosts often includes more than 15 hops, even if the two sites are just across town.

traceroute works by setting the time-to-live field (TTL, actually "hop count to live") of an outbound packet to an artificially low number. As packets arrive at a gateway, their TTL is decreased. When a gateway decreases the TTL to 0, it discards the packet and sends an ICMP "time exceeded" message back to the originating host.

See this page for more information about reverse DNS lookups.

The first three **traceroute** packets above have their TTL set to 1. The first gateway to see such a packet (lab-gw in this case) determines that the TTL has been exceeded and notifies jaguar of the dropped packet by sending back an ICMP message. The sender's IP address in the header of the error packet identifies the gateway, and **traceroute** looks up this address in DNS to find the gateway's hostname.

To identify the second-hop gateway, **traceroute** sends out a second round of packets with TTL fields set to 2. The first gateway routes the packets and decreases their TTL by 1. At the second gateway, the packets are then dropped and ICMP error messages are generated as before. This

process continues until the TTL is equal to the number of hops to the destination host and the packets reach their destination successfully.

Most routers send their ICMP messages from the interface “closest” to the destination. If you run **traceroute** backward from the destination host, you may see different IP addresses being used to identify the same set of routers. You might also discover that packets flowing in the reverse direction take a completely different path, a configuration known as asymmetric routing.

Since **traceroute** sends three packets for each value of the TTL field, you may sometimes observe an interesting artifact. If an intervening gateway multiplexes traffic across several routes, the packets might be returned by different hosts; in this case, **traceroute** simply prints them all.

Let’s look at a more interesting example from a host in Switzerland to caida.org at the San Diego Supercomputer Center:

```
linux$ traceroute caida.org
traceroute to caida.org (192.172.226.78), 30 hops max, 46 byte packets
 1  gw-oetiker.init7.net (213.144.138.193)  1.122 ms  0.182 ms  0.170 ms
 2  r1zur1.core.init7.net (77.109.128.209)  0.527 ms  0.204 ms  0.202 ms
 3  r1fra1.core.init7.net (77.109.128.250)  18.27 ms  6.99 ms  16.59 ms
 4  r1ams1.core.init7.net (77.109.128.154)  19.54 ms  21.85 ms  13.51 ms
 5  r1lon1.core.init7.net (77.109.128.150)  19.16 ms  21.15 ms  24.86 ms
 6  r1lax1.ce.init7.net (82.197.168.69)  158.23 ms  158.22 ms  158.27 ms
 7  cenic.laap.net (198.32.146.32)  158.34 ms  158.30 ms  158.24 ms
 8  dc-lax-core2-ge.cenic.net (137.164.46.119) 158.60 ms * 158.71 ms
 9  dc-tus-agg1-core2-10ge.cenic.net (137.164.46.7) 159 ms 159 ms 159 ms
10  dc-sdsc2-tus-dc-ge.cenic.net (137.164.24.174) 161 ms 161 ms 161 ms
11  pinot.sdsc.edu (198.17.46.56)  161.559 ms  161.381 ms  161.439 ms
12  rommie.caida.org (192.172.226.78)  161.442 ms  161.445 ms  161.532 ms
```

This output shows that packets travel inside Init Seven’s network for a long time. Sometimes we can guess the location of the gateways from their names. Init Seven’s core stretches all the way from Zurich (zur) to Frankfurt (fra), Amsterdam (ams), London (lon), and finally, Los Angeles (lax). Here, the traffic transfers to cenic.net, which delivers the packets to the caida.org host within the network of the San Diego Supercomputer Center (sdsc) in La Jolla, CA.

At hop 8, we see a star in place of one of the round trip times. This notation means that no response (error packet) was received in response to the probe. In this case, the cause is probably congestion, but that is not the only possibility. **traceroute** relies on low-priority ICMP packets, which many routers are smart enough to drop in preference to “real” traffic. A few stars shouldn’t send you into a panic.

If you see stars in all the time fields for a given gateway, no “time exceeded” messages came back from that machine. Perhaps the gateway is simply down. Sometimes, a gateway or firewall is configured to silently discard packets with expired TTLs. In this case, you can still see through the silent host to the gateways beyond. Another possibility is that the gateway’s error packets are slow to return and that **traceroute** has stopped waiting for them by the time they arrive.

Some firewalls block ICMP “time exceeded” messages entirely. If such a firewall lies along the path, you won’t get information about any of the gateways beyond it. However, you can still determine the total number of hops to the destination because the probe packets eventually get all the way there.

Also, some firewalls may block the outbound UDP datagrams that **traceroute** sends to trigger the ICMP responses. This problem causes **traceroute** to report no useful information at all. If you find that your own firewall is preventing you from running **traceroute**, make sure the firewall has been configured to pass UDP ports 33434–33534 as well as ICMP ECHO (type 8) packets.

A slow link does not necessarily indicate a malfunction. Some physical networks have a naturally high latency; UMTS/EDGE/GPRS wireless networks are a good example. Sluggishness can also be a sign of high load on the receiving network. Inconsistent round trip times would support such a hypothesis.

You may occasionally see the notation !N instead of a star or round trip time. The notation indicates that the current gateway sent back a “network unreachable” error, meaning that it doesn’t know how to route your packet. Other possibilities include !H for “host unreachable” and !P for “protocol unreachable.” A gateway that returns any of these error messages is usually the last hop you can get to. That host often has a routing problem (possibly caused by a broken network link): either its static routes are wrong or dynamic protocols have failed to propagate a usable route to the destination.

If **traceroute** doesn’t seem to be working for you or is working slowly, it may be timing out while trying to resolve the hostnames of gateways through DNS. If DNS is broken on the host you are tracing from, use **traceroute -n** to request numeric output. This option disables hostname lookups; it may be the only way to get **traceroute** to function on a crippled network.

traceroute needs root privileges to operate. To be available to normal users, it must be installed setuid root. Several Linux distributions include the **traceroute** command but turn off the setuid bit. Depending on your environment and needs, you can either turn the setuid bit back on or give interested users access to the command through **sudo**.

Recent years have seen the introduction of several new **traceroute**-like utilities that can bypass ICMP-blocking firewalls. See the PERTKB Wiki for an overview of these tools at goo.gl/fXpMeu. We especially like **mtr**, which has a **top**-like interface and shows a sort of live **traceroute**. Neat!

When debugging routing issues, look at your site from the perspective of the outside world. Several web-based route tracing services let you do this sort of inverse **traceroute** right from a browser window. Thomas Kernen maintains a list of these services at traceroute.org.

Packet sniffers

tcpdump and Wireshark belong to a class of tools known as packet sniffers. They listen to network traffic and record or print packets that meet criteria of your choice. For example, you can inspect all packets sent to or from a particular host, or TCP packets related to one particular network connection.

Packet sniffers are useful both for solving problems that you know about and for discovering entirely new problems. It's a good idea to take an occasional sniff of your network to make sure the traffic is in order.

Packet sniffers need to be able to intercept traffic that the local machine would not normally receive (or at least, pay attention to), so the underlying network hardware must allow access to every packet. Broadcast technologies such as Ethernet work fine, as do most other modern local area networks.

See this page for more information about network switches.

Since packet sniffers need to see as much of the raw network traffic as possible, they can be thwarted by network switches, which by design try to limit the propagation of "unnecessary" packets. However, it can still be informative to try out a sniffer on a switched network. You may discover problems related to broadcast or multicast packets. Depending on your switch vendor, you may be surprised at how much traffic you can see. Even if you don't see other systems' network traffic, a sniffer can be helpful when you are tracking down problems that involve the local host.

In addition to having access to all network packets, the interface hardware must transport those packets up to the software layer. Packet addresses are normally checked in hardware, and only broadcast/multicast packets and those addressed to the local host are relayed to the kernel. In "promiscuous mode," an interface lets the kernel read all packets on the network, even the ones intended for other hosts.

Packet sniffers understand many of the packet formats used by standard network services, and they can print these packets in human-readable form. This capability makes it easier to track the flow of a conversation between two programs. Some sniffers print the ASCII contents of a packet in addition to the packet header and so are useful for investigating high-level protocols.

Since some protocols send information (and even passwords) across the network as cleartext, take care not to invade the privacy of your users. On the other hand, nothing quite dramatizes the need for cryptographic security like the sight of a plaintext password captured in a network packet.

Sniffers read data from a raw network device, so they must run as root. Although this root limitation serves to decrease the chance that normal users will listen in on your network traffic, it

is really not much of a barrier. Some sites choose to remove sniffer programs from most hosts to reduce the chance of abuse. If nothing else, you should check your systems' interfaces to be sure they are not running in promiscuous mode without your knowledge or consent.

***tcpdump**: command-line packet sniffer*

tcpdump, yet another amazing network tool by Van Jacobson, runs on most systems. **tcpdump** has long been the industry-standard sniffer, and most other network analysis tools read and write trace files in **tcpdump** format, also known as **libpcap** format.

By default, **tcpdump** tunes in on the first network interface it comes across. If it chooses the wrong interface, you can force an interface with the **-i** flag. If DNS is broken or you just don't want **tcpdump** doing name lookups, use the **-n** option. This option is important because slow DNS service can cause the filter to start dropping packets before they can be dealt with by **tcpdump**.

The **-v** flag increases the information you see about packets, and **-vv** gives you even more data. Finally, **tcpdump** can store packets to a file with the **-w** flag and can read them back in with the **-r** flag.

Note that **tcpdump -w** saves only packet headers by default. This default makes for small dumps, but the most helpful and relevant information may be missing. So, unless you are sure you need only headers, use the **-s** option with a value on the order of 1560 (actual values are MTU-dependent) to capture whole packets for later inspection.

As an example, the following truncated output comes from the machine named nubark. The filter specification **host bull** limits the display of packets to those that directly involve the machine bull, either as source or as destination.

```
$ sudo tcpdump host bull
12:35:23.519339 bull.41537 > nubark.domain:  A? atrust.com. (28) (DF)
12:35:23.519961 nubark.domain > bull.41537:  A 66.77.122.161 (112) (DF)
```

The first packet shows bull sending a DNS lookup request about atrust.com to nubark. The response is the IP address of the machine associated with that name, which is 66.77.122.161. Note the time stamp on the left and **tcpdump**'s understanding of the application-layer protocol (in this case, DNS). The port number on bull is arbitrary and is shown numerically (41537), but since the server port number (53) is well known, **tcpdump** shows its symbolic name, domain.

Packet sniffers can produce an overwhelming amount of information—overwhelming not only for you but also for the underlying operating system. To avoid this problem on busy networks, **tcpdump** lets you specify complex filters. For example, the following filter collects only incoming web traffic from one subnet:

```
$ sudo tcpdump src net 192.168.1.0/24 and dst port 80
```

The **tcpdump** man page contains several good examples of advanced filtering along with a complete listing of primitives.

*Wireshark and TShark: **tcpdump** on steroids*

tcpdump has been around since approximately the dawn of time, but a newer open source package called Wireshark (formerly known as Ethereal) has been gaining ground rapidly. Wireshark is under active development and incorporates more functionality than most commercial sniffing products. It's an incredibly powerful analysis tool and should be included in every networking expert's tool kit. It's also an invaluable learning aid.

Wireshark includes both a GUI interface (**wireshark**) and a command-line interface (**tshark**). It's available as a core package on most operating systems. If it's not in your system's core repository, check wireshark.org, which hosts the source code and a variety of precompiled binaries.

Wireshark can read and write trace files in the formats used by many other packet sniffers. Another handy feature is that you can click on any packet in a TCP conversation and ask Wireshark to reassemble (splice together) the payload data of all the packets in the stream. This feature is useful if you want to examine the data transferred during a complete TCP exchange, such as a connection on which an email message is transmitted across the network.

Wireshark's capture filters are functionally identical to **tcpdump**'s since Wireshark uses the same underlying **libpcap** library. Watch out, though—one important gotcha with Wireshark is the added feature of "display filters," which affect what you see rather than what's actually captured by the sniffer. The display filter syntax is more powerful than the **libpcap** syntax supported at capture time. The display filters do look somewhat similar, but they are not the same.

See this page for more information about SANs.

Wireshark has built-in dissectors for a wide variety of network protocols, including many used to implement SANs. It breaks packets into a structured tree of information in which every bit of the packet is described in plain English.

A note of caution regarding Wireshark: although it has lots of neat features, it has also required many security updates over the years. Run a current copy, and do not leave it running indefinitely on sensitive machines; it might be a potential route of attack.

13.13 Network monitoring

Chapter 28, *Monitoring*, describes several general-purpose platforms that can help structure the ongoing oversight of your systems and networks. These systems accept data from a variety of sources, summarize it in a way that illuminates ongoing trends, and alert administrators to problems that require immediate attention.

The network is a key component of any computing environment, so it’s often one of the first parts of the infrastructure to benefit from systematic monitoring. If you don’t feel quite ready to commit to a single monitoring platform for all your administrative needs, the packages outlined in this section are good options for small-scale monitoring that’s focused on the network.

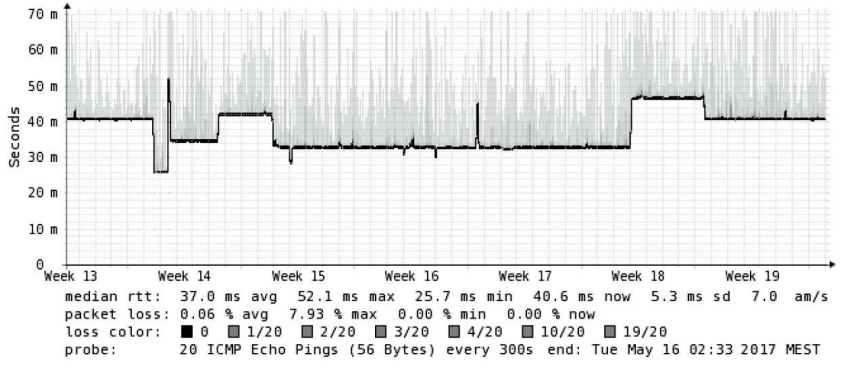
SmokePing: gather ping statistics over time

Even healthy networks drop an occasional packet. On the other hand, networks should not drop packets regularly, even at a low rate, because the impact on users can be disproportionately severe. Because high-level protocols often function even in the presence of packet loss, you might never notice dropped packets unless you're actively monitoring for them.

SmokePing, an open source tool by Tobias Oetiker, can help you develop a more comprehensive picture of your networks' behavior. SmokePing sends several ping packets to a target host at regular intervals. It shows the history of each monitored link through a web front end and can send alarms when things go amiss. You can get a copy from oss.oetiker.ch/smokeping.

Exhibit D shows a SmokePing graph. The vertical axis is the round trip time of pings, and the horizontal axis is time (weeks). The black line from which the gray spikes stick up indicates the median round trip time. The spikes themselves are the transit times of individual packets. Since the gray in this graph appears only above the median line, the great majority of packets must be traveling at close to the median speed, with just a few being delayed. This is a typical finding.

Exhibit D: Sample SmokePing graph



The stair-stepped shape of the median line indicates that the baseline transit time to this destination has changed several times during the monitoring period. The most likely hypotheses to explain this observation are either that the host is reachable by several routes or that it is actually a collection of several hosts that have the same DNS name but multiple IP addresses.

iPerf: track network performance

Ping-based tools are helpful for verifying reachability, but they're not really powerful enough to analyze and track network performance. Enter iPerf. The latest version, iPerf3, has an extensive set of features that administrators can use to fine tune network settings for maximum performance.

Here, we look only at iPerf's throughput monitoring. At the most basic level, iPerf opens a connection (TCP or UDP) between two servers, passes data between them, and records how long the process took.

Once you've installed **iperf** on both machines, start the server side.

```
$ iperf -s
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
```

Then, on the machine you want to test from, transfer some data as shown here.

```
$ iperf -c 10.211.55.11
------------------------------------------------------------
Client connecting to 10.211.55.11, TCP port 5001
TCP window size: 22.5 KByte (default)
------------------------------------------------------------
[  3] local 10.211.55.10 port 53862 connected with 10.211.55.11 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3] 0.0-10.0 sec   4.13 GBytes  3.55 Gbits/sec
```

iPerf returns great instantaneous data for tracking bandwidth. It's particularly helpful for assessing the effect of changes to kernel parameters that control the network stack, such as changes to the maximum transfer unit (MTU); see this page for more details.

Cacti: collect and graph data

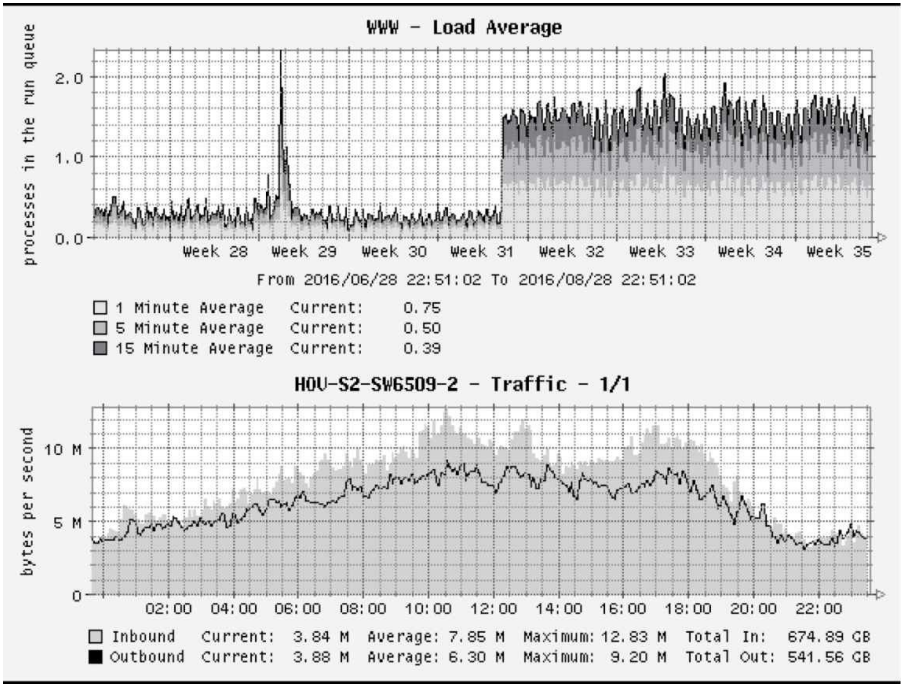
Cacti, available from cacti.net, offers several attractive features. It uses a separate package, RRDtool, as its back end, in which it stores monitoring data in the form of zero-maintenance, statically sized databases.

Cacti stores only enough data to create the graphs you want. For example, Cacti could store one sample every minute for a day, one sample every hour for a week, and one sample every week for a year. This consolidation scheme lets you maintain important historical context without having to store unimportant details or spend time on database administration.

Cacti can record and graph any SNMP variable (see this page), as well as many other performance metrics. You're free to collect whatever data you want. When combined with the NET-SNMP agent, Cacti generates a historical perspective on almost any system or network resource.

Exhibit E shows some examples of graphs created by Cacti. These graphs show the load average on a device over a period of multiple weeks along with a day's traffic on a network interface.

Exhibit E: Examples of Cacti graphs



Cacti sports easy web-based configuration as well as all the other built-in benefits of RRDtool, such as low maintenance and beautiful graphing. See the RRDtool home page at rrdtool.org for

links to the current versions of RRDtool and Cacti, as well as dozens of other monitoring tools.

13.14 Firewalls and NAT

We *do not* recommend the use of Linux, UNIX, or Windows systems as firewalls because of the insecurity inherent in running a full-fledged, general-purpose operating system. However, all operating systems have firewall features, and a hardened system is a workable substitute for organizations that don't have the budget for a high-dollar firewall appliance. Likewise, a Linux or UNIX firewall is a fine option for a security-savvy home user with a penchant for tinkering. (That said, many consumer-oriented networking devices, such as Linksys's router products, use Linux and **iptables** at their core.)

If you are set on using a general-purpose computer as a firewall, make sure that it's up to date with respect to security configuration and patches. A firewall machine is an excellent place to put into practice all the recommendations found in Chapter 27, *Security*. (The section that starts here discusses packet-filtering firewalls in general. If you are not familiar with the basic concept of a firewall, it would probably be wise to read that section before continuing.)

Microsoft has largely succeeded in convincing the world that every computer needs its own built-in firewall. However, that's not really true. In fact, machine-specific firewalls can lead to no end of inconsistent behavior and mysterious network problems if they are not managed in synchrony with site-wide standards.

Two main schools of thought deal with the issue of machine-specific firewalls. The first school considers them superfluous. According to this view, firewalls belong on gateway routers, where they can protect an entire network through the application of one consistent (and consistently applied) set of rules.

The second school considers machine-specific firewalls an important component of a "defense in depth" security plan. Although gateway firewalls are theoretically sufficient to control network traffic, they can be compromised, routed around, or administratively misconfigured. Therefore, it's prudent to implement the same network traffic restrictions through multiple, redundant firewall systems.

If you do choose to implement machine-specific firewalls, you need a system for deploying them in a consistent and easily updatable way. The configuration management systems described in Chapter 23 are excellent candidates for this task. Don't rely on manual configuration; it's just too vulnerable to entropy.

Linux iptables: rules, chains, and tables


Version 2.4 of the Linux kernel introduced an all-new packet-handling engine, called Netfilter, along with a command-line tool, **iptables**, to manage it. An even newer system, nftables, has been available since kernel version 3.13 from 2014. It's an elaboration of the Netfilter system that's configured with the **nft** command rather than the **iptables** command. We don't discuss nftables in this book, but it's worth evaluating at sites that run current kernels.


iptables configuration can be rather fiddly. Debian and Ubuntu include a simple front end, **ufw**, that facilitates common operations and configurations. It's worth checking out if your needs don't stray far from the mainstream.

iptables applies ordered "chains" of rules to network packets. Sets of chains make up "tables" and are used for handling specific kinds of traffic.

For example, the default **iptables** table is named "filter". Chains of rules in this table are used for packet-filtering network traffic. The filter table contains three default chains: FORWARD, INPUT, and OUTPUT. Each packet handled by the kernel is passed through exactly one of these chains.

Rules in the FORWARD chain are applied to all packets that arrive on one network interface and need to be forwarded to another. Rules in the INPUT and OUTPUT chains are applied to traffic addressed to or originating from the local host, respectively. These three standard chains are usually all you need for firewalling between two network interfaces. If necessary, you can define a custom configuration to support more complex accounting or routing scenarios.

In addition to the filter table, **iptables** includes the "nat" and "mangle" tables. The nat table contains chains of rules that control Network Address Translation (here, "nat" is the name of the **iptables** table and "NAT" is the name of the generic address translation scheme). The section *Private addresses and network address translation (NAT)* discusses NAT, and an example of the nat table in action is shown here. Later in this section, we use the nat table's PREROUTING chain for antispoofing packet filtering.

The mangle table contains chains that modify or alter the contents of network packets outside the context of NAT and packet filtering. Although the mangle table is handy for special packet handling, such as resetting IP time-to-live values, it is not typically used in most production environments. We discuss only the filter and nat tables in this section, leaving the mangle table to the adventurous.

***iptables** rule targets*

Each rule that makes up a chain has a "target" clause that determines what to do with matching packets. When a packet matches a rule, its fate is in most cases sealed; no additional rules are checked. Although many targets are defined internally to **iptables**, it is possible to specify another chain as a rule's target.

The targets available to rules in the filter table are ACCEPT, DROP, REJECT, LOG, ULOG, REDIRECT, RETURN, MIRROR, and QUEUE. When a rule results in an ACCEPT, matching packets are allowed to proceed on their way. DROP and REJECT both drop their packets; DROP is silent, and REJECT returns an ICMP error message. LOG gives you a simple way to track packets as they match rules, and ULOG expands logging.

REDIRECT shunts packets to a proxy instead of letting them go on their merry way. For example, you might use this feature to force all your site's web traffic to go through a web cache such as Squid. RETURN terminates user-defined chains and is analogous to the return statement in a subroutine call. The MIRROR target swaps the IP source and destination addresses before sending the packet. Finally, QUEUE hands packets to local user programs through a kernel module.

***iptables** firewall setup*

Before you can use **iptables** as a firewall, you must enable IP forwarding and make sure that various **iptables** modules have been loaded into the kernel. For more information on enabling IP forwarding, see *Linux TCP/IP options* or *Security-related kernel variables*. Packages that install **iptables** generally include startup scripts to achieve this enabling and loading.

A Linux firewall is usually implemented as a series of **iptables** commands contained in an **rc** startup script. Individual **iptables** commands usually take one of the following forms:

```
iptables -F chain-name
iptables -P chain-name target
iptables -A chain-name -i interface -j target
```

The first form (**-F**) flushes all prior rules from the chain. The second form (**-P**) sets a default policy (aka target) for the chain. We recommend that you use DROP for the default chain target. The third form (**-A**) appends the current specification to the chain. Unless you specify a table with the **-t** argument, your commands apply to chains in the filter table. The **-i** parameter applies the rule to the named *interface*, and **-j** identifies the *target*. **iptables** accepts many other clauses, some of which are shown in Table 13.10.

Table 13.10: Command-line flags for iptables filters

Clause	Meaning or possible values
-p *proto*	Matches by protocol: **tcp**, **udp**, or **icmp**
-s *source-ip*	Matches host or network source IP address (CIDR notation is OK)
-d *dest-ip*	Matches host or network destination address
--sport *port#*	Matches by source port (note the double dashes)
--dport *port#*	Matches by destination port (note the double dashes)
--icmp-type *type*	Matches by ICMP type code (note the double dashes)
!	Negates a clause
-t *table*	Specifies the table to which a command applies (default is filter)

A complete example

Below, we break apart a complete example. We assume that the eth1 interface goes to the Internet and that the eth0 interface goes to an internal network. The eth1 IP address is 128.138.101.4, the eth0 IP address is 10.1.1.1, and both interfaces have a netmask of 255.255.255.0. This example uses stateless packet filtering to protect the web server with IP address 10.1.1.2, which is the standard method of protecting Internet servers. Later in the example, we show how to use stateful filtering to protect desktop users.

Our first set of rules initializes the filter table. First, all chains in the table are flushed, then the INPUT and FORWARD chains' default target is set to DROP. As with any other network firewall, the most secure strategy is to drop any packets you have not explicitly allowed.

```
iptables -F
iptables -P INPUT DROP
iptables -P FORWARD DROP
```

Since rules are evaluated in order, we put our busiest rules at the front. However, we're careful to ensure that reordering the rules for performance doesn't modify functionality.

The first rule allows all connections through the firewall that originate from within the trusted net. The next three rules in the FORWARD chain allow connections through the firewall to network services on 10.1.1.2. Specifically, we allow SSH (port 22), HTTP (port 80), and HTTPS (port 443) through to our web server.

```
iptables -A FORWARD -i eth0 -p ANY -j ACCEPT
iptables -A FORWARD -d 10.1.1.2 -p tcp --dport 22 -j ACCEPT
iptables -A FORWARD -d 10.1.1.2 -p tcp --dport 80 -j ACCEPT
iptables -A FORWARD -d 10.1.1.2 -p tcp --dport 443 -j ACCEPT
```

The only TCP traffic we allow to our firewall host (10.1.1.1) is SSH, which is useful for managing the firewall itself. The second rule listed below allows loopback traffic, which stays local to the host. Administrators get nervous when they can't **ping** their default route, so the third rule here allows ICMP ECHO_REQUEST packets from internal IP addresses.

```
iptables -A INPUT -i eth0 -d 10.1.1.1 -p tcp --dport 22 -j ACCEPT
iptables -A INPUT -i lo -d 127.0.0.1 -p ANY -j ACCEPT
iptables -A INPUT -i eth0 -d 10.1.1.1 -p icmp --icmp-type 8 -j ACCEPT
```

For any IP host to work properly on the Internet, certain types of ICMP packets must be allowed through the firewall. The following eight rules allow a minimal set of ICMP packets to the firewall host, as well as to the network behind it.

```
iptables -A INPUT -p icmp --icmp-type 0 -j ACCEPT
iptables -A INPUT -p icmp --icmp-type 3 -j ACCEPT
iptables -A INPUT -p icmp --icmp-type 5 -j ACCEPT
iptables -A INPUT -p icmp --icmp-type 11 -j ACCEPT
iptables -A FORWARD -d 10.1.1.2 -p icmp --icmp-type 0 -j ACCEPT
iptables -A FORWARD -d 10.1.1.2 -p icmp --icmp-type 3 -j ACCEPT
iptables -A FORWARD -d 10.1.1.2 -p icmp --icmp-type 5 -j ACCEPT
iptables -A FORWARD -d 10.1.1.2 -p icmp --icmp-type 11 -j ACCEPT
```

See this page for more information about IP spoofing.

We next add rules to the PREROUTING chain in the nat table. Although the nat table is not intended for packet filtering, its PREROUTING chain is particularly useful for antispoofing filtering. If we put DROP entries in the PREROUTING chain, they need not be present in the INPUT and FORWARD chains, since the PREROUTING chain is applied to all packets that enter the firewall host. It's cleaner to put the entries in a single place rather than to duplicate them.

```
iptables -t nat -A PREROUTING -i eth1 -s 10.0.0.0/8 -j DROP
iptables -t nat -A PREROUTING -i eth1 -s 172.16.0.0/12 -j DROP
iptables -t nat -A PREROUTING -i eth1 -s 192.168.0.0/16 -j DROP
iptables -t nat -A PREROUTING -i eth1 -s 127.0.0.0/8 -j DROP
iptables -t nat -A PREROUTING -i eth1 -s 224.0.0.0/4 -j DROP
```

Finally, we end both the INPUT and FORWARD chains with a rule that forbids all packets not explicitly permitted. Although we already enforced this behavior with the **iptables -P** commands, the LOG target lets us see who is knocking on our door from the Internet.

```
iptables -A INPUT -i eth1 -j LOG
iptables -A FORWARD -i eth1 -j LOG
```

Optionally, we could set up IP NAT to disguise the private address space used on the internal network. See this page for more information about NAT.

One of the most powerful features that Netfilter brings to Linux firewalling is stateful packet filtering. Instead of allowing specific incoming services, a firewall for clients connecting to the Internet needs to allow incoming responses to the client's requests. The simple stateful

FORWARD chain below allows all traffic to leave our network but allows only incoming traffic that's related to connections initiated by our hosts.

```
iptables -A FORWARD -i eth0 -p ANY -j ACCEPT
iptables -A FORWARD -m state --state ESTABLISHED,RELATED -j ACCEPT
```

Certain kernel modules must be loaded to enable **iptables** to track complex network sessions such as those of FTP and IRC. If these modules are not loaded, **iptables** simply disallows those connections. Although stateful packet filters can increase the security of your site, they also add to the complexity of the network and can reduce performance. Be sure you need stateful functionality before implementing it in your firewall.

Perhaps the best way to debug your **iptables** rulesets is to use **iptables -L -v**. These options tell you how many times each rule in your chains has matched a packet. We often add temporary **iptables** rules with the LOG target when we want more information about the packets that get matched. You can often solve trickier problems by using a packet sniffer such as **tcpdump**.

Linux NAT and packet filtering

Linux traditionally implements only a limited form of Network Address Translation (NAT) that is more properly called Port Address Translation, or PAT. Instead of using a range of IP addresses as a true NAT implementation would, PAT multiplexes all connections onto a single address. The details and differences aren't of much practical importance.

iptables implements NAT as well as packet filtering. In earlier versions of Linux this functionality was a bit of a mess, but **iptables** makes a much cleaner separation between the NAT and filtering features. Of course, if you use NAT to let local hosts access the Internet, you *must* use a full complement of firewall filters as well.

To make NAT work, enable IP forwarding in the kernel by setting the kernel variable **/proc/sys/net/ipv4/ip_forward** to 1. Additionally, insert the appropriate kernel modules:

```
$ sudo modprobe iptable_nat
$ sudo modprobe ip_conntrack
$ sudo modprobe ip_conntrack_ftp
```

Many other modules track connections; see the **net/netfilter** subdirectory underneath **/lib/modules** for a more complete list and enable the ones you need.

The **iptables** command to route packets using NAT is of the form

```
iptables -t nat -A POSTROUTING -o eth0 -j SNAT --to 128.138.101.4
```

This example is for the same host as the filtering example in the previous section, so eth1 is the interface connected to the Internet. The eth1 interface does not appear directly in the command line above, but its IP address is the one that appears as the argument to **--to**. The eth0 interface is the one connected to the internal network.

To Internet hosts, it appears that all packets from hosts on the internal network have eth1's IP address. The host that implements NAT receives incoming packets, looks up their true destinations, rewrites them with the appropriate internal network IP address, and sends them on their merry way.

IPFilter for UNIX systems

IPFilter, an open source package developed by Darren Reed, supplies NAT and stateful firewall services on a variety of systems, including Linux and FreeBSD. You can use IPFilter as a loadable kernel module (which is recommended by the developers) or include it statically in the kernel.



IPFilter is mature and feature-complete. The package has an active user community and a history of continuous development. It is capable of stateful tracking even for stateless protocols such as UDP and ICMP.

IPFilter reads filtering rules from a configuration file (usually **/etc/ipf/ipf.conf** or **/etc/ipf.conf**) rather than obliging you to run a series of commands as does **iptables**. An example of a simple rule that could appear in **ipf.conf** is

```
block in all
```

This rule blocks all inbound traffic (i.e., network activity received by the system) on all network interfaces. Certainly secure, but not particularly useful!

Table 13.11 shows some of the possible conditions that can appear in an **ipf** rule.

Table 13.11: Commonly used ipf conditions

Condition	Meaning or possible values
`on` *interface*	Applies the rule to the specified interface
`proto` *protocol*	Selects packet according to protocol: `tcp`, `udp`, or `icmp`
`from` *source-ip*	Filters by source: host, network, or `any`
`to` *dest-ip*	Filters by destination: host, network, or `any`
`port =` *port#*	Filters by port name (from **/etc/services**) or number[a]
`flags` *flag-spec*	Filters according to TCP header flags bits
`icmp-type` *number*	Filters by ICMP type and code
`keep state`	Retains details about the flow of a session; see below

a. You can use any comparison operator: =, <, >, <=, >=, etc.

IPFilter evaluates rules in the sequence in which they are presented in the configuration file. The *last* match is binding. For example, inbound packets traversing the following filter will always pass:

```
block in all
pass in all
```

The `block` rule matches all packets, but so does the `pass` rule, and `pass` is the last match. To force a matching rule to apply immediately and make IPFilter skip subsequent rules, use the `quick`

keyword:

```
block in quick all
pass in all
```

An industrial-strength firewall typically contains many rules, so liberal use of quick is important in order to maintain the performance of the firewall. Without it, every packet is evaluated against every rule, and this wastefulness is costly.

Perhaps the most common use of a firewall is to control access to and from a specific network or host, often with respect to a specific port. IPFilter has powerful syntax to control traffic at this level of granularity. In the following rules, inbound traffic is permitted to the 10.0.0.0/24 network on TCP ports 80 and 443 and on UDP port 53.

```
block out quick all
pass in quick proto tcp from any to 10.0.0.0/24 port = 80 keep state
pass in quick proto tcp from any to 10.0.0.0/24 port = 443 keep state
pass in quick proto udp from any to 10.0.0.0/24 port = 53 keep state
block in all
```

The keep state keywords deserve special attention. IPFilter can keep track of connections by noting the first packet of new sessions. For example, when a new packet arrives addressed to port 80 on 10.0.0.10, IPFilter makes an entry in the state table and allows the packet through. It also allows the reply from the web server even though the first rule explicitly blocks all outbound traffic.

keep state is also useful for devices that offer no services but that must initiate connections. The following ruleset permits all conversations that are initiated by 192.168.10.10. It blocks all inbound packets except those related to connections that have already been initiated.

```
block in quick all
pass out quick from 192.168.10.10/32 to any keep state
```

The keep state keywords work for UDP and ICMP packets, too, but since these protocols are stateless, the mechanics are slightly more ad hoc: IPFilter permits responses to a UDP or an ICMP packet for 60 seconds after the inbound packet is seen by the filter. For example, if a UDP packet from 10.0.0.10, port 32,000, is addressed to 192.168.10.10, port 53, a UDP reply from 192.168.10.10 will be permitted until 60 seconds have passed. Similarly, an ICMP echo reply (ping response) is permitted after an echo request has been entered in the state table.

See this page for more information about NAT.

IPFilter uses the map keyword (in place of pass and block) to provide NAT services. In the following rule, traffic from the 10.0.0.0/24 network is mapped to the current routable address on the em0 interface.

```
map em0 10.0.0.0/24 -> 0/32
```

The filter must be reloaded if the address of em0 changes, as might happen if em0 leases a dynamic IP address through DHCP. For this reason, IPFilter's NAT features are best used at sites that have a static IP address on the Internet-facing interface.

Table 13.12 lists the command-line tools that come with the IPFilter package.

Table 13.12: IPFilter commands

Cmd	Function
ipf	Manages rules and filter lists
ipfstat	Obtains statistics about packet filtering
ipmon	Monitors logged filter information
ipnat	Manages NAT rules

Of the commands in Table 13.12, **ipf** is the most commonly used. **ipf** accepts a rule file as input and adds correctly parsed rules to the kernel's filter list. **ipf** adds rules to the end of the filter unless you use the **-Fa** argument, which flushes all existing rules. For example, to flush the kernel's existing set of filters and load the rules from **ipf.conf**, use the following syntax:

```
$ sudo ipf -Fa -f /etc/ipf/ipf.conf
```

IPFilter relies on pseudo-device files in **/dev** for access control, and by default only root can edit the filter list. We recommend leaving the default permissions in place and using **sudo** to maintain the filter.

Use **ipf**'s **-v** flag when loading the rules file to debug syntax errors and other problems in the configuration.

13.15 Cloud networking

One of the interesting features of the cloud is that you get to define the networking environment in which your virtual servers live. Ultimately, of course, cloud servers live on physical computers that are connected to real network hardware. However, that doesn't necessarily mean that virtual servers running on the same node are networked together. The combination of virtualization technology and programmable network switching equipment gives platform providers great flexibility to define the networking model they export to clients.

AWS's virtual private cloud (VPC)

VPC, the software-defined network technology for Amazon Web Services, creates private networks within the broader AWS network. VPC was first introduced in 2009 as a bridge between an on-premises data center and the cloud, opening up many hybrid use cases for enterprise organizations. Today, VPC is a central feature of AWS, and a default VPC is included for all accounts. EC2 instances for newer AWS accounts must be created within a VPC, and most new AWS services launch with native VPC support. (Longtime users gripe that AWS services are incomplete until they support VPC.)

The central features of VPC include

- An IPv4 address range selected from the RFC1918 private address space, expressed in CIDR notation (for example, 10.110.0.0/16 for the addresses 10.110.0.0-10.110.255.255; VPC has also recently added support for IPv6)
- Subnets to segment the VPC address space into smaller subnetworks
- Routing tables that determine where to send traffic
- Security groups that act as firewalls for EC2 instances
- Network Access Control Lists (NACLs) to isolate subnets from each other

You can create as many VPCs as you need, and no other AWS customer has access to network traffic within your VPCs. Depending on the state of your account, AWS may initially limit you to 5 VPCs. However, you can request a higher limit if you need it.

VPCs within the same region can be peered, creating private routes between separate networks. VPCs in different regions can be connected with software VPN tunnels over the Internet, or with expensive, custom, direct connections to AWS data centers over private circuits that you must lease from a telco.

VPCs can be as small as a /28 network or as large as a /16. It's important to plan ahead because the size cannot be adjusted after the VPC is created. Choose an address space that is large enough to accommodate future growth, but also ensure that it does not conflict with other networks that you may wish to connect.

Subnets and routing tables

Like traditional networks, VPCs are divided into subnets. Public subnets are for servers that must talk directly to clients on the Internet. They are akin to traditional DMZs. Private subnets are inaccessible from the Internet and are intended for trusted or sensitive systems.

See this page for more information about DMZs.

VPC routing is simpler than routing for a traditional hardware network because the cloud does not simulate physical topology. Every accessible destination is reachable in one logical hop.

In the world of physical networking, every device has a routing table that tells it how to route outbound network packets. But in VPC, routing tables are also an abstract entity that's defined through the AWS web console or its command-line equivalent. Every VPC subnet has an associated VPC routing table. When instances are created on a subnet, their routing tables are initialized from the VPC template.

The simplest routing table contains only a default static route for reaching other instances within the same VPC. You can add additional routes to access the Internet, on-premises networks (through VPN connections), or other VPCs (through peering connections).

A component called an Internet Gateway connects VPCs to the Internet. This entity is transparent to the administrator and is managed by AWS. However, you need to create one and attach it to your VPC if instances are to have Internet connectivity. Hosts in public subnets can access the Internet Gateway directly.

Instances in private subnets cannot be reached from the Internet even if they are assigned public IP addresses, a fact that results in much confusion for new users. For outbound access, they must hop through a NAT gateway on a public subnet. VPC offers a managed NAT feature which saves you the overhead of running your own gateway, but it incurs an additional hourly cost. The NAT gateway is a potential bottleneck for applications that have high throughput requirements, so it's better to locate the servers for such applications on public subnets, avoiding NAT.

AWS's implementation of IPv6 does not have NAT, and all instances set up for IPv6 receive "public" (i.e., routable) IPv6 addresses. You make IPv6 subnets private by connecting them through an egress-only Internet Gateway (aka eigw) which blocks inbound connections. The gateway is stateful, so external hosts can talk to servers on the private IPv6 network as long as the AWS server initiates the connection.

To understand the network routing for an instance, you'll find it more informative to review the VPC routing table for its subnet than to look at the instance's actual routing table (such as might be displayed by **netstat -r** or **ip route show** when logged in to the instance). The VPC version identifies gateways ("targets") by their AWS identifiers, which makes the table easy to parse at a glance.

In particular, you can easily distinguish public subnets from private subnets by looking at the VPC routing table. If the default gateway (i.e., the target associated with the address 0.0.0.0/0) is an Internet Gateway (an entity named igw-*something*), then that subnet is public. If the default gateway is a NAT device (a route target prefixed by an instance ID, i-*something*, or nat-*something*), then the subnet is private.

Table 13.13 shows an example routing table for a private subnet.

Table 13.13: Example VPC routing table for a private subnet

Destination	Target	Target type
10.110.0.0/16	local	Built-in route for the local VPC network
0.0.0.0/0	nat-a31ed812	Internet access through a VPC NAT gateway
10.120.0.0/16	pcx-38c3e8b2	Peering connection to another VPC
192.168.0.0/16	vgw-1e513d90	VPN gateway to an external network

VPCs are regional, but subnets are restricted to a single availability zone. To build highly available systems, create at least one subnet per zone and distribute instances evenly among all the subnets. A typical design puts load balancers or other proxies in public subnets and restricts web, application, and database servers to private subnets.

Security groups and NACLs

Security groups are firewalls for EC2 instances. Security group rules dictate which source addresses are allowed for ICMP, UDP, and TCP traffic (ingress rules), and which ports on other systems can be accessed by instances (egress rules). Security groups deny all connections by default, so any rules you add allow additional traffic.

All EC2 instances belong to at least one security group, but they may be in as many as five. To be perfectly accurate, security groups are associated with network interfaces, and an instance can have more than one network interface. So we should really say that the maximum number of security groups is the number of network interfaces times five.

The more security groups an instance belongs to, the more confusing it can be to determine precisely what traffic is and is not allowed. We prefer that each instance be in only one security group, even if that configuration results in some duplicate rules among groups.

When adding rules to security groups, always consider the principle of least privilege. Opening ports unnecessarily presents a security risk, especially for systems that have public, routable IP addresses. For example, a web server may only need ports 22 (SSH, used for management and control of the system), 80 (HTTP), and 443 (HTTPS).

In addition, all hosts should accept the ICMP packets used to implement path MTU discovery. Failure to admit these packets can lower network bandwidth considerably, so we find puzzling AWS's decision to block them by default. See goo.gl/WrETNq (deep link into docs.aws.amazon.com) for the steps to enable these packets.

Most security groups have granular inbound rules but allow all outbound traffic, as shown in Table 13.14. This configuration is convenient since you don't need to think about what outside connectivity your systems have. However, it's easier for attackers to set up shop if they can retrieve tools and communicate with their external control systems. The most secure networks have both inbound and outbound restrictions.

Table 13.14: Typical security group rules

Direction	Proto	Ports	CIDR	Notes
Ingress	TCP	22	10.110.0.0/16	SSH from the internal network
Ingress	TCP	80	0.0.0.0/0	HTTP from anywhere
Ingress	TCP	443	0.0.0.0/0	HTTPS from anywhere
Ingress	ICMP	n/a [a]	0.0.0.0/0	Allow path MTU discovery
Egress	ALL	ALL	0.0.0.0/0	Outbound traffic (all OK)

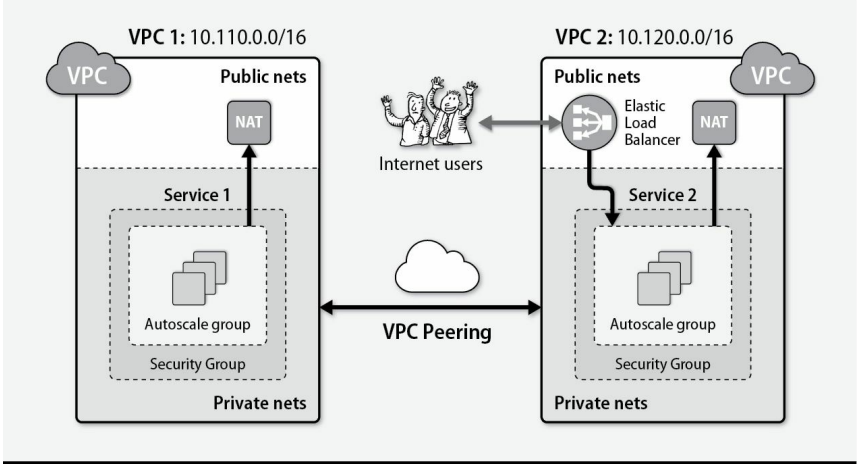
a. See goo.gl/WrETNq for detailed instructions; this record is a bit tricky to set up.

Much like access control lists on a firewall device, NACLs control traffic among subnets. Unlike security groups, NACLs are stateless: they don't distinguish between new and existing connections. They are similar in concept to NACLs on a hardware firewall. NACLs allow all traffic by default. In the wild, we see security groups used far more often than NACLs.

A sample VPC architecture

Exhibit F depicts two VPCs, each with public and private subnets. Network 2 hosts an Elastic Load Balancer in its public subnets. The ELB acts as a proxy for some autoscaling EC2 instances that live in the private subnet and protects those instances from the Internet. Service 2 in Network 2 may need access to Service 1 hosted in Network 1, and they can communicate privately through VPC peering.

Exhibit F: Peered VPCs with public and private subnets



Architecture diagrams like Exhibit F communicate dense technical details more clearly than written prose. We maintain diagrams like this one for every application we deploy.

Creating a VPC with Terraform

VPCs are composed of many resources, each of which has its own settings and options. The interdependencies among these objects are complex. It’s possible to create and manage almost everything by using the CLI or web console, but that approach requires that you keep all the minutiae in your head. Even if you can keep all the moving parts straight during the initial setup, it’s difficult to track your work over time.

Terraform, a tool from HashiCorp, creates and manages cloud resources. For example, Terraform can create a VPC, launch instances, and then initialize those instances by running scripts or other configuration management tools. Terraform configuration is expressed in HashiCorp Configuration Language (HCL), a declarative format that looks similar to JSON but adds variable interpolation and comments. The file can be tracked in revision control, so it’s simple to update and adapt.

The example below shows a Terraform configuration for a simple VPC with one public subnet. We think it’s rather self-documenting, intelligible even to a neophyte:

```
# Specify the VPC address range as a variable
variable "vpc_cidr" {
   default = "10.110.0.0/16"
}

# The address range for a public subnet
variable "public_subnet_cidr" {
   default = "10.110.0.0/24"
}
# The VPC
resource "aws_vpc" "default" {
   cidr_block = "${var.vpc_cidr}"
   enable_dns_hostnames = true
}

# Internet gateway to connect the VPC to the Internet
resource "aws_internet_gateway" "default" {
   vpc_id = "${aws_vpc.default.id}"
}

# Public subnet
resource "aws_subnet" "public-us-west-2a" {
   vpc_id = "${aws_vpc.default.id}"
   cidr_block = "${var.public_subnet_cidr}"
   availability_zone = "us-west-2a"
}

# Route table for the public subnet
resource "aws_route_table" "public-us-west-2a" {
   vpc_id = "${aws_vpc.default.id}"
   route {
      cidr_block = "0.0.0.0/0"
      gateway_id = "${aws_internet_gateway.default.id}"
   }
}

# Associate the route table with the public subnet
resource "aws_route_table_association" "public-us-west-2-a" {
   subnet_id = "${aws_subnet.public-us-west-2a.id}"
   route_table_id = "${aws_route_table.public-us-west-2a.id}"
}
```

The Terraform documentation is the authoritative syntax reference. You'll find many example configurations like this one in the Terraform GitHub repository and elsewhere on the Internet.

Run **terraform apply** to have Terraform create this VPC. It examines the current directory (by default) for **.tf** files and processes each of them, assembling an execution plan and then invoking API calls in the appropriate order. You can set the AWS API credentials in the configuration file or through the AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY environment variables, as we have done here.

```
$ AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
$ AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENGbPxRfiCYEXAMPLEKEY
$ time terraform apply
aws_vpc.default: Creating...
  cidr_block:                "" => "10.110.0.0/16"
  default_network_acl_id:    "" => "<computed>"
  default_security_group_id: "" => "<computed>"
  dhcp_options_id:           "" => "<computed>"
  enable_dns_hostnames:      "" => "1"
  enable_dns_support:        "" => "<computed>"
  main_route_table_id:       "" => "<computed>"
aws_vpc.default: Creation complete
aws_internet_gateway.default: Creating...
  vpc_id: "" => "vpc-a9ebe3cc"
aws_subnet.public-us-west-2a: Creating...
  availability_zone:       "" => "us-west-2a"
  cidr_block:              "" => "10.110.0.0/24"
  map_public_ip_on_launch: "" => "0"
  vpc_id:                  "" => "vpc-a9ebe3cc"
aws_subnet.public-us-west-2a: Creation complete
aws_route_table.public-us-west-2a: Creation complete
[snip]
Apply complete! Resources: 5 added, 0 changed, 0 destroyed.
real 0m4.530s
user 0m0.221s
sys  0m0.172s
```

time measures how long it takes to create all the resources in the configuration (about 4.5 seconds). The <computed> values indicate that Terraform chose defaults because we didn't specify those settings explicitly.

The state of all resources created by Terraform is saved in a file called **terraform.tfstate**. This file must be preserved so that Terraform knows which resources are under its control. In the future, Terraform will discover the managed resources on its own.

We can clean up the VPC just as easily:

```
$ terraform destroy -force
aws_vpc.default: Refreshing state... (ID: vpc-87ebe3e2)
aws_subnet.public-us-west-2a: Refreshing state... (ID: subnet-7c596a0b)
aws_internet_gateway.default: Refreshing state... (ID: igw-dc95edb9)
aws_route_table.public-us-west-2a: Refreshing state... (ID: rtb-2fc7214b)
aws_route_table_association.public-us-west-2-a: Refreshing state... (ID:
    rtbassoc-da479bbe)
aws_route_table_association.public-us-west-2-a: Destroying...
aws_route_table_association.public-us-west-2-a: Destruction complete
aws_subnet.public-us-west-2a: Destroying...
aws_route_table.public-us-west-2a: Destroying...
aws_route_table.public-us-west-2a: Destruction complete
aws_internet_gateway.default: Destroying...
aws_subnet.public-us-west-2a: Destruction complete
aws_internet_gateway.default: Destruction complete
aws_vpc.default: Destroying...
aws_vpc.default: Destruction complete
Apply complete! Resources: 0 added, 0 changed, 5 destroyed.
```

Terraform is cloud-agnostic, so it can manage resources for AWS, GCP, DigitalOcean, Azure, Docker, and other providers.

How do you know when to use Terraform and when to use the CLI? If you're building infrastructure for a team or project, or if you'll need to make changes and repeat the build later, use Terraform. If you need to fire off a quick instance as a test, if you need to inspect the details of a resource, or if you need to access the API from a shell script, use the CLI.

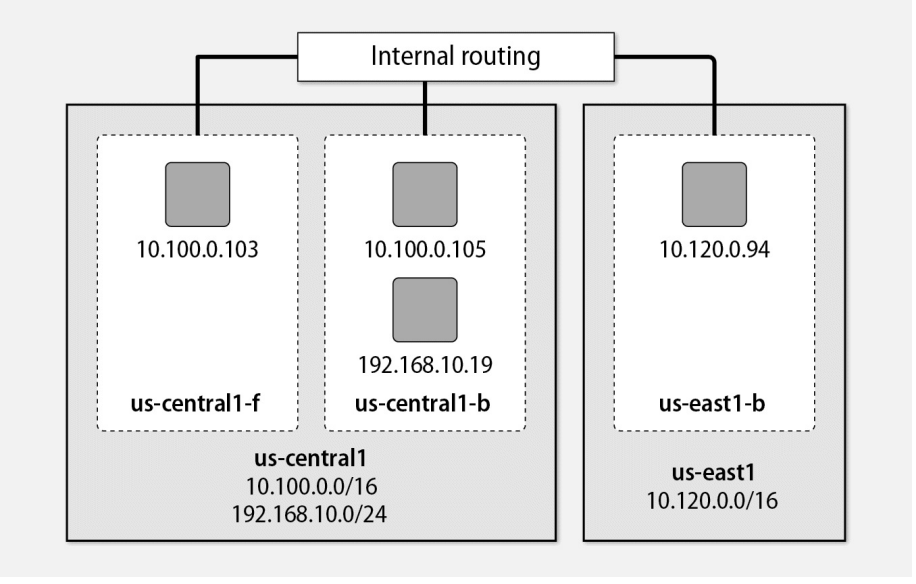
Google Cloud Platform networking

On the Google Cloud Platform, networking is functionally part of the platform, as opposed to being represented as a distinct service. GCP private networks are global: an instance in the us-east1 region can communicate with another instance in europe-west1 over the private network, a fact that makes it easy to build global network services. Network traffic among instances in the same zone is free, but there is a fee for traffic between zones or regions.

New projects have a default network with the 10.240.0.0/16 address range. You can create up to five separate networks per project, and instances are members of exactly one network. Many sites use this network architecture to isolate test and development from production systems.

Networks can be subdivided by region with subnetworks, a relatively recent addition to GCP that functions differently from subnets on AWS. The global network does not need to be part of a single IPv4 prefix range, and there can be multiple prefixes per region. GCP configures all the routing for you, so instances on different CIDR blocks within the same network can still reach each other. Exhibit G demonstrates this topology.

Exhibit G: A multiregion private GCP network with subnetworks



There is no concept of a subnetwork being public or private; instead, instances that don't need to accept inbound traffic from the Internet can simply not have a public, Internet-facing address. Google offers static external IP addresses that you can borrow for use in DNS records without fear that they will be assigned to another customer. When an instance does have an external address, you still won't see it if you run **ip addr show**; Google handles the address translation for you.

By default, firewall rules in a GCP network apply to all instances. To restrict rules to a smaller set of instances, you can tag instances and filter the rules according to the tags. The default, global firewall rules deny everything except the following:

- ICMP traffic for 0/0
- RDP (remote desktop for Windows, TCP port 3389) for 0/0
- SSH (TCP port 22) for 0/0
- All ports and protocols for the internal network (10.240.0.0/16 by default)

When it comes to decisions that impact security, we always come back to the principle of least privilege. In this case, we recommend narrowing these default rules to block RDP entirely, allow SSH only from your own source IPs, and further restrict traffic within the GCP network. You might also want to block ICMP, but be aware that you need to allow ICMP packets of type 3, code 4 to enable path MTU discovery.

DigitalOcean networking

DigitalOcean does not have a private network, or at least, not one similar to those of GCP and AWS. Droplets can have private interfaces that communicate over an internal network within the same region. However, that network is shared with all other DigitalOcean customers in the same region. This is a slight improvement over using the Internet, but firewalls and in-transit encryption become hard requirements.

We can examine a booted DigitalOcean droplet with the **tugboat** CLI:

```
$ tugboat info ulsah
Droplet fuzzy name provided. Finding droplet ID...done, 8857202
    (ulsah-ubuntu-15-10)
Name:             ulsah-ubuntu-15-10
ID:               8857202
Status:           active
IP4:              45.55.1.165
IP6:              2604:A880:0001:0020:0000:0000:01EF:D001
Private IP:       10.134.131.213
Region:           San Francisco 1 - sfo1
Image:            14169855 - ubuntu-15-10-x64
Size:             512MB
Backups Active:   false
```

The output includes an IPv6 address in addition to public and private IPv4 addresses.

On the instance, we can further explore by looking at the addresses on the local interfaces.

```
# tugboat ssh ulsah-ubuntu-15-10
# ip address show eth0
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast
    state UP group default qlen 1000
    link/ether 04:01:87:26:d6:01 brd ff:ff:ff:ff:ff:ff
    inet 45.55.1.165/19 brd 45.55.31.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet 10.12.0.8/16 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::601:87ff:fe26:d601/64 scope link
       valid_lft forever preferred_lft forever
# ip address show eth1
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast
    state UP group default qlen 1000
    link/ether 04:01:87:26:d6:02 brd ff:ff:ff:ff:ff:ff
    inet 10.134.131.213/16 brd 10.134.255.255 scope global eth1
       valid_lft forever preferred_lft forever
    inet6 fe80::601:87ff:fe26:d602/64 scope link
       valid_lft forever preferred_lft forever
```

The public address is assigned directly to the eth0 interface, not translated by the provider as on other cloud platforms. Each interface also has an IPv6 address, so it’s possible to serve traffic through IPv4 and IPv6 simultaneously.

13.16 Recommended reading

History

Comer, Douglas E. *Internetworking with TCP/IP Volume 1: Principles, Protocols, and Architectures (6th Edition)*. Upper Saddle River, NJ: Prentice Hall, 2013.

Doug Comer's *Internetworking with TCP/IP* series was for a long time the standard reference for the TCP/IP protocols. The books are designed as undergraduate textbooks and are a good introductory source of background material.

Salus, Peter H. *Casting the Net, From ARPANET to INTERNET and Beyond.* Reading, MA: Addison-Wesley Professional, 1995.

This is a lovely history of the ARPANET as it grew into the Internet, written by a historian who has been hanging out with UNIX people long enough to sound like one of them.

An excellent collection of documents about the history of the Internet and its various technologies can be found at isoc.org/internet/history.

Classics and bibles

STEVENS, W. RICHARD. *UNIX Network Programming.* Upper Saddle River, NJ: Prentice Hall, 1990.

STEVENS, W. RICHARD, BILL FENNER, AND ANDREW M. RUDOFF. *UNIX Network Programming, Volume 1, The Sockets Networking API (3rd Edition).* Upper Saddle River, NJ: Addison-Wesley, 2003.

STEVENS, W. RICHARD. *UNIX Network Programming, Volume 2: Interprocess Communications (2nd Edition).* Upper Saddle River, NJ: Addison-Wesley, 1999.

The *UNIX Network Programming* books are the student's bibles in networking classes that involve programming. If you need only the Berkeley sockets interface, the original edition is still a fine reference. If you need the STREAMS interface too, then the third edition, which includes IPv6, is a good bet. All three are clearly written in typical Rich Stevens style.

TANENBAUM, ANDREW S., AND DAVID J. WETHERALL. *Computer Networks (5th Edition).* Upper Saddle River, NJ: Prentice Hall PTR, 2011.

Computer Networks was the first networking text, and it is still a classic. It contains a thorough description of all the nitty-gritty details going on at the physical and link layers of the protocol stack. The latest edition includes coverage of wireless networks, gigabit Ethernet, peer-to-peer networks, voice over IP, cellular networks, and more.

Protocols

FALL, KEVIN R., AND W. RICHARD STEVENS. *TCP/IP Illustrated, Volume One: The Protocols (2nd Edition).* Reading, MA: Addison-Wesley, 2011.

WRIGHT, GARY R., AND W. RICHARD STEVENS. *TCP/IP Illustrated, Volume Two: The Implementation.* Reading, MA: Addison-Wesley, 1995.

The books in the *TCP/IP Illustrated* series are an excellent and thorough guide to the TCP/IP protocol stack.

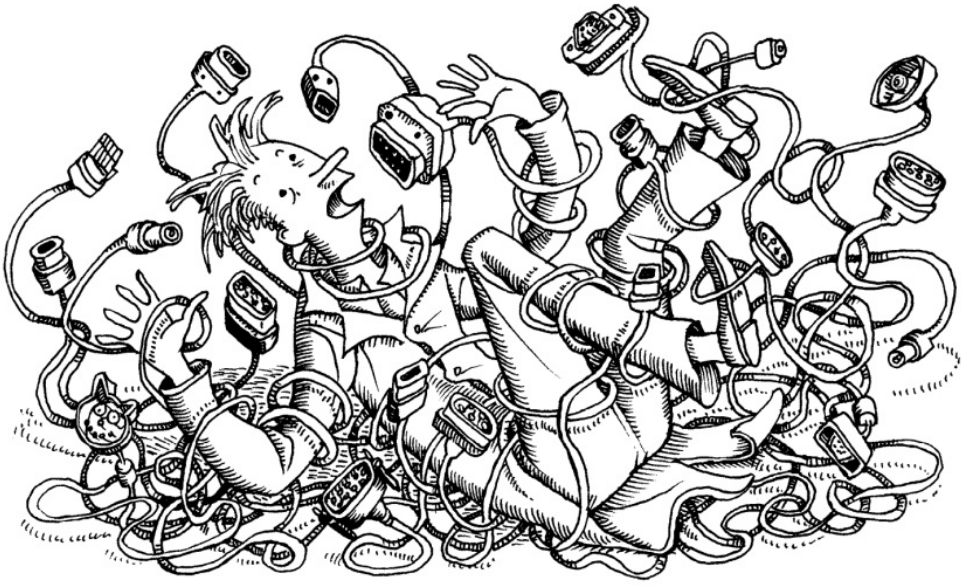
HUNT, CRAIG. *TCP/IP Network Administration (3rd Edition).* Sebastopol, CA: O'Reilly Media, 2002. Like other books in the nutshell series, this book is directed at administrators of UNIX systems. Half the book is about TCP/IP, and the rest deals with higher-level UNIX facilities such as email and remote login.

FARREL, ADRIAN. *The Internet and Its Protocols: A Comparative Approach*. San Francisco, CA: Morgan Kaufmann Publishers, 2004.

KOZIERAK, CHARLES M. *The TCP/IP Guide: A Comprehensive, Illustrated Internet Protocols Reference*. San Francisco, CA: No Starch Press, 2005.

DONAHUE, GARY A. *Network Warrior: Everything You Need to Know That Wasn't on the CCNA Exam*. Sebastopol, CA: O'Reilly Media, 2015.

14 Physical Networking



Regardless of whether your systems live in a data center, a cloud, or an old missile silo, one element they have in common is the need to communicate on a network. The ability to move data quickly and reliably is essential in every environment. If there's one area in which UNIX technology has touched human lives and influenced other operating systems, it's in the practical realization of large-scale packetized data transport.

Networks are following the same trail blazed by servers in that the physical and logical views of the network are increasingly separated by a virtualization layer that has its own configuration. That sort of setup is standard in the cloud, but even physical data centers often include a layer of software-defined networking (SDN) these days.

Administrators interact with real-world network hardware less frequently than they once did, but familiarity with traditional networking remains a crucial skill. Virtualized networks closely emulate physical networks in their features, terminology, architecture, and topology.

Many link-layer technologies have been promoted over the years, but Ethernet has emerged as the clear and decisive winner. Now that Ethernet is found on everything from game consoles to refrigerators, a thorough understanding of this technology is critical to success as a system administrator.

Obviously, the speed and reliability of your network have a direct effect on your organization's productivity. But today, networking has become so pervasive that the state of the network affects even such basic interactions as the ability to make telephone calls. A poorly designed network is

a personal and professional embarrassment that can lead to catastrophic social effects. It can also be expensive to fix.

At least four major factors contribute to success:

- Development of a reasonable network design
- Selection of high-quality hardware
- Proper installation and documentation
- Competent ongoing operations and maintenance

This chapter focuses on understanding, installing, and operating Ethernet networks in an enterprise environment.

14.1 Ethernet: the Swiss Army knife of networking

Having captured over 95% of the world-wide local area network (LAN) market, Ethernet can be found just about everywhere in its many forms. It started as Bob Metcalfe's Ph.D. thesis at MIT but is now described in a variety of IEEE standards.

Ethernet was originally specified at 3 Mb/s (mega*bits* per second), but it moved to 10 Mb/s almost immediately. Once a 100 Mb/s standard was finalized in 1994, it became clear that Ethernet would evolve rather than be replaced. This realization touched off a race to build increasingly faster versions of Ethernet, and that race continues today. Table 14.1 highlights the evolution of the various Ethernet standards. We have omitted a few of the less popular Ethernet standards that cropped up along the way.

Table 14.1: The evolution of Ethernet

Year	Speed	Common name	IEEE#	Dist	Media[a]
1973	3 Mb/s	Xerox Ethernet	–	?	Coax
1976	10 Mb/s	Ethernet 1	–	500m	RG-11 coax
1989	10 Mb/s	10BASE-T	802.3	100m	Cat 3 UTP copper
1994	100 Mb/s	100BASE-TX	802.3u	100m	Cat 5 UTP copper
1999	1 Gb/s	1000BASE-T ("gigabit")	802.3ab	100m	Cat 5e, 6 UTP copper
2006	10 Gb/s	10GBASE-T ("10 gig")	802.3an	100m	Cat 6a, 7, 7a UTP
2009	40 Gb/s	40GBASE-CR4	P802.3ba	10m	UTP copper
		40GBASE-SR4		100m	MM fiber
2009	100 Gb/s	100GBASE-CR10	P802.3ba	10m	UTP copper
		100GBASE-SR10		100m	MM fiber
2018[b]	200 Gb/s	200GBASE-FR4	802.3bs[c]	2km	CWDM fiber
		200GBASE-LR4		10km	CWDM fiber
2018[b]	400 Gb/s	400GBASE-SR16	802.3bs	100m	MM fiber (16 strand)
		400GBASE-DR4		500m	MM fiber (4 strand)
		400GBASE-FR8		2km	CWDM fiber
		400GBASE-LR8		10km	CWDM fiber
2020[b]	1 Tb/s	TbE	TBD	TBD	TBD

a. MM = Multimode, SM = Single-mode, UTP = Unshielded twisted pair, CWDM = Coarse wavelength division multiplexing
b. Industry projection
c. We'll give the benefit of the doubt and assume this lettering choice was an unfortunate coincidence.

Ethernet signaling

The underlying model used by Ethernet can be described as a polite dinner party at which guests (computers) don't interrupt each other but rather wait for a lull in the conversation (no traffic on the network cable) before speaking. If two guests start to talk at once (a collision) they both stop, excuse themselves, wait a bit, and then one of them starts talking again.

The technical term for this scheme is CSMA/CD:

- Carrier Sense: you can tell whether anyone is talking.
- Multiple Access: everyone can talk.
- Collision Detection: you know when you interrupt someone else.

The actual delay after a collision is somewhat random. This convention avoids the scenario in which two hosts simultaneously transmit to the network, detect the collision, wait the same amount of time, and then start transmitting again, thus flooding the network with collisions. This was not always true!

Today, the importance of the CSMA/CD conventions has been lessened by the advent of switches, which typically limit the number of hosts to two in a given collision domain. (To continue the "dinner party" analogy, you might think of this switched variant of Ethernet as being akin to the scenes found in old movies where two people sit at opposite ends of a long, formal dining table.)

Ethernet topology

The Ethernet topology is a branching bus with no loops. A packet can travel between two hosts on the same network in only one way.

Three types of packets can be exchanged on a segment: unicast, multicast, and broadcast. Unicast packets are addressed to only one host. Multicast packets are addressed to a group of hosts. Broadcast packets are delivered to all hosts on a segment.

A “broadcast domain” is the set of hosts that receive packets destined for the hardware broadcast address. Exactly one broadcast domain is defined for each logical Ethernet segment. Under the early Ethernet standards and media (e.g., 10BASE5), physical segments and logical segments were exactly the same because all the packets traveled on one big cable with host interfaces strapped onto the side of it. Attaching a new computer involved boring a hole into the outer sheath of the cable with a special drill to reach the center conductor. A “vampire tap” that bit into the outer conductor was then clamped on with screws.

With the advent of switches, today’s logical segments usually consist of many physical segments (possibly dozens or hundreds) to which only two devices are connected: a switch port and a host. The switches are responsible for escorting multicast and unicast packets to the physical (or wireless) segments on which the intended recipients reside. Broadcast traffic is forwarded to all ports in a logical segment.

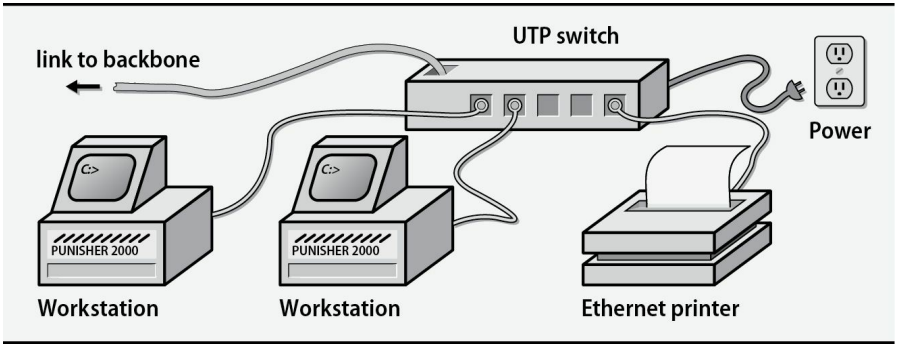
A single logical segment can consist of physical (or wireless) segments that operate at different speeds. Hence, switches must have buffering and timing capabilities to let them smooth over any potential timing conflicts.

Wireless networks are another common type of logical Ethernet segment. They behave more like the traditional forms of Ethernet which share one cable among many hosts. See this page.

Unshielded twisted-pair cabling

Unshielded twisted pair (UTP) has historically been the preferred cable medium for Ethernet in most office environments. Today, wireless networking has displaced UTP in many situations. The general "shape" of a UTP network is illustrated in Exhibit A.

Exhibit A: A UTP installation



UTP wire is commonly broken down into eight classifications. The performance rating system was first introduced by Anixter, a large cable supplier. These standards were formalized by the Telecommunications Industry Association (TIA) and are known today as Category 1 through Category 8, with a few special variants such as Category 5e and Category 6a thrown in for good measure.

The International Organization for Standardization (ISO) has also jumped into the exciting and highly profitable world of cable classification. They promote standards that are exactly or approximately equivalent to the higher-numbered TIA categories. For example, TIA Category 5 cable is equivalent to ISO Class D cable. For the geeks in the audience, Table 14.2 illustrates the differences among the various modern-day classifications. This is good information to memorize so you can impress your friends at parties.

Table 14.2: UTP cable characteristics

Parameter	Units	Cat 5[b] Class D	Cat 5e	Cat 6 Class E	Cat 6a Class EA	Cat 7 Class F	Cat 7a Class FA	Cat 8 Class I
Frequency range	MHz	100	100	250	500	600	1000	2000
Attenuation	dB	24	24	21.7	18.4	20.8	60	50
NEXT[a]	dB	27.1	30.1	39.9	59	62.1	60.4	36.5
ELFEXT[a]	dB	17	17.4	23.2	43.1	46.0	35.1	–
Return loss	dB	8	10	12	32	14.1	61.93	8
Propagation delay	ns	548	548	548	548	504	534	548

a. NEXT = Near-end crosstalk, ELFEXT = Equal level far-end crosstalk
b. Includes additional TIA and ISO requirements TSB95 and FDAM 2, respectively

Category 5 cable can support 100 Mb/s and is “table stakes” for network wiring today. Category 5e, Category 6, and Category 6a cabling support 1 Gb/s and are the most common standard currently in use for data cabling. Category 6a is the cable of choice for new installations because it is particularly resistant to interference from older Ethernet signaling standards (e.g., 10BASE-T), a problem that has plagued some Category 5/5e installations. Category 7 and Category 7a cable are intended for 10 Gb/s use, and Category 8 rounds out the family at 40 Gb/s.

Faster standards require multiple pairs of UTP. Having multiple conductors transports data across the link faster than any single pair can support. 100BASE-TX requires two pairs of Category 5 wire. 1000BASE-TX requires four pairs of Category 5e or Category 6/6a wire, and 10GBASE-TX requires four pairs of Category 6a, 7, or 7a wire. All these standards are limited to 100 meters in length.

Both PVC-coated and Teflon-coated wire are available. Your choice of jacketing should depend on the environment in which the cable will be installed. Enclosed areas that feed into the building’s ventilation system (“return air plenums”) typically require Teflon. PVC is less expensive and easier to work with but produces toxic fumes if it catches fire, hence the need to keep it out of air plenums. Check with your fire marshal or local fire department to determine the requirements in your area.

For terminating the four-pair UTP cable at patch panels and RJ-45 wall jacks, we suggest you use the TIA/EIA-568A RJ-45 wiring standard. This standard, which is compatible with other uses of RJ-45 (e.g., RS-232), is a convenient way to keep the wiring at both ends of the connection consistent, regardless of whether you can easily access the cable pairs themselves. Table 14.3 shows the pinouts.

Table 14.3: TIA/EIA-568A standard for wiring four-pair UTP to an RJ-45 jack

Pair	Colors	Wired to	Pair	Colors	Wired to
1	White/Blue	Pins 5/4	3	White/Green	Pins 1/2
2	White/Orange	Pins 3/6	4	White/Brown	Pins 7/8

Existing building wiring might or might not be suitable for network use, depending on how and when it was installed.

Optical fiber

Optical fiber is used in situations where copper cable is inadequate. Fiber carries signals farther than copper and is also resistant to electrical interference. In cases where fiber isn't absolutely necessary, copper is normally preferred because it's less expensive and easier to work with.

"Multimode" and "single mode" fiber are the two common types. Multimode fiber is typically used for applications within a building or campus. It's thicker than single-mode fiber and can carry multiple rays of light; this feature permits the use of less expensive electronics (e.g., LEDs as a light source).

Single-mode fiber is most often found in long-haul applications, such as intercity or interstate connections. It can carry only a single ray of light and requires expensive precision electronics on the endpoints.

A common strategy to increase the bandwidth across a fiber link is coarse wavelength division multiplexing (CWDM). It's a way to transmit multiple channels of data through a single fiber on multiple wavelengths (colors) of light. Some of the faster Ethernet standards use this scheme natively. However, it can also be employed to extend the capabilities of an existing dark fiber link through the use of CWDM multiplexers.

TIA-598C recommends color-coding the common types of fiber, as shown in Table 14.4. The key rule to remember is that everything must match. The fiber that connects the endpoints, the fiber cross-connect cables, and the endpoint electronics must all be of the same type and size. Note that although both OM1 and OM2 are colored orange, they are not interchangeable—check the size imprint on the cables to make sure they match. You will experience no end of difficult-to-isolate problems if you don't follow this rule.

Table 14.4: Attributes of standard optical fibers

Mode	ISO name[a]	Core diameter	Cladding diameter	Color
Multi	OM1	62.5 µm	125 µm	Orange
Multi	OM2	50 µm	125 µm	Orange
Multi	OM3	50 µm[b]	125 µm	Aqua
Single	OS1	8–10 µm	125 µm	Yellow

a. According to ISO 11801
b. OM3 is optimized for carrying laser light.

More than 30 types of connectors are used on the ends of optical fibers, and there is no real rhyme or reason as to which connectors are used where. The connectors you need to use in a particular case will most often be dictated by your equipment vendors or by your existing building fiber plant. The good news is that conversion jumpers are fairly easy to obtain.

Ethernet connection and expansion

Ethernets can be connected through several types of devices. The options below are ranked by approximate cost, with the cheapest options first. The more logic that a device uses to move bits from one network to another, the more hardware and embedded software the device needs to have and the more expensive it is likely to be.

Hubs

Devices from a bygone era, hubs are also referred to as concentrators or repeaters. They are active devices that connect Ethernet segments at the physical layer. They require external power.

A hub retimes and retransmits Ethernet frames but does not interpret them; it has no idea where packets are going or what protocol they are using. With the exception of extremely special cases, hubs should no longer be used in enterprise networks; we discourage their use in residential (consumer) networks as well. Switches make significantly more efficient use of network bandwidth and are just as cheap these days.

Switches

Switches connect Ethernets at the link layer. They join two physical networks in a way that makes them seem like one big physical network. Switches are the industry standard for connecting Ethernet devices today.

Switches receive, regenerate, and retransmit packets in hardware. Switches use a dynamic learning algorithm. They notice which source addresses come from one port and which from another. They forward packets between ports only when necessary. At first all packets are forwarded, but in a few seconds the switch has learned the locations of most hosts and can be more selective.

Since not all packets are forwarded among networks, each segment of cable that connects to a switch is less saturated with traffic than it would be if all machines were on the same cable. Given that most communication tends to be localized, the increase in apparent bandwidth can be dramatic. And since the logical model of the network is not affected by the presence of a switch, few administrative consequences result from installing one.

Switches can sometimes become confused if your network contains loops. The confusion arises because packets from a single host appear to be on two (or more) ports of the switch. A single Ethernet cannot have loops, but as you connect several Ethernets with routers and switches, the topology can include multiple paths to a host. Some switches can handle this situation by holding alternative routes in reserve in case the primary route goes down. They prune the network they see until the remaining sections present only one path to each node on the network. Some switches can also handle duplicate links between the same two networks and route traffic in a round robin fashion.

Switches must scan every packet to determine if it should be forwarded. Their performance is usually measured by both the packet scanning rate and the packet forwarding rate. Many vendors do not mention packet sizes in the performance figures they quote, and thus, actual performance might be less than advertised.

Although Ethernet switching hardware is getting faster all the time, it is still not a reasonable technology for connecting more than a hundred hosts in a single logical segment. Problems such as "broadcast storms" often plague large switched networks since broadcast traffic must be forwarded to all ports in a switched segment. To solve this problem, use a router to isolate broadcast traffic between switched segments, thereby creating more than one logical Ethernet.

Choosing a switch can be difficult. The switch market is a highly competitive segment of the computer industry, and it's plagued with marketing claims that aren't even partially true. When selecting a switch vendor, rely on independent evaluations rather than on data supplied by vendors themselves. In recent years, it has been common for one vendor to have the "best" product for a few months but then completely destroy its performance or reliability when trying to make improvements, thus elevating another manufacturer to the top of the heap.

In all cases, make sure that the backplane speed of the switch is adequate—that's the number that really counts at the end of a long day. A well-designed switch should have a backplane speed that exceeds the sum of the speeds of all its ports.

VLAN-capable switches

Large sites can benefit from switches that partition their ports (through software configuration) into subgroups called virtual local area networks or VLANs. A VLAN is a group of ports that belong to the same logical segment, as if the ports were connected to their own dedicated switch. Such partitioning increases the ability of the switch to isolate traffic, and that capability has beneficial effects on both security and performance.

Traffic among VLANs is handled by a router, or in some cases, by a layer 3 routing module or routing software layer within the switch. An extension of this system known as "VLAN trunking" (such as is specified by the IEEE 802.1Q protocol) allows physically separate switches to service ports on the same logical VLAN.

It's important to note that VLANs alone provide little additional security. You must filter the traffic among VLANs to reap any potential security benefit.

Routers

Routers (aka "layer 3 switches") direct traffic at the network layer, layer 3 of the OSI network model. They shuttle packets to their final destinations in accordance with the information in the TCP/IP protocol headers. In addition to simply moving packets from one place to another, routers can also perform other functions such as packet filtering (for security), prioritization (for quality of service), and big-picture network topology discovery. See Chapter 15 for all the gory details of how routing actually works.

Routers take one of two forms: fixed configuration and modular.

- Fixed configuration routers have network interfaces permanently installed at the factory. They are usually suitable for small, specialized applications. For example, a router with a T1 interface and an Ethernet interface might be a good choice to connect a small company to the Internet.
- Modular routers have a slot or bus architecture to which interfaces can be added by the end user. Although this approach is usually more expensive, it ensures greater flexibility down the road.

Depending on your reliability needs and expected traffic load, a dedicated router might or might not be cheaper than a UNIX or Linux system configured to act as a router. However, a dedicated router usually achieves superior performance and reliability. This is one area of network design in which it’s usually advisable to spend extra money up front to avoid headaches later.

Autonegotiation

With the introduction of a variety of Ethernet standards came the need for devices to figure out how their neighbors were configured and to adjust their settings accordingly. For example, the network won't work if one side of a link thinks the network is running at 1 Gb/s and the other side of the link thinks it's running at 10 Mb/s. The Ethernet autonegotiation feature of the IEEE standards is supposed to detect and solve this problem. And in some cases, it does. In other cases, it is easily misapplied and simply compounds the problem.

The two golden rules of autonegotiation are these:

- You *must* use autonegotiation on all interfaces capable of 1 Gb/s or above. It's required by the standard.
- On interfaces limited to 100 Mb/s or below, you must either configure *both* ends of a link in autonegotiation mode, or you must *manually* configure the speed and duplex (half vs. full) on *both* sides. If you configure only one side in autonegotiation mode, that side will not (in most cases) "learn" how the other side has been configured. The result will be a configuration mismatch and poor performance.

To see how to set a network interface's autonegotiation policy, see the system-specific sections in the *TCP/IP Networking* chapter; they start on this page.

Power over Ethernet

Power over Ethernet (PoE) is an extension of UTP Ethernet (standardized as IEEE 802.3af) that transmits power to devices over the same UTP cable that carries the Ethernet signal. It's especially handy for Voice over IP (VoIP) telephones or wireless access points (to name just two examples) that need a relatively small amount of power in addition to a network connection.

The power supply capacity of PoE systems has been stratified into four classes that range from 3.84 to 25.5 watts. Never satisfied, the industry is currently working on a higher power standard (802.3bt) that may provide more than 100 watts. Won't it be convenient to operate an Easy-Bake Oven off the network port in the conference room?

For those of you that are wondering: yes, it is possible to boot a small Linux system off a PoE port. Perhaps the simplest option is a Raspberry Pi with an add-on Pi PoE Switch HAT board.

PoE has two ramifications that are significant for sysadmins:

- You need to be aware of PoE devices in your infrastructure so that you can plan the availability of PoE-capable switch ports accordingly. They are more expensive than non-PoE ports.
- The power budget for data closets that house PoE switches must include the wattage of the PoE devices. Note that you don't have to budget the same amount of extra cooling for the closet because most of the heat generated by the consumption of PoE power is dissipated outside the closet (usually, in an office).

Jumbo frames

Ethernet is standardized for a typical packet size of 1,500 bytes (1,518 with framing), a value chosen long ago when networks were slow and memory for buffers was scarce. Today, these 1,500-byte packets look shrimpy in the context of a gigabit Ethernet. Because every packet consumes overhead and introduces latency, network throughput can be higher if larger packet sizes are allowed.

Unfortunately, the original IEEE standards for the various types of Ethernet forbid large packets because of interoperability concerns. But just as highway traffic often mysteriously flows faster than the stated speed limit, king-size Ethernet packets are a common sight on today's networks. Egged on by customers, most manufacturers of network equipment have built support for large frames into their gigabit products.

To use these so-called jumbo frames, all you need do is bump up your network interfaces' MTUs. Throughput gains vary with traffic patterns, but large transfers over TCP (e.g., NFSv4 or SMB file service) benefit the most. Expect a modest but measurable improvement on the order of 10%.

Be aware of these points, though:

- All network equipment on a subnet must support and use jumbo frames, including switches and routers. You cannot mix and match.
- Because jumbo frames are nonstandard, you usually have to enable them explicitly. Devices may accept jumbo frames by default, but they probably will not generate them.
- Since jumbo frames are a form of outlawry, there's no universal consensus on exactly how large a jumbo frame can or should be. The most common value is 9,000 bytes, or 9,018 bytes with framing. You'll have to investigate your devices to determine the largest packet size they have in common. Frames larger than 9K or so are sometimes called "super jumbo frames," but don't be scared off by the extreme-sounding name. Larger is generally better, at least up to 64K or so.

We endorse the use of jumbo frames on gigabit Ethernets, but be prepared to do some extra debugging if things go wrong. It's perfectly reasonable to deploy new networks with the default MTU and convert to jumbo frames later once the reliability of the underlying network has been confirmed.

14.2 Wireless: Ethernet for nomads

A wireless network consists of wireless access points (WAPs, or simply APs) and wireless clients. WAPs can be connected to traditional wired networks (the typical configuration) or wirelessly connected to other access points, a configuration known as a “wireless mesh.”

Wireless standards

The common wireless standards today are IEEE 802.11g, 802.11n, and 802.11ac. 802.11g operates in the 2.4 GHz frequency band and affords LAN-like access at up to 54 Mb/s. Operating range varies from 100 meters to 40 kilometers, depending on equipment and terrain.

802.11n delivers up to 600 Mb/s of bandwidth and can use both the 5 GHz frequency band and the 2.4 GHz band (though 5 GHz is recommended for deployment). Typical operating range is approximately double that of 802.11g.

The 600 Mb/s bandwidth of 802.11n is largely theoretical. In practice, bandwidth in the neighborhood of 400 Mb/s is a more realistic expectation for an optimized configuration. The environment and capabilities of client devices explain most of the difference between theoretical and real-life throughput. When it comes to wireless, “your mileage may vary” always applies!

802.11ac is an extension of 802.11n with support for up to 1 Gb/s of multistation throughput.

All these standards are covered by the generic term “Wi-Fi.” In theory, the Wi-Fi label is restricted to Ethernet implementations from the IEEE 802.11 family. However, that’s the only kind of wireless Ethernet hardware you can actually buy, so all wireless Ethernet is Wi-Fi.

Today, 802.11g and 802.11n are commonplace. The transceivers are inexpensive and are built into most laptops. Add-in cards are widely and cheaply available for desktop PCs, too.

Wireless client access

You can configure a UNIX or Linux box to connect to a wireless network as a client if you have the right hardware and driver. Since most PC-based wireless cards are still designed for Microsoft Windows, they might not come from the factory with FreeBSD or Linux drivers.

When attempting to add wireless connectivity to a FreeBSD or Linux system, you'll likely need these commands:

- **ifconfig** – configure a wireless network interface
- **iwlist** – list available wireless access points
- **iwconfig** – configure wireless connection parameters
- **wpa_supplicant** – authenticate to a wireless (or wired 802.1x) network

Unfortunately, the industry's frantic scramble to sell low-cost hardware often means that getting a wireless adapter to work correctly under UNIX or Linux might require hours of trial and error. Plan ahead, or buy the same adapter that someone else on the Internet has had good luck with on the same OS version you're running.

Wireless infrastructure and WAPs

Everyone wants wireless everything everywhere, and a wide variety of products are available to provide wireless service. But as with so many things, you get what you pay for. Inexpensive devices often meet the needs of home users but fail to scale well in an enterprise environment.

Wireless topology

WAPs are usually dedicated appliances that consist of one or more radios and some form of embedded network operating system, often a stripped-down version of Linux. A single WAP can provide a connection point for multiple clients, but not for an unlimited number of clients. A good rule of thumb is to serve no more than forty simultaneous clients from a single enterprise-grade WAP. Any device that communicates through a wireless standard supported by your WAPs can act as a client.

WAPs are configured to advertise one or more "service set identifiers," aka SSIDs. The SSID acts as the name of a wireless LAN and must be unique within a particular area. When a client wants to connect to a wireless LAN, it listens to see what SSIDs are being advertised and lets the user select from among these networks.

You can choose to name your SSID something helpful and easy to remember such as "Third Floor Public," or you can get creative. Some of our favorite SSID names are

- FBI Surveillance Van
- The Promised LAN
- IP Freely
- Get Off My LAN
- Virus Distribution Center
- Access Denied

Nothing better than geeks at play... In the simplest scenarios, a WAP advertises a single SSID, your client connects to that SSID, and voila! You're on the network.

However, few aspects of wireless networking are truly simple. What if your house or building is too big to be served by a single WAP? Or what if you need to provide different networks to different groups of users (such as employees vs. guests)? For these cases, you need to strategically structure your wireless network.

You can use multiple SSIDs to break up groups of users or functions. Typically, you map them to separate VLANs, which you can then route or filter as desired, just like wired networks.

The frequency spectrum allocated to 802.11 wireless is broken up into bands, commonly called channels. Left on its own, a WAP selects a quiet radio channel to advertise an SSID. Clients and the WAP then use that channel for communication, forming a single broadcast domain. Nearby

WAPs will likely choose other channels so as to maximize available bandwidth and minimize interference.

The theory is that as clients move around the environment, they will dissociate from one WAP when its signal becomes weak and connect to a closer WAP with a stronger signal. Theory and reality often don't cooperate, however. Many clients hold onto weak WAP signals with a death grip and ignore better options.

In most situations, you should allow WAPs to automatically select their favorite channels. If you must manually interfere with this process and are using 802.11b/g/n, consider selecting channel 1, 6, or 11. The spectrum allocated to these channels does not overlap, so combinations of these channels create the greatest likelihood of a wide-open wireless highway. The default channels for 802.11a/ac don't overlap at all, so just pick your favorite number.

Some WAPs have multiple antennas and take advantage of multiple-input, multiple-output technology (MIMO). This practice can increase available bandwidth by exploiting multiple transmitters and receivers to take advantage of signal offsets resulting from propagation delay. The technology can provide a slight performance improvement in some situations, though probably not as much improvement as the dazzling proliferation of antennas might lead you to expect.

If you need a physically larger coverage area, then deploy multiple WAPs. If the area is completely open, you can deploy them in a grid structure. If the physical plant includes walls and other obstructions, you may want to invest in a professional wireless survey. The survey will identify the best options for WAP placement given the physical attributes of your space.

Small money wireless

We like the products made by Ubiquiti (ubnt.com) for inexpensive, high-performing home networks. Google Wifi is a nice, cloud-managed solution, great if you support remote family members. Another option is to run a stripped down version of Linux (such as OpenWrt or LEDE) on a commercial WAP. See openwrt.org for more information and a list of compatible hardware.

Literally dozens of vendors are hawking wireless access points these days. You can buy them at Home Depot and even at the grocery store. El cheapo access points (those in the $30 range) are likely to perform poorly when handling large file transfers or more than one active client.

Big money wireless

Big wireless means big money. Providing reliable, high-density wireless on a large scale (think hospitals, sports arenas, schools, cities) is a challenge complicated by physical plant constraints, user density, and the pesky laws of physics. For situations like this, you need enterprise-grade wireless gear that's aware of the location and condition of each WAP and that actively adjusts the WAPs' channels, signal strengths, and client associations to yield the best results. These systems usually support transparent roaming, which allows a client's association with a particular VLAN and session to seamlessly follow it as the client moves among WAPs.

Our favorite large wireless platforms are those made by Aerohive and Meraki (the latter now owned by Cisco). These next-generation platforms are managed from the cloud, allowing you to sip martinis on the beach as you monitor your network through a browser. You can even eject individual users from the wireless network from the comfort of your beach chair. Take that, hater!

If you are deploying a wireless network on a large scale, you'll probably need to invest in a wireless network analyzer. We highly recommend the analysis products made by AirMagnet.

Wireless security

The security of wireless networks has traditionally been poor. Wired Equivalent Privacy (WEP) is a protocol used in conjunction with legacy 802.11b networks to encrypt packets traveling over the airwaves. Unfortunately, this standard contains a fatal design flaw that makes it little more than a speed bump for snoopers. Someone sitting outside your building or house can access your network directly and undetectably, usually in under a minute.

More recently, the Wi-Fi Protected Access (WPA) security standards have engendered new confidence in wireless security. Today, WPA (specifically, WPA2) should be used instead of WEP in all new installations. Without WPA2, wireless networks should be considered completely insecure and should never be found inside an enterprise firewall. Don’t even use WEP at home!

To remember that it’s WEP that’s insecure and WPA that’s secure, just remember that WEP stands for Wired Equivalent Privacy. The name is accurate; WEP gives you as much protection as letting someone connect directly to your wired network. (That is, no protection at all—at least at the IP level.)

14.3 SDN: SOFTWARE-DEFINED NETWORKING

Just as with server virtualization, the separation of physical network hardware from the functional architecture of the network can significantly increase flexibility and manageability. The best traction along this path is the software-defined networking (SDN) movement.

The main idea of SDN is that the components managing the network (the control plane) are physically separate from the components that forward packets (the data plane). The data plane is programmable through the control plane, so you can fine-tune or dynamically configure data paths to meet performance, security, and accessibility goals.

As with so many things in our industry, SDN for enterprise networks has become somewhat of a marketing gimmick. The original goal was to standardize vendor-independent ways to reconfigure network components. Although some of this idealism has been realized, many vendors now offer proprietary enterprise SDN products that run somewhat counter to SDN's original purpose. If you find yourself exploring the enterprise SDN space, choose products that conform to open standards and are interoperable with other vendors' products.

For large cloud providers, SDN adds a layer of flexibility that reduces your need to know (or care) where a particular resource is physically located. Although these solutions may be proprietary, they are tightly integrated into cloud providers' platforms and can make configuring your virtual infrastructure effortless.

SDN and its API-driven configuration system offer you, the sysadmin, a tempting opportunity to integrate network topology management with other DevOps-style tools for continuous integration and deployment. Perhaps in some ideal world, you always have a "next at bat" production environment staged and ready to activate with a single click. As the new environment is promoted to production, the network infrastructure magically morphs, eliminating user-visible downtime and the need for you to schedule maintenance windows.

14.4 Network testing and debugging

The key to debugging a network is to break it down into its component parts and then test each piece until you've isolated the offending device or cable. The "idiot lights" on switches and hubs (such as "link status" and "packet traffic") often hold immediate clues to the source of the problem. Top-notch documentation of your wiring scheme is essential for making these indicator lights work in your favor.

As with most tasks, having the right tools for the job is a big part of being able to get the job done right and without delay. The market offers two major types of network debugging tools, although these are quickly converging into unified devices.

The first type of tool is the hand-held cable analyzer. This device can measure the electrical characteristics of a given cable, including its length, with a groovy technology called "time domain reflectrometry." Usually, these analyzers can also point out simple faults such as broken or miswired cables.

Our favorite product for LAN cable analysis is the Fluke LanMeter. It's an all-in-one analyzer that can even perform IP pings across the network. High-end versions have their own web server that can show you historical statistics. For WAN (telco) circuits, the T-BERD line analyzer is the cat's meow. It's made by Viavi (viavisolutions.com).

The second type of debugging tool is the network sniffer. A sniffer captures the bytes that travel across the wire and disassembles network packets to look for protocol errors, misconfigurations, and general snafus. Sniffers operate at the link layer of the network rather than the electrical layer, so they cannot diagnose cabling problems or electrical issues that might be affecting network interfaces.

Commercial sniffers are available, but we find that the freely available program Wireshark running on a fat laptop is usually the best option. See the *Packet sniffers* section for details. (Like so many popular programs, Wireshark is often the target of attacks by hackers. Make sure you stay up to date with the most current version.)

14.5 Building wiring

If you’re embarking on a building wiring project, the most important advice we can give you is to “do it right the first time.” This is not an area in which to skimp or cut corners. Buying quality materials, selecting a competent wiring contractor, and installing extra connections (drops) will save you years of frustration and heartburn down the road.

UTP cabling options

Category 6a wire typically offers the best price vs. performance tradeoff in today's market. Its normal format is four pairs per sheath, which is just right for a variety of data connections from RS-232 to gigabit Ethernet.

Category 6a specifications require that the twist be maintained to the point of contact. Special training and termination equipment are necessary to satisfy this requirement. You must use Category 6a jacks and patch panels. We've had the best luck with parts manufactured by Siemon.

Connections to offices

For many years, there has been an ongoing debate about how many connections should be wired per office. One connection per office is clearly not enough. But should you use two or four? With the advent of high-bandwidth wireless, we now recommend two, for several reasons:

- A nonzero number of wired connections is typically needed to support voice telephones and other specialty devices.
- Most user devices can now be connected through wireless networking, and users prefer this to being chained down by cables.
- Your network wiring budget is better spent on core infrastructure (fiber to closets, etc.) than on more drops to individual offices.

If you're in the process of wiring your entire building, you might consider installing a few outlets in hallways, conference rooms, lunch rooms, bathrooms, and of course, ceilings (for wireless access points). Don't forget to keep security in mind, however, and put publicly accessible ports on a "guest" VLAN that doesn't have access to your internal network resources. You can also secure public ports by implementing 802.1x authentication.

Wiring standards

Modern buildings often require a large and complex wiring infrastructure to support all the various activities that take place inside. Walking into the average telecommunications closet can be a shocking experience for the weak of stomach, as identically colored, unlabeled wires often cover the walls.

In an effort to increase traceability and standardize building wiring, the Telecommunications Industry Association in February 1993 released TIA/EIA-606 (*Administration Standard for Commercial Telecommunications Infrastructure*), later updated to TIA/EIA-606-B in 2012.

EIA-606 specifies requirements and guidelines for the identification and documentation of telecommunications infrastructure. Items covered by EIA-606 include

- Termination hardware
- Cables
- Cable pathways
- Equipment spaces
- Infrastructure color coding
- Labeling requirements
- Symbols for standard components

In particular, the standard specifies colors to be used for wiring. Table 14.5 shows the details.

Table 14.5: EIA-606 color chart

Termination type	Color	Code[a]	Comments
Demarcation point	Orange	150C	Central office terminations
Network connections	Green	353C	Also used for aux circuit terminations
Common equipment[b]	Purple	264C	Major switching/data eqpt. terminations
First-level backbone	White	–	Cable terminations
Second-level backbone	Gray	422C	Cable terminations
Station	Blue	291C	Horizontal cable terminations
Interbuilding backbone	Brown	465C	Campus cable terminations
Miscellaneous	Yellow	101C	Maintenance, alarms, etc.
Key telephone systems	Red	184C	–

a. Pantone Matching System color code
b. PBXes, hosts, LANs, muxes, etc.

Pantone sells software to map between the Pantone systems for ink-on-paper, textile dyes, and colored plastic. Hey, you could color-coordinate the wiring, the uniforms of the installers, and the wiring documentation! On second thought…

14.6 Network design issues

This section addresses the logical and physical design of networks. It's targeted at medium-sized installations. The ideas presented here scale up to a few hundred hosts but are overkill for three machines and inadequate for thousands. We also assume that you have an adequate budget and are starting from scratch, which is probably only partially true.

Most of network design consists of specifying

- The types of media that will be used
- The topology and routing of cables
- The use of switches and routers

Another key issue in network design is congestion control. For example, file-sharing protocols such as NFS and SMB tax the network quite heavily, and so file serving on a backbone cable is undesirable.

The issues presented in the following sections are typical of those that must be considered in any network design.

Network architecture vs. building architecture

Network architecture is usually more flexible than building architecture, but the two must coexist. If you are lucky enough to be able to specify the network before a building is constructed, be lavish. For most of us, both the building and a facilities-management department already exist and are somewhat rigid.

In existing buildings, the network must use the building architecture, not fight it. Modern buildings often contain utility raceways for data and telephone cables in addition to high-voltage electrical wiring and water or gas pipes. They often use drop ceilings, a boon to network installers. Many campuses and organizations have underground utility tunnels that facilitate network installation.

The integrity of firewalls must be maintained; if you route a cable through a firewall, the hole must be snug and filled in with a noncombustible substance. (This type of firewall is a concrete, brick, or flame-retardant wall that prevents a fire from spreading and burning down the building. Although different from a network security firewall, it's probably just as important.)

Respect return air plenums in your choice of cable. If you are caught violating fire codes, you might be fined and will be required to fix the problems you have created, even if that means tearing down the entire network and rebuilding it correctly.

Your network's logical design must fit into the physical constraints of the buildings it serves. As you specify the network, keep in mind that it's easy to draw a logically good solution and then find that it is physically difficult or impossible to implement.

Expansion

It’s difficult to predict needs ten years into the future, especially in the computer and networking fields. Therefore, design the network with expansion and increased bandwidth in mind. As you install cable, especially in out-of-the-way, hard-to-reach places, pull three to four times the number of pairs you actually need. Remember: the majority of installation cost is labor, not materials.

Even if you have no plans to use fiber, it’s wise to install some when wiring your building, especially in situations where it will be hard to install cable later. Run both multimode and single-mode fiber. The kind you need in the future is always the kind you didn’t install.

Congestion

A network is like a chain: it is only as good as its weakest or slowest link. The performance of Ethernet, like that of many other network architectures, degrades nonlinearly as the network becomes loaded.

Overtaxed switches, mismatched interfaces, and low-speed links can all lead to congestion. It’s helpful to isolate local traffic by creating subnets and by using interconnection devices such as routers. Subnets can also be used to cordon off machines that are used for experimentation. It’s difficult to run an experiment that involves several machines if you cannot isolate those machines both physically and logically from the rest of the network.

Maintenance and documentation

We have found that the maintainability of a network correlates highly with the quality of its documentation. Accurate, complete, up-to-date documentation is indispensable.

Cables should be labeled at all termination points. It’s a good idea to post copies of local cable maps inside communications closets so that the maps can be updated on the spot when changes are made. Once every few weeks, have someone copy down the changes for entry into a wiring database.

Joints between major population centers in the form of switches or routers can facilitate debugging by allowing parts of the network to be isolated and debugged separately. It’s also helpful to put joints between political and administrative domains, for similar reasons.

14.7 Management issues

If a network is to work correctly, some things must be centralized, some distributed, and some local. Reasonable ground rules and “good citizen” guidelines must be formulated and agreed on.

A typical environment includes

- A backbone network among buildings
- Departmental subnets connected to the backbone
- Group subnets within a department
- Connections to the outside world (Internet or field office VPNs)

Several facets of network design and implementation must have site-wide control, responsibility, maintenance, and financing. Networks with chargeback algorithms for each connection grow in bizarre but predictable ways as departments try to minimize their own local costs. Prime targets for central control are

- The network design, including the use of subnets, routers, switches, etc.
- The backbone network itself, including the connections to it
- Host IP addresses, hostnames, and subdomain names
- Protocols, mostly to ensure their interoperation
- Routing policy to the Internet

Domain names, IP addresses, and network names are in some sense already controlled centrally by authorities such as ARIN (the American Registry for Internet Numbers) and ICANN. However, your site’s use of these items must be coordinated locally as well.

A central authority has an overall view of the network: its design, capacity, and expected growth. It can afford to own monitoring equipment (and the staff to run it) and to keep the backbone network healthy. It can insist on correct network design, even when that means telling a department to buy a router and build a subnet to connect to the campus backbone. Such a decision might be necessary to ensure that a new connection does not adversely impact the existing network.

If a network serves many types of machines, operating systems, and protocols, it is almost essential to have a layer 3 device as a gateway between networks.

14.8 Recommended vendors

In the past 30+ years of installing networks around the world, we've gotten burned more than a few times by products that didn't quite meet specs or were misrepresented, overpriced, or otherwise failed to meet expectations. Below is a list of vendors in the United States that we still trust, recommend, and use ourselves today.

Cables and connectors

AMP (part of Tyco)
(800) 522-6752
amp.com

Anixter
(800) 264-9837
anixter.com

Black Box Corporation
(724) 746-5500
blackbox.com

Belden Cable
(800) 235-3361
(765) 983-5200
belden.com

Siemon
(860) 945-4395
siemon.com

Newark Electronics
(800) 463-9275
newark.com

Test equipment

Fluke	Siemon	Viavi
(800) 443-5853	(860) 945-4395	(844) 468-4284
fluke.com	siemon.com	viavisolutions.com

Routers/switches

Cisco Systems	Juniper Networks
(415) 326-1941	(408) 745-2000
cisco.com	juniper.net

14.9 Recommended reading

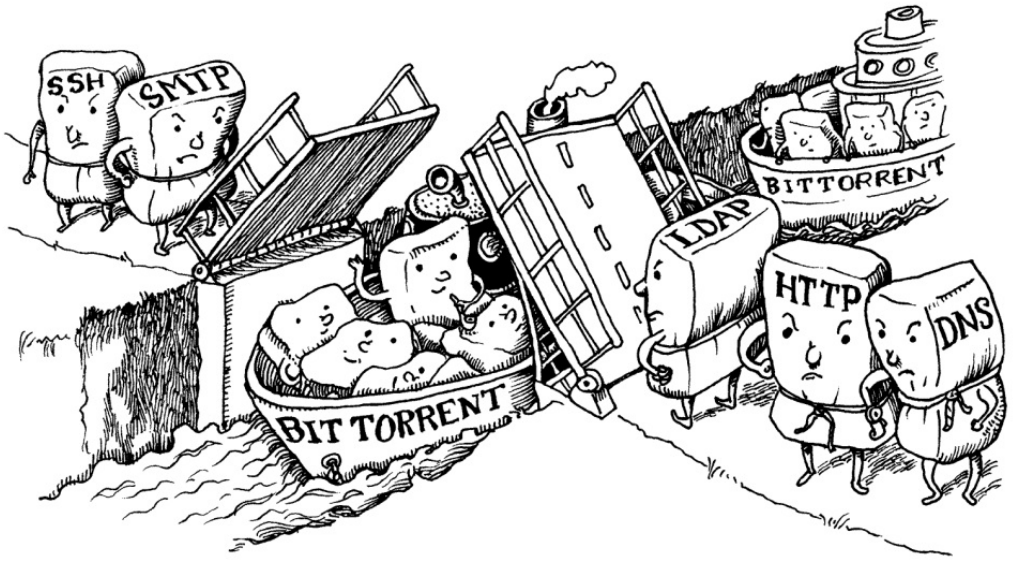
ANSI/TIA/EIA-568-A, *Commercial Building Telecommunications Cabling Standard*, and ANSI/TIA/EIA-606, *Administration Standard for the Telecommunications Infrastructure of Commercial Buildings*, are the telecommunication industry's standards for building wiring. Unfortunately, they are not free. See tiaonline.org.

Barnett, David, David Groth, and Jim McBee. *Cabling: The Complete Guide to Network Wiring (3rd Edition)*. San Francisco, CA: Sybex, 2004.

Goransson, Paul, and Chuck Black. *Software Defined Networks, A Comprehensive Approach (2nd Edition)*. Burlington, MA: Morgan Kaufman, 2016.

Spurgeon, Charles, and Joann Zimmerman. *Ethernet: The Definitive Guide: Designing and Managing Local Area Networks (2nd Edition)*. Sebastopol, CA: O'Reilly, 2014.

15 IP Routing



More than 4.3 billion IP addresses are available world-wide, so getting packets to the right place on the Internet is no easy task. Chapter 13, *TCP/IP Networking*, briefly introduced IP packet forwarding. In this chapter, we examine the forwarding process in more detail and investigate several network protocols that allow routers to automatically discover efficient routes. Routing protocols not only lessen the day-to-day administrative burden of maintaining routing information, but they also allow network traffic to be redirected quickly if a router, link, or network should fail.

It's important to distinguish between the process of actually forwarding IP packets and the management of the routing table that drives this process, both of which are commonly called "routing." Packet forwarding is simple, whereas route computation is tricky; consequently, the second meaning is used more often in practice. This chapter describes only unicast routing; multicast routing (sending packets to groups of subscribers) involves an array of very different problems and is beyond the scope of this book.

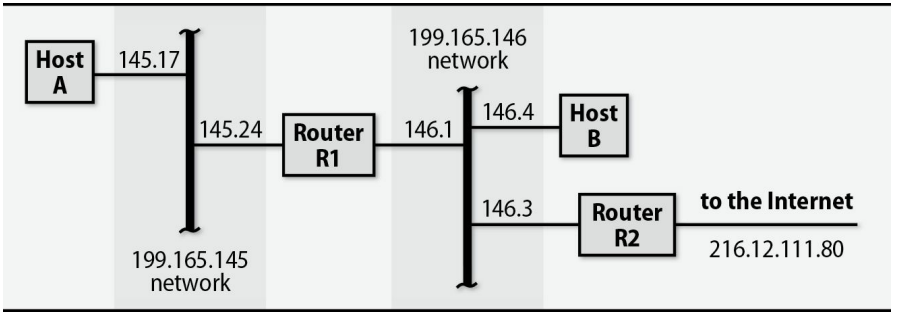
For most cases, the information covered in Chapter 13 is all you need to know about routing. If the appropriate network infrastructure is already in place, you can set up a single static default route (as described in the *Routing* section) and voilà, you have enough information to reach just about anywhere on the Internet. If you must live within a complex network topology, or if you are using UNIX or Linux systems as part of your network infrastructure, then this chapter's information about dynamic routing protocols and tools can come in handy. (However, we do not recommend the use of UNIX or Linux systems as network routers in a production infrastructure. Buy a dedicated router.)

IP routing (both for IPv4 and for IPv6) is “next hop” routing. At any given point, the system handling a packet needs to determine only the *next* host or router in the packet’s journey to its final destination. This is a different approach from that of many legacy protocols, which determine the exact path a packet will travel before it leaves its originating host, a scheme known as source routing. (IP packets can also be source-routed—at least in theory—but this is almost never done. The feature is not widely supported because of security considerations.)

15.1 PACKET FORWARDING: A CLOSER LOOK

Before we jump into the management of routing tables, we need a more detailed look at how the tables are used. Consider the network shown in Exhibit A.

Exhibit A: Example network



For simplicity, we start this example with IPv4; for an IPv6 routing table, see this page.

Router R1 connects two networks, and router R2 connects one of these nets to the outside world. A look at the routing tables for these hosts and routers lets us examine some specific packet forwarding scenarios. First, host A's routing table:

```
A$ netstat -rn
Destination     Gateway         Genmask         Flags MSS Window irtt Iface
127.0.0.0       0.0.0.0         255.0.0.0       U     0   0      0    lo
199.165.145.0   0.0.0.0         255.255.255.0   U     0   0      0    eth0
0.0.0.0         199.165.145.24  0.0.0.0         UG    0   0      0    eth0
```

The example above uses the venerable **netstat** tool to query the routing table. This tool is distributed with FreeBSD and is available for Linux as part of the **net-tools** package. **net-tools** is no longer actively maintained, and as a result it is considered deprecated. The less featureful **ip route** command is the officially recommended way to obtain this information on Linux:

```
A$ ip route
default via 199.165.145.24 dev eth0 onlink
199.165.145.0/24 dev eth0  proto kernel  scope link  src 199.165.145.17
```

The output from **netstat -rn** is slightly easier to read, so we use that for subsequent examples and for the following exploration of Exhibit A.

Host A has the simplest routing configuration of the four machines. The first two routes describe the machine's own network interfaces in standard routing terms. These entries exist so that forwarding to directly connected networks need not be handled as a special case. eth0 is host A's

Ethernet interface, and lo is the loopback interface, a virtual interface emulated in software. Entries such as these are normally added automatically when a network interface is configured.

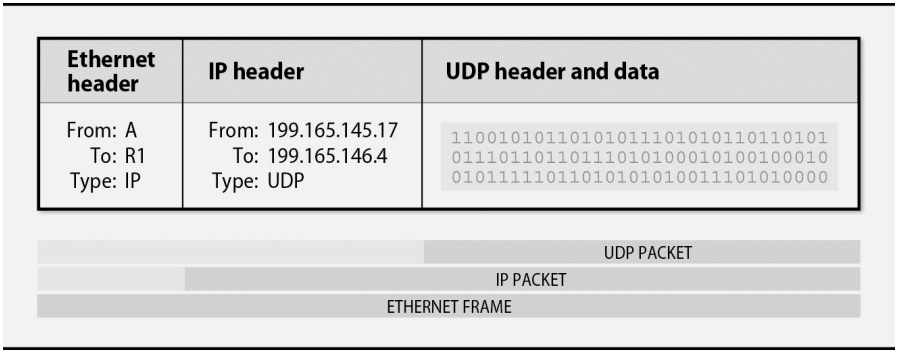
See the discussion of netmasks starting on this page.

The default route on host A forwards all packets not addressed to the loopback address or to the 199.165.145 network to the router R1, whose address on this network is 199.165.145.24. Gateways must be only one hop away.

See this page for more information about addressing.

Suppose a process on A sends a packet to B, whose address is 199.165.146.4. The IP implementation looks for a route to the target network, 199.165.146, but none of the routes match. The default route is invoked and the packet is forwarded to R1. Exhibit B shows the packet that actually goes out on the Ethernet. The addresses in the Ethernet header are the MAC addresses of A's and R1's interfaces on the 145 net.

Exhibit B: Ethernet packet



The Ethernet destination address is that of router R1, but the IP packet hidden within the Ethernet frame does not mention R1 at all. When R1 inspects the packet it has received, it sees from the IP destination address that it is not the ultimate destination of the packet. It then uses its own routing table to forward the packet to host B without rewriting the IP header; the header still shows the packet coming from A.

Here's the routing table for host R1:

```
R1$ netstat -rn
Destination     Gateway         Genmask         Flags MSS Window irtt Iface
127.0.0.0       0.0.0.0         255.0.0.0       U     0   0      0    lo
199.165.145.0   0.0.0.0         255.255.255.0   U     0   0      0    eth0
199.165.146.0   0.0.0.0         255.255.255.0   U     0   0      0    eth1
0.0.0.0         199.165.146.3   0.0.0.0         UG    0   0      0    eth1
```

This table is similar to that of host A, except that it shows two physical network interfaces. The default route in this case points to R2, since that's the gateway through which the Internet can be reached. Packets bound for either of the 199.165 networks can be delivered directly.

Like host A, host B has only one real network interface. However, B needs an additional route to function correctly because it has direct connections to two different routers. Traffic for the 199.165.145 net must travel through R1, but other traffic should go out to the Internet through R2.

```
B$ netstat -rn
Destination     Gateway         Genmask         Flags MSS Window irtt Iface
127.0.0.0       0.0.0.0         255.0.0.0       U     0   0      0    lo
199.165.145.0   199.165.146.1   255.255.255.0   U     0   0      0    eth0
199.165.146.0   0.0.0.0         255.255.255.0   U     0   0      0    eth0
0.0.0.0         199.165.146.3   0.0.0.0         UG    0   0      0    eth0
```

See [this page] for an explanation of ICMP redirects.

In theory, you can configure host B with initial knowledge of only one gateway and rely on help from ICMP redirects to eliminate extra hops. For example, here is one possible initial configuration for host B:

```
B$ netstat -rn
Destination     Gateway         Genmask          Flags MSS Window irtt Iface
127.0.0.0       0.0.0.0         255.0.0.0        U     0   0      0    lo
199.165.146.0   0.0.0.0         255.255.255.0    U     0   0      0    eth0
0.0.0.0         199.165.146.3 0.0.0.0          UG    0   0      0    eth0
```

If B then sends a packet to host A (199.165.145.17), no route matches and the packet is forwarded to R2 for delivery. R2 (which, being a router, presumably has complete information about the network) sends the packet on to R1. Since R1 and B are on the same network, R2 also sends an ICMP redirect notice to B, and B enters a host route for A into its routing table:

```
199.165.145.17  199.165.146.1 255.255.255.255 UGHD  0   0      0    eth0
```

This route sends all future traffic for A directly through R1. However, it does not affect routing for other hosts on A's network, all of which have to be routed by separate redirects from R2.

Some sites use ICMP redirects this way as a sort of low-rent routing “protocol,” thinking that this approach is dynamic. Unfortunately, systems and routers all handle redirects differently. Some hold on to them indefinitely. Others remove them from the routing table after a relatively short period (5–15 minutes). Still others ignore them entirely, which is probably the correct approach from a security perspective.

Redirects have several other potential disadvantages: increased network load, increased load on R2, routing table clutter, and dependence on extra servers, to name a few. Therefore, we don’t recommend their use. In a properly configured network, redirects should never appear in the routing table.

If you are using IPv6 addresses, the same model applies. Here’s a routing table from a FreeBSD host that is running IPv6:

```
$ netstat -rn
Destination          Gateway            Flags  Netif Expire
default              2001:886b:4452::1  UGS     re0
2001:886b:4452::/64  link#1             U       re0
fe80::/10            ::1                UGRS    lo0
fe80::%re0/64        link#1             U       re0
```

As in IPv4, the first route is a default that’s used when no more-specific entries match. The next line contains a route to the global IPv6 network where the host lives, 2001:886b:4452::/64. The final two lines are special; they represent a route to the reserved IPv6 network fe80, known as the link-local unicast network. This network is used for traffic that is scoped to the local broadcast domain (typically, the same physical network segment). It is most often used by network services that need to find each other on a unicast network, such as OSPF. Don’t use link-local addresses for normal networking purposes.

15.2 Routing daemons and routing protocols

In simple networks such as the one shown in Exhibit A, it is perfectly reasonable to configure routing by hand. At some point, however, networks become too complicated to be managed this way. Instead of having to explicitly tell every computer on every network how to reach every other computer and network, it would be nice if the computers could just cooperate and figure it all out. This is the job of routing protocols and the daemons that implement them.

Routing protocols have a major advantage over static routing systems in that they can react and adapt to changing network conditions. If a link goes down, then the routing daemons can discover and propagate alternative routes to the networks served by that link, if any such routes exist.

Routing daemons collect information from three sources: configuration files, the existing routing tables, and routing daemons on other systems. This information is merged to compute an optimal set of routes, and the new routes are then fed back into the system routing table (and possibly fed to other systems through a routing protocol). Because network conditions change over time, routing daemons must periodically check in with one another for reassurance that their routing information is still current.

The exact manner in which routes are computed depends on the routing protocol. Two general types of protocols are in common use: distance-vector protocols and link-state protocols.

Distance-vector protocols

Distance-vector (aka "gossipy") protocols are based on the general idea, "If router X is five hops away from network Y, and I'm adjacent to router X, then I must be six hops away from network Y." You announce how far you think you are from the networks you know about. If your neighbors don't know of a better way to get to each network, they mark you as being the best gateway. If they already know a shorter route, they ignore your advertisement. Over time, everyone's routing tables are supposed to converge to a steady state.

This is a really elegant idea. If it worked as advertised, routing would be relatively simple. Unfortunately, the basic algorithm does not deal well with changes in topology. The problem is that changes in topology can lengthen the optimal routes. Some DV protocols, such as EIGRP, maintain information about multiple possible routes so that they always have a fallback plan. The exact details are not important.

In some cases, infinite loops (e.g., router X receives information from router Y and sends it on to router Z, which sends it back to router Y) can prevent routes from converging at all. Real-world distance-vector protocols must avoid such problems by introducing complex heuristics or by enforcing arbitrary restrictions such as the RIP (Routing Information Protocol) notion that any network more than 15 hops away is unreachable.

Even in nonpathological cases, it can take many update cycles for all routers to reach a steady state. Therefore, to guarantee that routing does not jam for an extended period, the cycle time must be made short, and for this reason distance-vector protocols as a class tend to be talkative. For example, RIP requires that routers broadcast all their routing information every 30 seconds. EIGRP sends updates every 90 seconds.

On the other hand, BGP, the Border Gateway Protocol, transmits the entire table once and then transmits changes as they occur. This optimization substantially reduces the potential for "chatty" (and mostly unnecessary) traffic.

Table 15.1 lists the distance-vector protocols in common use today.

Table 15.1: Common distance-vector routing protocols

Name	Long name	Application
RIP	Routing Information Protocol	Internal LANs (if that)
RIPng	Routing Information Protocol, next generation	IPv6 LANs
EIGRP [a]	Enhanced Interior Gateway Routing Protocol	WANs, corporate LANs
BGP	Border Gateway Protocol	Internet backbone routing

a. This protocol (EIGRP) is proprietary to Cisco.

Link-state protocols

Link-state protocols distribute information in a relatively unprocessed form. The records traded among routers are of the form “Router X is adjacent to router Y, and the link is up.” A complete set of such records forms a connectivity map of the network from which each router can compute its own routing table. The primary advantage that link-state protocols offer over distance-vector protocols is the ability to quickly converge on an operational routing solution after a catastrophe occurs. The tradeoff is that maintaining a complete map of the network at each node requires memory and CPU power that would not be needed by a distance-vector routing system.

Because the communications among routers in a link-state protocol are not part of the actual route-computation algorithm, they can be implemented in such a way that transmission loops do not occur. Updates to the topology database propagate across the network efficiently, at a lower cost in network bandwidth and CPU time.

Link-state protocols tend to be more complicated than distance-vector protocols, but this complexity can be explained in part by the fact that link-state protocols make it easier to implement advanced features such as type-of-service routing and multiple routes to the same destination.

The only true link-state protocol in general use is OSPF.

Cost metrics

For a routing protocol to determine which path to a network is shortest, the protocol has to define what is meant by “shortest.” Is it the path involving the fewest number of hops? The path with the lowest latency? The largest minimal intermediate bandwidth? The lowest financial cost?

For routing, the quality of a link is represented by a number called the cost metric. A path cost is the sum of the costs of each link in the path. In the simplest systems, every link has a cost of 1, leading to hop counts as a path metric. But any of the considerations mentioned above can be converted to a numeric cost metric.

Routing protocol designers have labored long and hard to make the definition of cost metrics flexible, and some protocols even allow different metrics to be used for different kinds of network traffic. Nevertheless, in 99% of cases, all this hard work can be safely ignored. The default metrics for most systems work just fine.

You might encounter situations in which the actual shortest path to a destination is not a good default route for political or financial reasons. To handle these cases, you can artificially boost the cost of the critical links to make them seem less appealing. Leave the rest of the routing configuration alone.

Interior and exterior protocols

An “autonomous system” (AS) is a group of networks under the administrative control of a single entity. The definition is vague; real-world autonomous systems can be as large as a world-wide corporate network or as small as a building or a single academic department. It all depends on how you want to manage routing. The general tendency is to make autonomous systems as large as you can. This convention simplifies administration and makes routing as efficient as possible.

Routing within an autonomous system is somewhat different from routing between autonomous systems. Protocols for routing among ASs (“exterior” protocols) must often handle routes for many networks (e.g., the entire Internet), and they must deal gracefully with the fact that neighboring routers are under other people’s control. Exterior protocols do not reveal the topology inside an autonomous system, so in a sense they can be thought of as a second level of routing hierarchy that deals with collections of nets rather than individual hosts or cables.

In practice, small- and medium-sized sites rarely need to run an exterior protocol unless they are connected to more than one ISP. With multiple ISPs, the easy division of networks into local and Internet domains collapses, and routers must decide which route to the Internet is best for any particular address. (However, that is not to say that *every* router must know this information. Most hosts can stay stupid and route their default packets through an internal gateway that is better informed.)

Although exterior protocols are not much different from their interior counterparts, this chapter concentrates on the interior protocols and the daemons that support them. If your site must use an external protocol as well, see the recommended reading list here for some suggested references.

15.3 Protocols on parade

Several routing protocols are in common use. In this section, we introduce the major players and summarize their main advantages and weaknesses.

RIP and RIPng: Routing Information Protocol

RIP is an old Xerox protocol that was adapted for IP networks. The IP version was originally specified in RFC1058, circa 1988. The protocol has existed in three versions: RIP, RIPv2, and the IPv6-only RIPng ("next generation").

All versions of RIP are simple distance-vector protocols that use hop counts as a cost metric. Because RIP was designed in an era when computers were expensive and networks small, RIPv1 considers any host 15 or more hops away to be unreachable. Later versions of RIP have maintained the hop-count limit, mostly to encourage the administrators of complex sites to migrate to more sophisticated routing protocols.

See this page for more information about CIDR.

RIPv2 is a minor revision of RIP that distributes netmasks along with next-hop addresses, so its support for subnetted networks and CIDR is better than that of RIPv1. A vague gesture toward increasing the security of RIP was also included.

RIPv2 can be run in a compatibility mode that preserves most of its new features without entirely abandoning vanilla RIP receivers. In most respects, RIPv2 is identical to the original protocol and should be used in preference to it.

See this page for details on IPv6.

RIPng is a restatement of RIP in terms of IPv6. It is an IPv6-only protocol, and RIP remains IPv4-only. If you want to route both IPv4 and IPv6 with RIP, you'll need to run RIP and RIPng as separate protocols.

Although RIP is known for its profligate broadcasting, it does a good job when a network is changing often or when the topology of remote networks is not known. However, it can be slow to stabilize after a link goes down.

It was originally thought that the advent of more sophisticated routing protocols such as OSPF would make RIP obsolete. However, RIP continues to fill a need for a simple, easy-to-implement protocol that doesn't require much configuration, and it works well on low-complexity networks.

RIP is widely implemented on non-UNIX platforms. A variety of common devices, from printers to SNMP-manageable network components, can listen to RIP advertisements to learn about network gateways. In addition, some form of RIP client is available for all versions of UNIX and Linux, so RIP is a de facto lowest-common-denominator routing protocol. Often, RIP is used for LAN routing, and a more featureful protocol is used for wide-area connectivity.

Some sites run passive RIP daemons (usually **routed** or Quagga's **ripd**) that listen for routing updates on the network but do not broadcast any information of their own. The actual route computations are performed with a more efficient protocol such as OSPF (see the next section). RIP is used only as a distribution mechanism.

OSPF: Open Shortest Path First

OSPF is the most popular link-state protocol. “Shortest path first” refers to the mathematical algorithm that calculates routes; “open” is used in the sense of “nonproprietary.” RFC2328 defines the basic protocol (OSPF version 2), and RFC5340 extends it to include support for IPv6 (OSPF version 3). OSPF version 1 is obsolete and is not used.

OSPF is an industrial-strength protocol that works well for large, complicated topologies. It offers several advantages over RIP, including the ability to manage several paths to a single destination and the ability to partition the network into sections (“areas”) that share only high-level routing information. The protocol itself is complex and hence only worthwhile at sites of significant size, where routing protocol behavior really makes a difference. To use OSPF effectively, your site’s IP addressing scheme should be reasonably hierarchical.

The OSPF protocol specification does not mandate any particular cost metric. Cisco’s implementation uses a bandwidth-related value by default.

EIGRP: Enhanced Interior Gateway Routing Protocol

EIGRP is a proprietary routing protocol that runs only on Cisco routers. Its predecessor IGRP was created to address some of the shortcomings of RIP before robust standards like OSPF existed. IGRP has now been deprecated in favor of EIGRP, which accommodates CIDR masks. IGRP and EIGRP are configured similarly despite being quite different in their underlying protocol design.

EIGRP supports IPv6, but as with other routing protocols, the IPv6 world and IPv4 world are configured separately and act as separate, though parallel, routing domains.

EIGRP is a distance-vector protocol, but it’s designed to avoid the looping and convergence problems found in other DV systems. It’s widely regarded as the most evolved distance-vector protocol. For most purposes, EIGRP and OSPF are equally functional.

BGP: Border Gateway Protocol

BGP is an exterior routing protocol; that is, a protocol that manages traffic among autonomous systems rather than among individual networks. There were once several exterior routing protocols in common use, but BGP has outlasted them all.

BGP is now the standard protocol used for Internet backbone routing. As of mid-2017, the Internet routing table contains about 660,000 prefixes. It should be clear from this number that backbone routing has scaling requirements very different from those for local routing.

15.4 Routing protocol multicast coordination

Routers need to talk to each other to learn how to get to places on the network, but to get to places on the network they need to talk to a router. This chicken-and-egg problem is most commonly solved through multicast communication. This is the networking equivalent of agreeing to meet your friend on a particular street corner if you get separated. The process is normally invisible to system administrators, but you might occasionally see this multicast traffic in your packet traces or when doing other kinds of network debugging. Table 15.2 lists the agreed-on multicast addresses for various routing protocols.

Table 15.2: Routing protocol multicast addresses

Description	IPv6	IPv4
All systems on this subnet	ff02::1	224.0.0.1
All routers on this subnet	ff02::2	224.0.0.2
Unassigned	ff02::3	224.0.0.3
DVMRP Routers	ff02::4	224.0.0.4
OSPF Routers	ff02::5	224.0.0.5
OSPF DR Routers	ff02::6	224.0.0.6
RIP Routers	ff02::9	224.0.0.9
EIGRP Routers	ff02::10	224.0.0.10

15.5 Routing strategy selection criteria

Routing for a network can be managed at essentially four levels of complexity:

- No routing
- Static routes only
- Mostly static routes, but clients listen for RIP updates
- Dynamic routing everywhere

The topology of the overall network has a dramatic effect on each individual segment's routing requirements. Different nets might need very different levels of routing support. The following rules of thumb can help you choose a strategy:

- A stand-alone network requires no routing.
- If a network has only one way out, clients (nongateway machines) on that network should have a static default route to the lone gateway. No other configuration is necessary, except perhaps on the gateway itself.
- A gateway with a small number of networks on one side and a gateway to "the world" on the other side can have explicit static routes pointing to the former and a default route to the latter. However, dynamic routing is advisable if both sides have more than one routing choice.
- If networks cross political or administrative boundaries, use dynamic routing at those points, even if the complexity of the networks involved would not otherwise suggest the use of a routing protocol.
- RIP works OK and is widely supported. Don't reject it out of hand just because it's an older protocol with a reputation for chattiness.

  The problem with RIP is that it doesn't scale indefinitely; an expanding network will eventually outgrow it. That fact makes RIP something of a transitional protocol with a narrow zone of applicability. That zone is bounded on one side by networks too simple to require any routing protocol and on the other side by networks too complicated for RIP. If your network plans include continued growth, it's probably reasonable to skip over the "RIP zone" entirely.

- Even when RIP isn't a good choice for your global routing strategy, it's still a good way to distribute routes to leaf nodes. But don't use it where it's not needed: systems on a network that has only one gateway never need dynamic updates.

- EIGRP and OSPF are about equally functional, but EIGRP is proprietary to Cisco. Cisco makes excellent and cost-competitive routers; nevertheless, standardizing on EIGRP limits your choices for future expansion.
- Routers connected to the Internet through multiple upstream providers must use BGP. However, most routers have only one upstream path and can therefore use a simple static default route.

A good default strategy for a medium-sized site with a relatively stable local structure and a connection to someone else's net is to use a combination of static and dynamic routing. Routers within the local structure that do not lead to external networks can use static routing, forwarding all unknown packets to a default machine that understands the outside world and does dynamic routing.

A network that is too complicated to be managed with this scheme should rely on dynamic routing. Default static routes can still be used on leaf nets, but machines on networks with more than one router should run **routed** or some other RIP receiver in passive mode.

15.6 Routing daemons

You should not use UNIX and Linux systems as routers for production networks. Dedicated routers are simpler, more reliable, more secure, and faster (even if they are secretly running a Linux kernel). That said, it's nice to be able to set up a new subnet with only a $6 network card and a $20 switch. That's a reasonable approach for lightly populated test and auxiliary networks.

Systems that act as gateways to such subnets don't need any help managing their own routing tables. Static routes are perfectly adequate, both for the gateway machine and for the machines on the subnet itself. However, if you want the subnet to be reachable by other systems at your site, you need to advertise the subnet's existence and to identify the router to which packets bound for that subnet should be sent. The usual way to do this is to run a routing daemon on the gateway.

UNIX and Linux systems can participate in most routing protocols through various routing daemons. The notable exception is EIGRP, which, as far as we are aware, has no widely available UNIX or Linux implementation.

Because routing daemons are uncommon on production systems, we don't describe their use and configuration in detail. However, the following sections outline the common software options and point to detailed configuration information.

routed: obsolete RIP implementation

routed was for a long time the only standard routing daemon, and it's still included on a few systems. **routed** speaks only RIP, and poorly at that: even support for RIPv2 is scattershot. **routed** does not speak RIPng, implementation of that protocol being confined to modern daemons such as Quagga.

Where available, **routed** is useful chiefly for its "quiet" mode (**-q**), in which it listens for routing updates but does not broadcast any information of its own. Aside from the command-line flag, **routed** normally does not require configuration. It's an easy and cheap way to get routing updates without having to deal with much configuration hassle.

See this page for more about manual maintenance of routing tables.

---

routed adds its discovered routes to the kernel's routing table. Routes must be reheard at least every four minutes or they will be removed. However, **routed** knows which routes it has added and does not remove static routes that were installed with the **route** or **ip** commands.

Quagga: mainstream routing daemon

Quagga (quagga.net) is a development fork of Zebra, a GNU project started by Kunihiro Ishiguro and Yoshinari Yoshikawa to implement multiprotocol routing with a collection of independent daemons instead of a single monolithic application. In real life, the quagga—a subspecies of zebra last photographed in 1870—is extinct, but in the digital realm it is Quagga that survives and Zebra that is no longer under active development.

Quagga currently implements RIP (all versions), OSPF (versions 2 and 3), and BGP. It runs on Linux, FreeBSD, and several other platforms. Quagga is either installed by default or is available as an optional package through the system's standard software repository.

In the Quagga system, the core **zebra** daemon acts as a central clearing-house for routing information. It manages the interaction between the kernel's routing table and the daemons for individual routing protocols (**ripd**, **ripngd**, **ospfd**, **ospf6d**, and **bgpd**). It also controls the flow of routing information among protocols. Each daemon has its own configuration file in the **/etc/quagga** directory.

You can connect to any of the Quagga daemons through a command-line interface (**vtysh**) to query and modify its configuration. The command language itself is designed to be familiar to users of Cisco's IOS operating system; see the section on Cisco routers below for some additional details. As in IOS, you use **enable** to enter "superuser" mode, **config term** to enter configuration commands, and **write** to save your configuration changes back to the daemon's configuration file.

The official documentation at quagga.net is available in HTML or PDF form. Although complete, it's for the most part a workmanlike catalog of options and does not provide much of an overview of the system. The real documentation action is at quagga.net/docs. Look there for well-commented example configurations, FAQs, and tips.

Although the configuration files have a simple format, you'll need to understand the protocols you're configuring and have some idea of which options you want to enable or configure. See the recommended reading list here for some good books on routing protocols.

XORP: router in a box

XORP, the eXtensible Open Router Platform project, was started at around the same time as Zebra, but its ambitions are more general. Instead of focusing on routing, XORP aims to emulate all the functions of a dedicated router, including packet filtering and traffic management. Check it out at xorp.org.

One interesting aspect of XORP is that in addition to running under several operating systems (Linux, FreeBSD, macOS, and Windows Server), it’s also available as a live CD that runs directly on PC hardware. The live CD is secretly based on Linux, but it does go a long way toward turning a generic PC into a dedicated routing appliance.

15.7 Cisco routers

Routers made by Cisco Systems, Inc., are the de facto standard for Internet routing today. Having captured over 56% of the router market, Cisco's products are well known, and staff that know how to operate them are relatively easy to find. Before Cisco, UNIX boxes with multiple network interfaces were often used as routers. Today, dedicated routers are the favored gear to put in datacom closets and above ceiling tiles where network cables come together.

Most of Cisco's router products run an operating system called Cisco IOS, which is proprietary and unrelated to UNIX. Its command set is rather large; the full documentation set fills up about 4.5 feet of shelf space. We could never fully cover Cisco IOS here, but knowing a few basics can get you a long way.

By default, IOS defines two levels of access (user and privileged), both of which are password protected. By default, you can simply **ssh** to a Cisco router to enter user mode.

You are prompted for the user-level access password:

```
$ ssh acme-gw.acme.com
Password: <password>
```

Upon entering the correct password, you receive a prompt from Cisco's EXEC command interpreter.

```
acme-gw.acme.com>
```

At this prompt, you can enter commands such as **show interfaces** to see the router's network interfaces or **show ?** to list the other things you can see.

To enter privileged mode, type **enable** and when asked, type the privileged password. Once you have reached the privileged level, your prompt ends in a #:

```
acme-gw.acme.com#
```

Be careful—you can do anything from this prompt, including erasing the router's configuration information and its operating system. When in doubt, consult Cisco's manuals or one of the comprehensive books published by Cisco Press.

You can type **show running** to see the current running configuration of the router and **show config** to see the current nonvolatile configuration. Most of the time, these are the same.

Here's a typical configuration:

```
acme-gw.acme.com# show running
Current configuration:
version 12.4
hostname acme-gw
enable secret xxxxxxxx
ip subnet-zero

interface Ethernet0
    description Acme internal network
        ip address 192.108.21.254 255.255.255.0
        no ip directed-broadcast
interface Ethernet1
    description Acme backbone network
        ip address 192.225.33.254 255.255.255.0
        no ip directed-broadcast

ip classless
line con 0
transport input none

line aux 0
    transport input telnet
line vty 0 4
    password xxxxxxxx
    login

end
```

The router configuration can be modified in a variety of ways. Cisco offers graphical tools that run under some versions of UNIX/Linux and Windows. Real network administrators never use these; the command prompt is always the sure bet. You can also **scp** a config file to or from a router so you can edit it with your favorite editor.

To modify the configuration from the command prompt, type **config term**.

```
acme-gw.acme.com# config term
Enter configuration commands, one per line.  End with CNTL/Z.
acme-gw(config)#
```

You can then type new configuration commands exactly as you want them to appear in the **show running** output. For example, if you wanted to change the IP address of the Ethernet0 interface in the configuration above, you could enter

```
interface Ethernet0
ip address 192.225.40.253 255.255.255.0
```

When you've finished entering configuration commands, press <Control-Z> to return to the regular command prompt. If you're happy with the new configuration, enter **write mem** to save the configuration to nonvolatile memory.

Here are some tips for a successful Cisco router experience:

- Name the router with the **hostname** command. This precaution helps prevent accidents caused by configuration changes to the wrong router. The hostname always appears in the command prompt.
- Always keep a backup router configuration on hand. You can **scp** or **tftp** the running configuration to another system each night for safekeeping.
- It's often possible to store a copy of the configuration in NVRAM or on a removable jump drive. Do so!
- Control access to the router command line by putting access lists on the router's VTYs (VTYs are like PTYs on a UNIX system). This precaution prevents unwanted parties from trying to break into your router.
- Control the traffic flowing through your networks (and possibly to the outside world) by setting up access lists on each router interface.
- Keep routers physically secure. It's easy to reset the privileged password if you have physical access to a Cisco box.

If you have multiple routers and multiple router wranglers, check out the free tool RANCID from shrubbery.net. With a name like RANCID it practically markets itself, but here's the elevator pitch: RANCID logs into your routers every night to retrieve their configuration files. It diffs the configurations and lets you know about anything that's changed. It also automatically keeps the configuration files under revision control (see this page).

15.8 Recommended reading

Perlman, Radia. *Interconnections: Bridges, Routers, Switches, and Internetworking Protocols (2nd Edition)*. Reading, MA: Addison-Wesley, 2000. This is the definitive work in this topic area. If you buy just one book about networking fundamentals, this should be it. Also, don't ever pass up a chance to hang out with Radia—she's a lot of fun and holds a shocking amount of knowledge in her brain.

Edgeworth, Brad, Aaron Foss, and Ramiro Garza Rios. *IP Routing on Cisco IOS, IOS XE, and IOS XR: An Essential Guide to Understanding and Implementing IP Routing Protocols*. Indianapolis, IN: Cisco Press, 2014.

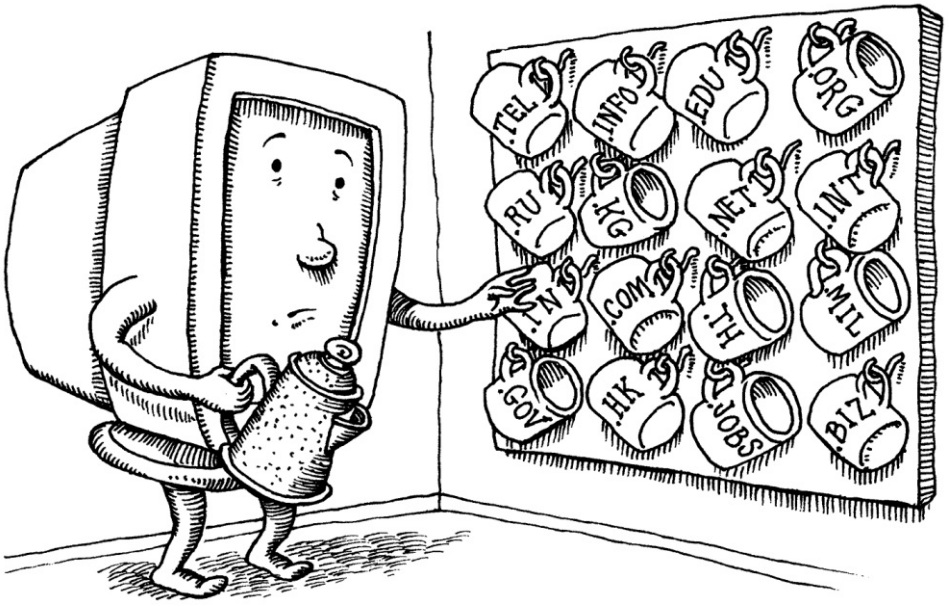
Huitema, Christian. *Routing in the Internet (2nd Edition)*. Upper Saddle River, NJ: Prentice Hall PTR, 2000. This book is a clear and well-written introduction to routing from the ground up. It covers most of the protocols in common use and also some advanced topics such as multicasting.

There are many routing-related RFCs. Table 15.3 shows the main ones.

Table 15.3: Routing-related RFCs

RFC	Title	Authors
1256	ICMP Router Discovery Messages	Deering
1724	RIP Version 2 MIB Extension	Malkin, Baker
2080	RIPng for IPv6	Malkin, Minnear
2328	OSPF Version 2	Moy
2453	Routing Information Protocol Version 2	Malkin
4271	A Border Gateway Protocol 4 (BGP-4)	Rekhter, Li, et al.
4552	Authentication/Confidentiality for OSPFv3	Gupta, Melam
4822	RIPv2 Cryptographic Authentication	Atkinson, Fanto
4861	Neighbor Discovery for IPv6	Narten et al.
5175	IPv6 Router Advertisement Flags Option	Haberman, Hinden
5308	Routing IPv6 with IS-IS	Hopps
5340	OSPF for IPv6	Coltun et al.
5643	Management Information Base for OSPFv3	Joyal, Manral, et al.

16 DNS: The Domain Name System



The Internet delivers instant access to resources all over the world, and each of those computers or sites has a unique name (e.g., google.com). However, anyone who has tried to find a friend or a lost child in a crowded stadium knows that simply knowing a name and yelling it loudly is not enough. Essential to finding anything (or anyone) is an organized system for communicating, updating, and distributing names and their locations.

Users and user-level programs like to refer to resources by name (e.g., amazon.com), but low-level network software understands only IP addresses (e.g., 54.239.17.6). Mapping between names and addresses is the best known and arguably most important function of DNS, the Domain Name System. DNS includes other elements and features, but almost without exception they exist to support this primary objective.

Over the history of the Internet, DNS has been both praised and criticized. Its initial elegance and simplicity encouraged adoption in the early years and enabled the Internet to grow quickly with little centralized management. As needs for additional functionality grew, so did the DNS system. Sometimes, these functions were bolted on in a way that looks ugly today. Naysayers point out weaknesses in the DNS infrastructure as evidence that the Internet is on the verge of collapse.

Say what you will, but the fundamental concepts and protocols of DNS have so far withstood growth from a few hundred hosts in a single country to a world-wide network that supports over 3 billion users across more than 1 billion hosts. Nowhere else can we find an information system

that has grown to this scale with so few issues. Without DNS, the Internet would have failed long ago.

16.1 DNS ARCHITECTURE

DNS is a distributed database. Under this model, one site stores the data for computers it knows about, another site stores the data for its own set of computers, and the sites cooperate and share data when one site needs to look up the other's data. From an administrative point of view, the DNS servers you have configured for your domain answer queries from the outside world about names in your domain; they also query other domains' servers on behalf of your users.

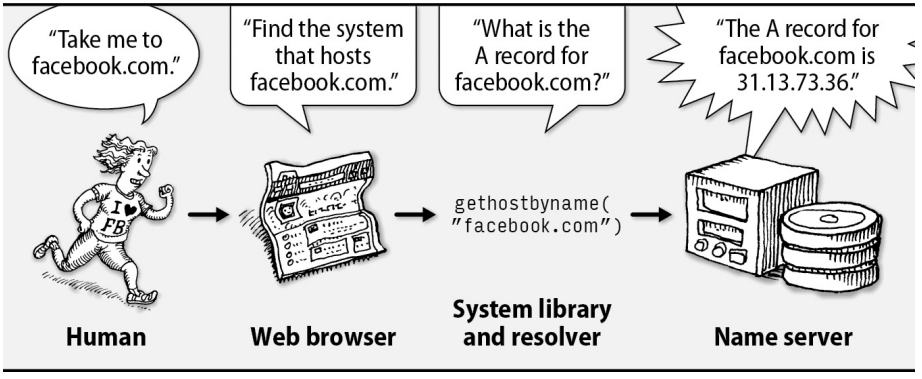
Queries and responses

A DNS query consists of a name and a record type. The answer returned is a set of "resource records" (RRs) that are responsive to the query (or alternatively, a response indicating that the name and record type you asked for do not exist).

"Responsive" doesn't necessarily mean "dispositive." DNS servers are arranged into a hierarchy, and it might be necessary to contact servers at several layers to answer a particular query (see this page). Servers that don't know the answer to a query return resource records that help the client locate a server that does.

The most common query is for an A record, which returns the IP address associated with a name. Exhibit A illustrates a typical scenario.

Exhibit A: A simple name lookup



First, a human types the name of a desired site into a web browser. The browser then calls the DNS "resolver" library to look up the corresponding address. The resolver library constructs a query for an A record and sends it to a name server, which returns the A record in its response. Finally, the browser opens a TCP connection to the target host through the IP address returned by the name server.

DNS service providers

Years ago, one of the core tasks of every system administrator was to set up and maintain a DNS server for their organization. Today, the landscape has changed. If an organization maintains a DNS server at all, it is frequently for internal use only.

Microsoft's Active Directory system includes an integrated DNS server that meshes nicely with the other Microsoft-flavored services found in corporate environments. However, Active Directory is suitable only for internal use. It should never be used as an external (Internet-facing) DNS server because of potential security concerns.

Every organization still needs an external-facing DNS server, but it's now common to use one of the many commercial "managed" DNS providers for this function. These services offer a GUI management interface and highly available, secure DNS infrastructure for only pennies (or dollars) a day. Amazon Route 53, CloudFlare, GoDaddy, DNS Made Easy, and Rackspace are just a few of the major providers.

Of course, you can still set up and maintain your own DNS server (internal or external) if you wish. You have dozens of DNS implementations to choose from, but the Berkeley Internet Name Domain (BIND) system still dominates the Internet. Over 75% of DNS servers run some form of it, according to the July, 2015 ISC Internet Domain Survey.

Regardless of which path you choose, as a system administrator you need to understand the basic concepts and architecture of DNS. The first few sections of this chapter focus on that important foundational knowledge. Starting on this page, we show some specific configurations for BIND.

16.2 DNS FOR LOOKUPS

Regardless of whether you run your own name server, use a managed DNS service, or have someone else providing DNS service for you, you'll certainly want to configure all of your systems to look up names in DNS.

Two steps are needed to make this happen. First, you configure your systems as DNS clients. Second, you tell the systems when to use DNS as opposed to other name lookup methods such as a static **/etc/hosts** file.

resolv.conf: client resolver configuration

Each host on the network should be a DNS client. You configure the client-side resolver in the file **/etc/resolv.conf**. This file lists the name servers to which the host can send queries.

See this page for more information about DHCP.

If your host gets its IP address and network parameters from a DHCP server, the **/etc/resolv.conf** file is normally set up for you automatically. Otherwise, you must edit the file by hand. The format is

```
search domainname ...
nameserver ipaddr
```

Up to three name servers can be listed. Here's a complete example:

```
search atrust.com booklab.atrust.com
nameserver 63.173.189.1         ; ns1
nameserver 174.129.219.225      ; ns2
```

The search line lists the domains to query if a hostname is not fully qualified. For example, if a user issues the command **ssh coraline**, the resolver completes the name with the first domain in the search list and looks for coraline.atrust.com. If no such name exists, the resolver also tries coraline.booklab.atrust.com. The number of domains that can be specified in a search directive is resolver-specific; most allow between six and eight, with a limit of 256 characters.

The name servers listed in **resolv.conf** must be configured to allow your host to submit queries. They must also be recursive; that is, they must answer queries to the best of their ability and not try to refer you to other name servers; see this page.

DNS servers are contacted in order. As long as the first one continues to answer queries, the others are ignored. If a problem occurs, the query eventually times out and the next name server is tried. Each server is tried in turn, up to four times. The timeout interval increases with each failure. The default timeout interval is five seconds, which seems like forever to impatient users.

nsswitch.conf: who do I ask for a name?

Both FreeBSD and Linux use a switch file, **/etc/nsswitch.conf,** to specify how hostname-to-IP-address mappings should be performed and whether DNS should be tried first, last, or not at all. If no switch file is present, the default behavior is

```
hosts: dns [!UNAVAIL=return] files
```

The !UNAVAIL clause means that if DNS is available but a name is not found there, the lookup attempt should fail rather than continuing to the next entry (in this case, the **/etc/hosts** file). If no name server is running (as might be the case during boot), the lookup process does consult the **hosts** file.

Our example distributions all provide the following default **nsswitch.conf** entry:

```
hosts: files dns
```

This configuration gives precedence to the **/etc/hosts** file, which is always checked. DNS is consulted only for names that are unresolvable through **/etc/hosts**.

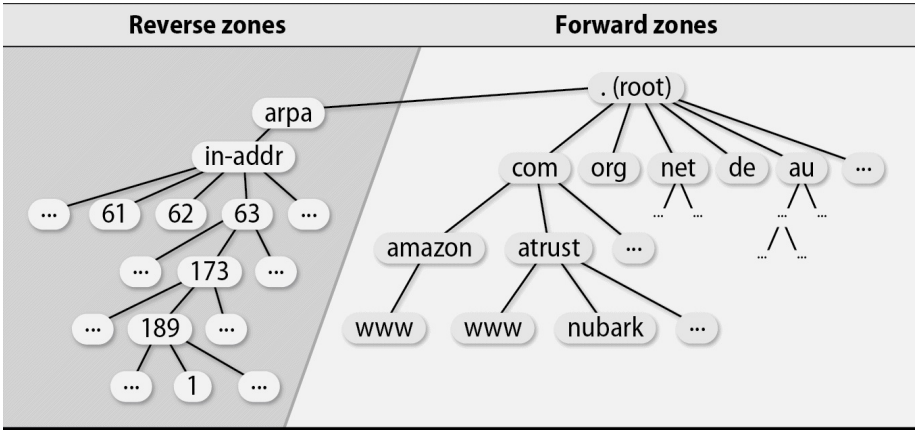
There is really no best way to configure lookups—it depends on how your site is managed. In general, we prefer to keep as much host information as possible in DNS but always preserve the ability to fall back to the static **hosts** file during the boot process if necessary.

If name service is provided for you by an outside organization, you might be done with DNS configuration after setting up **resolv.conf** and **nsswitch.conf**. If so, you can skip the rest of this chapter, or read on to learn more.

16.3 The DNS namespace

The DNS namespace is organized into a tree that contains both forward mappings and reverse mappings. Forward mappings map hostnames to IP addresses (and other records), and reverse mappings map IP addresses to hostnames. Every complete hostname (e.g., nubark.atrust.com) is a node in the forward branch of the tree, and (in theory) every IP address is a node in the reverse branch. Exhibit B shows the general layout of the naming tree.

Exhibit B: DNS zone tree



To allow the same DNS system to manage both names (which have the most significant information on the right), and IP addresses (which have the most significant part on the left), the IP branch of the namespace is inverted by listing the octets of the IP address backwards. For example, if host nubark.atrust.com has IP address 63.173.189.1, the corresponding node of the forward branch of the naming tree is “nubark.atrust.com.” and the node of the reverse branch is “1.189.173.63.in-addr.arpa.”. The in-addr.arpa portion of the name is a fixed suffix.

Both of these names end with a dot, just as the full pathnames of files always start with a slash. That makes them “fully qualified domain names” or FQDNs for short. Outside the context of DNS, names like nubark.atrust.com (without the final dot) are sometimes referred to as “fully qualified hostnames,” but this is a colloquialism. Within the DNS system itself, the presence or absence of the trailing dot is of crucial importance.

Two types of top-level domains exist: country code domains (ccTLDs) and generic top-level domains (gTLDs). ICANN, the Internet Corporation for Assigned Names and Numbers, accredits various agencies to be part of its shared registry project for registering names in the gTLDs such as com, net, and org. To register for a ccTLD name, check the IANA (Internet Assigned Numbers Authority) web page iana.org/cctld to find the registry in charge of a particular country’s registration.

Registering a domain name

To obtain a second-level domain name (such as blazedgoat.com), you must apply to a registrar for the appropriate top-level domain. To complete the domain registration forms, you must choose a name that is not already taken and identify a technical contact person, an administrative contact person, and at least two hosts that will be name servers for your domain. Fees vary among registrars, but these days they are all generally quite inexpensive.

Creating your own subdomains

The procedure for creating a subdomain is similar to that for creating a second-level domain, except that the central authority is now local (or more accurately, within your own organization). Specifically, the steps are as follows:

- Choose a name that is unique in the local context.
- Identify two or more hosts to be servers for your new domain.
- Coordinate with the administrator of the parent domain.

The two-or-more-servers rule is a policy, not a technical requirement. You make the rules in your own subdomains, so you can get away with a single server if you want.

Parent domains should check to be sure that a child domain's name servers are up and running before performing the delegation. If the servers are not working, a "lame delegation" results, and you might receive nasty email asking you to clean up your DNS act. Lame delegations are covered in more detail here.

16.4 How DNS works

Name servers around the world work together to answer queries. Typically, they distribute information maintained by whichever administrator is closest to the query target. Understanding the roles and relationships of name servers is important both for day-to-day operations and for debugging.

Name servers

A name server performs several chores:

- It answers queries about your site's hostnames and IP addresses.
- It asks about both local and remote hosts on behalf of your users.
- It caches the answers to queries so that it can answer faster next time.
- It communicates with other local name servers to keep DNS data synchronized.

Name servers deal with "zones," where a zone is essentially a domain minus its subdomains. You will often see the term "domain" used where a zone is what's actually meant, even in this book.

Name servers can operate in several different modes. The distinctions among them fall along several axes, so the final categorization is often not tidy. To make things even more confusing, a single server can play different roles with respect to different zones. Table 16.1 lists some of the adjectives used to describe name servers.

Table 16.1: Name server taxonomy

Type of server	Description
authoritative	Officially represents a zone
master	The master server for a zone; gets its data from a disk file
primary	Another name for the master server
slave	Copies its data from the master
secondary	Another name for a slave server
stub	Like a slave, but copies only name server data (not host data)
distribution	A server advertised only within a domain (aka "stealth server")
nonauthoritative[a]	Answers a query from cache; doesn't know if the data is still valid
caching	Caches data from previous queries; usually has no local zones
forwarder	Performs queries on behalf of many clients; builds a large cache
recursive	Queries on your behalf until it returns either an answer or an error
nonrecursive	Refers you to another server if it can't answer a query

a. Strictly speaking, "nonauthoritative" is an attribute of a DNS query response, not a server.

These categorizations vary according to the name server's source of data (authoritative, caching, master, slave), the type of data saved (stub), the query path (forwarder), the completeness of answers handed out (recursive, nonrecursive), and finally, the visibility of the server (distribution). The next few sections provide additional details on the most important of these distinctions; the others are described elsewhere in this chapter.

Authoritative and caching-only servers

Master, slave, and caching-only servers are distinguished by two characteristics: where the data comes from, and whether the server is authoritative for the domain. Each zone typically has one master name server. The master server keeps the official copy of the zone's data on disk. The system administrator changes the zone's data by editing the master server's data files.

Some sites use multiple masters or even no masters. However, these are unusual configurations. We describe only the single-master case.

See this page for more information about zone transfers.

A slave server gets its data from the master server through a "zone transfer" operation. A zone can have several slave name servers and *must* have at least one. A stub server is a special kind of slave that loads only the NS (name server) records from the master. It's fine for the same machine to be both a master server for some zones and a slave server for other zones.

A caching-only name server loads the addresses of the servers for the root domain from a startup file and accumulates the rest of its data by caching answers to the queries it resolves. A caching-only name server has no data of its own and is not authoritative for any zone (except perhaps the localhost zone).

An authoritative answer from a name server is "guaranteed" to be accurate; a nonauthoritative answer might be out of date. However, a very high percentage of nonauthoritative answers are perfectly correct. Master and slave servers are authoritative for their own zones, but not for information they may have cached about other domains. Truth be told, even authoritative answers can be inaccurate if a sysadmin changes the master server's data but forgets to propagate the changes (e.g., doesn't change the zone's serial number).

At least one slave server is required for each zone. Ideally, there should be at least two slaves, one of which is in a location that does not share common infrastructure with the master. On-site slaves should live on different networks and different power circuits. When name service stops, all normal network access stops, too.

Recursive and nonrecursive servers

Name servers are either recursive or nonrecursive. If a nonrecursive server has the answer to a query cached from a previous transaction or is authoritative for the domain to which the query pertains, it provides an appropriate response. Otherwise, instead of returning a real answer, it returns a referral to the authoritative servers of another domain that are more likely to know the answer. A client of a nonrecursive server must be prepared to accept and act on referrals.

Although nonrecursive servers might seem lazy, they usually have good reason not to take on extra work. Authoritative-only servers (e.g., root servers and top-level domain servers) are all nonrecursive, but since they may process tens of thousands of queries per second we can excuse them for cutting corners.

A recursive server returns only real answers and error messages. It follows referrals itself, relieving clients of this responsibility. In other respects, the basic procedure for resolving a query is essentially the same.

For security, an organization's externally accessible name servers should always be nonrecursive. Recursive name servers that are visible to the world can be vulnerable to cache poisoning attacks.

Note well: resolver libraries *do not* understand referrals. Any local name server listed in a client's **resolv.conf** file must be recursive.

Resource records

Each site maintains one or more pieces of the distributed database that makes up the world-wide DNS system. Your piece of the database consists of text files that contain records for each of your hosts; these are known as "resource records." Each record is a single line consisting of a name (usually a hostname), a record type, and some data values. The name field can be omitted if its value is the same as that of the previous line.

For example, the lines

```
nubark              IN     A    63.173.189.1
                    IN     MX   10 mailserver.atrust.com.
```

in the "forward" file (called **atrust.com**), and the line

```
1                   IN     PTR  nubark.atrust.com.
```

See this page for more information about MX records.

---

in the "reverse" file (called **63.173.189.rev**) associate nubark.atrust.com with the IP address 63.173.189.1. The MX record routes email addressed to this machine to the host mailserver.atrust.com.

The IN fields denote the record classes. In practice, this field is always IN for Internet.

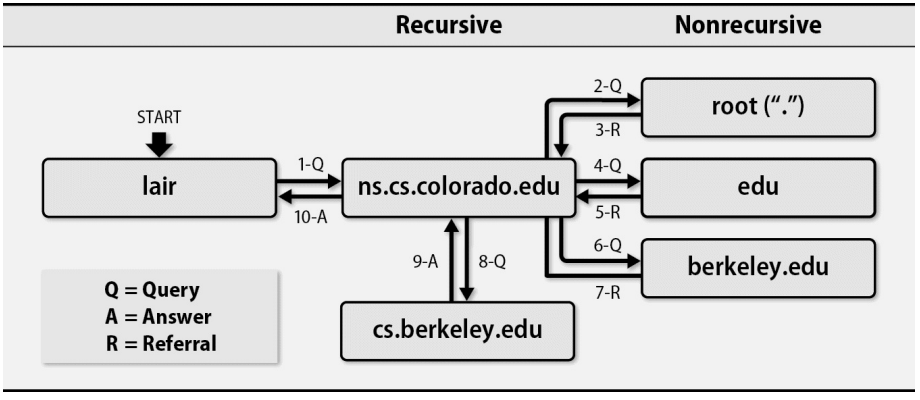
Resource records are the lingua franca of DNS and are independent of the configuration files that control the operation of any given DNS server implementation. They are also the pieces of data that flow around the DNS system and become cached at various locations.

Delegation

All name servers read the identities of the root servers from a local config file or have them built into the code. The root servers know the name servers for com, net, edu, fi, de, and other top-level domains. Farther down the chain, edu knows about colorado.edu, berkeley.edu, and so on. Each domain can delegate authority for its subdomains to other servers.

Let's inspect a real example. Suppose we want to look up the address for the machine vangogh.cs.berkeley.edu from the machine lair.cs.colorado.edu. The host lair asks its local name server, ns.cs.colorado.edu, to figure out the answer. The following illustration (Exhibit C) shows the subsequent events.

Exhibit C: DNS query process for vangogh.cs.berkeley.edu



The numbers on the arrows between servers show the order of events, and a letter denotes the type of transaction (query, referral, or answer). We assume that none of the required information was cached before the query, except for the names and IP addresses of the servers of the root domain.

The local server doesn't know vangogh's address. In fact, it doesn't know anything about cs.berkeley.edu or berkeley.edu or even edu. It does know servers for the root domain, however, so it queries a root server about vangogh.cs.berkeley.edu and receives a referral to the servers for edu.

The local name server is a recursive server. When the answer to a query consists of a referral to another server, the local server resubmits the query to the new server. It continues to follow referrals until it finds a server that has the data it's looking for.

In this case, the local name server sends its query to a server of the edu domain (asking, as always, about vangogh.cs.berkeley.edu) and gets back a referral to the servers for berkeley.edu. The local name server then repeats this same query on a berkeley.edu server. If the Berkeley server doesn't have the answer cached, it returns a referral to the servers for cs.berkeley.edu. The

cs.berkeley.edu server is authoritative for the requested information, looks the answer up in its zone files, and returns vangogh's address.

When the dust settles, ns.cs.colorado.edu has cached vangogh's address. It has also cached data on the servers for edu, berkeley.edu, and cs.berkeley.edu.

You can view the query process in detail with **dig +trace** or **drill -T**. (**dig** and **drill** are DNS query tools: **dig** from the BIND distribution and **drill** from NLnet Labs.)

Caching and efficiency

Caching increases the efficiency of lookups: a cached answer is almost free and is usually correct because hostname-to-address mappings change infrequently. An answer is saved for a period of time called the “time to live” (TTL), which is specified by the owner of the data record in question.

Most queries are for local hosts and can be resolved quickly. Users also inadvertently help with efficiency because they repeat many queries; after the first instance of a query, the repeats are more or less free.

Under normal conditions, your site’s resource records should use a TTL that is somewhere between an hour and a day. The longer the TTL, the less network traffic will be consumed by Internet clients obtaining fresh copies of the record.

If you have a specific service that is load-balanced across logical subnets (often called “global server load balancing”), you may be required by your load-balancing vendor to choose a shorter TTL, such as 10 seconds or 1 minute. The short TTL lets the load balancer react quickly to inoperative servers and denial of service attacks. The system still works correctly with short TTLs, but your name servers have to work hard.

In the vangogh example above, the TTLs were 42 days for the roots, 2 days for edu, 2 days for berkeley.edu, and 1 day for vangogh.cs.berkeley.edu. These are reasonable values. If you are planning a massive renumbering, change the TTLs to a shorter value well before you start.

DNS servers also implement negative caching. That is, they remember when a query fails and do not repeat that query until the negative caching TTL value has expired. Negative caching can potentially save answers of the following types:

- No host or domain matches the name queried.
- The type of data requested does not exist for this host.
- The server is not responding.
- The server is unreachable because of network problems.

The BIND implementation caches the first two types of negative data and allows the negative cache times to be configured.

Multiple answers and round robin DNS load balancing

A name server often receives multiple records in response to a query. For example, the response to a query for the name servers of the root domain would list all 13 servers.

You can take advantage of this balancing effect for your own servers by assigning several different IP addresses (for different machines) to a single hostname:

```
www             IN A     192.168.0.1
                IN A     192.168.0.2
                IN A     192.168.0.3
```

Most name servers return multirecord sets in a different order each time they receive a query, rotating them in round robin fashion. When a client receives a response with multiple records, the most common behavior is to try the addresses in the order returned by the DNS server. However, this behavior is not required. Some clients may behave differently.

This scheme is commonly referred to as round robin DNS load balancing. However, it is a crude solution at best. Large sites use load-balancing software (such as HAProxy; see this page) or dedicated load-balancing appliances.

Debugging with query tools

See this page for more information about DNSSEC.

Five command-line tools that query the DNS database are distributed with BIND: **nslookup**, **dig**, **host**, **drill** and **delv**. **nslookup** and **host** are simple and have pretty output, but you need **dig** or **drill** to get all the details. **drill** is better for following DNSSEC signature chains. The name **drill** is a pun on **dig** (the Domain Information Groper), implying you can get even more info from DNS with **drill** than you can with **dig**. **delv** is new to BIND 9.10 and will eventually replace **drill** for DNSSEC debugging.

See this page for more information about split DNS.

By default, **dig** and **drill** query the name servers configured in **/etc/resolv.conf**. A **@***nameserver* argument makes either command query a specific name server. The ability to query a particular server lets you check to be sure that any changes you make to a zone have been propagated to secondary servers and to the outside world. This feature is especially useful if you use views (split DNS) and need to verify that you have configured them correctly.

If you specify a record type, **dig** and **drill** query for that type only. The pseudo-type **any** is a bit sneaky: instead of returning all data associated with a name, it returns all *cached* data associated with the name. So, to get all records, you might have to do **dig** *domain* **NS** followed by **dig @ns1.***domain domain* **any**. (Authoritative data counts as cached in this context.)

dig has about 50 options and **drill** about half that many. Either command accepts an **-h** flag to list the various options. (You'll probably want to pipe the output through **less**.) For both tools, **-x** reverses the bytes of an IP address and does a reverse query. The **+trace** flag to **dig** or **-T** to **drill** shows the iterative steps in the resolution process from the roots down.

dig and **drill** include the notation aa in the output flags if an answer is authoritative (i.e., it comes directly from a master or slave server of that zone). The code ad indicates that an answer was authenticated by DNSSEC. When testing a new configuration, be sure that you look up data for both local and remote hosts. If you can access a host by IP address but not by name, DNS is probably the culprit.

The most common use of **dig** is to determine what records are currently being returned for a particular name. If only an AUTHORITY response is returned, you have been referred to another name server. If an ANSWER response is returned, your question has been directly answered (and other information may be included as well).

It's often useful to follow the delegation chain manually from the root servers to verify that everything is in the right place. Below we look at an example of that process for the name

www.viawest.com. First, we query a root server to see who is authoritative for viawest.com by requesting the start-of-authority (SOA) record:

```
$ dig @a.root-servers.net viawest.com soa
; <<>> DiG 9.8.3-P1 <<>> @a.root-servers.net viawest.com soa
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 7824
;; flags: qr rd; QUERY: 1, ANSWER: 0, AUTHORITY: 13, ADDITIONAL: 14
;; WARNING: recursion requested but not available

;; QUESTION SECTION:
;viawest.com.                IN SOA

;; AUTHORITY SECTION:

com.                 172800  IN NS     c.gtld-servers.net.
com.                 172800  IN NS     b.gtld-servers.net.
com.                 172800  IN NS     a.gtld-servers.net.
...

;; ADDITIONAL SECTION:
c.gtld-servers.net.  172800  IN A      192.26.92.30
b.gtld-servers.net.  172800  IN A      192.33.14.30
b.gtld-servers.net.  172800  IN AAAA   2001:503:231d::2:30
a.gtld-servers.net.  172800  IN A      192.5.6.30
...

;; Query time: 62 msec
;; SERVER: 198.41.0.4#53(198.41.0.4)
;; WHEN: Wed Feb  3 18:37:37 2016
;; MSG SIZE  rcvd: 489
```

Note that the status returned is NOERROR. That tells us that the query returned a response without notable errors. Other common status values are NXDOMAIN, which indicates the name requested doesn't exist (or isn't registered), and SERVFAIL, which usually indicates a configuration error on the name server itself.

This AUTHORITY SECTION tells us that the global top-level domain (gTLD) servers are the next link in the authority chain for this domain. So, we pick one at random and repeat the same query:

```
$ dig @c.gtld-servers.net viawest.com soa
; <<>> DiG 9.8.3-P1 <<>> @c.gtld-servers.net viawest.com soa
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 9760
;; flags: qr rd; QUERY: 1, ANSWER: 0, AUTHORITY: 2, ADDITIONAL: 2
;; WARNING: recursion requested but not available

;; QUESTION SECTION:
;viawest.com.                IN SOA

;; AUTHORITY SECTION:
viawest.com.         172800  IN NS     ns1.viawest.net.
viawest.com.         172800  IN NS     ns2.viawest.net.

;; ADDITIONAL SECTION:
ns1.viawest.net.     172800  IN A      216.87.64.12
ns2.viawest.net.     172800  IN A      209.170.216.2

;; Query time: 52 msec
;; SERVER: 192.26.92.30#53(192.26.92.30)
;; WHEN: Wed Feb  3 18:40:48 2016
;; MSG SIZE  rcvd: 108
```

This response is much more succinct, and we now know that the next server to query is ns1.viawest.com (or ns2.viawest.com).

```
$ dig @ns1.viawest.net viawest.com soa
; <<>> DiG 9.8.3-P1 <<>> @ns2.viawest.net viawest.com soa
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 61543
;; flags: qr aa rd; QUERY: 1, ANSWER: 1, AUTHORITY: 1, ADDITIONAL: 1
;; WARNING: recursion requested but not available

;; QUESTION SECTION:
;viawest.com.                IN SOA

;; ANSWER SECTION:
viawest.com.         3600    IN SOA    mvec.viawest.net. hostmaster.
    viawest.net. 2007112567 3600 1800 1209600 3600

;; AUTHORITY SECTION:
viawest.com.         86400   IN NS     ns2.viawest.net.

;; ADDITIONAL SECTION:
ns2.viawest.net.     3600    IN A      209.170.216.2

;; Query time: 5 msec
;; SERVER: 216.87.64.12#53(216.87.64.12)
;; WHEN: Wed Feb  3 18:42:20 2016
;; MSG SIZE  rcvd: 126
```

This query returns an ANSWER for the viawest.com domain. We now know an authoritative name server and can query for the name we actually want, www.viawest.com.

```
$ dig @ns1.viawest.net www.viawest.com any
; <<>> DiG 9.8.3-P1 <<>> @ns1.viawest.net www.viawest.com any
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 29968
;; flags: qr aa rd; QUERY: 1, ANSWER: 1, AUTHORITY: 1, ADDITIONAL: 1
;; WARNING: recursion requested but not available

;; QUESTION SECTION:
;www.viawest.com.           IN ANY

;; ANSWER SECTION:
www.viawest.com.    60      IN CNAME hm-d8ebfa-via1.threatx.io.

;; AUTHORITY SECTION:
viawest.com.        86400   IN NS    ns2.viawest.net.

;; ADDITIONAL SECTION:
ns2.viawest.net.    3600    IN A     209.170.216.2

;; Query time: 6 msec
;; SERVER: 216.87.64.12#53(216.87.64.12)
;; WHEN: Wed Feb  3 18:46:38 2016
;; MSG SIZE  rcvd: 117
```

This final query shows us that www.viawest.com has a CNAME record pointed at hm-d8ebfa-via1.threatx.io, meaning that it is another name for the threatx host (a host operated by a cloud-based distributed denial-of-service provider).

Of course, if you query a recursive name server, it will follow the entire delegation chain on your behalf. But when debugging, it’s typically more useful to investigate the chain link by link.

16.5 The DNS database

A zone's DNS database is a set of text files maintained by the system administrator on the zone's master name server. These text files are often called zone files. They contain two types of entries: parser commands (things like $ORIGIN and $TTL) and resource records. Only the resource records are really part of the database; the parser commands just provide some shorthand ways to enter records.

Parser commands in zone files

Zone file commands are standardized in RFCs 1035 and 2308.

Commands can be embedded in zone files to make the zone files more readable and easier to maintain. The commands either influence the way the parser interprets subsequent records or they expand into multiple DNS records themselves. Once a zone file has been read and interpreted, none of these commands remain a part of the zone's data (at least, not in their original forms).

Three commands (`$ORIGIN`, `$INCLUDE`, and `$TTL`) are standard for all DNS implementations, and a fourth, `$GENERATE`, is found only in BIND. Commands must start in column one and occur on a line by themselves.

Zone files are read and parsed from top to bottom in a single pass. As the name server reads a zone file, it adds the default domain (or "origin") to any names that are not already fully qualified. The origin defaults to the domain name specified in the name server's configuration file. However, you can set the origin or change it within a zone file by using the `$ORIGIN` directive:

```
$ORIGIN domain-name
```

The use of relative names where fully qualified names are expected saves lots of typing and makes zone files much easier to read.

Many sites use the `$INCLUDE` directive in their zone database files to separate overhead records from data records, to separate logical pieces of a zone file, or to keep cryptographic keys in a file with restricted permissions. The syntax is

```
$INCLUDE filename [origin]
```

The specified file is read into the database at the point of the `$INCLUDE` directive. If *filename* is not an absolute path, it is interpreted relative to the home directory of the running name server.

If you supply an *origin* value, the parser acts as if an `$ORIGIN` directive precedes the contents of the file being read. Watch out: the origin does not revert to its previous value after the `$INCLUDE` has been executed. You'll probably want to reset the origin, either at the end of the included file or on the line following the `$INCLUDE` statement.

The `$TTL` directive sets a default value for the time-to-live field of the records that follow it. It must be the first line of the zone file. The default units for the `$TTL` value are seconds, but you can also qualify numbers with `h` for hours, `m` for minutes, `d` for days, or `w` for weeks. For example, the lines

```
$TTL 86400
$TTL 24h
$TTL 1d
```

all set the $TTL to one day.

Resource records

Each zone of the DNS hierarchy has a set of resource records associated with it. The basic format of a resource record is

```
[name] [ttl] [class] type data
```

Fields are separated by whitespace (tabs or spaces) and can contain the special characters shown in Table 16.2.

Table 16.2: Special characters in resource records

Character	Meaning
;	Introduces a comment
@	The current zone name
( )	Allows data to span lines
*	Wild card (*name* field only)[a]

a. See this page for some cautionary statements.

The *name* field identifies the entity (usually a host or domain) that the record describes. If several consecutive records refer to the same entity, the name can be omitted after the first record as long as the subsequent records begin with white-space. If present, the *name* field must begin in column one.

A name can be either relative or absolute. Absolute names end with a dot and are complete. Internally, the software deals only with absolute names; it appends the current origin and a dot to any name that does not already end in a dot. This feature allows names to be shorter, but it also invites mistakes.

For example, if cs.colorado.edu were the current domain, the name “anchor” would be interpreted as “anchor.cs.colorado.edu.”. If by mistake you entered the name as “anchor.cs.colorado.edu”, the lack of a final dot would still imply a relative name, resulting in the name “anchor.cs.colorado.edu.cs.colorado.edu.” This kind of mistake is common.

The *ttl* (time to live) field specifies the length of time, in seconds, that the record can be cached and still be considered valid. It is often omitted, except in the root server hints file. It defaults to the value set by the $TTL directive, which must be the first line of the zone data file.

Increasing the value of the *ttl* parameter to about a week substantially reduces network traffic and DNS load. However, once records have been cached outside your local network, you cannot force them to be discarded. If you plan a massive renumbering and your old *ttl* was a week, lower the $TTL value (e.g., to one hour) at least a week before your intended renumbering. This preparatory step makes sure that records with week-long *ttl*s are expired and replaced with

records that have one-hour *ttl*s. You can then be certain that all your updates will propagate together within an hour. Set the *ttl*s back to their original value after you've completed your update campaign.

Some sites set the TTL on the records for Internet-facing servers to a low value so that if a server experiences problems (network failure, hardware failure, denial-of-service attack, etc.), the administrators can respond by changing the server's name-to-IP-address mapping. Because the original TTLs were low, the new values will propagate quickly. For example, the name google.com has a five-minute TTL, but Google's name servers have a TTL of four days (345,600 seconds):

```
google.com.         300     IN A     216.58.217.46
google.com.         345600  IN NS    ns1.google.com.
ns1.google.com.     345600  IN A     216.239.32.10
```

We used **dig** to obtain these records; we truncated the output.

The *class* specifies the network type. IN for Internet is the default.

Many different types of DNS records are defined, but fewer than 10 are in common use; IPv6 adds a few more. We divide the resource records into four groups:

- Zone infrastructure records, which identify domains and their name servers
- Basic records, which map between names and addresses and route mail
- Security records, which add authentication and signatures to zone files
- Optional records, which provide extra information about hosts or domains

The contents of the *data* field depend on the record type. A DNS query for a particular domain and record type returns all matching resource records from the zone file. Table 16.3 lists the common record types.

Table 16.3: DNS record types

	Type	Name	Function
Zone	SOA	Start Of Authority	Defines a DNS zone
	NS	Name Server	Identifies servers, delegates subdomains
Basics	A	IPv4 Address	Name-to-address translation
	AAAA	IPv6 Address	Name-to-IPv6-address translation
	PTR	Pointer	Address-to-name translation
	MX	Mail Exchanger	Controls email routing
Security	DS	Delegation Signer	Hash of signed child zone's key-signing key
	DNSKEY	Public Key	Public key for a DNS name
	NSEC	Next Secure	Used with DNSSEC for negative answers
	NSEC3	Next Secure v3	Used with DNSSEC for negative answers
	RRSIG	Signature	Signed, authenticated resource record set
Optional	CNAME	Canonical Name	Nicknames or aliases for a host
	SRV	Service	Gives locations of a well-known service
	TXT	Text	Comments or untyped information

Some record types are obsolete, experimental, or not widely used. See your name server's implementation documentation for a complete list. Most records are maintained by hand (by editing text files or by entering them in a web GUI), but the security resource records require cryptographic processing and so must be managed with software tools. These records are described in the DNSSEC section beginning here.

The order of resource records in the zone file is arbitrary, but traditionally the SOA record is first, followed by the NS records. The records for each host are usually kept together. It's common practice to sort by the *name* field, although some sites sort by IP address so that it's easier to identify unused addresses.

As we describe each type of resource record in detail in the next sections, we inspect some sample records from the atrust.com domain's data files. The default domain in this context is "atrust.com.", so a host specified as "bark" really means "bark.atrust.com.".

See this page for more information about RFCs.

The format and interpretation of each type of resource record is specified by the IETF in the RFC series. In the upcoming sections, we list the specific RFCs relevant to each record type (along with their years of origin) in a margin note.

The SOA record

SOA records are specified in RFC1035 (1987).

An SOA (Start of Authority) record marks the beginning of a zone, a group of resource records located at the same place within the DNS namespace. The data for a DNS domain usually includes at least two zones: one for translating hostnames to IP addresses, called the forward zone, and others that map IP addresses back to hostnames, called reverse zones.

Each zone has exactly one SOA record. The SOA record includes the name of the zone, the primary name server for the zone, a technical contact, and various timeout values. Comments are introduced by a semicolon. Here's an example:

```
; Start of authority record for atrust.com
atrust.com.      IN SOA    ns1.atrust.com. hostmaster.atrust.com. (
    2017110200     ; Serial number
    10800          ; Refresh  (3 hours)
    1200           ; Retry    (20 minutes)
    3600000        ; Expire   (40+ days)
    3600 )         ; Minimum  (1 hour)
```

The *name* field of the SOA record (atrust.com. in this example) often contains the symbol `@`, which is shorthand for the name of the current zone. The value of `@` is the domain name specified in the `zone` statement of **named.conf**. This value can be changed from within the zone file with the `$ORIGIN` parser directive (see this page).

This example has no *ttl* field. The class is IN for Internet, the type is SOA, and the remaining items form the *data* field. The numerical parameters in parentheses are timeout values and are often written on one line without comments.

"ns1.atrust.com." is the zone's master name server. Actually, any name server for the zone can be listed in the SOA record unless you are using dynamic DNS. In that case, the SOA record must name the master server.

"hostmaster.atrust.com." was originally intended to be the email address of the technical contact in the format "*user.host.*" rather than the standard *user@host.* Unfortunately, due to spam concerns and other reasons, most sites do not keep this contact info updated.

The parentheses continue the SOA record over several lines.

The first numeric parameter is the serial number of the zone's configuration data. The serial number is used by slave servers to determine when to get fresh data. It can be any 32-bit integer and should be incremented every time the data file for the zone is changed. Many sites encode

the file's modification date in the serial number. For example, 2017110200 would be the first change to the zone on November 2, 2017.

Serial numbers need not be continuous, but they must increase monotonically. If by accident you set a really large value on the master server and that value is transferred to the slaves, then correcting the serial number on the master will not work. The slaves request new data only if the master's serial number is larger than theirs.

You can fix this problem in two ways:

- One fix is to exploit the properties of the sequence space in which the serial numbers live. This procedure involves adding a large value (2^{31}) to the bloated serial number, letting all the slave servers transfer the data, and then setting the serial number to just what you want. This weird arithmetic, with explicit examples, is covered in detail in the O'Reilly book titled *DNS and BIND*; RFC1982 describes the sequence space.
- A sneaky but more tedious way to fix the problem is to change the serial number on the master, kill the slave servers, remove the slaves' backup data files so they are forced to reload from the master, and restart the slaves. It does not work to just remove the files and reload; you must kill and restart the slave servers. This method gets hard if you follow best-practices advice and have your slave servers geographically distributed, especially if you are not the sysadmin for those slave servers.

It's a common mistake to change the data files but forget to update the serial number. Your name server will punish you by failing to propagate your changes to slave servers.

The next four entries in the SOA record are timeout values, in seconds, that control how long data can be cached at various points throughout the world-wide DNS database. Times can also be expressed in units of minutes, hours, days, or weeks by addition of a suffix of `m`, `h`, `d`, or `w`, respectively. For example, `1h30m` means 1 hour and 30 minutes. Timeout values represent a tradeoff between efficiency (it's cheaper to use an old value than to fetch a new one) and accuracy (new values are more accurate). The four timeout fields are called *refresh*, *update*, *expire*, and *minimum*.

The *refresh* timeout specifies how often slave servers should check with the master to see if the serial number of the zone's configuration has changed. Whenever the zone changes, slaves must update their copy of the zone's data. The slave compares the serial numbers; if the master's serial number is larger, the slave requests a zone transfer to update the data. Common values for the *refresh* timeout range from one to six hours (3,600 to 21,600 seconds).

Instead of just waiting passively for slave servers to time out, master servers for BIND notify their slaves every time a zone changes. However, it's possible for an update notification to be lost because of network congestion, so the refresh timeout should still be set to a reasonable value.

If a slave server tries to check the master's serial number but the master does not respond, the slave tries again after the *retry* timeout period has elapsed. Our experience suggests that 20–60 minutes (1,200–3,600 seconds) is a good value.

If a master server is down for a long time, slaves will try to refresh their data many times but always fail. Each slave should eventually decide that the master is never coming back and that its data is surely out of date. The *expire* parameter determines how long the slaves will continue to serve the domain's data authoritatively in the absence of a master. The system should be able to survive if the master server is down for a few days, so this parameter should have a longish value. We recommend a month or two.

The *minimum* parameter in the SOA record sets the time to live for negative answers that are cached. The default for positive answers (i.e., actual records) is specified at the top of the zone file with the $TTL directive. Experience suggests values of several hours to a few days for $TTL and an hour to a few hours for the *minimum*. BIND silently discards any *minimum* values greater than 3 hours.

The $TTL, *expire*, and *minimum* parameters eventually force everyone that uses DNS to discard old data values. The initial design of DNS relied on the fact that host data was relatively stable and did not change often. However, DHCP, mobile hosts, and the Internet explosion have changed the rules. Name servers are desperately trying to cope with the dynamic update and incremental zone transfer mechanisms described later.

NS records

NS records are specified in RFC1035 (1987).

NS (name server) records identify the servers that are authoritative for a zone (that is, all the master and slave servers) and delegate subdomains to other organizations. NS records are usually placed directly after a zone's SOA record.

The format is

```
zone [ttl] [IN] NS hostname
```

For example:

```
                    NS     ns1.atrust.com.
                    NS     ns2.atrust.com.
booklab             NS     ubuntu.booklab.atrust.com.
                    NS     ns1.atrust.com.
```

The first two lines define name servers for the atrust.com domain. No *name* is listed because it is the same as the *name* field of the SOA record that precedes the records; the *name* can therefore be left blank. The *class* is also not listed because IN is the default and does not need to be stated explicitly.

The third and fourth lines delegate a subdomain called booklab.atrust.com to the name servers ubuntu.booklab.atrust.com and ns1.atrust.com. These records are actually part of the booklab subdomain, but they must also appear in the parent zone, atrust.com, in order for the delegation to work. In a similar fashion, NS records for atrust.com are stored in the .com zone file to define the atrust.com subdomain and identify its servers. The .com servers refer queries about hosts in atrust.com to the servers listed in NS records for atrust.com within the .com domain.

See this page for more information about delegation.

The list of name servers in the parent zone should be kept up to date with those in the zone itself, if possible. Nonexistent servers listed in the parent zone can delay name service, although clients will eventually stumble onto one of the functioning name servers. If none of the name servers listed in the parent exist in the child, a so-called lame delegation results; see this page.

Extra servers in the child are OK as long as at least one of the child's servers still has an NS record in the parent. Check your delegations occasionally with **dig** or **drill** to be sure they specify an appropriate set of servers; see this page.

A records

A records are specified in RFC1035 (1987).

A (address) records are the heart of the DNS database. They provide the mapping from hostnames to IP addresses. A host usually has one A record for each of its network interfaces. The format is

```
hostname [ttl] [IN] A ipaddr
```

For example:

```
ns1             IN A      63.173.189.1
```

In this example, the *name* field is not dot-terminated, so the name server adds the default domain to it to form the fully qualified name “ns1.atrust.com.”. The record associates that name with the IP address 63.173.189.1.

AAAA records

AAAA records are specified in RFC3596 (2003).

AAAA records are the IPv6 equivalent of A records. Records are independent of the transport protocol used to deliver them; publishing IPv6 records in your DNS zones does not mean that you must answer DNS queries over IPv6.

The format of an AAAA record is

```
hostname [ttl] [IN] AAAA ipaddr
```

For example:

```
f.root-servers.net.           IN AAAA   2001:500:2f::f
```

Each colon-separated chunk of the address represents four hex digits, with leading zeros usually omitted. Two adjacent colons stand for “enough zeros to fill out the 128 bits of a complete IPv6 address.” An address can contain at most one such double colon.

PTR records

PTR records are specified in RFC1035 (1987).

PTR (pointer) records map from IP addresses back to hostnames. As described in *The DNS namespace*, reverse mapping records live under the in-addr.arpa domain and are named with the bytes of the IP address in reverse order. For example, the zone for the 189 subnet in this example is 189.173.63.in-addr.arpa.

The general format of a PTR record is

```
addr [ttl] [IN] PTR hostname
```

For example, the PTR record in the 189.173.63.in-addr.arpa zone that corresponds to ns1's A record above is

```
1               IN PTR    ns1.atrust.com.
```

The name 1 does not end in a dot and therefore is relative. But relative to what? Not atrust.com—for this sample record to be accurate, the default zone has to be "189.173.63.in-addr.arpa.".

You can set the zone by putting the PTR records for each subnet in their own file. The default domain associated with the file is set in the name server configuration file. Another way to do reverse mappings is to include records such as

```
1.189           IN PTR    ns1.atrust.com.
```

with a default domain of 173.63.in-addr.arpa. Some sites put all reverse records in the same file and use $ORIGIN directives (see this page) to specify the subnet. Note that the hostname ns1.atrust.com ends with a dot to prevent the default domain, 173.63.in-addr.arpa, from being appended to its name.

Since atrust.com and 189.173.63.in-addr.arpa are different regions of the DNS namespace, they constitute two separate zones. Each zone must have its own SOA record and resource records. In addition to defining an in-addr.arpa zone for each real network, you should also define one that takes care of the loopback network (127.0.0.0), at least if you run BIND. See this page for an example.

See this page for more details about subnetting.

This all works fine if subnets are defined on byte boundaries. But how do you handle the reverse mappings for a subnet such as 63.173.189.0/26, where that last byte can be in any of four

subnets: 0-63, 64-127, 128-191, or 192-255? An elegant hack defined in RFC2317 exploits CNAME resource records to accomplish this feat.

It's important that A records match their corresponding PTR records. Mismatched and missing PTR records cause authentication failures that can slow your system to a crawl. This problem is annoying in itself; it can also facilitate denial-of-service attacks against any application that requires the reverse mapping to match the A record.

For IPv6, the reverse mapping information that corresponds to an AAAA address record is a PTR record in the ip6.arpa top-level domain.

The "nibble" format reverses an AAAA address record by expanding each colon-separated address chunk to the full 4 hex digits and then reversing the order of those digits and tacking on ip6.arpa at the end. For example, the PTR record that corresponds to our sample AAAA record on this page would be

```
f.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.f.2.0.0.0.0.5.0.1.0.0.2.ip6.arpa.
    PTR f.root-servers.net.
```

This line has been folded to fit the page. It's unfortunately not very friendly for a sysadmin to have to type or debug or even read. Of course, in your actual DNS zone files, the $ORIGIN statement could hide some of the complexity.

MX records

MX records are specified in RFC1035 (1987).

The mail system uses mail exchanger (MX) records to route mail more efficiently. An MX record preempts the destination specified by the sender of a message, in most cases directing the message to a hub at the recipient's site. This feature puts the flow of mail into a site under the control of local sysadmins instead of senders.

The format of an MX record is

```
name [ttl] [IN] MX preference host ...
```

The records below route mail addressed to user@somehost.atrust.com to the machine mailserver.atrust.com if it is up and accepting email. If mailserver is not available, mail goes to mail-relay3.atrust.com. If neither machine named in the MX records is accepting mail, the fallback behavior is to deliver the mail as originally addressed.

```
somehost        IN MX    10 mailserver.atrust.com.
                IN MX    20 mail-relay3.atrust.com.
```

Hosts with low preference values are tried first: 0 is the most desirable, and 65,535 is as bad as it gets.

MX records are useful in many situations:

- When you have a central mail hub or service provider for incoming mail
- When you want to filter mail for spam or viruses before delivering it
- When the destination host is down
- When the destination host isn't directly reachable from the Internet
- When the local sysadmin knows where mail should be sent better than your correspondents do (i.e., always)

A machine that accepts email on behalf of another host may need to configure its mail transport program to enable this function. See here and here for a discussion of how to set up this configuration on **sendmail** and Postfix email servers, respectively.

Wild card MX records are also sometimes seen in the DNS database:

```
*               IN MX    10 mailserver.atrust.com.
```

At first glance, this record seems like it would save lots of typing and add a default MX record for all hosts. But wild card records don’t quite work as you might expect. They match anything in the *name* field of a resource record that is *not* already listed as an explicit name in another resource record.

Thus, you *cannot* use a star to set a default value for all your hosts. But perversely, you can use it to set a default value for names that are not your hosts. This setup causes lots of mail to be sent to your hub only to be rejected because the hostname matching the star does not in fact belong to your domain. Ergo, avoid wild card MX records.

CNAME records

CNAME records are specified in RFC1035 (1987).

CNAME records assign additional names to a host. These nicknames are commonly used either to associate a function with a host or to shorten a long hostname. The real name is sometimes called the canonical name (hence, “CNAME”). Some examples:

```
ftp             IN CNAME  anchor
kb              IN CNAME  kibblesnbits
```

The format of a CNAME record is

nickname [*ttl*] [IN] CNAME *hostname*

When DNS software encounters a CNAME record, it stops its query for the nickname and re-queries for the real name. If a host has a CNAME record, other records (A, MX, NS, etc.) for that host must refer to its real name, not its nickname. (This rule for CNAMEs was explicitly relaxed for DNSSEC, which adds digital signatures to each DNS resource record set. The RRSIG record for the CNAME refers to the nickname.)

CNAME records can nest eight deep. That is, a CNAME record can point to another CNAME, and that CNAME can point to a third CNAME, and so on, up to seven times; the eighth target must be the real hostname. If you use CNAMEs, the PTR record should point to the real name, not a nickname.

You can avoid CNAMEs altogether by publishing A records for both a host’s real name and its nicknames. This configuration makes lookups slightly faster because the extra layer of indirection is not needed.

RFC1033 requires the “apex” of a zone (sometimes called the “root domain” or “naked domain”) to resolve to one or more A (and/or AAAA) records. The use of a CNAME record is forbidden. In other words, you can do this:

```
www.yourdomain.com.       CNAME some-name.somecloud.com.
```

but not this:

```
yourdomain.com.           CNAME some-name.somecloud.com.
```

This restriction is potentially vexatious, especially when you want the apex to point somewhere within a cloud provider’s network and the server’s IP address is subject to change. In this situation, a static A record is not a reliable option.

To fix the problem, you'll need to use a managed DNS provider (such as AWS Route 53 or CloudFlare) that has developed some kind of system for hacking around the RFC1033 requirement. Typically, these systems let you specify your apex records in a manner similar to a CNAME, but they actually serve A records to the outside world. The DNS provider does the work of keeping the A records synchronized to the actual target.

SRV records

SRV records are specified in RFC2782 (2000).

An SRV record specifies the location of services within a domain. For example, an SRV record lets you query a remote domain for the name of its FTP server. Before SRV, you had to hope the remote sysadmins had followed the prevailing custom and added a CNAME for “ftp” to their server’s DNS records.

SRV records make more sense than CNAMEs for this application and are certainly a better way for sysadmins to move services around and control their use. However, SRV records must be explicitly sought and parsed by clients, so they are not used in all the places they probably should be. They are used extensively by Windows, however.

SRV records resemble generalized MX records with fields that let the local DNS administrator steer and load-balance connections from the outside world. The format is

```
service.proto.name [ttl] [IN] SRV pri weight port target
```

where *service* is a service defined in the IANA assigned numbers database (see the list at iana.org/numbers.htm), *proto* is either tcp or udp, *name* is the domain to which the SRV record refers, *pri* is an MX-style priority, *weight* is a weight used for load balancing among several servers, *port* is the port on which the service runs, and *target* is the hostname of the server that provides the service. To avoid a second round trip, DNS servers usually return the A record of the target with the answer to a SRV query.

A value of 0 for the *weight* parameter means that no special load balancing should be done. A value of “.” for the target means that the service is not run at this site.

Here is an example snitched from RFC2782 and adapted for atrust.com:

```
_ftp._tcp            SRV    0 0 21 ftp-server.atrust.com.

; 1/4 of the connections to old box, 3/4 to the new one
_ssh._tcp            SRV    0 1 22 old-slow-box.atrust.com.
                     SRV    0 3 22 new-fast-box.atrust.com.

; main server on port 80, backup on new box, port 8000
_http._tcp           SRV    0 0 80 www-server.atrust.com.
                     SRV    10 0 8000 new-fast-box.atrust.com.

; so both http://www.atrust.com and http://atrust.com work
_http._tcp.www       SRV    0 0 80 www-server.atrust.com.
                     SRV    10 0 8000 new-fast-box.atrust.com.

; block all other services (target = .)
*._tcp               SRV    0 0 0 .
*._udp               SRV    0 0 0 .
```

This example illustrates the use of both the weight parameter (for SSH) and the priority parameter (HTTP). Both SSH servers are used, with the work being split between them. The backup HTTP server is only used when the principal server is unavailable. All other services are blocked, both for TCP and UDP. However, the fact that other services do not appear in DNS does not mean that they are not actually running, just that you can’t locate them through DNS.

TXT records

TXT records are specified in RFC1035 (1987).

A TXT record adds arbitrary text to a host's DNS records. For example, some sites have a TXT record that identifies them:

```
                IN TXT    "Applied Trust Engineering, Boulder, CO, USA"
```

This record directly follows the SOA and NS records for the atrust.com zone and so inherits the *name* field from them.

The format of a TXT record is

```
name [ttl] [IN] TXT info ...
```

All *info* items must be quoted. Be sure the quotes are balanced—missing quotes wreak havoc with your DNS data because all the records between the missing quote and the next occurrence of a quote mysteriously disappear.

As with other resource records, servers return TXT records in random order. To encode long items such as addresses, use long text lines rather than a collection of several TXT records.

Because TXT records have no particular format, they are sometimes used to add information for other purposes without requiring changes to the DNS system itself.

SPF, DKIM, and DMARC records

SPF (Sender Policy Framework), DKIM (DomainKeys Identified Mail), and DMARC (Domain-based Message Authentication, Reporting, and Conformance) are standards that attempt to stem the Internet's ever-increasing flow of unsolicited commercial email (aka UCE or spam). Each of these systems distributes spam-fighting information through DNS in the form of TXT records, so they are not true DNS record types. For that reason, we cover these systems in Chapter 18, *Electronic Mail*. See the material that starts here. (There is in fact a defined DNS record type for SPF; however, the TXT record version is preferred.)

DNSSEC records

Five resource record types are currently associated with DNSSEC, the cryptographically secured version of DNS.

DS and DNSKEY records store various types of keys and fingerprints. RRSIGs contain the signatures of other records in the zone (record sets, really). Finally, NSEC and NSEC3 records give DNS servers a way to sign nonexistent records, thus extending cryptographic security to negative query responses. These six records differ from most in that they are generated by software tools rather than being typed in by hand.

DNSSEC is a big topic in its own right, so we discuss these records and their use in the DNSSEC section, which begins here.

16.6 The BIND software

BIND, the Berkeley Internet Name Domain system, is an open source software package from the Internet Systems Consortium (ISC) that implements DNS for Linux, UNIX, macOS, and Windows systems. There have been three main flavors of BIND: BIND 4, BIND 8, and BIND 9, with BIND 10 currently under development by ISC. We cover only BIND 9 in this book.

Components of BIND

The BIND distribution has four major components:

- A name server daemon called **named** that answers queries
- A resolver library that queries DNS servers on behalf of users
- Command-line interfaces to DNS: **nslookup**, **dig**, and **host**
- A program to remotely control **named** called **rndc**

The hardest BIND-related sysadmin chore is probably sorting through all the myriad options and features that BIND supports and determining which ones make sense for your situation.

Configuration files

A complete configuration for **named** consists of the config file (**named.conf**), the zone data files that contain address mappings for each host, and the root name server hints file. Authoritative servers need **named.conf** and zone data files for each zone for which they are the master server; caching servers need **named.conf** and the root hints file.

named.conf has its own format; all the other files are collections of individual DNS data records whose formats were discussed in *The DNS database*.

The **named.conf** file specifies the roles of this host (master, slave, stub, or caching-only) and the manner in which it should obtain its copy of the data for each zone it serves. It's also the place where options are specified—both global options related to the overall operation of **named** and server- or zone-specific options that apply to only a portion of the DNS traffic.

The config file consists of a series of statements whose syntax we describe as they are introduced in subsequent sections. The format is unfortunately quite fragile—a missing semicolon or unbalanced quote can wreak havoc.

Comments can appear anywhere that whitespace is appropriate. C, C++, and shell-style comments are all understood:

```
/* This is a comment that can span lines. */
// Everything to the end of the line is a comment.
# Everything to the end of the line is a comment.
```

Each statement begins with a keyword that identifies the type of statement. There can be more than one instance of each type of statement, except for options and logging. Statements and parts of statements can also be left out, invoking default behavior for the missing items.

Table 16.4 lists the available statements; the Link column points to our discussion of each statement in the upcoming sections.

Table 16.4: Statements used in named.conf

Statement	Link	Function
`include`	here	Interpolates a file
`options`	here	Sets global configuration options/defaults
`acl`	here	Defines access control lists
`key`	here	Defines authentication information
`server`	here	Specifies per-server options
`masters`	here	Defines a list of masters for stub and slave zones
`logging`	here	Specifies logging categories and their destinations
`statistics-channels`	here	Outputs real-time statistics in XML format
`zone`	here	Defines a zone of resource records
`controls`	here	Defines channels used to control **named** with **rndc**
`view`	here	Defines a view of the zone data
`lwres`	–	Specifies that **named** should be a resolver, too

Before describing these statements and the way they are used to configure **named**, we need to describe a data structure that is used in many of the statements, the address match list. An address match list is a generalization of an IP address that can include the following items:

- An IP address, either IPv4 or IPv6 (e.g., 199.165.145.4 or fe80::202:b3ff:fe1e:8329)
- An IP network specified with a CIDR netmask (e.g., 199.165/16; see this page)
- The name of a previously defined access control list (see this page)
- The name of a cryptographic authentication key
- The `!` character to negate things

Address match lists are used as parameters to many statements and options. Some examples:

```
{ ! 1.2.3.13; 1.2.3/24; };
{ 128.138/16; 198.11.16/24; 204.228.69/24; 127.0.0.1; };
```

The first of these lists excludes the host 1.2.3.13 but includes the rest of the 1.2.3.0/24 network; the second defines the networks assigned to the University of Colorado. The braces and final semicolons are not really part of the address match lists but are included here for illustration; they would be part of the enclosing statements of which the address match lists are a part.

When an IP address or network is compared to a match list, the list is searched in order until a match is found. This “first match” algorithm makes the ordering of entries important. For example, the first address match list above would not have the desired effect if the two entries were reversed, because 1.2.3.13 would succeed in matching 1.2.3.0/24 and the negated entry would never be encountered.

Now, on to the statements! Some are short and sweet; others almost warrant a chapter unto themselves.

The include statement

To break up or better organize a large configuration, you can put different portions of the configuration in separate files. Subsidiary files are brought into **named.conf** with an include statement:

```
include "path";
```

If the *path* is relative, it is interpreted with respect to the directory specified in the directory option.

A common use of include is to incorporate cryptographic keys that should not be world-readable. Rather than forbidding read access to the entire **named.conf** file, some sites keep keys in files with restrictive permissions that only **named** can read. Those files are then included into **named.conf**.

Many sites put zone statements in a separate file and use the include statement to pull them in. This configuration helps separate the parts of the configuration that are relatively static from those that are likely to change frequently.

The options statement

The options statement specifies global options, some of which might later be overridden for particular zones or servers. The general format is

```
options {
    option;
    option;
    ...
};
```

If no options statement is present in **named.conf**, default values are used.

BIND has a lot of options—too many, in fact. The 9.9 release has more than 170, which is a lot for sysadmins to wrap their heads around. Unfortunately, as soon as the BIND folks think about removing some of the options that were a bad idea or that are no longer necessary, they get pushback from sites who use and need those obscure options. We do not cover the whole gamut of BIND options here; we have biased our coverage and discuss only the ones whose use we recommend. (We also asked the BIND developers for their suggestions on which options to cover, and we followed their advice.)

For more complete coverage of the options, see one of the books on DNS and BIND listed at the end of this chapter. You can also refer to the documentation shipped with BIND. The **ARM** document in the **doc** directory of the distribution describes each option and shows both syntax and default values. The file **doc/misc/options** also contains a complete list of options.

The default values are listed in square brackets beside each option. For most sites, the default values are just fine. Options are listed in no particular order.

File location options

```
directory "path";          [directory where the server was started]
key-directory "path";      [same as directory entry]
```

The directory statement causes **named** to **cd** to the specified directory. Wherever relative pathnames appear in **named**'s configuration files, they are interpreted relative to this directory. The *path* should be an absolute path. Any output files (debugging, statistics, etc.) are also written in this directory. The key-directory is where cryptographic keys are stored; it should not be world-readable.

We like to put all the BIND-related configuration files (other than **named.conf** and **resolv.conf**) in a subdirectory beneath **/var** (or wherever you keep your configuration files for other programs). We use **/var/named** or **/var/domain**.

Name server identity options

```
version "string";          [real version number of the server]
hostname "string";         [real hostname of the server]
server-id "string";        [none]
```

The version string identifies the version of the name server software running on the server. The hostname string identifies the server itself, as does the server-id string. These options let you lie about the true values. Each of them puts data into CHAOS-class (as opposed to IN-class, the default) TXT records where curious onlookers can search for them with the **dig** command.

The hostname and server-id parameters are additions motivated by the use of anycast routing to duplicate instances of the root and gTLD servers.

Zone synchronization options

```
notify yes | master-only | explicit | no;   [yes]
also-notify server-ipaddrs;                 [empty]
allow-notify address-match-list;            [empty]
```

The notify and also-notify clauses apply only to master servers. allow-notify applies only to slave servers.

Early versions of BIND synchronized zone files between master and slave servers only when the refresh timeout in the zone's SOA record had expired. These days, the master **named** automatically notifies its peers whenever the corresponding zone database has been reloaded, as long as notify is set to yes. The slave servers can then rendezvous with the master to see if the file has changed, and if so, to update their copies of the zone data.

You can use notify both as a global option and as a zone-specific option. It makes the zone files converge much more quickly after you make changes. By default, every authoritative server sends updates to every other authoritative server (a system termed "splattercast" by Paul Vixie). Setting notify to master-only curbs this chatter by sending notifications only to slave servers of zones for which this server is the master. If the notify option is set to explicit, then **named** only notifies the servers listed in the also-notify clause.

See this page for more information about stub zones.

---

named normally figures out which machines are slave servers of a zone by looking at the zone's NS records. If also-notify is specified, a set of additional servers that are not advertised with NS records can also be notified. This tweak is sometimes necessary when your site has internal servers.

The target of an also-notify is a list of IP addresses and, optionally, ports. You must use the allow-notify clause in the secondaries' **named.conf** files if you want a name server other than the master to notify them.

For servers with multiple network interfaces, additional options specify the IP address and port to use for outgoing notifications.

Query recursion options

```
recursion yes | no;                          [yes]
allow-recursion { address-match-list };  [all hosts]
```

The recursion option specifies whether **named** should process queries recursively on behalf of your users. You can enable this option on an authoritative server of your zones' data, but that's frowned upon. The best-practice recommendation is to keep authoritative servers and caching servers separate.

If this name server should be recursive for your clients, set recursion to yes and include an allow-recursion clause so that **named** can distinguish queries that originate at your site from remote queries. **named** will act recursively for the former and nonrecursively for the latter. If your name server answers recursive queries for everyone, it is called an open resolver and can become a reflector for certain kinds of attacks; see RFC5358.

Cache memory use options

```
recursive-clients number;     [1000]
max-cache-size number;        [unlimited]
```

If your server is handling an extraordinary amount of traffic, you may need to tweak the recursive-clients and max-cache-size options. recursive-clients controls the number of recursive lookups the server will process simultaneously; each requires about 20KiB of memory. max-cache-size limits the amount of memory the server will use for caching answers to queries. If the cache grows too large, **named** deletes records before their TTLs expire to keep memory use under the limit.

IP port utilization options

```
use-v4-udp-ports { range begin end; };    [range 1024 65535]
use-v6-udp-ports { range begin end; };    [range 1024 65535]

avoid-v4-udp-ports { port-list };     [empty]
avoid-v6-udp-ports { port-list };     [empty]

query-source v4-address [port]    [any]    # CAUTION, don't use port
query-source-v6 v6-address [port] [any]    # CAUTION, don't use port
```

Source ports have become important in the DNS world because of a DNS protocol weakness discovered by Dan Kaminsky, one that allows DNS cache poisoning when name servers use predictable source ports and query IDs. The use- and avoid- options for UDP ports (together with changes to the **named** software) have mitigated this attack.

Some sysadmins formerly set a specific outgoing port number so they could configure their firewalls to recognize it and accept UDP packets only for that port. However, this configuration is no longer safe in the post-Kaminsky era. Don't use the query-source address options to specify a fixed outgoing port for DNS queries or you will forfeit the Kaminsky protection that a large range of random ports provides.

The defaults for the use-* ranges are fine, and you shouldn't need to change them. But be aware of the implications: queries go out from random high-numbered ports, and the answers come back to those same ports. Ergo, your firewall must be prepared to accept UDP packets on random high-numbered ports.

If your firewall blocks certain ports in this range (for example, port 2049 for RPC) then you have a small problem. When your name server sends a query and uses one of the blocked ports as its source, the firewall blocks the answer. The name server eventually stops waiting and sends out the query again. Not fatal, but annoying to the user caught in the crossfire.

To forestall this problem, use the avoid-* options to make BIND stay away from the blocked ports. Any high-numbered UDP ports blocked by your firewall should be included in the list. (Some firewalls are stateful and may be smart enough to recognize the DNS answer as being paired with the corresponding query of a second ago. Such firewalls don't need help from this option.) If you update your firewall in response to some threatened attack, be sure to update the port list here, too.

The query-source options let you specify the IP address to be used on outgoing queries. For example, you might need to use a specific IP address to get through your firewall or to distinguish between internal and external views.

Forwarding options

```
forwarders { in_addr; in_addr; ... }; [empty list]
forward only | first;                 [first]
```

Instead of having every name server perform its own external queries, you can designate one or more servers as forwarders. A run-of-the-mill server can look in its cache and in the records for which it is authoritative. If it doesn't find the answer it's looking for, it can then send the query on to a forwarder host. That way, the forwarders build up caches that benefit the entire site. The designation is implicit—nothing in the configuration file of the forwarder explicitly says "Hey, you're a forwarder."

The forwarders option lists the IP addresses of the servers you want to use as forwarders. They are queried in turn. The use of a forwarder circumvents the normal DNS procedure of starting at a root server and following the chain of referrals. Be careful not to create forwarding loops.

A forward-only server caches answers and queries forwarders, but it never queries anyone else. If the forwarders do not respond, queries fail. A forward-first server prefers to deal with forwarders, but if they do not respond, the forward-first server will complete queries on its own.

Since the forwarders option has no default value, forwarding does not occur unless it has been specifically configured.

You can turn on forwarding either globally or within individual zone statements.

Permission options

```
allow-query { address-match-list };          [all hosts]
allow-query-cache { address-match-list };    [all hosts]
allow-transfer { address-match-list };       [all hosts]
allow-update { address-match-list };         [none]
blackhole { address-match-list };            [empty]
```

These options specify which hosts (or networks) can query your name server or its cache, request block transfers of your zone data, or dynamically update your zones. These match lists are a low-rent form of security and are susceptible to IP address spoofing, so there's some risk in relying on them. It's probably not a big deal if someone tricks your server into answering a DNS query, but avoid the allow_update and allow_transfer clauses; use cryptographic keys instead.

The blackhole address list identifies servers that you never want to talk to; **named** does not accept queries from these servers and will never ask them for answers.

Packet size options

```
edns-udp-size number;   [4096]
max-udp-size number;    [4096]
```

All machines on the Internet must be capable of reassembling a fragmented UDP packet of 512 bytes or fewer. Although this conservative requirement made sense in the 1980s, it is laughably small by modern standards. Modern routers and firewalls can handle much larger packets, but it only takes one bad link in the IP chain to spoil the whole path.

Since DNS by default uses UDP for queries and since DNS responses are often larger than 512 bytes, DNS administrators have to worry about large UDP packets being dropped. If a large reply gets fragmented and your firewall only lets the first fragment through, the receiver gets a truncated answer and retries the query with TCP. TCP is a more expensive protocol than UDP, and busy servers at the root or TLDs don't need increased TCP traffic because of everybody's broken firewalls.

The edns-udp-size option sets the reassembly buffer size that the name server advertises through EDNS0, the extended DNS protocol. The max-udp-size option sets the maximum packet size that the server will actually send. Both sizes are in bytes. Reasonable values are in the 512–4,096 byte range.

DNSSEC control options

```
dnssec-enable yes | no;                    [yes]
dnssec-validation yes | no;                [yes]
dnssec-must-be-secure domain yes | no;     [none]
```

These options configure support for DNSSEC. See the sections starting on this page for a general discussion of DNSSEC and a detailed description of how to set up DNSSEC at your site.

An authoritative server needs the dnssec-enable option turned on. A recursive server needs the dnssec-enable and dnssec-validation options turned on.

dnssec-enable and dnssec-validation are turned on by default, which has various implications:

- An authoritative server of a signed zone answering a query with the DNSSEC-aware bit turned on answers with the requested resource records and their signatures.
- An authoritative server of a signed zone answering a query with the DNSSEC-aware bit *not* set answers with just the requested resource records, as in the pre-DNSSEC era.
- An authoritative server of an unsigned zone answers queries with just the requested resource records; there are no signatures to include.
- A recursive server sends queries on behalf of users with the DNSSEC-aware bit set.
- A recursive server validates the signatures included with signed replies before returning data to the user.

The dnssec-must-be-secure option allows you to specify that you will only accept secure answers from particular domains, or, alternatively, that you don't care and that insecure answers are OK. For example, you might say yes to the domain important-stuff.mybank.com and no to the domain marketing.mybank.com.

Statistics-gathering option

```
zone-statistics yes | no      [no]
```

This option causes **named** to maintain per-zone statistics as well as global statistics. Run **rndc stats** to dump the statistics to a file.

Performance tuning options

```
clients-per-query int;     [10]        # Clients waiting on same query
max-clients-per-query int; [100]       # Max clients before server drops
datasize int;              [unlimited] # Max memory server may use
files int;                 [unlimited] # Max # of concurrent open files
lame-ttl int;              [10min]     # Secs to cache lame server data
max-acache-size int;       [ ]         # Cache size for additional data
max-cache-size int;        [ ]         # Max memory for cached answers
max-cache-ttl int;         [1week]     # Max TTL for caching positive data
max-journal-size int;      [ ]         # Max size of transaction journal
max-ncache-ttl int;        [3hrs]      # Max TTL for caching negative data
tcp-clients int;           [100]       # Max simultaneous TCP clients
```

This long list of options can be used to tune **named** to run well on your hardware. We don't describe them in detail, but if you are having performance problems, these options may suggest a starting point for your tuning efforts.

Whew, we are finally done with the options. Let's get on to the rest of the configuration language!

The acl statement

An access control list is just an address match list with a name:

```
acl acl-name {
    address-match-list
};
```

You can use an *acl-name* anywhere an address match list is called for.

An `acl` must be a top-level statement in **named.conf**, so don't try sneaking it in amid your other option declarations. Also keep in mind that **named.conf** is read in a single pass, so access control lists must be defined before they are used. Four lists are predefined:

- `any` – all hosts
- `localnets` – all hosts on the local network(s)
- `localhost` – the machine itself
- `none` – nothing

The `localnets` list includes all of the networks to which the host is directly attached. In other words, it's a list of the machine's network addresses modulo their netmasks.

The (TSIG) key statement

The key statement defines a "shared secret" (that is, a password) that authenticates communication between two servers; for example, between the master server and a slave for a zone transfer, or between a server and the **rndc** process that controls it. Background information about BIND's support for cryptographic authentication is given in the *DNS security issues* section starting on this page. Here, we touch briefly on the mechanics of the process.

To build a key record, you specify both the cryptographic algorithm that you want to use and the shared secret, represented as a base-64-encoded string (see page 561 for details):

```
key key-id {
    algorithm string;
    secret string;
};
```

As with access control lists, the *key-id* must be defined with a key statement before it is used. To associate the key with a particular server, just include *key-id* in the keys clause of that server's server statement. The key is used both to verify requests from that server and to sign the responses to those requests.

The shared secret is sensitive information and should not be kept in a world-readable file. Use an include statement to bring it into the **named.conf** file.

The server statement

named can potentially talk to many servers, not all of them running current software and not all of them even nominally sane. The server statement tells **named** about the characteristics of its remote peers. The server statement can override defaults for a particular server; it's not required unless you want to configure keys for zone transfers.

```
server ip_addr {
    bogus yes | no;                          [no]
    provide-ixfr yes | no;                   [yes]
    request-ixfr yes | no;                   [yes]
    keys { key-id; key-id; ... };            [none]
    transfer-source ip-address [port];       [closest interface]
    transfer-source-v6 ipv6-address [port];  [closest interface]
};
```

You can use a server statement to override the values of global configuration options for individual servers. Just list the options for which you want nondefault behavior. We have not shown all the server-specific options, just the ones we think you might need. See the BIND documentation for a complete list.

If you mark a server as being bogus, **named** won't send any queries its way. This directive should be reserved for servers that are in fact bogus. bogus differs from the global option blackhole in that it suppresses only outbound queries. By contrast, the blackhole option completely eliminates all forms of communication with the listed servers.

A BIND name server acting as master for a dynamically updated zone performs incremental zone transfers if provide-ixfr is set to yes. Likewise, a server acting as a slave requests incremental zone transfers from the master if request-ixfr is set to yes. Dynamic DNS is discussed in detail here.

The keys clause identifies a key ID that has been previously defined in a key statement for use with transaction signatures (see this page). Any requests sent to the remote server are signed with this key. Requests originating at the remote server are not required to be signed, but if they are, the signature will be verified.

The transfer-source clauses give the IPv4 or IPv6 address of the interface (and optionally, the port) that should be used as a source address (port) for zone transfer requests. This clause is only needed when the system has multiple interfaces and the remote server has specified a specific IP address in its allow-transfer clause; the addresses must match.

The masters statement

The masters statement lets you name a set of one or more master servers by specifying their IP addresses and cryptographic keys. You can then use this defined name in the masters clause of zone statements instead of repeating the IP addresses and keys.

How can there be more than one master? See this page.

The masters facility is helpful when multiple slave or stub zones get their data from the same remote servers. If the addresses or cryptographic keys of the remote servers change, you can update the masters statement that introduces them rather than changing many different zone statements.

The syntax is

```
masters name { ip_addr [port ip_port] [key key] ; ... } ;
```

The logging statement

named is the current holder of the “most configurable logging system on Earth” award. Syslog put the prioritization of log messages into the programmer’s hands and the disposition of those messages into the sysadmin’s hands. But for a given priority, the sysadmin had no way to say, “I care about this message but not about that message.” BIND added categories that classify log messages by type, and channels that broaden the choices for the disposition of messages. Categories are determined by the programmer, and channels by the sysadmin.

Since logging requires quite a bit of explanation and is somewhat tangential, we discuss it in the debugging section beginning here.

The statistics-channels statement

The statistics-channels statement lets you connect to a running **named** with a browser to view statistics as they are accumulated. Since the stats of your name server might be sensitive, you should restrict access to this data to trusted hosts at your own site. The syntax is

```
statistics-channels {
    inet (ip-addr | *) port port# allow { address-match-list } ;
    ...
}
```

You can include multiple inet-port-allow sequences. The defaults are open, so be careful! The IP address defaults to any, the port defaults to port 80 (normal HTTP), and the allow clause defaults to letting anyone connect. To use statistics channels, **named** must have been compiled with **libxml2**.

The zone statement

zone statements are the heart of the **named.conf** file. They tell **named** about the zones for which it is authoritative and set the options that are appropriate for managing each zone. A zone statement is also used by a caching server to preload the root server hints (that is, the names and addresses of the root servers, which bootstrap the DNS lookup process).

The exact format of a zone statement varies, depending on the role that **named** is to play with respect to that zone. The possible zone types are master, slave, hint, forward, stub, and delegation-only. We do not describe stub zones (used by BIND only) or delegation-only zones (used to stop the use of wild card records in top-level zones to advertise a registrar's services). The following brief sections describe the other zone types.

Many of the global options covered earlier can become part of a zone statement and override the previously defined values. We have not repeated those options here, except to mention certain ones that are frequently used.

Configuring the master server for a zone

Here's the format you need for a zone of which this **named** is the master server:

```
zone "domain-name" {
    type master;
    file "path";
};
```

The *domain-name* in a zone specification must always appear in double quotes.

The zone's data is kept on disk in a human-readable (and human-editable) file. The filename has no default, so you must provide a file statement when declaring a master zone. A zone file is just a collection of DNS resource records in the formats described starting here.

Other server-specific attributes are also frequently specified within the zone statement. For example:

```
    allow-query { address-match-list };      [any]
    allow-transfer { address-match-list };   [any]
    allow-update { address-match-list };     [none]
    zone-statistics yes | no                 [no]
```

The access control options are not required, but it's a good idea to use them. They accept any kind of address match list, so you can configure security either in terms of IP addresses or in terms of TSIG encryption keys. As usual, encryption keys are safer.

See this page for more information about dynamic updates.

If dynamic updates are used for this zone, the allow-update clause must be present with an address match list that limits the hosts from which updates can occur. Dynamic updates apply only to master zones; the allow-update clause cannot be used for a slave zone. Be sure that this clause includes just your own machines (e.g., DHCP servers) and not the whole Internet. (You also need ingress filtering at your firewall; see this page. Better yet, use TSIG for authentication.)

The zone-statistics option makes **named** keep track of query/response statistics such as the number and percentage of responses that were referrals, that resulted in errors, or that demanded recursion.

With all these zone-specific options (and about 40 more we have not covered!), the configuration is starting to sound complicated. However, a master zone declaration consisting of nothing but a pathname to the zone file is perfectly reasonable. Here is an example, slightly modified, from the BIND documentation:

```
zone "example.com" {
    type master;
    file "forward/example.com";
    allow-query { any; };
    allow-transfer { my-slaves; };
}
```

Here, my-slaves would be an access control list you had previously defined.

Configuring a slave server for a zone

The zone statement for a slave is similar to that of a master:

```
zone "domain-name" {
    type slave;
    file "path";
    masters { ip_addr [port ip_port] [key keyname]; ... };
    allow-query { address-match-list };          [any]
};
```

Slave servers normally maintain a complete copy of their zone's database. The file statement specifies a local file in which the replicated database can be stored. Each time the server fetches a new copy of the zone, it saves the data in this file. If the server crashes and reboots, the file can then be reloaded from the local disk without being transferred across the network.

You shouldn't edit this cache file, since it's maintained by **named**. However, it can be interesting to inspect if you suspect you have made an error in the master server's data file. The slave's disk file shows you how **named** has interpreted the original zone data. In particular, relative names and $ORIGIN directives have all been expanded. If you see a name in the data file that looks like one of these

128.138.243.151.cs.colorado.edu.

anchor.cs.colorado.edu.cs.colorado.edu.

you can be pretty sure that you forgot a trailing dot somewhere.

The masters clause lists the IP addresses of one or more machines from which the zone database can be obtained. It can also contain the name of a list of masters defined by a previous masters statement.

We said that only one machine can be the master for a zone, so why is it possible to list more than one address? Two reasons. First, the master machine might have more than one network interface and therefore more than one IP address. It's possible for one interface to become unreachable (because of network or routing problems) while others are still accessible. Therefore, it's a good practice to list all the master server's topologically distinct addresses.

Second, **named** doesn't care where the zone data comes from. It can pull the database just as easily from a slave server as from the master. You could use this feature to allow a well-connected slave server to serve as a sort of backup master, since the IP addresses are tried in order until a working server is found. In theory, you can also set up a hierarchy of servers, with one master serving several second-level servers, which in turn serve many third-level servers.

Setting up the root server hints

Another form of zone statement points **named** toward a file from which it can preload its cache with the names and addresses of the root name servers.

```
zone "." {
    type hint;
    file "path";
};
```

The "hints" are a set of DNS records that list servers for the root domain. They're needed to give a recursive, caching instance of **named** a place to start searching for information about other sites' domains. Without them, **named** would only know about the domains it serves and their subdomains.

When **named** starts, it reloads the hints from one of the root servers. Ergo, you'll be fine as long as your hints file contains at least one valid, reachable root server. As a fallback, the root server hints are also compiled into **named**.

The hints file is often called **root.cache**. It contains the response you would get if you queried any root server for the name server records in the root domain. In fact, you can generate the hints file in exactly this way with **dig**. For example:

```
$ dig @f.root-servers.net . ns > root.cache
```

Mind the dot. If f.root-servers.net is not responding, you can run the query without specifying a particular server:

```
$ dig . ns > root.cache
```

The output will be similar; however, you will be obtaining the list of root servers from the cache of a local name server, not from an authoritative source. That should be just fine—even if you have not rebooted or restarted your name server for a year or two, it has been refreshing its root server records periodically as their TTLs expire.

Setting up a forwarding zone

A zone of type forward overrides **named**'s default query path (ask the root first, then follow referrals, as described on this page) for a particular domain:

```
zone "domain-name" {
    type forward;
    forward only | first;
    forwarders { ip_addr; ip_addr; ... };
};
```

You might use a forward zone if your organization had a strategic working relationship with some other group or company and you wanted to funnel traffic directly to that company's name servers, bypassing the standard query path.

The controls statement for rndc

The controls statement limits the interaction between the running **named** process and **rndc**, the program a sysadmin can use to signal and control it. **rndc** can start and stop **named**, dump its state, put it in debug mode, etc. **rndc** operates over the network, and with improper configuration it might let anyone on the Internet mess with your name server. The syntax is

```
controls {
    inet addr port port allow { address-match-list } keys { key_list };
}
```

rndc talks to **named** on port 953 if you don't specify a different port.

Allowing your name server to be controlled remotely is both handy and dangerous. Strong authentication through a key entry in the allow clause is required; keys in the address match list are ignored and must be explicitly stated in the keys clause of the controls statement.

You can use the **rndc-confgen** command to generate an authentication key for use between **rndc** and **named**. There are essentially two ways to set up use of the key: you can have both **named** and **rndc** consult the same configuration file to learn the key (e.g., **/etc/rndc.key**), or you can include the key in both the **rndc**'s and **named**'s configuration files (**/etc/rndc.conf** for **rndc** and **/etc/named.conf** for **named**). The latter option is more complicated, but it's necessary when **named** and **rndc** will be running on different computers. **rndc-confgen -a** sets up keys for localhost access.

When no controls statement is present, BIND defaults to the loopback address for the address match list and looks for the key in **/etc/rndc.key**. Because strong authentication is mandatory, the **rndc** command cannot control **named** if no key exists. This precaution may seem draconian, but consider: even if **rndc** worked only from 127.0.0.1 and this address was blocked from the outside world at your firewall, you would still be trusting all local users not to tamper with your name server. Any user could **telnet** to the control port and type "stop"—quite an effective denial-of-service attack.

Here is an example of the output (to standard out) from **rndc-confgen** when a 256-bit key is requested. We chose 256 bits because it fits on the page; you would normally choose a longer key and redirect the output to **/etc/rndc.conf**. The comments at the bottom of the output show the lines you need to add to **named.conf** to make **named** and **rndc** play together.

```
$ ./rndc-confgen -b 256
# Start of rndc.conf
key "rndc-key" {
    algorithm hmac-md5;
    secret "orZuz5amkUnEp52zlHxD6cd5hACldOGsG/elP/dv2IY=";
};

options {
    default-key "rndc-key";
    default-server 127.0.0.1;
    default-port 953;
};
# End of rndc.conf

# Use the following in named.conf, adjusting the allow list as needed:
# key "rndc-key" {
#   algorithm hmac-md5;
#   secret "orZuz5amkUnEp52zlHxD6cd5hACldOGsG/elP/dv2IY=";
# };
#
# controls {
#   inet 127.0.0.1 port 953
#   allow { 127.0.0.1; } keys { "rndc-key"; };
# };
# End of named.conf
```

16.7 Split DNS and the view statement

Many sites want the internal view of their network to be different from the view seen from the Internet. For example, you might reveal all of a zone's hosts to internal users but restrict the external view to a few well-known servers. Or, you might expose the same set of hosts in both views but supply additional (or different) records to internal users. For example, the MX records for mail routing might point to a single mail hub machine from outside the domain but point to individual workstations from the perspective of internal users.

See this page for more information about private address spaces.

A split DNS configuration is especially useful for sites that use RFC1918 private IP addresses on their internal networks. For example, a query for the hostname associated with IP address 10.0.0.1 can never be answered by the global DNS system, but it is meaningful within the context of the local network. Of the queries arriving at the root name servers, 4%–5% are either *from* an IP address in one of the private address ranges or *about* one of these addresses. Neither can be answered; both are the result of misconfiguration, either of BIND's split DNS or of Microsoft's "domains."

The view statement packages up a couple of access lists that control which clients see which view, some options that apply to all the zones in the view, and finally, the zones themselves. The syntax is

```
view view-name {
    match-clients { address-match-list } ;        [any]
    match-destinations { address-match-list } ;   [any]
    match-recursive-only yes | no;                [no]
    view-option; ...
    zone-statement; ...
} ;
```

Views have always had a match-clients clause that filters on queries' source IP addresses. It typically serves internal and external views of a site's DNS data. For finer control, you can now also filter on the query destination address and can require recursive queries.

The match-destinations clause looks at the destination address to which a query was sent. It's useful on multihomed machines (that is, machines with more than one network interface) when you want to serve different DNS data depending on the interface on which the query arrived. The match-recursive-only clause requires queries to be recursive as well as to originate at a permitted client. Iterative queries let you see what is in a site's cache; this option prevents it.

Views are processed in order, so put the most restrictive views first. Zones in different views can have the same names but take their data from different files. Views are an all-or-nothing

proposition; if you use them, all zone statements in your **named** configuration file must appear in the context of a view.

Here is a simple example from the BIND 9 documentation. The two views define the same zone, but with different data.

```
view "internal" {
    match-clients { our_nets; };  // Only internal networks
    recursion yes;                // Internal clients only
    zone "example.com" {          // Complete view of zone
        type master;
        file "example-internal.db";
    };
};

view "external" {
    match-clients { any; };       // Allow all queries
    recursion no;                 // But no recursion
    zone "example.com" {          // Only "public" hosts
        type master;
        file "example-external.db";
    }
};
```

If the order of the views were reversed, no one would ever see the internal view. Internal hosts would match the any value in the match-clients clause of the external view before they reached the internal view.

Our second DNS configuration example starting on this page provides an additional example of views.

16.8 BIND CONFIGURATION EXAMPLES

Now that we have explored the wonders of **named.conf**, let’s look at two complete configuration examples:

- The localhost zone
- A small security company that uses split DNS

The localhost zone

The IPv4 address 127.0.0.1 refers to a host itself and should be mapped to the name “localhost.”. Some sites map the address to “localhost.*localdomain*.” and some do both. The corresponding IPv6 address is ::1.

If you forget to configure the localhost zone, your site may end up querying the root servers for localhost information. The root servers receive so many of these queries that the operators are considering adding a generic mapping between localhost and 127.0.0.1 at the root level. Other unusual names in the popular “bogus TLD” category are lan, home, localdomain, and domain.

The forward mapping for the name localhost can be defined in the forward zone file for the domain (with an appropriate $ORIGIN statement) or in its own file. Each server, even a caching server, is usually the master for its own reverse localhost domain.

Here are the lines in **named.conf** that configure localhost:

```
zone "localhost" {               // localhost forward zone
    type master;
    file "localhost";
    allow-update { none; };
};

zone "0.0.127.in-addr.arpa" {    // localhost reverse zone
    type master;
    file "127.0.0";
    allow-update { none; };
};
```

The corresponding forward zone file, **localhost**, contains the following lines:

```
$TTL 30d
; localhost.
@   IN          SOA       localhost. postmaster.localhost. (
                            2015050801 ; Serial
                            3600       ; Refresh
                            1800       ; Retry
                            604800     ; Expiration
                            3600 )     ; Minimum

                NS      localhost.
                A       127.0.0.1
```

The reverse file, **127.0.0**, contains:

```
$TTL 30d
; 0.0.127.in-addr.arpa
@   IN          SOA     localhost. postmaster.localhost. (
                            2015050801 ; Serial
                            3600       ; Refresh
                            1800       ; Retry
                            604800     ; Expiration
                            3600 )     ; Minimum

                NS      localhost.
1               PTR     localhost.
```

The mapping for the localhost address (127.0.0.1) never changes, so the timeouts can be large. Note the serial number, which encodes the date; the file was last changed in 2015. Also note that only the master name server is listed for the localhost domain. The meaning of @ here is “0.0.127.in-addr.arpa.”.

A small security company

Our second example is for a small company that specializes in security consulting. They run BIND 9 on a recent version of Red Hat Enterprise Linux and use views to implement a split DNS system in which internal and external users see different host data. They also use private address space internally; queries about those addresses should never escape to the Internet to clutter up the global DNS system. Here is their **named.conf** file, reformatted and commented a bit:

```
options {
    directory "/var/domain";
    version "root@atrust.com";
    allow-transfer { 82.165.230.84; 71.33.249.193; 127.0.0.1; };
    listen-on { 192.168.2.10; 192.168.2.1; 127.0.0.1; 192.168.2.12; };
};

include "atrust.key";             // Mode 600 file

controls {
    inet 127.0.0.1 allow { 127.0.0.1; } keys { atkey; };
};

view "internal" {

    match-clients { 192.168.0.0/16; 206.168.198.192/28; 172.29.0.0/24; };
    recursion yes;

    include "infrastructure.zones";   // Root hints, localhost forw + rev

    zone "atrust.com" {           // Internal forward zone
        type master;
        file "internal/atrust.com";
    };
    zone "1.168.192.in-addr.arpa" {  // Internal reverse zone
        type master;
        file "internal/192.168.1.rev";
        allow-update { none; };
    };
    ... // Many zones omitted

    include "internal/tmark.zones";  // atrust.net, atrust.org slaves

}; // End of internal view
```

```
view "world" {                    // External view

    match-clients { any; };
    recursion no;

    zone "atrust.com" {           // External forward zone
        type master;
        file "world/atrust.com";
        allow-update { none; };
    };
    zone "189.173.63.in-addr.arpa" {    // External reverse zone
        type master;
        file "world/63.173.189.rev";
        allow-update { none; };
    };
    include "world/tmark.zones";     // atrust.net, atrust.org masters
    zone "admin.com" {            // Master zones only in world view
        type master;
        file "world/admin.com";
        allow-update { none; };
    };
    ... // Lots of master+slave zones omitted

}; // End of external view
```

The file **atrust.key** defines the key named atkey:

```
key "atkey" {
    algorithm hmac-md5;
    secret "shared secret key goes here";
};
```

The file **tmark.zones** includes variations on the name atrust.com, both in different top-level domains (net, org, us, info, etc.) and with different spellings (applied-trust.com, etc.). The file **infrastructure.zones** contains the root hints and localhost files.

Zones are organized by view (internal or world) and type (master or slave), and the naming convention for zone data files reflects this scheme. This server is recursive for the internal view, which includes all local hosts, including many that use private addressing. The server is not recursive for the external view, which contains only selected hosts at atrust.com and the external zones for which they provide either master or slave DNS service.

Snippets of the files **internal/atrust.com** and **world/atrust.com** are shown below. First, the **internal** file:

```
; atrust.com – internal file

$TTL 86400
$ORIGIN atrust.com.
@   3600    SOA ns1.atrust.com. trent.atrust.com. (
                    2015110200 10800 1200 3600000 3600 )
    3600    NS  NS1.atrust.com.
    3600    NS  NS2.atrust.com.
    3600    MX  10 mailserver.atrust.com.
    3600    A   66.77.122.161

ns1         A   192.168.2.11
ns2         A   66.77.122.161
www         A   66.77.122.161
mailserver  A   192.168.2.11
exchange    A   192.168.2.100
secure      A   66.77.122.161
...
```

You can see from the IP address ranges that this site is using RFC1918 private addresses internally. Note also that instead of assigning nicknames to hosts through CNAMEs, this site has multiple A records that point to the same IP addresses. This approach works fine, but each IP address should have only one PTR record in the reverse zone.

A records were at one time potentially faster to resolve than CNAMEs because they relieved clients of the need to perform a second DNS query to obtain the address of a CNAME's target. These days, DNS servers are smarter and automatically include an A record for the target in the original query response (if they know it).

Here is the external view of that same domain from the file **world/atrust.com**:

```
; atrust.com – external file

$TTL 57600
$ORIGIN atrust.com.
@                   SOA ns1.atrust.com. trent.atrust.com. (
                            2015110200 10800 1200 3600000 3600 )
                    NS  NS1.atrust.com.
                    NS  NS2.atrust.com.
                    MX  10 mailserver.atrust.com.
                    A   66.77.122.161
ns1.atrust.com.     A   206.168.198.209
ns2.atrust.com.     A   66.77.122.161
www                 A   66.77.122.161
mailserver          A   206.168.198.209
secure              A   66.77.122.161

; reverse maps
exterior1           A   206.168.198.209
209.198.168.206     PTR exterior1.atrust.com.
exterior2           A   206.168.198.213
213.198.168.206     PTR exterior2.atrust.com.
...
```

As in the internal view, nicknames are implemented with A records. Only a few hosts are actually visible in the external view (although that's not immediately apparent from these truncated excerpts). Machines that appear in both views (for example, ns1) have RFC1918 private addresses internally but publicly registered and assigned addresses externally.

The TTL in these zone files is set to 16 hours (57,600 seconds). For internal zones, the TTL is one day (86,400 seconds).

16.9 ZONE FILE UPDATING

To change a domain's data (e.g., to add or delete a host), you update the zone data files on the master server. You must also increment the serial number in the zone's SOA record. Finally, you must get your name server software to pick up and distribute your changes.

This final step varies depending on your software. For BIND, just run **rndc reload** to signal **named** to pick up the changes. You can also kill and restart **named**, but if your server is both authoritative for your zone and recursive for your users, this operation discards cached data from other domains.

Updated zone data is propagated to slave servers of BIND masters right away because the notify option is on by default. If notifications are not turned on, your slave servers will not pick up the changes until after *refresh* seconds, as set in the zone's SOA record (typically an hour later).

If you have the notify option turned off, you can force BIND slaves to update themselves by running **rndc reload** on each slave. This command makes the slave check with the master, see that the data has changed, and request a zone transfer.

Don't forget to modify both the forward and reverse zones when you change a hostname or IP address. Forgetting the reverse files leaves sneaky errors: some commands work and some don't.

Changing a zone's data but forgetting to change the serial number makes the changes take effect on the master server (after a reload) but not on the slaves.

Do not edit data files on slave servers. These files are maintained by the name server, and sysadmins should not meddle with them. It's fine to look at the BIND data files as long as you don't make changes.

Zone transfers

DNS servers are synchronized through a mechanism called a zone transfer. A zone transfer can include the entire zone (called AXFR) or be limited to incremental changes (called IXFR). By default, zone transfers use the TCP protocol on port 53. BIND logs transfer-related information with category “xfer-in” or “xfer-out.”

A slave wanting to refresh its data must request a zone transfer from the master server and make a backup copy of the zone data on disk. If the data on the master has not changed, as determined by a comparison of the serial numbers (not the actual data), no update occurs and the backup files are just touched. (That is, their modification times are set to the current time.)

Both the sending and receiving servers remain available to answer queries during a zone transfer. Only after the transfer is complete does the slave begin to use the new data.

When zones are huge (like com) or dynamically updated (see the next section), changes are typically small relative to the size of the entire zone. With IXFR, only the changes are sent (unless they are larger than the complete zone, in which case a regular AXFR transfer is done). The IXFR mechanism is analogous to the **patch** program in that it makes changes to an old database to bring it into conformity with a new database.

In BIND, IXFR is the default for any zones configured for dynamic update, and **named** maintains a transaction log called *zonename***.jnl**. You can set the options provide-ixfr and request-ixfr in the server statements for individual peers. The provide-ixfr option enables or disables IXFR service for zones for which this server is the master. The request-ixfr option requests IXFRs for zones for which this server is a slave.

```
provide-ixfr yes ;    # In BIND server statement
request-ixfr yes ;    # In BIND server statement
```

IXFRs work for zones that are edited by hand, too. Use the BIND zone option called ixfr-from-differences to enable this behavior. IXFR requires the zone file to be sorted in a canonical order. An IXFR request to a server that does not support it automatically falls back to the standard AXFR zone transfer.

Dynamic updates

See this page for more information about DHCP.

DNS was originally designed under the assumption that name-to-address mappings are relatively stable and do not change frequently. However, a site that uses DHCP to dynamically assign IP addresses as machines boot and join the network breaks this rule constantly. Two basic solutions are available: either add generic (and static) entries to the DNS database, or provide some way to make small, frequent changes to zone data.

The first solution should be familiar to anyone who has looked up the PTR record for the IP address assigned to them by a mass-market (home) ISP. The DNS configuration usually looks something like this:

```
dhcp-host1.domain.  IN  A  192.168.0.1
dhcp-host2.domain.  IN  A  192.168.0.2
...
```

Although this is a simple solution, it means that hostnames are permanently associated with particular IP addresses and that computers therefore change hostnames whenever they receive new IP addresses. Hostname-based logging and security measures become very difficult in this environment.

The dynamic update feature outlined in RFC2136 offers an alternative solution. It extends the DNS protocol to include an update operation, thereby allowing entities such as DHCP daemons to notify name servers of the address assignments they make. Dynamic updates can add, delete, or modify resource records.

When dynamic updates are enabled in BIND, **named** maintains a journal of dynamic changes (*zonename*.**jnl**) that it can consult in the event of a server crash. **named** recovers the in-memory state of the zone by reading the original zone files and then replaying the changes from the journal.

You cannot hand-edit a dynamically updated zone without first stopping the dynamic update stream. **rndc freeze** *zone* or **rndc freeze** *zone class view* will do the trick. These commands sync the journal file to the master zone file on disk and then delete the journal. You can then edit the zone file by hand. Unfortunately, the original formatting of the zone file will have been destroyed by **named**'s monkeying—the file will look like those maintained by **named** for slave servers.

Dynamic update attempts are refused while a zone is frozen. To reload the zone file from disk and reenable dynamic updates, use **rndc thaw** with the same arguments you used to freeze the zone.

The **nsupdate** program supplied with BIND 9 comes with a command-line interface for making dynamic updates. It runs in batch mode, accepting commands from the keyboard or a file. A blank line or the **send** command signals the end of an update and sends the changes to the server. Two blank lines signify the end of input. The command language includes a primitive if statement to express constructs such as "if this hostname does not exist in DNS, add it." As predicates for an **nsupdate** action, you can require a name to exist or not exist, or require a resource record set to exist or not exist.

For example, here is a simple **nsupdate** script that adds a new host and also adds a nickname for an existing host if the nickname is not already in use. The angle bracket prompt is produced by **nsupdate** and is not part of the command script.

```
$ nsupdate
> update add newhost.cs.colorado.edu 86400 A 128.138.243.16
>
> prereq nxdomain gypsy.cs.colorado.edu
> update add gypsy.cs.colorado.edu CNAME evi-laptop.cs.colorado.edu
```

Dynamic updates to DNS are scary. They can potentially provide uncontrolled write access to your important system data. Don't try to use IP addresses for access control—they are too easily forged. TSIG authentication with a shared-secret key is better; it's available and is easy to configure. BIND 9 supports both:

```
$ nsupdate -k keydir:keyfile
```

or

```
$ nsupdate -y keyname:secretkey
```

Since the password goes on the command line in the **-y** form, anyone running **w** or **ps** at the right moment can see it. For this reason, the **-k** form is preferred. For more details on TSIG, see the section starting here.

Dynamic updates to a zone are enabled in **named.conf** with an allow-update or update-policy clause. allow-update grants permission to update any records in accordance with IP- or key-based authentication. update-policy is a BIND 9 extension that allows fine-grained control for updates according to the hostname or record type. It requires key-based authentication. Both forms can be used only within zone statements, and they are mutually exclusive within a particular zone.

A good default for zones with dynamic hosts is to use update-policy to allow clients to update their A or PTR records but not to change the SOA record, NS records, or KEY records.

The syntax of an update-policy rule (of which there can be several) is

```
(grant|deny) identity nametype name [types] ;
```

The *identity* is the name of the cryptographic key needed to authorize the update. The *nametype* has one of four values: name, subdomain, wildcard, or self. The self option is particularly prized because it allows hosts to update only their own records. Use it if your situation allows.

The *name* is the zone to be updated, and the *types* are the resource record types that can be updated. If no types are specified, all types except SOA, NS, RRSIG, and NSEC or NSEC3 can be updated.

Here's a complete example:

```
update-policy { grant dhcp-key subdomain dhcp.cs.colorado.edu A } ;
```

This configuration allows anyone who knows the key dhcp-key to update address records in the dhcp.cs.colorado.edu subdomain. This statement would appear in the master server's **named.conf** file within the zone statement for dhcp.cs.colorado.edu. (There would be a key statement somewhere to define dhcp-key as well.)

The snippet below from the **named.conf** file at the computer science department at the University of Colorado uses the update-policy statement to allow students in a system administration class to update their own subdomains but not to mess with the rest of the DNS environment.

```
zone "saclass.net" {
    type master;
    file "saclass/saclass.net";
    update-policy {
        grant feanor_mroe. subdomain saclass.net.;
        grant mojo_mroe. subdomain saclass.net.;
        grant dawdle_mroe. subdomain saclass.net.;
        grant pirate_mroe. subdomain saclass.net.;
        ...
    };
...
```

16.10 DNS SECURITY ISSUES

DNS started out as an inherently open system, but it has steadily grown more and more secure—or at least, securable. By default, anyone on the Internet can investigate your domain with individual queries from tools such as **dig**, **host**, **nslookup**, and **drill**. In some cases, they can dump your entire DNS database.

To address such vulnerabilities, name servers support various types of access control that key off of host and network addresses or cryptographic authentication. Table 16.5 summarizes the security features that can be configured in **named.conf**. The Link column points to more information for each topic.

Table 16.5: Security features in BIND

Feature	Context	Link	What it specifies
`acl`	Various	here	Access control lists
`allow-query`	`options`, `zone`	here	Who can query a zone or server
`allow-recursion`	`options`	here	Who can make recursive queries
`allow-transfer`	`options`, `zone`	here	Who can request zone transfers
`allow-update`	`zone`	here	Who can make dynamic updates
`blackhole`	`options`	here	Servers to completely ignore
`bogus`	`server`	here	Servers never to query
`update-policy`	`zone`	here	Who can make dynamic updates

BIND can run in a **chroot**ed environment under an unprivileged UID to minimize security risks. It can also use transaction signatures to control communication between master and slave servers and between the name servers and their control programs.

Access control lists in BIND, revisited

ACLs are named address-match lists that can appear as arguments to statements such as allow-query, allow-transfer, and blackhole. Their basic syntax was described here. ACLs can help beef up DNS security in a variety of ways.

Every site should at least have one ACL for bogus addresses and one ACL for local addresses. For example:

```
acl bogusnets {          // ACL for bogus networks
    0.0.0.0/8 ;          // Default, wild card addresses
    1.0.0.0/8 ;          // Reserved addresses
    2.0.0.0/8 ;          // Reserved addresses
    169.254.0.0/16 ;     // Link-local delegated addresses
    192.0.2.0/24 ;       // Sample addresses, like example.com
    224.0.0.0/3 ;        // Multicast address space
    10.0.0.0/8 ;         // Private address space (RFC1918)
    172.16.0.0/12 ;      // Private address space (RFC1918)
    192.168.0.0/16 ;     // Private address space (RFC1918)
} ;

acl cunets {             // ACL for University of Colorado networks
    128.138.0.0/16 ;     // Main campus network
    198.11.16/24 ;
    204.228.69/24 ;
};
```

In the global options section of your config file, you could then include

```
allow-recursion { cunets; } ;
blackhole { bogusnets; } ;
```

It's also a good idea to restrict zone transfers to legitimate slave servers. An ACL makes things nice and tidy:

```
acl ourslaves {
    128.138.242.1 ;         // anchor
    ...
} ;
acl measurements {
    198.32.4.0/24 ;         // Bill manning's measurements, v4 address
    2001:478:6:0::/48 ;     // Bill manning's measurements, v6 address
} ;
```

The actual restriction is implemented with a line such as

```
allow-transfer { ourslaves; measurements; } ;
```

Here, transfers are limited to our own slave servers and to the machines of an Internet measurement project that walks the reverse DNS tree to determine the size of the Internet and the percentage of misconfigured servers. Limiting transfers in this way makes it impossible for other sites to dump your entire database with a tool such as **dig** (see this page).

Of course, you should still protect your network at a lower level through router access control lists and standard security hygiene on each host. If those measures are not possible, you can refuse DNS packets except to a gateway machine that you monitor closely.

Open resolvers

An open resolver is a recursive, caching name server that accepts and answers queries from anyone on the Internet. Open resolvers are bad. Outsiders can consume your resources without your permission or knowledge, and if they are bad guys, they might be able to poison your resolver's cache.

Worse, open resolvers are sometimes used by miscreants to amplify distributed denial of service attacks. The attacker sends queries to your resolver with a faked source address that points back to the victim of the attack. Your resolver dutifully answers the queries and sends some nice fat packets to the victim. The victim didn't initiate the queries, but it still has to route and process the network traffic. Multiply by a bunch of open resolvers and it's real trouble for the victim.

Statistics show that between 70% and 75% of caching name servers are currently open resolvers —yikes! The site dns.measurement-factory.com/tools can help you test your site. Go there, select the "open resolver test," and type in the IP addresses of your name servers. Alternatively, you can enter a network number or WHOIS identifier to test all the associated servers.

Use access control lists in **named.conf** to limit your caching name servers to answering queries from your own users.

Running in a chrooted jail

If hackers compromise your name server, they can potentially gain access to the system under the guise of the user as whom it runs. To limit the damage that someone could do in this situation, you can run the server in a **chroot**ed environment, run it as an unprivileged user, or both.

For **named**, the command-line flag **-t** specifies the directory to **chroot** to, and the **-u** flag specifies the UID under which **named** should run. For example,

```
$ sudo named -u 53
```

initially starts **named** as root, but after **named** completes its rootly chores, it relinquishes its root privileges and runs as UID 53.

Many sites don't bother to use the **-u** and **-t** flags, but when a new vulnerability is announced, they must be faster to upgrade than the hackers are to attack.

The **chroot** jail cannot be empty since it must contain all the files the name server normally needs to run: **/dev/null**, **/dev/random**, the zone files, configuration files, keys, syslog target files, UNIX domain socket for syslog, **/var**, etc. It takes a bit of work to set all this up. The **chroot** system call is performed after libraries have been loaded, so you need not copy shared libraries into the jail.

Secure server-to-server communication with TSIG and TKEY

During the time when DNSSEC (covered in the next section) was being developed, the IETF developed a simpler mechanism called TSIG (RFC2845) to allow secure communication among servers through the use of “transaction signatures.” Access control through transaction signatures is more secure than access control by IP source addresses alone. TSIG can secure zone transfers between a master server and its slaves and can also secure dynamic updates.

The TSIG seal on a message authenticates the peer and verifies that the data has not been tampered with. Signatures are checked at the time a packet is received and are then discarded; they are not cached and do not become part of the DNS data.

TSIG uses symmetric encryption. That is, the encryption key is the same as the decryption key. This single key is called the “shared secret.” The TSIG specification allows multiple encryption methods, and BIND implements quite a few. Use a different key for each pair of servers that want to communicate securely.

TSIG is much less expensive computationally than public key cryptography, but because it requires manual configuration, it is only appropriate for a local network on which the number of pairs of communicating servers is small. It does not scale to the global Internet.

Setting up TSIG for BIND

First, use BIND's **dnssec-keygen** utility to generate a shared-secret host key for the two servers, say, master and slave1:

```
$ dnssec-keygen -a HMAC-SHA256 -b 128 -n HOST master-slave1
Kmaster-slave1.+163+15496
```

The **-b 128** flag tells **dnssec-keygen** to create a 128-bit key. We use 128 bits here just to keep the keys short enough to fit on our printed pages. In real life, you might want to use a longer key; 512 bits is the maximum allowed.

This command produces the following two files:

Kmaster-slave1.+163+15496.private
Kmaster-slave1.+163+15496.key

The **163** represents the SHA-256 algorithm, and **15496** is a number used as a key identifier in case you have multiple keys for the same pair of servers. The number looks random, but it is actually just a hash of the TSIG key. Both files include the same key, but in different formats.

The **.private** file looks like this:

```
Private-key-format: v1.3
Algorithm: 163 (HMAC_SHA256)
Key: owKt6ZWOlu0gaVFkwOqGxA==
Bits: AAA=
Created: 20160218012956
Publish: 20160218012956
Activate: 20160218012956
```

and the **.key** file like this:

```
master-slave1.  IN  KEY  512 3 163 owKt6ZWOlu0gaVFkwOqGxA==
```

Note that **dnssec-keygen** added a dot to the end of the key names in both the filenames and the contents of the **.key** file. The motivation for this convention is that when **dnssec-keygen** is used for DNSSEC keys that are added to zone files, the key names must be fully qualified domain names and must therefore end in a dot. There should probably be two tools, one that generates shared-secret keys and one that generates public-key key pairs.

You don't actually need the **.key** file—it's an artifact of **dnssec-keygen**'s being used for two different jobs. Just delete it. The 512 in the KEY record is not the key length but rather a flag bit that identifies the record as a DNS host key.

After all this complication, you may be disappointed to learn that the generated key is really just a long random number. You could generate the key manually by writing down an ASCII string of

the right length (divisible by 4) and pretending that it's a base-64 encoding of something, or you could use **mmencode** to encode a random string. The way you create the key is not important; it just has to exist on both machines.

***scp** is part of the OpenSSH suite. See this page for details.*

Copy the key from the **.private** file to both master and slave1 with **scp**, or cut and paste it. Do not use **telnet** or **ftp** to copy the key; even internal networks might not be secure.

The key must be included in both machines' **named.conf** files. Since **named.conf** is usually world-readable and keys should not be, put the key in a separate file that is included in **named.conf**. The key file should have mode 600 and should be owned by the **named** user.

For example, you could put the snippet

```
key master-slave1. {
    algorithm hmac-md5 ;
    secret "shared-key-you-generated" ;
} ;
```

in the file **master-slave1.tsig**. In the **named.conf** file, add the line

```
include "master-slave1.tsig" ;
```

near the top.

This part of the configuration simply defines the keys. For them to actually be used to sign and verify updates, the master needs to require the key for transfers and the slave needs to identify the master with a server statement and keys clause. For example, you might add the line

```
allow-transfer { key master-slave1. ;} ;
```

to the zone statement on the master server, and the line

```
server master's-IP-address { keys { master-slave1. ; } ; } ;
```

to the slave's **named.conf** file. If the master server allows dynamic updates, it can also use the key in its allow-update clause in the zone statement.

Our example key name is pretty generic. If you use TSIG keys for many zones, you might want to include the name of the zone in the key name to help you keep everything straight.

When you first turn on transaction signatures, run **named** at debug level 1 (see this page for information about debug mode) for a while to see any error messages that are generated.

When using TSIG keys and transaction signatures between master and slave servers, keep the clocks of the servers synchronized with NTP. If the clocks are too far apart (more than about 5 minutes), signature verification will not work. This problem can be very hard to identify.

TKEY is a BIND mechanism that lets two hosts generate a shared-secret key automatically, without phone calls or secure copies to distribute the key. It uses an algorithm called the Diffie-Hellman key exchange in which each side makes up a random number, does some math on it, and sends the result to the other side. Each side then mathematically combines its own number with the transmission it received to arrive at the same key. An eavesdropper might overhear the transmission but will be unable to reverse the math. (The math involved is called the discrete log problem and relies on the fact that for modular arithmetic, taking powers is easy but taking logs to undo the powers is close to impossible.)

Microsoft servers use TSIG in a nonstandard way called GSS-TSIG that exchanges the shared secret through TKEY. If you need a Microsoft server to communicate with BIND, use the `tkey-domain` and `tkey-gssapi-credential` options.

SIG(0) is another mechanism for signing transactions between servers or between dynamic updaters and the master server. It uses public key cryptography; see RFCs 2535 and 2931 for details.

DNSSEC

DNSSEC is a set of DNS extensions that authenticate the origin of zone data and verify its integrity by using public key cryptography. That is, the extensions allow DNS clients to ask the questions “Did this DNS data really come from the zone’s owner?” and “Is this really the data sent by that owner?”

DNSSEC relies on a cascading chain of trust. The root servers validate information for the top-level domains, the top-level domains validate information for the second-level domains, and so on.

Public key cryptosystems use two keys: one to encrypt (sign) and a different one to decrypt (verify). Publishers sign their data with the secret “private” key. Anyone can verify the validity of a signature with the matching “public” key, which is widely distributed. If a public key correctly decrypts a zone file, then the zone must have been encrypted with the corresponding private key. The trick is to make sure that the public keys you use for verification are authentic. Public key systems allow one entity to sign the public key of another, thereby vouching for the legitimacy of the key; hence the term “chain of trust.”

The data in a DNS zone is too voluminous to be encrypted with public key cryptography—the encryption would be too slow. Instead, since the data is not secret, a secure hash is run on the data and the results of the hash are signed (encrypted) by the zone’s private key. The results of the hash are like a fingerprint of the data and are called a digital signature. The signatures are appended to the data they authenticate as RRSIG records in the signed zone file.

To verify the signature, you decrypt it with the public key of the signer, run the data through the same secure hash algorithm, and compare the computed hash value with the decrypted hash value. If they match, you have authenticated the signer and verified the integrity of the data.

In the DNSSEC system, each zone has its own public and private keys. In fact, it has two sets of keys: a zone-signing key pair and a key-signing key pair. The private zone-signing key signs each RRset (that is, each set of records of the same type for the same host). The public zone-signing key verifies the signatures and is included in the zone’s data in the form of a DNSKEY resource record.

Parent zones contain DS records that are hashes of the child zones’ self-signed key-signing-key DNSKEY records. A name server verifies the authenticity of a child zone’s DNSKEY record by checking it against the parent zone’s signature. To verify the authenticity of the parent zone’s key, the name server can check the parent’s parent, and so on back to the root. The public key for the root zone is widely published and is included in the root hints file.

The DNSSEC specifications require that if a zone has multiple keys, each is tried until the data is validated. This behavior is required so that keys can be rolled over (changed) without interruptions in DNS service. If a DNSSEC-aware recursive name server queries an unsigned zone, the unsigned answer that comes back is accepted as valid. But problems occur when

signatures expire or when parent and child zones do not agree on the child’s current DNSKEY record.

DNSSEC policy

Before you begin deployment of DNSSEC, you should nail down (or at least think about) a few policies and procedures. For example:

- What size keys will you use? Longer keys are more secure, but they make for larger packets.
- How often will you change keys in the absence of a security incident?

We suggest that you keep a key log that records the date you generated each key, the hardware and operating system used, the key tag assigned, the version of the key generator software, the algorithm used, the key length, and the signature validity period. If a cryptographic algorithm is later compromised, you can check your log to see if you are vulnerable.

DNSSEC resource records

DNSSEC uses five resource record types that were referred to in the DNS database section back on this page but were not described in detail: DS, DNSKEY, RRSIG, NSEC, and NSEC3. We describe them here in general and then outline the steps involved in signing a zone. Each of these records is created by DNSSEC tools rather than by being typed into a zone file with a text editor.

The DS (Designated Signer) record appears only in the parent zone and indicates that a subzone is secure (signed). It also identifies the key used by the child to self-sign its own KEY resource record set. The DS record includes a key identifier (a five-digit number), a cryptographic algorithm, a digest type, and a digest of the public key record allowed (or used) to sign the child's key resource record. Here's an example.

```
example.com.    IN  DS  682 5 1 12898DCF9F7C2E89A1AD20DBCE159E7...
```

(In this section, base-64-encoded hashes and keys have all been truncated to save space and better illustrate the structure of the records.)

The question of how to change existing keys in the parent and child zones has been a thorny one that seemed destined to require cooperation and communication between parent and child. The creation of the DS record, the use of separate key-signing and zone-signing keys, and the use of multiple key pairs have helped address this problem.

Keys included in a DNSKEY resource record can be either key-signing keys (KSKs) or zone-signing keys (ZSKs). A flag called SEP for "secure entry point" distinguishes them. Bit 15 of the flags field is set to 1 for KSKs and to 0 for ZSKs. This convention makes the flags field of KSKs odd and of ZSKs even when they are treated as decimal numbers. The values are currently 257 and 256, respectively.

Multiple keys can be generated and signed so that a smooth transition from one key to the next is possible. The child can change its zone-signing keys without notifying the parent; it must only coordinate with the parent if it changes its key-signing key. As keys roll over, both the old key and the new key are valid for a certain interval. Once cached values on the Internet have expired, the old key can be retired.

An RRSIG record is the signature of a resource record set (that is, the set of all records of the same type and name within a zone). RRSIG records are generated by zone-signing software and added to the signed version of the zone file.

An RRSIG record contains a wealth of information:

- The type of record set being signed
- The signature algorithm used, encoded as a small integer
- The number of labels (dot-separated pieces) in the name field

- The TTL of the record set that was signed
- The time the signature expires (as *yyyymmddhhssss*)
- The time the record set was signed (also *yyyymmddhhssss*)
- A key identifier (a 5-digit number)
- The signer's name (domain name)
- And finally, the digital signature itself (base-64-encoded)

Here's an example:

```
RRSIG  NS 5 2 57600 20090919182841 (
           20090820182841 23301 example.com.
           pMKZ76waPVTbIguEQNUojNVlVewHau4p...== )
```

NSEC or NSEC3 records are also produced as a zone is signed. Rather than signing record sets, they certify the intervals *between* record set names and so allow for a signed answer of "no such domain" or "no such resource record set." For example, a server might respond to a query for A records named bork.atrust.com with an NSEC record that certifies the nonexistence of any A records between bark.atrust.com and bundt.atrust.com.

Unfortunately, the inclusion of the endpoint names in NSEC records allows someone to walk through the zone and obtain all of its valid hostnames. NSEC3 fixes this feature by including hashes of the endpoint names rather than the endpoint names themselves, but it is more expensive to compute: more security, less performance. NSEC and NSEC3 are both in current use, and you can choose between them when you generate your keys and sign your zones.

Unless protecting against a zone walk is critically important for your site, we recommend that you use NSEC for now.

Turning on DNSSEC

Two separate workflows are involved in deploying signed zones: a first that creates keys and signs zones, and a second that serves the contents of those signed zones. These duties need not be implemented on the same machine. In fact, it is better to quarantine the private key and the CPU-intensive signing process on a machine that is not publicly accessible from the Internet. (Of course, the machine that actually serves the data must be visible to the Internet.)

The first step in setting up DNSSEC is to organize your zone files so that all the data files for a zone are in a single directory. The tools that manage DNSSEC zones expect this organization.

Next, enable DNSSEC on your servers with the **named.conf** options

```
options {
    dsnsec-enable yes;
}
```

for authoritative servers, and

```
options {
    dsnsec-enable yes;
    dnssec-validation yes;
}
```

for recursive servers. The dnssec-enable option tells your authoritative servers to include DNSSEC record set signatures in their responses when answering queries from DNSSEC-aware name servers. The dnssec-validation option makes **named** verify the legitimacy of signatures it receives in responses from other servers.

Key pair generation

You must generate two key pairs for each zone you want to sign: a zone-signing (ZSK) pair and a key-signing (KSK) pair. Each pair consists of a public key and a private key. The KSK's private key signs the ZSK and creates a secure entry point for the zone. The ZSK's private key signs the zone's resource records. The public keys are then published to allow other sites to verify your signatures.

The commands

```
$ dnssec-keygen -a RSASHA256 -b 1024 -n ZONE example.com
Kexample.com.+008+29718
$ dnssec-keygen -a RSASHA256 -b 2048 -n ZONE -f KSK example.com
Kexample.com.+008+05005
```

generate for example.com a 1,024-bit ZSK pair that uses the RSA and SHA-256 algorithms and a corresponding 2,048-bit KSK pair. The outstanding issue of UDP packet size limits suggests that it's best to use short zone-signing keys and to change them often. You can use longer key-signing keys to help recover some security.

It can take a while—minutes—to generate these keys. The limiting factor is typically not CPU power but the entropy available for randomization. On Linux, you can install the **haveged** daemon to harvest entropy from additional sources and thereby speed up key generation.

dnssec-keygen prints to standard out the base filename of the keys it has generated. In this example, **example.com** is the name of the key, **008** is the identifier of the RSA/SHA-256 algorithm suite, and **29718** and **05005** are hashes called the key identifiers, key footprints, or key tags. As when generating TSIG keys, each run of **dnssec-keygen** creates two files (**.key** and **.private**):

```
Kexample.com.+008+29718.key     # Public zone-signing key
Kexample.com.+008+29718.private # Private zone-signing key
```

Several encryption algorithms are available, each with a range of possible key lengths. You can run **dnssec-keygen** with no arguments to see the current list of supported algorithms. BIND can also use keys generated by other software.

Depending on the version of your software, some of the available algorithm names might have NSEC3 appended or prepended to them. If you want to use NSEC3 records instead of NSEC records for signed negative answers, you must generate NSEC3-compatible keys with one of the NSEC3-specific algorithms; see the man page for **dnssec-keygen**.

The **.key** files each contain a single DNSKEY resource record for example.com. For example, here is the zone-signing public key, truncated to fit the page. You can tell it's a ZSK because the flags field is 256, rather than 257 for a KSK.

```
example.com.    IN  DNSKEY 256 3 8 AwEAAcyLrgENt8OJ4PIQiv2ZhWwSviA...
```

These public keys must be $INCLUDEd or inserted into the zone file, either at the end or right after the SOA record. To copy the keys into the zone file, you can append them with **cat** or paste them in with a text editor. Use a command like **cat Kexample.com.+*.key >> zonefile**. The >> appends to the **zonefile** rather than replacing it entirely, as > would. (Don't mess this one up!)

Ideally, the private key portion of any key pair would be kept off-line, or at least on a machine that is not on the public Internet. This precaution is impossible for dynamically updated zones and impractical for zone-signing keys, but it is perfectly reasonable for key-signing keys, which are presumably quite long-lived. Consider a hidden master server that is not accessible from outside for the ZSKs. Print out the private KSK or write it to a USB memory stick and then lock it in a safe until you need it again.

While you're locking away your new private keys, it's also a good time to enter the new keys into your key log file. You don't need to include the keys themselves, just the IDs, algorithms, date, purpose, and so on.

The default signature validity periods are one month for RRSIG records (ZSK signatures of resource record sets) and three months for DNSKEY records (KSK signatures of ZSKs). Current best practice suggests ZSKs of length 1,024 that are used for three months to a year and KSKs of length 1,280 that are used for a year or two.The web site keylength.com tabulates a variety of organizations' recommendations regarding the suggested lengths of cryptographic keys.

Since the recommended key retention periods are longer than the default signature validity periods, you must either specify a longer validity period when signing zones or periodically re-sign the zones, even if the key has not changed.

Zone signing

Now that you've got keys, you can sign your zones with the **dnssec-signzone** command, which adds RRSIG and NSEC or NSEC3 records for each resource record set. These commands read your original zone file and produce a separate, signed copy named *zonefile*.**signed**.

The syntax is

```
dnssec-signzone [-o zone] [-N increment] [-k KSKfile] zonefile [ZSKfile]
```

where *zone* defaults to *zonefile* and the key files default to the filenames produced by **dnssec-keygen** as outlined above.

If you name your zone data files after the zones and maintain the names of the original key files, the command reduces to

```
dnssec-signzone [-N increment ] zonefile
```

The **-N increment** flag automatically increments the serial number in the SOA record so that you can't forget. You can also specify the value **unixtime** to update the serial number to the current UNIX time (seconds since January 1, 1970) or the value **keep** to prevent **dnssec-signzone** from modifying the original serial number. The serial number is incremented in the signed zone file but not in the original zone file.

Here's a spelled-out example that uses the keys generated above:

```
$ sudo dnssec-signzone -o example.com -N increment
    -k Kexample.com.+008+05005 example.com Kexample.com.+008+29718
```

The signed file is sorted in alphabetical order and includes the DNSKEY records we added by hand and the RRSIG and NSEC records generated during signing. The zone's serial number has been incremented.

If you generated your keys with an NSEC3-compatible algorithm, you would sign the zone as above but with a **-3** *salt* flag. Table 16.6 shows some other useful options.

Table 16.6: Useful options for dnssec-signzone

Option	Function
-g	Generates DS record(s) to be included in the parent zone
-s *start-time*	Sets the time at which the signatures become valid
-e *end-time*	Sets the time at which the signatures expire
-t	Prints statistics

The dates and times for signature validity can be expressed as absolute times in the format *yyyymmddhhmmss* or as times relative to now in the format *+N*, where *N* is in seconds. The default signature validity period is from an hour in the past to 30 days in the future. Here is an example in which we specify that signatures should be valid until the end of the calendar year 2017:

```
$ sudo dnssec-signzone -N increment -e 20171231235959 example.com
```

Signed zone files are typically four to ten times larger than the original zone, and all your nice logical ordering is lost. A line such as

```
mail-relay          A   63.173.189.2
```

becomes several lines:

```
mail-relay.example.com.  57600  A  63.173.189.2
    57600   RRSIG   A 8 3 57600 20090722234636 (
                      20150622234636 23301 example.com.
                      Y7s9jDWYuuXvozeU7zGRdFCl+rzU8cLiwoev
                      0I2TGfLlbhsRgJfkpEYFVRUB7kKVRNguEYwk
                      d2RSkDJ9QzRQ+w== )
    3600    NSEC    mail-relay2.example.com. A RRSIG NSEC
    3600    RRSIG   NSEC 8 3 3600 20090722234636 (
                      20150622234636 23301 example.com.
                      42QrXP8vpoChsGPseProBMZ7twf7eS5WK+4O
                      WNsN84hF0notymRxZRIZypqWzLIPBZAUJ77R
                      HP0hLfBDoqmZYw== )
```

In practical terms, a signed zone file is no longer human-readable, and it cannot be edited by hand because of the RRSIG and NSEC or NSEC3 records. No user-serviceable parts inside!

With the exception of DNSKEY records, each resource record set (resource records of the same type for the same name) gets one signature from the ZSK. DNSKEY resource records are signed by both the ZSK and the KSK, so they have two RRSIGs. The base-64 representation of a signature ends in however many equal signs are needed to make the length a multiple of 4.

Once your zones are signed, all that remains is to point your name server at the signed versions of the zone files. If you're using BIND, look for the zone statement that corresponds to each zone in **named.conf** and change the file parameter from **example.com** to **example.com.signed**.

Finally, restart the name server daemon, telling it to reread its configuration file with **sudo rndc reconfig** followed by **sudo rndc flush**.

You are now serving a DNSSEC signed zone! To make changes, you can edit either the original unsigned zone or the signed zone and then re-sign the zone. Editing a signed zone is something of a logistical nightmare, but it is much quicker than re-signing the entire zone. Be sure to

remove the RRSIG records that correspond to any records that you change. You probably want to make identical changes to the unsigned zone to avoid version skew.

If you pass a signed zone as the argument to **dnssec-signzone**, any unsigned records are signed and the signatures of any records that are close to expiring are renewed. “Close to expiring” is defined as being three-quarters of the way through the validity period. Re-signing typically results in changes, so make sure you increment the zone’s serial number by hand or use **dnssec-signzone -N increment** to automatically increment the zone’s serial number.

That’s all there is to the local part of DNSSEC configuration. What’s left is the thorny problem of getting your island of secure DNS connected to other trusted, signed parts of the DNS archipelago.

The DNSSEC chain of trust

Continuing with our example DNSSEC setup, example.com is now signed and its name servers have DNSSEC enabled. This means that when querying they use EDNS0, the extended DNS protocol, and set the DNSSEC-aware option in the DNS header of the packet. When answering a query that arrives with that bit set, they include the signature data with their answer.

A client that receives signed answers can validate the response by checking the record's signatures with the appropriate public key. But it gets this key from the zone's own DNSKEY record, which is rather suspicious if you think about it. What's to stop an impostor from serving up both fake records and a fake key that validates them?

The canonical solution is that you give your parent zone a DS record to include in its zone file. By virtue of coming from the parent zone, the DS record is certified by the parent's private key. If the client trusts your parent zone, it should then trust that the parent zone's DS record accurately reflects your zone's public key.

The parent zone is in turn certified by its parent, and so on back to the root.

DNSSEC key rollover

Key rollover has always been a troublesome issue in DNSSEC. In fact, the original specifications were changed specifically to address the issue of the communication needed between parent and child zones whenever keys were created, changed, or deleted. The new specifications are called DNSSEC-bis.

ZSK rollover is relatively straightforward and does not involve your parent zone or any trust anchor issues. The only tricky part is the timing. Keys have an expiration time, so rollover must occur well before that time. However, keys also have a TTL, defined in the zone file. To illustrate, assume that the TTL is one day and that keys don't expire for another week. The following steps are then involved:

1. Generate a new ZSK.
2. Include it in the zone file.
3. Sign or re-sign the zone with the KSK and the *old* ZSK.
4. Signal the name server to reload the zone; the new key is now there.
5. Wait 24 hours (the TTL); now everyone has both the old and new keys.
6. Sign the zone again with the KSK and the *new* ZSK.
7. Signal the name server to reload the zone.
8. Wait another 24 hours; now everyone has the new signed zone.
9. Remove the old ZSK at your leisure, e.g., the next time the zone changes.

This scheme is called prepublishing. Obviously, you must start the process at least two TTLs before the point at which you need to have everyone using the new key. The waiting periods guarantee that any site with cached values always has a cached key that corresponds to the cached data.

Another variable that affects this process is the time it takes for your slowest slave server to update its copy of your zone when notified by the master server. So don't wait until the last minute to start your rollover process or to re-sign zones whose signatures are expiring. Expired signatures do not validate, so sites that verify DNSSEC signatures will not be able to do DNS lookups for your domain.

The mechanism to roll over a KSK is called double signing and it's also pretty straightforward. However, you will need to communicate your new DS record to your parent. Make sure you have positive acknowledgement from the parent before you switch to just the new key. Here are the steps:

1. Create a new KSK.

2. Include it in the zone file.
3. Sign the zone with both old and new KSKs and the ZSK.
4. Signal the name server to reload the zone.
5. Wait 24 hours (the TTL); now everyone has the new key.
6. After confirmation, delete the old KSK record from the zone.
7. Re-sign the zone with the new KSK and ZSK.

DNSSEC tools

With the advent of BIND 9.10 comes a new debugging tool. The Domain Entity Lookup and Validation engine (DELV) looks much like **dig** but has a better understanding of DNSSEC. In fact, **delv** checks the DNSSEC validation chain with the same code that is used by the BIND 9 **named** itself.

In addition to the DNSSEC tools that come with BIND, four other deployment and testing toolsets might be helpful: **ldns**, DNSSEC-Tools (formerly Sparta), RIPE, and OpenDNSSEC (opendnssec.org).

***ldns** tools, nlnetlabs.nl/projects/ldns*

ldns, from the folks at NLnet Labs, is a library of routines for writing DNS tools and a set of example programs that use this library. We list the tools and what each one does below. The tools are all in the **examples** directory except for **drill**, which has its own directory in the distribution. Man pages can be found with the commands. The top-level **README** file gives very brief installation instructions.

- **ldns-chaos** shows the name server ID info stored in the CHAOS class.
- **ldns-compare-zones** shows the differences between two zone files.
- **ldns-dpa** analyzes DNS packets in **tcpdump** trace files.
- **ldns-key2ds** converts a DNSKEY record to a DS record.
- **ldns-keyfetcher** fetches DNSSEC public keys for zones.
- **ldns-keygen** generates TSIG keys and DNSSEC key pairs.
- **ldns-notify** makes a zone's slave servers check for updates.
- **ldns-nsec3-hash** prints the NSEC3 hash for a name.
- **ldns-read-zone** reads a zone and prints it in various formats.
- **ldns-revoke** sets the revoke flag on a DNSKEY key RR (RFC5011).
- **ldns-rrsig** prints human-readable expiration dates from RRSIGs.
- **ldns-signzone** signs a zone file with either NSEC or NSEC3.
- **ldns-update** sends a dynamic update packet.
- **ldns-verify-zone** makes sure RRSIG, NSEC, and NSEC3 records are OK.
- **ldns-walk** walks through a zone by following the DNSSEC NSEC records.

- **ldns-zcat** reassembles zone files split with **ldns-zsplit**.
- **ldns-zsplit** splits a zone into chunks so it can be signed in parallel.

Many of these tools are simple and do only one tiny DNS chore. They were written as example uses of the **ldns** library and demonstrate how simple the code becomes when the library does all the hard bits for you.

dnssec-tools.org

DNSSec-tools builds on the BIND tools and includes the following commands:

- **dnspktflow** traces the flow of DNS packets during a query/response sequence captured by **tcpdump** and produces a cool diagram.
- **donuts** analyzes zone files and finds errors and inconsistencies.
- **donutsd** runs **donuts** at intervals and warns of problems.
- **mapper** maps zone files, showing secure and insecure portions.
- **rollerd**, **rollctl**, and **rollinit** automate key rollovers by using the prepublishing scheme for ZSKs and the double signature method for KSKs. See this page for the details of these schemes.
- **trustman** manages trust anchors and includes an implementation of RFC5011 key rollover.
- **validate** validates signatures from the command-line.
- **zonesigner** generates keys and signs zones.

The web site contains good documentation and tutorials for all of these tools. The source code is available for download and is covered by the BSD license.

RIPE tools, ripe.net

RIPE's tools act as a front end to BIND's DNSSEC tools and focus on key management. They have friendlier messages since they run and package up the many arguments and commands into more intuitive forms.

OpenDNSSEC, opendnssec.org

OpenDNSSEC is a set of tools that takes unsigned zones, adds the signatures and other records for DNSSEC, and passes it on to the authoritative name servers for that zone. This automation greatly simplifies the initial setup of DNSSEC.

Debugging DNSSEC

DNSSEC interoperates with both signed and unsigned zones, and with both DNSSEC-aware and DNSSEC-oblivious name servers. Ergo, incremental deployment is possible, and it usually just works. But not always.

DNSSEC is a distributed system with lots of moving parts. Servers, resolvers, and the paths among them can all experience problems. A problem seen locally may originate far away, so tools like SecSpider and Vantages that monitor the distributed state of the system can be helpful. Those tools, the utilities mentioned in the previous section, and your name server log files are your primary debugging weapons.

Make sure that you route the DNSSEC logging category in **named.conf** to a file on the local machine. It's helpful to separate out the DNSSEC-related messages so that you don't route any other logging categories to this file. Here is an example logging specification for **named**:

```
channel dnssec-log {
    file "/var/log/named/dnssec.log" versions 4 size 10m ;
    print-time yes ;
    print-category yes ;
    print-severity yes ;
    severity debug 3 ;
} ;
category dnssec { dnssec-log; } ;
```

In BIND, set the debugging level to 3 or higher to see the validation steps taken by a recursive BIND server trying to validate a signature. This logging level produces about two pages of logging output per signature verified. If you are monitoring a busy server, log data from multiple queries will likely be interleaved. Sorting through the mess can be challenging and tedious.

drill has two particularly useful flags: **-T** to trace the chain of trust from the root to a specified host, and **-S** to chase the signatures from a specified host back to the root. Here's some made-up sample output from **drill -S** snitched from the *DNSSEC HOWTO* at NLnet Labs:

```
$ drill -S -k ksk.keyfile example.net SOA
DNSSEC Trust tree:
example.net. (SOA)
|---example.net. (DNSKEY keytag: 17000)
    |---example.net. (DNSKEY keytag: 49656)
    |---example.net. (DS keytag: 49656)
        |---net. (DNSKEY keytag: 62972)
            |---net. (DNSKEY keytag: 13467)
            |---net. (DS keytag: 13467)
                |---. (DNSKEY keytag: 63380)
                    |---. (DNSKEY keytag: 63276)  ;; Chase successful
```

If a validating name server cannot verify a signature, it returns a SERVFAIL indication. The underlying problem could be a configuration error by someone at one of the zones in the chain of trust, bogus data from an interloper, or a problem in the setup of the validating recursive server itself. Try **drill** to chase the signatures along the chain of trust and see where the problem lies.

If all the signatures are verified, try querying the troublesome site with **dig** and then with **dig +cd**. (The **cd** flag turns off validation.) Try this at each of the zones in the chain of trust to see if you can find the problem. You can work your way up or down the chain of trust. The likely result will be an expired trust anchor or expired signatures.

16.11 BIND DEBUGGING

BIND provides three basic debugging tools: logging, described below; a control program, described starting here; and a command-line query tool, described here.

Logging in BIND

See Chapter 10 for more information about syslog.

named's logging facilities are flexible enough to make your hair stand on end. BIND originally just used syslog to report error messages and anomalies. Recent versions generalize the syslog concepts by adding another layer of indirection and support for logging directly to files. Before you dive in, check the mini-glossary of BIND logging terms shown in Table 16.7.

Table 16.7: A BIND logging lexicon

Term	What it means
category	A class of messages that **named** can generate; for example, messages about dynamic updates or messages about answering queries
module	The name of the source module that generates a message
severity	The "badness" of an error message; what syslog refers to as a priority
channel	A place where messages can go: syslog, a file, or **/dev/null**[a]
facility	A syslog facility name. DNS does not have its own specific facility, but you have your pick of all the standard ones

a. **/dev/null** is a pseudo-device that throws away all input.

You configure BIND logging with a logging statement in **named.conf**. You first define channels, the possible destinations for messages. You then direct various categories of message to go to particular channels.

When a message is generated, it is assigned a category, a module, and a severity at its point of origin. It is then distributed to all the channels associated with its category and module. Each channel has a severity filter that tells what severity level a message must have to get through. Channels that lead to syslog stamp messages with the designated facility name. Messages that go to syslog are also filtered according to the rules in **/etc/syslog.conf**. Here's the outline of a logging statement:

```
logging {
    channel-def;
    channel-def;
    ...
    category category-name {
        channel-name;
        channel-name;
        ...
    };
};
```

Channels

A *channel-def* looks slightly different according to whether the channel is a file channel or a syslog channel. You must choose file or syslog for each channel; a channel can't be both at the same time.

```
channel channel-name {
    file path [ versions numvers | unlimited ] [ size sizespec ];
    syslog facility;
    severity severity;
    print-category yes | no;
    print-severity yes | no;
    print-time yes | no;
};
```

For a file channel, *numvers* tells how many backup versions of a file to keep, and *sizespec* specifies how large the file should be allowed to grow (examples: 2048, 100k, 20m, unlimited, default) before it is automatically rotated. If you name a file channel **mylog**, the rotated versions are **mylog.0**, **mylog.1**, and so on.

See this page for a list of syslog facility names.

In the syslog case, *facility* names the syslog facility under which to log the message. It can be any standard facility. In practice, only daemon and local0 through local7 are reasonable choices.

The rest of the statements in a *channel-def* are optional. *severity* can have the values (in descending order) critical, error, warning, notice, info, or debug (with an optional numeric level, e.g., severity debug 3). The value dynamic is also recognized and matches the server's current debug level.

The various print options add or suppress message prefixes. Syslog prepends the time and reporting host to each message logged, but not the severity or the category. The source filename (module) that generated the message is also available as a print option. It makes sense to enable print-time only for file channels—syslog adds its own time stamps, so there's no need to duplicate them.

The four channels listed in Table 16.8 are predefined by default. These defaults should be fine for most installations.

Table 16.8: Predefined logging channels in BIND

Channel name	What it does
`default_syslog`	Sends to syslog with facility daemon, severity info
`default_debug`	Logs to the file **named.run** with severity set to `dynamic`
`default_stderr`	Sends to standard error of the **named** process with severity info
`null`	Discards all messages

Categories

Categories are determined by the programmer at the time the code is written. They organize log messages by topic or functionality instead of just by severity. Table 16.9 shows the current list of message categories.

Table 16.9: BIND logging categories

Category	What it includes
`client`	Client requests
`config`	Configuration file parsing and processing
`database`	Messages about database operations
`default`	Default for categories without specific logging options
`delegation-only`	Queries forced to NXDOMAIN by delegation-only zones
`dispatch`	Dispatching of incoming packets to server modules
`dnssec`	DNSSEC messages
`edns-disabled`	Info about broken servers
`general`	Catchall for unclassified messages
`lame-servers`	Servers that are supposed to be serving a zone, but aren't[a]
`network`	Network operations
`notify`	Messages about the "zone changed" notification protocol
`queries`	A short log message for every query the server receives (!)
`resolver`	DNS resolution, e.g., recursive lookups for clients
`security`	Approved/unapproved requests
`unmatched`	Queries **named** cannot classify (bad class, no view)
`update`	Messages about dynamic updates
`update-security`	Approval or denial of update requests
`xfer-in`	Zone transfers that the server is receiving
`xfer-out`	Zone transfers that the server is sending

a. Either the parent zone or the child zone could be at fault; hard to tell without investigating.

Log messages

The default logging configuration is

```
logging {
    category default { default_syslog; default_debug; };
};
```

You should watch the log files when you make major changes to BIND and perhaps increase the logging level. Later, reconfigure to preserve only serious messages once you have verified that **named** is stable.

Query logging can be quite educational. You can verify that your `allow` clauses are working, see who is querying you, identify broken clients, etc. It's a good check to perform after major reconfigurations, especially if you have a good sense of what your query load looked like before the changes.

To start query logging, just direct the `queries` category to a channel. Writing to syslog is less efficient than writing directly to a file, so use a file channel on a local disk when you are logging every query. Have lots of disk space and be ready to turn query logging off once you obtain enough data. (**rndc querylog** dynamically toggles query logging on and off.)

Views can be pesky to debug, but fortunately, the view that matched a particular query is logged along with the query.

Some common log messages are listed below:

- *Lame server resolving xxx.* If you get this message about one of your own zones, you have configured something incorrectly. The message is harmless if it's about some zone out on the Internet; it's someone else's problem. A good one to throw away by directing it to the `null` channel.
- *…query (cache) xxx denied*. This can be either misconfiguration of the remote site, abuse, or a case in which someone has delegated a zone to you, but you have not configured it.
- *Too many timeouts resolving xxx: disabling EDNS*. This message can result from a broken firewall not admitting UDP packets over 512 bytes long or not admitting fragments. It can also be a sign of problems at the specified host. Verify that the problem is not your firewall and consider redirecting these messages to the null channel.
- *Unexpected RCODE (SERVFAIL) resolving xxx*. This can be an attack or, more likely, a sign of something repeatedly querying a lame zone.
- *Bad referral*. This message indicates a miscommunication among a zone's name servers.
- *Not authoritative for*. A slave server is unable to get authoritative data for a zone. Perhaps it's pointing to the wrong master, or perhaps the master had trouble loading

the zone in question.

- *Rejected zone*. **named** rejected a zone file because it contained errors.
- *No NS RRs found*. A zone file did not include NS records after the SOA record. It could be that the records are missing, or it could be that they don't start with a tab or other whitespace. In the latter case, the records are not attached to the zone of the SOA record and are therefore misinterpreted.
- *No default TTL set*. The preferred way to set the default TTL for resource records is with a $TTL directive at the top of the zone file. This error message indicates that the $TTL is missing; it is required in BIND 9.
- *No root name server for class*. Your server is having trouble finding the root name servers. Check your hints file and the server's Internet connectivity.
- *Address already in use*. The port on which **named** wants to run is already being used by another process, probably another copy of **named**. If you don't see another **named** around, it might have crashed and left an **rndc** control socket open that you'll have to track down and remove. A good way to fix the problem is to stop the **named** process with **rndc** and then restart **named**:

```
$ sudo rndc stop
$ sudo /usr/sbin/named ...
```

- *…updating zone xxx: update unsuccessful*. A dynamic update for a zone was attempted but refused, most likely because of the allow-update or update-policy clause in **named.conf** for this zone. This is a common error message and often is caused by misconfigured Windows boxes.

Sample BIND logging configuration

The following snippet from the ISC **named.conf** file for a busy TLD name server illustrates a comprehensive logging regimen.

```
logging {
    channel default-log { # Default channel, to a file
        file "log/named.log" versions 3 size 10m;
        print-time yes;
        print-category yes;
        print-severity  yes;
        severity info;
    };
    channel xfer-log { # Zone transfers channel, to a file
        file "log/xfer.log" versions 3 size 10m;
        print-category yes;
        print-severity yes;
        print-time yes;
        severity info;
    };
    channel dnssec-log { # DNSSEC channel, to a file
        file "log/dnssec.log" versions 3 size 1M;
        severity debug 1;
        print-severity yes;
        print-time yes;
    };
    category default { default-log; default_debug; };
    category dnssec { dnssec-log; };
    category xfer-in { xfer-log; };
    category xfer-out { xfer-log; };
    category notify { xfer-log; };
};
```

Debug levels in BIND

named debug levels are denoted by integers from 0 to 100. The higher the number, the more verbose the output. Level 0 turns debugging off. Levels 1 and 2 are fine for debugging your configuration and database. Levels beyond about 4 are appropriate for the maintainers of the code.

You invoke debugging on the **named** command line with the **-d** flag. For example,

```
$ sudo named -d2
```

would start **named** at debug level 2. By default, debugging information is written to the file **named.run** in the current working directory from which **named** is started. The **named.run** file grows fast, so don’t go out for a beer while debugging or you might have bigger problems when you return.

You can also turn on debugging while **named** is running with **rndc trace**, which increments the debug level by 1, or with **rndc trace** *level*, which sets the debug level to the value specified. **rndc notrace** turns debugging off completely. You can also enable debugging by defining a logging channel that includes a severity specification such as

```
severity debug 3;
```

which sends all debugging messages up to level 3 to that particular channel. Other lines in the channel definition specify the destination of those debugging messages. The higher the severity level, the more information is logged.

Watching the logs or the debugging output illustrates how often DNS is misconfigured in the real world. That pesky little dot at the end of names (or rather, the lack thereof) accounts for an alarming amount of DNS traffic.

Name server control with rndc

Table 16.10 shows some of the options accepted by **rndc**. Typing **rndc** with no arguments lists the available commands and briefly describes what they do. Earlier incantations of **rndc** used signals, but with over 25 commands, the BIND folks ran out of signals long ago. Commands that produce files put them in whatever directory is specified as **named**'s home in **named.conf**.

Table 16.10: rndc commands

Command	Function
dumpdb	Dumps the DNS database to **named_dump.db**
flush [*view*]	Flushes all caches or those for a specified *view*
flushname *name* [*view*]	Flushes the specified *name* from the server's cache
freeze *zone* [*class* [*view*]] [a]	Suspends updates to a dynamic *zone*
thaw *zone* [*class* [*view*]] [a]	Resumes updates to a dynamic *zone*
halt	Halts **named** without writing pending updates
querylog	Toggles tracing of incoming queries
notify *zone* [*class* [*view*]] [a]	Resends notification messages for *zone*
notrace	Turns off debugging
reconfig	Reloads the config file and loads any new zones
recursing	Dumps queries currently recursing, **named.recursing**
refresh *zone* [*class* [*view*]] [a]	Schedules maintenance for a *zone*
reload	Reloads **named.conf** and zone files
reload *zone* [*class* [*view*]] [a]	Reloads only the specified *zone* or *view*
retransfer *zone* [*class* [*view*]] [a]	Recopies the data for *zone* from the master server
stats	Dumps statistics to **named.stats**
status	Displays the current status of the running **named**
stop	Saves pending updates and then stops **named**
trace	Increments the debug level by 1
trace *level*	Changes the debug level to the value *level*
validation *newstate*	Enables/disables DNSSEC validation on the fly

a. The *class* argument here is the same as for resource records, typically IN for Internet.

rndc reload makes **named** reread its configuration file and reload zone files. The **reload** *zone* command is handy when only one zone has changed and you don't want to reload all the zones, especially on a busy server. You can also specify a *class* and *view* to reload only the selected view of the zone's data.

Note that **rndc reload** is not sufficient to add a completely new zone; that requires **named** to read both the **named.conf** file and the new zone file. For new zones, use **rndc reconfig**, which rereads the config file and loads any new zones without disturbing existing zones.

rndc freeze *zone* stops dynamic updates and reconciles the journal of dynamic updates to the data files. After freezing the zone, you can edit the zone data by hand. As long as the zone is frozen, dynamic updates are refused. Once you've finished editing, use **rndc thaw** *zone* to start accepting dynamic updates again.

rndc dumpdb instructs **named** to dump its database to **named_dump.db**. The dump file is big and includes not only local data but also any cached data the name server has accumulated.

Your versions of **named** and **rndc** must match or you will see an error message about a protocol version mismatch. They're normally installed together on individual machines, but version skew can be an issue when you are trying to control a **named** on another computer.

Command-line querying for lame delegations

When you apply for a domain name, you are asking for a part of the DNS naming tree to be delegated to your name servers and your DNS administrator. If you never use the domain or you change the name servers or their IP addresses without coordinating with your parent zone, a "lame delegation" results.

The effects of a lame delegation can be really bad. If one of your servers is lame, your DNS system is less efficient. If all the name servers for a domain are lame, no one can reach you. All queries start at the root unless answers are cached, so lame servers and lazy software that doesn't do negative caching of SERVFAIL errors increase the load of everyone on the path from the root to the lame domain.

The **doc** ("domain obscenity control") command can help you identify lame delegations, but you can also find them just by reviewing your log files. Here's an example log message:

```
Jul 19 14:37:50 nubark named[757]: lame server resolving 'w3w3.com'
    (in 'w3w3.com'?): 216.117.131.52#53
```

Digging for name servers for w3w3.com at one of the .com gTLD servers yields the results below. We have truncated the output to tame **dig**'s verbosity; the **+short** flag to **dig** limits the output even more.

```
$ dig @e.gtld-servers.net w3w3.com ns
;; ANSWER SECTION:
w3w3.com.  172800  IN  NS  ns0.nameservices.net.
w3w3.com.  172800  IN  NS  ns1.nameservices.net.
```

If we query each of these servers in turn, we get an answer from ns0 but not from ns1:

```
$ dig @ns0.nameservices.net w3w3.com ns
;; ANSWER SECTION:
w3w3.com.  14400  IN  NS  ns0.nameservices.net.
w3w3.com.  14400  IN  NS  ns1.nameservices.net.
```

```
$ dig @ns1.nameservices.net w3w3.com ns
;; QUESTION SECTION:
;w3w3.com. IN  NS

;; AUTHORITY SECTION:
com.    92152   IN  NS  M.GTLD-SERVERS.NET.
com.    92152   IN  NS  I.GTLD-SERVERS.NET.
com.    92152   IN  NS  E.GTLD-SERVERS.NET.
```

The server ns1.nameservices.net has been delegated responsibility for w3w3.com by the .com servers, but it does not accept that responsibility. It is misconfigured, resulting in a lame delegation. Clients trying to look up w3w3.com will experience slow service. If w3w3.com is paying nameservices.net for DNS service, they deserve a refund!

Many sites point their lame-servers logging channel to **/dev/null** and don't fret about other people's lame delegations. That's fine as long as your own domain is squeaky clean and is not itself a source or victim of lame delegations.

Sometimes when you **dig** at an authoritative server in an attempt to find lameness, **dig** returns no information. Try the query again with the **+norecurse** flag so that you can see exactly what the server in question knows.

16.12 Recommended reading

DNS and BIND are described by a variety of sources, including the documentation that comes with the distributions, chapters in several books on Internet topics, books in the O'Reilly Nutshell series, books from other publishers, and various on-line resources.

Books and other documentation

THE NOMINUM AND ISC BIND DEVELOPMENT TEAMS. *BIND 9 Administrator Reference Manual*. This manual is included in the BIND distribution (**doc/arm**) from isc.org and is also available separately from the same site. It outlines the administration and management of BIND 9.

LIU, CRICKET, AND PAUL ALBITZ. *DNS and BIND (5th Edition)*. Sebastopol, CA: O'Reilly Media, 2006. This is pretty much the BIND bible, although it's getting a bit long in the tooth.

LIU, CRICKET. *DNS & BIND Cookbook*. Sebastopol, CA: O'Reilly Media, 2002. This baby version of the O'Reilly DNS book is task oriented and gives clear instructions and examples for various name server chores. Dated, but still useful.

LIU, CRICKET. *DNS and BIND on IPv6*. Sebastopol, CA: O'Reilly Media, 2011. This is an IPv6-focused addendum to DNS and BIND. It's short and includes only IPv6-related material.

LUCAS, MICHAEL W. *DNSSEC Mastery: Securing the Domain Name System with BIND*. Grosse Point Woods, MI: Tilted Windmill Press, 2013.

On-line resources

The web sites isc.org, dns-oarc.net, ripe.net, and nlnetlabs.nl contain a wealth of DNS information, research, measurement results, presentations, and other good stuff.

All the nitty-gritty details of the DNS protocol, resource records, and the like are summarized at iana.org/assignments/dns-parameters. This document contains a nice mapping from a DNS fact to the RFC that specifies it.

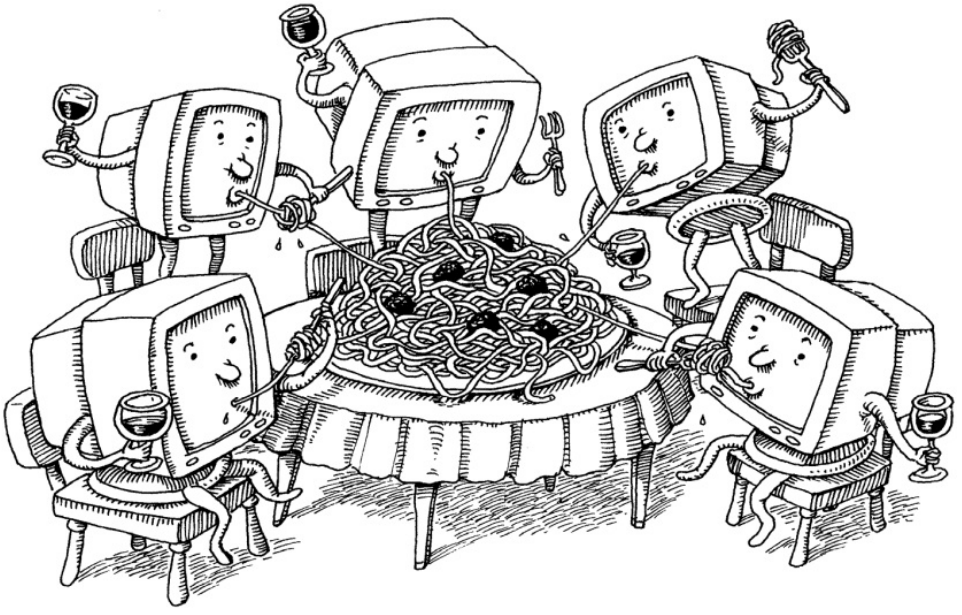
The *DNSSEC HOWTO*, a tutorial in disguise by Olaf Kolkman, is a 70-page document that covers the ins and outs of deploying and debugging DNSSEC. Get it at nlnetlabs.nl/dnssec_howto/dnssec_howto.pdf.

The RFCs

The RFCs that define the DNS system are available from rfc-editor.org. We formerly listed a page or so of the most important DNS-related RFCs, but there are now so many (more than 100, with another 50 Internet drafts) that you are better off searching rfc-editor.org to access the entire archive.

Refer to the **doc/rfc** and **doc/draft** directories of the current BIND distribution to see the entire complement of DNS-related RFCs.

17 Single Sign-On



Both users and system administrators would like account information to magically propagate to all an environment's computers so that a user can log in to any system with the same credentials. The common term for this feature is "single sign-on" (SSO), and the need for it is universal.

SSO involves two core security concepts: identity and authentication. A user identity is the abstract representation of an individual who needs access to a system or an application. It typically includes attributes such as a username, password, user ID, and email address. Authentication is the act of proving that an individual is the legitimate owner of an identity.

This chapter focuses on SSO as a component of UNIX and Linux systems within a single organization. For interorganizational SSO (such as might be needed to integrate your systems with a Software-as-a-Service provider), several standards-based and commercial SSO solutions are available. For those cases, we recommend learning about Security Assertion Markup Language (SAML) as a first step on your journey.

17.1 Core SSO elements

Although there are many ways to set up SSO, four elements are typically required in every scenario:

- A centralized directory store that contains user identity and authorization information. The most common solutions are directory services based on the Lightweight Directory Access Protocol (LDAP). In environments that mix Windows, UNIX, and Linux systems, the ever-popular Microsoft Active Directory service is a good choice. Active Directory includes a customized, nonstandard LDAP interface.

- A tool for managing user information in the directory. For native LDAP implementations, we recommend phpLDAPadmin or Apache Directory Studio. Both are easy-to-use, web-based tools that let you import, add, modify, and delete directory entries. If you're a Microsoft Active Directory fan-person, you can use the Windows-native MMC snap-in "Active Directory Users and Computers" to manage information in the directory.

- A mechanism for authenticating user identities. You can authenticate users directly against an LDAP store, but it's also common to use the Kerberos ticket-based authentication system originally developed at MIT. In Windows environments, Active Directory supplies LDAP access to user identities and uses a customized version of Kerberos for authentication.

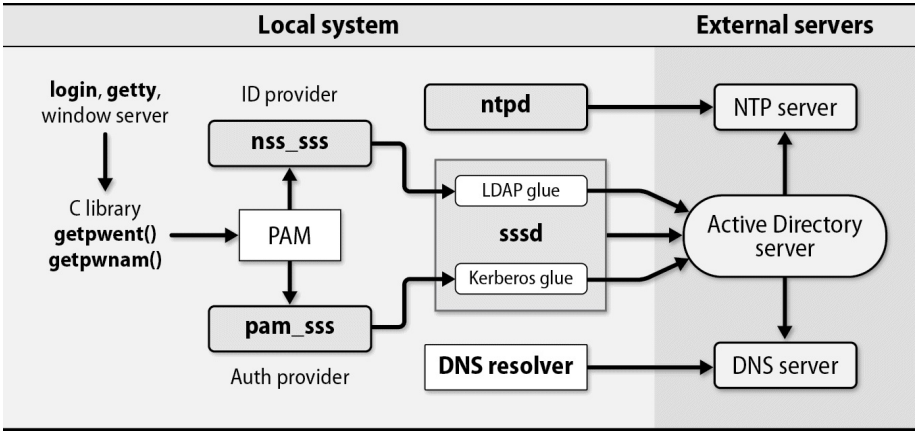
  Authentication on modern UNIX and Linux systems goes through the Pluggable Authentication Module system, aka PAM. You can use the System Security Services Daemon (**sssd**) to aggregate access to user identity and authentication services, then point PAM at **sssd**.

- Centralized-identity-and-authentication-aware versions of the C library routines that look up user attributes. These routines (e.g., **getpwent**) historically read flat files such as **/etc/passwd** and **/etc/group** and answered queries from their contents. These days, the data sources are configured in the name service switch file, **/etc/nsswitch.conf**.

The security community is divided over whether authentication is most secure when performed through LDAP or Kerberos. The road of life is paved with flat squirrels that couldn't decide. Pick an option and don't look back.

Exhibit A illustrates the high-level relationships of the various components in a typical configuration. This example uses Active Directory as the directory server. Note that both time synchronization (NTP) and hostname mapping (DNS) are critical for environments that use Kerberos because authentication tickets are time stamped and have a limited validity period.

Exhibit A: SSO components



In this chapter, we cover core LDAP concepts and introduce two specific LDAP servers for UNIX and Linux. We then discuss the steps needed to make a machine use a centralized directory service to process logins.

17.2 LDAP: "LIGHTWEIGHT" DIRECTORY SERVICES

A directory service is just a database, but one that makes a few assumptions. Any kind of data that matches the assumptions is a candidate for inclusion in the directory. The basic assumptions are as follows:

- Data objects are relatively small.
- The database will be widely replicated and cached.
- The information is attribute-based.
- Data are read often but written infrequently.
- Searching is a common operation.

The current IETF standards-track protocol that fills this role is the Lightweight Directory Access Protocol (LDAP). Ironically, LDAP is anything but lightweight. It was originally a gateway protocol that allowed TCP/IP clients to talk to an older directory service called X.500, which is now obsolete.

Microsoft's Active Directory is the most common instantiation of LDAP, and many sites use Active Directory for both Windows and UNIX/Linux authentication. For environments in which Active Directory isn't a candidate, the OpenLDAP package (openldap.org) has become the standard implementation. The 389 Directory Server (formerly known as the Fedora Directory Server and the Netscape Directory Server) is also open source and can be found at port389.org. The name derives from the fact that TCP port 389 is the default port for all LDAP implementations.

Uses for LDAP

Until you've had some experience with it, LDAP can be a slippery fish to grab hold of. LDAP by itself doesn't solve any specific administrative problem. Today, the most common use of LDAP is to act as a central repository for login names, passwords, and other account attributes. However, LDAP can be used in many other ways:

- LDAP can store additional directory information about users, such as phone numbers, home addresses, and office locations.
- Most mail systems—including **sendmail**, Exim, and Postfix—can draw a large part of their routing information from LDAP. See this page for more information about using LDAP with **sendmail**.
- LDAP makes it easy for applications (even those written by other teams and departments) to authenticate users without having to worry about the exact details of account management.
- LDAP is well supported by common scripting languages such as Perl and Python through code libraries. Ergo, LDAP can be an elegant way to distribute configuration information for locally written scripts and administrative utilities.
- LDAP is well supported as a public directory service. Most major email clients can read user directories stored in LDAP. Simple LDAP searches are also supported by many web browsers through an LDAP URL type.

The structure of LDAP data

LDAP data takes the form of property lists, which are known in the LDAP world as "entries." Each entry consists of a set of named attributes (such as `description` or `uid`) along with those attributes' values. Every attribute can have multiple values. Windows users might recognize this structure as being similar to that of the Windows registry.

As an example, here's a typical (but simplified) **/etc/passwd** line expressed as an LDAP entry:

```
dn: uid=ghopper,ou=People,dc=navy,dc=mil
objectClass: top
objectClass: person
objectClass: organizationalPerson
objectClass: inetOrgPerson
objectClass: posixAccount
objectClass: shadowAccount
uid: ghopper
cn: Grace Hopper
userPassword: {crypt}$1$pZaGA2RL$MPDJoc0afuhHY6yk8HQFp0
loginShell: /bin/bash
uidNumber: 1202
gidNumber: 1202
homeDirectory: /home/ghopper
```

This notation is a simple example of LDIF, the LDAP Data Interchange Format, which is used by most LDAP-related tools and server implementations. The fact that LDAP data can be easily converted back and forth from plain text is part of the reason for its success.

Entries are organized into a hierarchy through the use of "distinguished names" (attribute name: `dn`) that form a sort of search path. As in DNS, the "most significant bit" goes on the right. In the example above, the DNS name navy.mil has structured the top levels of the LDAP hierarchy. It has been broken down into two domain components (`dc`'s), "navy" and "mil," but this is only one of several common conventions.

Every entry has exactly one distinguished name. Entries are entirely separate from one another and have no hierarchical relationship except as is implicitly defined by the `dn` attributes. This approach enforces uniqueness and gives the implementation a hint as to how to efficiently index and search the data. Various LDAP consumers use the virtual hierarchy defined by `dn` attributes, but that's more a data-structuring convention than an explicit feature of the LDAP system. There are, however, provisions for symbolic links between entries and for referrals to other servers.

LDAP entries are typically schematized through the use of an `objectClass` attribute. Object classes specify the attributes that an entry can contain, some of which may be required for validity. The schemata also assign a data type to each attribute. Object classes nest and combine in the traditional object-oriented fashion. The top level of the object class tree is the class named `top`, which specifies merely that an entry must have an `objectClass` attribute.

Table 17.1 shows some common LDAP attributes whose meanings might not be immediately apparent. These attributes are case-insensitive.

Table 17.1: Some common attribute names found in LDAP hierarchies

Attribute	Stands for	What it is
`o`	Organization	Often identifies a site's top-level entry[a]
`ou`	Organizational unit	A logical subdivision, e.g., "marketing"
`cn`	Common name	The most natural name to represent the entry
`dc`	Domain component	Used at sites that model their hierarchy on DNS
`objectClass`	Object class	Schema to which this entry's attributes conform

a. Typically not used by sites that model their LDAP hierarchy on DNS

OpenLDAP: the traditional open source LDAP server

OpenLDAP is an extension of work originally done at the University of Michigan; it now continues as an open source project. It's shipped with most Linux distributions, although it is not necessarily included in the default installation. The documentation is perhaps best described as "brisk."

In the OpenLDAP distribution, **slapd** is the standard LDAP server daemon. In an environment with multiple OpenLDAP servers, **slurpd** runs on the master server and handles replication by pushing changes out to slave servers. A selection of command-line tools enable the querying and modification of LDAP data.

Setup is straightforward. First, create an **/etc/openldap/slapd.conf** file by copying the sample installed with the Open-LDAP server. These are the lines you need to pay attention to:

```
database bdb
suffix "dc=mydomain, dc=com"
rootdn "cn=admin, dc=mydomain, dc=com"
rootpw {crypt}abJnggxhB/yWI
directory /var/lib/ldap
```

The database format defaults to Berkeley DB, which is fine for data that will live within the OpenLDAP system. You can use a variety of other back ends, including ad hoc methods such as scripts that create the data on the fly.

suffix is your "LDAP basename." It's the root of your portion of the LDAP namespace, similar in concept to your DNS domain name. In fact, this example illustrates the use of a DNS domain name as an LDAP basename, which is a common practice.

rootdn is your administrator's name, and rootpw is the administrator's hashed password. Note that the domain components leading up to the administrator's name must also be specified. You can use **slappasswd** to generate the value for this field; just copy and paste its output into the file.

Because of the presence of this password hash, make sure that the **slapd.conf** file is owned by root and that its permissions are 600.

Edit **/etc/openldap/ldap.conf** to set the default server and basename for LDAP client requests. It's pretty straightforward—just set the argument of the host entry to the hostname of your server and set the base to the same value as the suffix in the **slapd.conf** file. Make sure both lines are uncommented. Here's an example from atrust.com:

```
BASE    dc=atrust,dc=com
URI     ldap://atlantic.atrust.com
```

At this point, you can start up **slapd** simply by running it with no arguments.

389 Directory Server: alternative open source LDAP server

Like OpenLDAP, the 389 Directory Server (port389.org) is an extension of the work done at the University of Michigan. However, it spent some years in the commercial world (at Netscape) before returning as an open source project.

There are several reasons to consider the 389 Directory Server as an alternative to OpenLDAP, but its superior documentation is one clear advantage. The 389 Directory Server comes with several professional grade administration and use guides, including detailed installation and deployment instructions.

A few other key features of the 389 Directory Server are

- Multimaster replication for fault tolerance and superior write performance
- Active Directory user and group synchronization
- A graphical console for all facets of user, group, and server management
- On-line, zero-downtime, LDAP-based update of schema, configuration, management, and in-tree Access Control Information (ACIs)

389 Directory Server has a much more active development community than does OpenLDAP. We generally recommend it over OpenLDAP for new installations.

From an administrative standpoint, the structure and operation of the two open source servers are strikingly similar. This fact is perhaps not too surprising since both packages were built on the same original code base.

LDAP Querying

To administer LDAP, you need to be able to see and manipulate the contents of the database. The phpLDAPadmin tool mentioned earlier is one of the nicer free tools for this purpose because it gives you an intuitive point-and-click interface. If phpLDAPadmin isn't an option, **ldapsearch** (distributed with both OpenLDAP and 389 Directory Server) is an analogous command-line tool that produces output in LDIF format. **ldapsearch** is especially good for use in scripts and for debugging environments in which Active Directory is acting as the LDAP server.

The following example query uses **ldapsearch** to look up directory information for every user whose cn starts with "ned." In this case, there's only one result. The meanings of the various command-line flags are discussed below.

```
$ ldapsearch -h atlantic.atrust.com -p 389
    -x -D "cn=trent,cn=users,dc=boulder,dc=atrust,dc=com" -W
    -b "cn=users,dc=boulder,dc=atrust,dc=com" "cn=ned*"

Enter LDAP Password: <password>

# LDAPv3
# base <cn=users,dc=boulder,dc=atrust,dc=com> with scope sub
# filter: cn=ned*
# requesting: ALL
#
# ned, Users, boulder.atrust.com
dn: cn=ned,cn=Users,dc=boulder,dc=atrust,dc=com
objectClass: top
objectClass: person
objectClass: organizationalPerson
objectClass: user
cn: ned
sn: McClain
telephoneNumber: 303 555 4505
givenName: Ned
distinguishedName: cn=ned,cn=Users,dc=boulder,dc=atrust,dc=com
displayName: Ned McClain
memberOf: cn=Users,cn=Builtin,dc=boulder,dc=atrust,dc=com
memberOf: cn=Enterprise Admins,cn=Users,dc=boulder,dc=atrust,dc=com
name: ned
sAMAccountName: ned
userPrincipalName: ned@boulder.atrust.com
lastLogonTimestamp: 129086952498943974
mail: ned@atrust.com
```

ldapsearch's **-h** and **-p** flags specify the host and port of the LDAP server you want to query, respectively.

You usually need to authenticate yourself to the LDAP server. In this case, the **-x** flag requests simple authentication (as opposed to SASL). The **-D** flag identifies the distinguished name of a user account that has the privileges needed to execute the query, and the **-W** flag makes **ldapsearch** prompt for the corresponding password.

The **-b** flag tells **ldapsearch** where in the LDAP hierarchy to start the search. This parameter is known as the baseDN; hence the **b**. By default, **ldapsearch** returns all matching entries below the baseDN. You can tweak this behavior with the **-s** flag.

The last argument is a "filter," which is a description of what you're searching for. It doesn't require an option flag. This filter, **cn=ned***, returns all LDAP entries that have a common name that starts with "ned". The filter is quoted to protect the star from shell globbing.

To extract all entries below a given baseDN, just use **objectClass=*** as the search filter—or leave the filter out, since this is the default.

Any arguments that follow the filter select specific attributes to return. For example, if you added **mail givenName** to the command line above, **ldapsearch** would return only the values of matching attributes.

Conversion of passwd and group files to LDAP

If you are moving to LDAP and your existing user and group information is stored in flat files, you may want to migrate your existing data. RFC2307 defines the standard mapping from traditional UNIX data sets, such as the **passwd** and **group** files, into the LDAP namespace. It's a useful reference document for sysadmins who want to use LDAP in a UNIX environment, at least in theory. In practice, the specifications are a lot easier for computers to read than for humans; you're better off looking at examples.

Padl Software offers a free set of Perl scripts that migrate existing flat files or NIS maps to LDAP. It's available from padl.com/OSS/MigrationTools.html, and the scripts are straightforward to run. They can be used as filters to generate LDIF, or they can be run against a live server to upload the data directly. For example, the **migrate_group** script converts this line from **/etc/group**:

```
csstaff:x:2033:evi,matthew,trent
```

to the following LDIF:

```
dn: cn=csstaff,ou=Group,dc=domainname,dc=com
cn: csstaff
objectClass: posixGroup
objectClass: top
userPassword: {crypt}x
gidNumber: 2033
memberuid: evi
memberuid: matthew
memberuid: trent
```

17.3 USING DIRECTORY SERVICES FOR LOGIN

Once you have a directory service set up, complete the following configuration chores so your system can enter SSO paradise:

- If you're planning to use Active Directory with Kerberos, configure Kerberos and join the system to the Active Directory domain.
- Configure **sssd** to communicate with the appropriate identity and authentication stores (LDAP, Active Directory, or Kerberos).
- Configure the name service switch, **nsswitch.conf**, to use **sssd** as a source of user, group, and password information.
- Configure PAM to service authentication requests through **sssd**.

Some software uses the traditional **getpwent** family of library routines to look up user information, whereas modern services often directly call the PAM authentication routines. Configure both PAM and **nsswitch.conf** to ensure a fully functional environment.

We walk through these procedures below.

Kerberos

Kerberos is a ticket-based authentication system that uses symmetric key cryptography. Its recent popularity has been driven primarily by Microsoft, which uses it as part of Active Directory and Windows authentication. For SSO purposes, we describe how to integrate with an Active Directory Kerberos environment on both Linux and FreeBSD. If you're using an LDAP server other than Active Directory or if you want to authenticate against Active Directory through the LDAP path rather than the Kerberos path, you can skip to the discussion of **sssd** here.

See this page for general information about Kerberos.

Linux Kerberos configuration for AD integration

Sysadmins often want their Linux systems to be members of an Active Directory domain. In the past, the complexity of this configuration drove some of those sysadmins to drink. Fortunately, the debut of **realmd** has made this task much simpler. **realmd** acts as a configuration tool for both **sssd** and Kerberos.



Before attempting to join an Active Directory domain, verify the following:

- **realmd** is installed on the Linux system you're joining to the domain.
- **sssd** is installed (see below).
- **ntpd** is installed and running.
- You know the correct name of your AD domain.
- You have credentials for an AD account that is allowed to join systems to the domain. This action results in a Kerberos ticket-granting ticket (TGT) being issued to the system so that it can perform authentication operations going forward without access to an administrator's password.

For example, if your AD domain name is ULSAH.COM and the AD account trent is allowed to join systems to the domain, you can use the following command to join your system to the domain:

```
$ sudo realm join --user=trent ULSAH.COM
```

You can then verify the result:

```
$ realm list
ulsah.com
  type: kerberos
  realm-name: ULSAH.COM
  domain-name: ulsah.com
  configured: kerberos-member
  server-software: active-directory
  client-software: sssd
  required-package: sssd
  required-package: adcli
  required-package: samba-common
  login-formats: %U@ulsah.com
  login-policy: allow-real logins
```

FreeBSD Kerberos configuration for AD integration

Kerberos is infamous for its complex configuration process, especially on the server side. Unfortunately, FreeBSD has no slick tool akin to Linux's **realmd** that configures Kerberos and joins an Active Directory domain in one step. However, you need to set up only the client side of Kerberos. The configuration file is **/etc/krb5.conf**.



First, double-check that the system's fully qualified domain name has been included in **/etc/hosts** and that NTP is configured and working. Then edit **krb5.conf** to add the realm as shown in the following example. Substitute the name of your site's AD domain for ULSAH.COM.

```
[logging]
    default = FILE:/var/log/krb5.log
[libdefaults]
    clockskew = 300
    default_realm = ULSAH.COM
    kdc_timesync = 1
    ccache_type = 4
    forwardable = true
    proxiable = true
[realms]
    ULSAH.COM = {
        kdc = dc.ulsah.com
        admin_server = dc.ulsah.com
        default_domain = ULSAH
    }
[domain_realm]
    .ulsah.com = ULSAH.COM
    ulsah.com = ULSAH.COM
```

Several values are of interest in the example above. A 5-minute clock skew is allowed even though the time is set through NTP. This leeway allows the system to function even in the event of an NTP problem. The default realm is set to the AD domain, and the key distribution center

(or KDC) is configured as an AD domain controller. **krb5.log** might come in handy for debugging.

Request a ticket from the Active Directory controller by running the **kinit** command. Specify a valid domain user account. The “administrator” account is usually a good test, but any account will do. When prompted, type the domain password.

```
$ kinit administrator@ULSAH.COM
Password for administrator@ULSAH.COM: <password>
```

Use **klist** to show the Kerberos ticket:

```
$ klist
Ticket cache: FILE:/tmp/krb5cc_1000
Default principal: administrator@ULSAH.COM

Valid starting       Expires              Service principal
04/30/17 13:40:19    04/30/17 23:40:21    krbtgt/ULSAH.COM@ULSAH.COM
        renew until 05/01/17 13:40:19

Kerberos 4 ticket cache: /tmp/tkt1000
klist: You have no tickets cached
```

If a ticket is displayed, authentication was successful. In this case, the ticket is valid for 10 hours and can be renewed for 24 hours. You can use the **kdestroy** command to invalidate the ticket.

The last step is to join the system to the domain, as shown below. The administrator account used (in this case, trent) must have the appropriate privileges on the Active Directory server to join systems to the domain.

```
$ net ads join -U trent
Enter trent's password: <password>
Using short domain -- ULSAH
Joined 'example.ulsah.com' to domain 'ULSAH.COM'
```

See the man page for **krb5.conf** for additional configuration options.

sssd: the System Security Services Daemon



The UNIX and Linux road to SSO nirvana has been a rough one. Years ago, it was common to set up independent authentication for every service or application. This approach often resulted in a morass of separate configurations and undocumented dependencies that were impossible to manage over time. Users' passwords would work with one application but not another, causing frustration for everyone.

Microsoft formerly published extensions (originally called "Services for UNIX," then "Windows Security and Directory Services for UNIX," and finally, "Identity Management for UNIX" in Windows Server 2012) that facilitated the housing of UNIX users and groups within Active Directory. Putting the authority for managing these attributes in a non-UNIX system was an unnatural fit, however. To the relief of many, Microsoft discontinued this feature as of Windows Server 2016.

These issues needed some kind of comprehensive solution, and that's just what we got with **sssd**, the System Security Services Daemon. Available for both Linux and FreeBSD, **sssd** is a one-stop shop for user identity wrangling, authentication, and account mapping. It can also cache credentials off-line, which is useful for mobile devices. **sssd** supports authentication both through native LDAP and through Kerberos.

You configure **sssd** through the **sssd.conf** file. Here's a basic example for an environment that uses Active Directory as the directory service:

```
[sssd]
services = nss, pam
domains = ULSAH.COM

[domain/ULSAH.COM]
id_provider = ad
access_provider = ad
```

If you are using a non-AD LDAP server, your **sssd.conf** file might look more like this:

```
[sssd]
services = nss, pam
domains = LDAP

[domain/LDAP]
id_provider = ldap
auth_provider = ldap
ldap_uri = ldap://ldap.ulsah.com
ldap_user_search_base = dc=ulsah,dc=com
tls_reqcert = demand
ldap_tls_cacert = /etc/pki/tls/certs/ca-bundle.crt
```

For obvious security reasons, **sssd** does not allow authentication over an unencrypted channel, so the use of LDAPS/TLS is required. Setting the tls_reqcert attribute to demand in the example above forces **sssd** to validate the server certificate as an additional check. **sssd** drops the connection if the certificate is found to be deficient.

Once **sssd** is up and running, you must tell the system to use it as the source for identity and authentication information. Configuring the name service switch and configuring PAM are the next steps in this process.

nsswitch.conf: the name service switch

The name service switch (NSS) was developed to ease selection among various configuration databases and name resolution mechanisms. All the configuration goes into the **/etc/nsswitch.conf** file.

The syntax is simple: for a given type of lookup, you simply list the sources in the order they should be consulted. The system's local **passwd** and **group** files should always be consulted first (specified by files), but you can then punt to Active Directory or another directory service by way of **sssd** (specified by sss). These entries do the trick:

```
passwd: files sss
group:  files sss
shadow: files sss
```

Once you've configured the **nsswitch.conf** file, you can test the configuration with the command **getent passwd.** This command prints the user accounts defined by all sources in **/etc/passwd** format:

```
$ getent passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
...
bwhaley:x:10006:10018::/home/bwhaley:/bin/sh
guest:*:10001:10001:Guest:/home/ULSAH/guest:/bin/bash
ben:*:10002:10000:Ben Whaley:/home/ULSAH/ben:/bin/bash
krbtgt:*:10003:10000:krbtgt:/home/ULSAH/krbtgt:/bin/bash
```

The only way to distinguish local users from domain accounts is by user ID and by the path of the home directory, as seen in the last three entries above.

PAM: cooking spray or authentication wonder?

PAM stands for “pluggable authentication modules.” The PAM system relieves programmers of the chore of implementing direct connections to authentication systems and gives sysadmins flexible, modular control over the system’s authentication methods. Both the concept and the term come from Sun Microsystems (now part of Oracle *sniff*) and from a 1996 paper by Samar and Lai of SunSoft.

In the distant past, commands like **login** included hardwired authentication code that prompted the user for a password, tested the password against the encrypted version obtained from **/etc/shadow** (**/etc/passwd** at that time), and rendered a judgment as to whether the two passwords matched. Of course, other commands (e.g., **passwd**) contained similar code. It was impossible to change authentication methods without source code, and administrators had little or no control over details such as whether the system should accept “password” as a valid password. PAM changed all that.

PAM puts the system’s authentication routines into a shared library that **login** and other programs can call. By separating authentication functions into a discrete subsystem, PAM makes it easy to integrate new advances in authentication and encryption into the computing environment. For instance, multifactor authentication is supported without changes to the source code of **login** and **passwd**.

For the sysadmin, setting the right level of security for authentication has become a simple configuration task. Programmers win, too, since they no longer have to write tedious authentication code. More importantly, their authentication systems are implemented correctly on the first try. PAM can authenticate all sorts of activities: user logins, other forms of system access, use of protected web sites, even the configuration of applications.

PAM configuration

PAM configuration files are a series of one-liners, each of which names a particular PAM module to be used on the system. The general format is

```
module-type control-flag module-path [ arguments ]
```

Fields are separated by whitespace.

The order in which modules appear in the PAM configuration file is important. For example, the module that prompts the user for a password must come before the module that checks that password for validity. One module can pass its output to the next by setting either environment variables or PAM variables.

The *module-type* parameter—auth, account, session, or password—determines what the module is expected to do. auth modules identify the user and grant group memberships. Modules that do account chores enforce restrictions such as limiting logins to particular times of day, limiting the

number of simultaneous users, or limiting the ports on which logins can occur. (For example, you would use an account-type module to restrict root logins to the console.) session chores include tasks that are done before or after a user is granted access; for example, mounting the user's home directory. Finally, password modules change a user's password or passphrase.

The *control-flag* specifies how the modules in the stack should interact to produce an ultimate result for the stack. Table 17.2 shows the common values.

Table 17.2: PAM control flags

Flag	Stop on failure?	Stop on success?	Comments
`include`	–	–	Includes another file at this point in the stack
`optional`	No	No	Significant only if this is the lone module
`required`	No	No	Failure eventually causes the stack to fail
`requisite`	Yes	No	Same as `required`, but fails stack immediately
`sufficient`	No	Yes	The name is kind of a lie; see comments below

If PAM could simply return a failure code as soon as the first individual module in a stack failed, the *control-flags* system would be simpler. Unfortunately, the system is designed so that most modules get a chance to run regardless of their sibling modules' success or failure, and this fact causes some subtleties in the flow of control. (The intent is to prevent an attacker from learning which module in the PAM stack caused the failure.)

required modules are required to succeed; a failure of any one of them guarantees that the stack as a whole will eventually fail. However, the failure of a module that is marked required doesn't immediately stop execution of the stack. If you want that behavior, use the requisite control flag instead of required.

The success of a sufficient module aborts the stack immediately. However, the ultimate result of the stack isn't guaranteed to be a success because sufficient modules can't override the failure of earlier required modules. If an earlier required module has already failed, a successful sufficient module aborts the stack *and* returns failure as the overall result.

Before you modify your systems' security settings, make sure you understand the system thoroughly and that you double-check the particulars. (You won't configure PAM every day. How long will you remember which version is requisite and which is required?)

PAM example

An example **/etc/pam.d/login** file from a Linux system running **sssd** is reproduced below. We expanded the included files to form a more coherent example.

```
auth      requisite   pam_nologin.so
auth      [user_unknown=ignore success=ok ignore=ignore auth_err=die
            default=bad] pam_securetty.so
auth      required    pam_env.so
auth      sufficient  pam_unix2.so
auth      sufficient  pam_sss.so use_first_pass

account   required    pam_unix2.so
account   [default=bad success=ok user_unknown=ignore] pam_sss.so

password  requisite   pam_pwcheck.so nullok cracklib
password  required    pam_unix2.so use_authtok nullok
password  sufficient  pam_sss.so use_authtok

session   required    pam_loginuid.so
session   required    pam_limits.so
session   required    pam_unix2.so
session   sufficient  pam_sss.so
session   optional    pam_umask.so
session   required    pam_lastlog.so nowtmp
session   optional    pam_mail.so standard
session   optional    pam_ck_connector.so
```

The auth stack includes several modules. On the first line, the pam_nologin module checks for the existence of the **/etc/nologin** file. If it exists, the module aborts the login immediately unless the user is root. The pam_securetty module ensures that root can log in only on terminals listed in **/etc/securetty**. This line uses an alternative Linux syntax described in the **pam.conf** man page. In this case, the requested behavior is similar to that of the required control flag. pam_env sets environment variables from **/etc/security/pam_env.conf**, then pam_unix2 checks the user's credentials by performing standard UNIX authentication. If the user doesn't have a local UNIX account, pam_sss attempts authentication through **sssd**. If any of these modules fail, the auth stack returns an error.

The account stack includes only the pam_unix2 and pam_sss modules. In this context, they assess the validity of the account itself. The modules return an error if, for example, the account has expired or the password must be changed. In the latter case, the relevant module collects a new password from the user and passes it on to the password modules.

The pam_pwcheck line checks the strength of proposed new passwords by calling the **cracklib** library. It returns an error if a new password does not meet the requirements. However, it also allows empty passwords because of the nullok flag. The pam_unix2 and pam_sss lines update the actual password.

Finally, the session modules perform several housekeeping chores. pam_loginuid sets the kernel's loginuid process attribute to the user's UID. pam_limits reads resource usage limits from **/etc/security/limits.conf** and sets the corresponding process parameters that enforce them. pam_unix2 and pam_sss log the user's access to the system, and pam_umask sets an initial file creation

mode. The pam_lastlog module displays the user's last login time as a security check, and the pam_mail module prints a note if the user has new mail. Finally, pam_ck_connector notifies the **ConsoleKit** daemon (a system-wide daemon that manages login sessions) of the new login.

At the end of the process, the user has been successfully authenticated and PAM returns control to **login**.

17.4 Alternative approaches

Although LDAP is currently the most popular method for centralizing user identity and authentication information within an organization, many other approaches have emerged over the decades. Two older options, NIS and **rsync**, are still in use in some isolated pockets.

NIS: the Network Information Service

NIS, released by Sun in the 1980s, was the first “prime time” administrative database. It was originally called the Sun Yellow Pages, but eventually had to be renamed for legal reasons. NIS commands still begin with the letters **yp**, so it’s hard to forget the original name. NIS was widely adopted and is still supported in both FreeBSD and Linux.

These days, however, NIS is an old gray mare. NIS should not be used for new deployments, and existing deployments should be migrated to a modern day alternative such as LDAP.

rsync: transfer files securely

rsync, written by Andrew Tridgell and Paul Mackerras, is a bit like a souped-up version of **scp** that is scrupulous about preserving links, modification times, and permissions. It is network efficient because it looks inside individual files and attempts to transmit only the differences between versions.

One quick-and-dirty approach to distributing files such as **/etc/passwd** and **/etc/group** is to set up a **cron** job to **rsync** them from a master server. Although this scheme is easy to set up and might be useful in a pinch, it requires that all changes be applied directly to the master, including user password changes.

As an example, the command

```
# rsync -gopt -e ssh /etc/passwd /etc/shadow lollipop:/etc
```

transfers the **/etc/passwd** and **/etc/shadow** files to the machine lollipop. The **-gopt** options preserve the permissions, ownerships, and modification times of the file. **rsync** uses **ssh** as the transport, and so the connection is encrypted. However, **sshd** on lollipop must be configured not to require a password if you want to run this command from a script. Of course, such a setup has significant security implications. Coder beware!

With the **--include** and **--exclude** flags you can specify a list of regular expressions to match against filenames, so you can set up a sophisticated set of transfer criteria. If the command line gets too unwieldy, you can read the patterns from separate files with the **--include-file** and **--exclude-file** options.

Configuration management tools such as Ansible are another common way to distribute files among systems. See Chapter 23, *Configuration Management*, for more details.

17.5 Recommended reading

A good general introduction to LDAP is *LDAP for Rocket Scientists*, which covers LDAP architecture and protocol. Find it on-line at zytrax.com/books/ldap. Another good source of information is the LDAP-related RFCs, which are numerous and varied. As a group, they tend to convey an impression of great complexity, which is somewhat unrepresentative of average use. Table 17.3 list some of the most important of these RFCs.

Table 17.3: Important LDAP-related RFCs

RFC	Title
2307	An Approach for Using LDAP as a Network Information Service
2820	Access Control Requirements for LDAP
2849	LDAP Data Interchange Format (LDIF)—Technical Specification
3112	LDAP Authentication Password Schema
3672	Subentries in the Lightweight Directory Access Protocol (LDAP)
4511	LDAP: The Protocol
4512	LDAP: Directory Information Models
4513	LDAP: Authentication Methods and Security Mechanisms
4514	LDAP: String Representation of Distinguished Names
4515	LDAP: String Representation of Search Filters
4516	LDAP: Uniform Resource Locator
4517	LDAP: Syntaxes and Matching Rules
4519	LDAP: Schema for User Applications

In addition, there are a couple of oldie-but-goodie books on LDAP:

Carter, Gerald. *LDAP System Administration*. Sebastopol, CA: O'Reilly Media, 2003.

Voglmaier, Reinhard. *The ABCs of LDAP: How to Install, Run, and Administer LDAP Services*. Boca Raton, FL: Auerbach Publications, 2004.

There's also a decent book focused entirely on PAM:

Lucas, Michael. *PAM Mastery*. North Charleston, SC: CreateSpace, 2016.

Finally, the O'Reilly book on Active Directory is excellent:

Desmond, Brian, Joe Richards, Robbie Allen, and Alistair G. Lowe-Norris. *Active Directory: Designing, Deploying, and Running Active Directory*. Sebastopol, CA: O'Reilly Media, 2013.

18 Electronic Mail



Decades ago, cooking a chicken dinner involved not just frying the chicken, but selecting a tender young chicken out of the coop, terminating it with a kill signal, plucking the feathers, etc. Today, most of us just buy a package of chicken at the grocery store or butcher shop and skip the mess.

Email has evolved in a similar way. Ages ago, it was common for organizations to hand-craft their email infrastructure, sometimes to the point of predetermining exact mail routing. Today, many organizations use packaged, cloud-hosted email services such as Google Gmail or Microsoft Office 365.

Even if your email system runs in the cloud, you will still have occasion to understand, support, and interact with it as an administrator. If your site uses local email servers, the workload expands even further to include configuration, monitoring, and testing chores.

If you find yourself in one of these more hands-on scenarios, this chapter is for you. Otherwise, skip this material and spend your email administration time responding to messages from wealthy foreigners who need help moving millions of dollars in exchange for a large reward. (Just kidding, of course.)

18.1 Mail system architecture

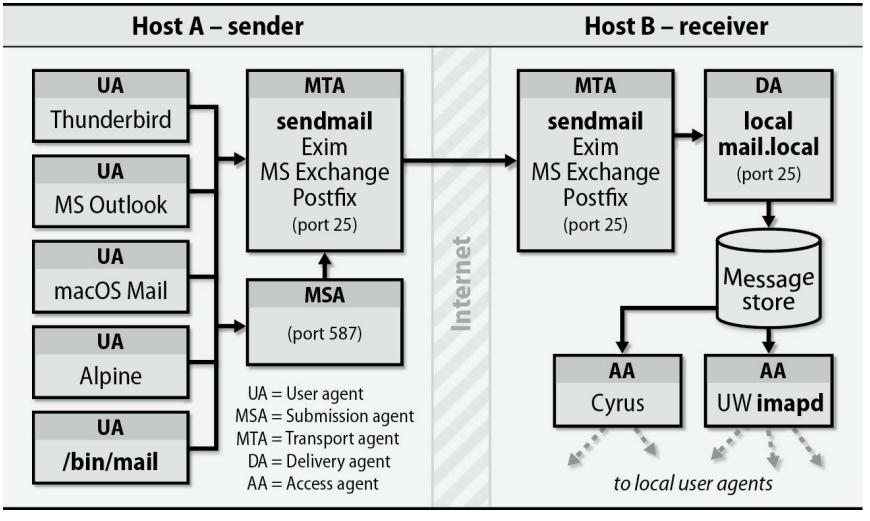
A mail system consists of several distinct components:

- A “mail user agent” (MUA or UA) that lets users read and compose mail
- A “mail submission agent” (MSA) that accepts outgoing mail from an MUA, grooms it, and submits it to the transport system
- A “mail transport agent” (MTA) that routes messages among machines
- A “delivery agent” (DA) that places messages in a local message store (the receiving users’ mailboxes or, sometimes, a database)
- An optional “access agent” (AA) that connects the user agent to the message store (e.g., through the IMAP or POP protocol)

Note that these functional divisions are somewhat abstract. Real-world mail systems break out these roles into somewhat different packages.

Attached to some of these functions are tools for recognizing spam, viruses, and (outbound) internal company secrets. Exhibit A illustrates how the various pieces fit together as a message winds its way from sender to receiver.

Exhibit A: Mail system components



User agents

Email users run a user agent (sometimes called an email client) to read and compose messages. Email messages originally consisted only of text, but a standard known as Multipurpose Internet Mail Extensions (MIME) now encodes text formats and attachments (including viruses) into email. It is supported by most user agents. Since MIME generally does not affect the addressing or transport of mail, we do not discuss it further.

/bin/mail was the original user agent, and it remains the "good ol' standby" for reading text email messages at a shell prompt. Since email on the Internet has moved far beyond the text era, text-based user agents are no longer practical for most users. But we shouldn't throw **/bin/mail** away; it's still a handy interface for scripts and other programs.

One of the elegant features illustrated in Exhibit A is that a user agent doesn't necessarily need to be running on the same system—or even on the same platform—as the rest of your mail system. Users can reach their email from a Windows laptop or smartphone through access agent protocols such as IMAP and POP.

Submission agents

MSAs, a late addition to the email pantheon, were invented to offload some of the computational tasks of MTAs. MSAs make it easy for mail hub servers to distinguish incoming from outbound email (when making decisions about allowing relaying, for example) and give user agents a uniform and simple configuration for outbound mail.

The MSA is a sort of “receptionist” for new messages being injected into the system by local user agents. An MSA sits between the user agent and the transport agent and takes over several functions that were formerly a part of the MTA’s job. An MSA implements secure (encrypted and authenticated) communication with user agents and often does minor header rewriting and cleanup on incoming messages. In many cases, the MSA is really just the MTA listening on a different port with a different configuration applied.

MSAs speak the same mail transfer protocol used by MTAs, so they appear to be MTAs from the perspective of user agents. However, they typically listen for connections on port 587 rather than port 25, the MTA standard. For this scheme to work, user agents must connect on port 587 instead of port 25. If your user agents cannot be taught to use port 587, you can still run an MSA on port 25, but you must do so on a system other than the one that runs your MTA; only one process at a time can listen on a particular port.

If you use an MSA, be sure to configure your transport agent so that it doesn’t duplicate any of the rewriting or header fix-up work done by the MSA. Duplicate processing won’t affect the correctness of mail handling, but it does represent useless extra work.

See this page for more information about SMTP authentication.

Since your MSA uses your MTA to relay messages, the MSA and MTA must use SMTP-AUTH to authenticate each other. Otherwise, you create a so-called open relay that spammers can exploit and that other sites will blacklist you for.

Transport agents

A transport agent must accept mail from a user agent or submission agent, understand the recipients' addresses, and somehow get the mail to the correct hosts for delivery. Transport agents speak the Simple Mail Transport Protocol (SMTP), which was originally defined in RFC821 but has now been superseded and extended by RFC5321. The extended version is called ESMTP.

An MTA's list of chores, as both a mail sender and receiver, includes

- Receiving email messages from remote mail servers
- Understanding the recipients' addresses
- Rewriting addresses to a form understood by the delivery agent
- Forwarding the message to the next responsible mail server or passing it to a local delivery agent to be saved to a user's mailbox

The bulk of the work involved in setting up a mail system relates to the configuration of the MTA. In this book, we cover three open source MTAs: **sendmail**, Exim, and Postfix.

Local delivery agents

A delivery agent, sometimes called a local delivery agent (LDA), accepts mail from a transport agent and delivers it to the appropriate recipients' mailboxes on the local machine. As originally specified, email can be delivered to a person, to a mailing list, to a file, or even to a program. However, the last two types of recipients can weaken the security and safety of your system.

MTAs usually include a built-in local delivery agent for easy deliveries. **procmail** (procmail.org) and Maildrop (courier-mta.org/maildrop) are LDAs that can filter or sort mail before delivering it. Some access agents (AAs) also have built-in LDAs that do both delivery and local housekeeping chores.

Message stores

A message store is the final resting place of an email message once it has completed its journey across the Internet and been delivered to recipients.

Mail has traditionally been stored in either **mbox** format or **Maildir** format. The former stores all mail in a single file, typically **/var/mail/***username*, with individual messages separated by a special From line. **Maildir** format stores each message in a separate file. A file for each message is more convenient but creates directories with many, many small files; some filesystems may not be amused.

Flat files in **mbox** or **Maildir** format are still widely used, but ISPs with thousands or millions of email clients have typically migrated to other technologies for their message stores, usually databases. Unfortunately, that means that message stores are becoming more opaque.

Access agents

Two protocols access message stores and download email messages to a local device (workstation, laptop, smartphone, etc.): Internet Message Access Protocol version 4 (IMAP4) and Post Office Protocol version 3 (POP3). Earlier versions of these protocols had security issues. Be sure to use a version (IMAPS or POP3S) that incorporates SSL encryption and hence does not transmit passwords in cleartext over the Internet.

IMAP is significantly better than POP. It delivers your mail one message at a time rather than all at once, which is kinder to the network (especially on slow links) and better for someone who travels from location to location. IMAP is especially good at dealing with the giant attachments that some folks like to send: you can browse the headers of your messages and not download the attachments until you are ready to deal with them.

18.2 Anatomy of a mail message

A mail message has three distinct parts:

- Envelope
- Headers
- Body of the message

The envelope determines where the message will be delivered or, if the message can’t be delivered, to whom it should be returned. The envelope is invisible to users and is not part of the message itself; it’s used internally by the MTA.

Envelope addresses generally agree with the From and To lines of the header when the sender and recipient are individuals. The envelope and headers might not agree if the message was sent to a mailing list or was generated by a spammer who is trying to conceal his identity.

Headers are a collection of property/value pairs as specified in RFC5322 (updated by RFC6854). They record all kinds of information about the message, such as the date and time it was sent, the transport agents through which it passed on its journey, and who it is to and from. The headers are a bona fide part of the mail message, but user agents typically hide the less interesting ones when displaying messages for the user.

The body of the message is the content to be sent. It usually consists of plain text, although that text often represents a mail-safe encoding for various types of binary or rich-text content.

Dissecting mail headers to locate problems within the mail system is an essential sysadmin skill. Many user agents hide the headers, but there is usually a way to see them, even if you have to use an editor on the message store.

Below are most of the headers (with occasional truncations indicated by …) from a typical nonspam message. We removed another half page of headers that Gmail uses as part of its spam filtering. (In memory of Evi, who originally owned this chapter, this historical example has been kept intact.)

```
Delivered-To: sailingevi@gmail.com
Received: by 10.231.39.205 with SMTP id…; Fri, 24 May 2013 08:14:27
    -700 (PDT)
Received: by 10.114.163.26 with SMTP id…; Fri, 24 May 2013 08:14:26
    -700 (PDT)
Return-Path: <david@schweikert.ch>
Received: from mail-relay.atrust.com
    (mail-relay.atrust.com [63.173.189.2]) by mx.google.com with
    ESMTP id 17si2166978pxi.34.2009.10.16.08.14.20; Fri, 24 May 2013
    08:14:25 -0700 (PDT)
Received-SPF: fail (google.com: domain of david@schweikert.
    ch does not designate 63.173.189.2 as permitted sender)
    client-ip=63.173.189.2;
Authentication-Results: mx.google.com; spf=hardfail (google.com: domain
    of david@schweikert.ch does not designate 63.173.189.2 as permitted
    sender) smtp.mail=david@schweikert.ch
Received: from mail.schweikert.ch (nigel.schweikert.ch [88.198.52.145])
    by mail-relay.atrust.com (8.12.11/8.12.11) with ESMTP id n9GFEDKA0
    for <evi@atrust.com>; Fri, 24 May 2013 09:14:14 -0600
Received: from localhost (localhost.localdomain [127.0.0.1]) by mail.
    schweikert.ch (Postfix) with ESMTP id 3251112DA79; Fri, 24 May 2013
    17:14:12 +0200 (CEST)
X-Virus-Scanned: Debian amavisd-new at mail.schweikert.ch
Received: from mail.schweikert.ch ([127.0.0.1]) by localhost (mail.
    schweikert.ch [127.0.0.1]) (amavisd-new, port 10024) with ESMTP id
    dV8BpT7rhJKC; Fri, 24 May 2013 17:14:07 +0200 (CEST)...
Received: by mail.schweikert.ch (Postfix, from userid 1000)
    id 2A15612DB89; Fri, 24 May 2013 17:14:07 +0200 (CEST)
Date: Fri, 24 May 2013 17:14:06 +0200
From: David Schweikert <david@schweikert.ch>
To: evi@atrust.com
Cc: Garth Snyder <garth@garthsnyder.com>
Subject: Email chapter comments

Hi evi,

I just finished reading the email chapter draft, and I was pleased to see
...
```

To decode this beast, start reading the Received lines, but start from the bottom (sender side). This message went from David Schweikert's home machine in the schweikert.ch domain to his mail server (mail.schweikert.ch), where it was scanned for viruses. It was then forwarded to the recipient evi@atrust.com. However, the receiving host mail-relay.atrust.com sent it on to sailingevi@gmail.com, where it entered Evi's mailbox.

See [this page] for more information about SPF.

Midway through the headers, you see an SPF (Sender Policy Framework) validation failure, an indication that the message has been flagged as spam. This failure happened because Google checked the IP address of mail-relay.atrust.com and compared it with the SPF record at schweikert.ch; of course, it doesn't match. This is an inherent weakness of relying on SPF records to identify forgeries—they don't work for mail that has been relayed.

You can often see the MTAs that were used (Postfix at schweikert.ch, **sendmail** 8.12 at atrust.com), and in this case, you can also see that virus scanning was performed through **amavisd-new** on port 10,024 on a machine running Debian Linux. You can follow the progress of the message from the Central European Summer Time zone (CEST +0200), to Colorado (-0600), and on to the Gmail server (PDT -0700); the numbers are the differences between local time and UTC, Coordinated Universal Time. A lot of info is stashed in the headers!

Here are the headers, again truncated, from a spam message:

```
Delivered-To: sailingevi@gmail.com
Received: by 10.231.39.205 with SMTP id…; Fri, 19 Oct 2009 08:59:32
    -0700…
Received: by 10.231.5.143 with SMTP id…; Fri, 19 Oct 2009 08:59:31
    -0700…
Return-Path: <smotheringl39@sherman.dp.ua>
Received: from mail-relay.atrust.com (mail-relay.atrust.com
    [63.173.189.2]) …
Received-SPF: neutral (google.com: 63.173.189.2 is neither
    permitted nor denied by best guess record for domain of
    smotheringl39@sherman.dp.ua) client-ip=63.173.189.2;
Authentication-Results: mx.google.com; spf=neutral (google.
    com: 63.173.189.2 is neither permitted nor denied by best
    guess record for domain of smotheringl39@sherman.dp.ua)
    smtp.mail=smotheringl39@sherman.dp.ua
Received: from SpeedTouch.lan (187-10-167-249.dsl.telesp.net.br
    [187.10.167.249] (may be forged)) by mail-relay.atrust.com …
Received: from 187.10.167.249 by relay2.trifle.net; Fri, 19 Oct 2009
    13:59: …
From: "alert@atrust.com" <alert@atrust.com>
To: <ned@atrust.com>
Subject: A new settings file for the ned@atrust.com mailbox
Date: Fri, 19 Oct 2009 13:59:12 -0300 …
```

According to the From header, this message's sender is alert@atrust.com. But according to the Return-Path header, which contains a copy of the envelope sender, the originator was smotheringl39@sherman.dp.ua, an address in the Ukraine. The first MTA that handled the message is at IP address 187.10.167.249, which is in Brazil. Sneaky spammers… It's important to note that many of the lines in the header, including the Received lines, may have been forged. Use this data with extreme caution.

The SPF check at Google fails again, this time with a “neutral” result because the domain sherman.dp.ua does not have an SPF record with which to compare the IP address of mail-relay.atrust.com.

The recipient information is also at least partially untrue. The To header says the message is addressed to ned@atrust.com. However, the envelope recipient addresses must have included evi@atrust.com in order for the message to be forwarded to sailingevi@gmail.com for delivery.

18.3 The SMTP protocol

The Simple Mail Transport Protocol (SMTP) and its extended version, ESMTP, have been standardized in the RFC series (RFC5321, updated by RFC7504) and are used for most message hand-offs among the various pieces of the mail system:

- UA-to-MSA or -MTA as a message is injected into the mail system
- MSA-to-MTA as the message starts its delivery journey
- MTA- or MSA-to-antivirus or -antispam scanning programs
- MTA-to-MTA as a message is forwarded from one site to another
- MTA-to-DA as a message is delivered to the local message store

Because the format of messages and the transfer protocol are both standardized, my MTA and your MTA don't have to be the same or even know each other's identity; they just have to both speak SMTP or ESMTP. Your various mail servers can run different MTAs and interoperate just fine.

True to its name, SMTP is…simple. An MTA connects to your mail server and says, "Here's a message; please deliver it to user@your.domain." Your MTA says "OK."

Requiring strict adherence to the SMTP protocol has become a technique for fighting spam and malware, so it's important for mail administrators to be somewhat familiar with the protocol. The language has only a few commands; Table 18.1 shows the most important ones.

Table 18.1: SMTP commands

Command	Function
HELO *hostname*	Identifies the connecting host if speaking SMTP
EHLO *hostname*	Identifies the connecting host if speaking ESMTP
MAIL FROM: *revpath*	Initiates a mail transaction (envelope sender)
RCPT TO: *fwdpath* [a]	Identifies envelope recipient(s)
VRFY *address*	Verifies that *address* is valid (deliverable)
EXPN *address*	Shows expansion of aliases and **.forward** mappings
DATA	Begins the message body (preceded by headers) [b]
QUIT	Ends the exchange and closes the connection
RSET	Resets the state of the connection
HELP	Prints a summary of SMTP commands

a. There can be multiple RCPT commands for a message.
b. You terminate the body by entering a dot on its own line.

You had me at EHLO

ESMTP speakers start conversations with EHLO instead of HELO. If the process at the other end understands and responds with an OK, then the participants negotiate supported extensions and agree on a lowest common denominator for the exchange. If the peer returns an error in response to the EHLO, then the ESMTP speaker falls back to SMTP. But today, almost everything uses ESMTP.

A typical SMTP conversation to deliver an email message goes as follows: HELO or EHLO, MAIL FROM:, RCPT TO:, DATA, and QUIT. The sender does most of the talking, with the recipient contributing error codes and acknowledgments.

SMTP and ESMTP are both text-based protocols, so you can use them directly when debugging the mail system. Just **telnet** to TCP port 25 or 587 and start entering SMTP commands. See the example here.

SMTP error codes

Also specified in the RFCs that define SMTP are a set of temporary and permanent error codes. These were originally three-digit codes (e.g., 550), with each digit being interpreted separately. A first digit of 2 indicated success, a 4 signified a temporary error, and a 5 indicated a permanent error.

The three-digit error code system did not scale, so RFC3463 (updated by RFCs 3886, 4468, 4865, 4954, and 5248) restructured it to create more flexibility. It defined an expanded error code format known as a delivery status notification or DSN. DSNs have the format X.X.X instead of the old XXX, and each of the individual Xs can be a multidigit number. The initial X must still be 2, 4, or 5. The second digit specifies a topic, and the third provides the details. The new system uses the second number to distinguish host errors from mailbox errors. Table 18.2 lists a few of the DSN codes. RFC3463's Appendix A shows them all.

Table 18.2: RFC3463 delivery status notifications

Temporary	Permanent	Meaning
4.2.1	5.2.1	Mailbox is disabled
4.2.2	5.2.2	Mailbox is full
4.2.3	5.2.3	Message is too long
4.4.1	5.4.1	No answer from host
4.4.4	5.4.4	Unable to route
4.5.3	5.5.3	Too many recipients
4.7.1	5.7.1	Delivery not authorized, message refused
4.7.*	5.7.*	Site policy violation

SMTP authentication

RFC4954 (updated by RFC5248) defines an extension to the original SMTP protocol that allows an SMTP client to identify and authenticate itself to a mail server. The server might then let the client relay mail through it. The protocol supports several different authentication mechanisms. The exchange is as follows:

1. The client says EHLO, announcing that it speaks ESMTP.
2. The server responds and advertises its authentication mechanisms.
3. The client says AUTH and names a specific mechanism that it wants to use, optionally including its authentication data.
4. The server accepts the data sent with AUTH or starts a challenge and response sequence with the client.
5. The server either accepts or denies the authentication attempt.

To see what authentication mechanisms a server supports, you can **telnet** to port 25 and say EHLO. For example, here is a truncated conversation with the mail server mail-relay.atrust.com (the commands we typed are in bold):

```
$ telnet mail-relay.atrust.com 25
Trying 192.168.2.1...
Connected to mail-relay.atrust.com.
Escape character is '^]'.
220 mail-relay.atrust.com ESMTP AT Mail Service 28.1.2/28.1.2; Mon, 12
    Sep 2016 18:05:55 -0600
ehlo booklab.atrust.com
250-mail-relay.atrust.com Hello [192.168.22.35], pleased to meet you
250-ENHANCEDSTATUSCODES
250-PIPELINING
250-8BITMIME
250-SIZE
250-DSN
250-ETRN
250-AUTH LOGIN PLAIN
250-DELIVERBY
250 HELP
```

In this case, the mail server supports the LOGIN and PLAIN authentication mechanisms. **sendmail**, Exim, and Postfix all support SMTP authentication; details of configuration are covered here, here, and here, respectively.

18.4 SPAM AND MALWARE

Spam is the jargon word for junk mail, also known as unsolicited commercial email or UCE. It is one of the most universally hated aspects of the Internet. Once upon a time, system administrators spent many hours each week hand-tuning block lists and adjusting decision weights in home-grown spam filtering tools. Unfortunately, spammers have become so crafty and commercialized that these measures are no longer an effective use of system administrators' time.

In this section we cover the basic antispam features of each MTA. However, there's a certain futility to any attempt to fight spam as a lone vigilante. You should really pay for a cloud-based spam-fighting service (such as McAfee SaaS Email Protection, Google G Suite, or Barracuda) and leave the spam fighting to the professionals who love that stuff. They have better intelligence about the state of the global emailsphere and can react far more quickly to new information than you can.

Spam has become a serious problem because although the absolute response rate is low, the responses per dollar spent is high. (A list of 30 million email addresses costs about $20.) If it didn't work for the spammers, it wouldn't be such a problem. Surveys show that 95%–98% of all mail is spam.

There are even venture-capital-funded companies whose entire mission is to deliver spam less expensively and more efficiently (although they typically call it "marketing email" rather than spam). If you work at or buy services from one of these companies, we're not sure how you sleep at night.

In all cases, advise your users to simply delete the spam they receive. Many spam messages contain instructions that purport to explain how recipients can be removed from the mailing list. If you follow those instructions, however, the spammers may remove you from the current list, but they immediately add you to several other lists with the annotation "reaches a real human who reads the message." Your email address is then worth even more.

Forgeries

Forging email is trivial; many user agents let you fill in the sender's address with anything you want. MTAs can use SMTP authentication between local servers, but that doesn't scale to Internet sizes. Some MTAs add warning headers to outgoing local messages that they think might be forged.

Any user can be impersonated in mail messages. Be careful if email is your organization's authorization vehicle for things like door keys, access cards, and money. The practice of targeting users with forged email is commonly called "phishing." You should warn administrative users of this fact and suggest that if they see suspicious mail that appears to come from a person in authority, they should verify the validity of the message. Caution is doubly appropriate if the message asks that unreasonable privileges be given to an unusual person.

SPF and Sender ID

The best way to fight spam is to stop it at its source. This sounds simple and easy, but in reality it's almost an impossible challenge. The structure of the Internet makes it difficult to track the real source of a message and to verify its authenticity. The community needs a sure-fire way to verify that the entity sending an email is actually who or what it claims to be. Many proposals have addressed this problem, but SPF and Sender ID have achieved the most traction.

SPF, or Sender Policy Framework, has been described by the IETF in RFC7208. SPF defines a set of DNS records through which an organization can identify its official outbound mail servers. MTAs can then refuse email purporting to be from that organization's domain if the email does not originate from one of these official sources. Of course, the system only works well if the majority of organizations publish SPF records.

Sender ID and SPF are virtually identical in form and function. However, key parts of Sender ID are patented by Microsoft, and hence it has been the subject of much controversy. As of this writing (2017), Microsoft is still trying to strong-arm the industry into adopting its proprietary standards. The IETF chose not to choose and published RFC4406 on Sender ID and RFC7208 on SPF. Organizations that implement this type of spam avoidance strategy typically use SPF.

Messages that are relayed break both SPF and Sender ID, which is a serious flaw in these systems. The receiver consults the SPF record for the original sender to discover its list of authorized servers. However, those addresses won't match any relay machines that were involved in transporting the message. Be careful what decisions you make in response to SPF failures.

DKIM

DKIM (DomainKeys Identified Mail) is a cryptographic signature system for email messages. It lets the receiver verify not only the sender's identity but also the fact that a message has not been tampered with in transit. The system uses DNS records to publish a domain's cryptographic keys and message-signing policy. DKIM is supported by all the MTAs described in this chapter, but real-world deployment has been extremely rare.

18.5 Message privacy and encryption

By default, all mail is sent unencrypted. Educate your users that they should never send sensitive data through email unless they make use of an external encryption package or your organization has provided a centralized encryption solution for email. Even *with* encryption, electronic communication can never be guaranteed to be 100% secure. You pays your money and you takes your chances.

Historically, the most common external encryption packages have been Pretty Good Privacy (PGP), its GNUified clone GPG, and S/MIME. Both S/MIME and PGP are documented in the RFC series, with S/MIME being on the standards track. Most common user agents support plug-ins for both solutions.

These standards offer a basis for email confidentiality, authentication, message integrity assurance, and nonrepudiation of origin. But although PGP/GPG and S/MIME are potentially viable solutions for tech-savvy users who care about privacy, they have proved too cumbersome for unsophisticated users. Both require some facility with cryptographic key management and an understanding of the underlying encryption strategy. (Pro tip: If you use PGP/GPG or S/MIME, you can increase your odds of remaining secure by ensuring that your public key or certificate is expired and replaced frequently. Long-term use of a key increases the likelihood that it will be compromised without your awareness.)

Most organizations that handle sensitive data in email (especially ones that communicate with the public, such as health care institutions) opt for a centralized service that uses proprietary technology to encrypt messages. Such systems can use either on-premises solutions (such as Cisco's IronPort) that you deploy in your data center or cloud-based services (such as Zix, zixcorp.com) that can be configured to encrypt outbound messages according to their contents or other rules. Centralized email encryption is one category of service for which it's best to use a commercial solution rather than rolling your own.

At least in the email realm, data loss prevention (DLP) is a kissing cousin to centralized encryption. DLP systems seek to avoid—or at least, detect—the leakage of proprietary information into the stream of email leaving your organization. They scan outbound email for potentially sensitive content. Suspicious messages can be flagged, blocked, or returned to their senders. Our recommendation is that you choose a centralized encryption platform that also includes DLP capability; it's one less platform to manage.

See this page for more information about TLS.

In addition to encrypting transport between MTAs, it's important to ensure that user-agent-to-access-agent communication is always encrypted, especially because this channel typically employs some form of user credentials to connect. Make sure that only the secure, TLS-using

versions of the IMAP and POP protocols are allowed by access agents. (These are known as IMAPS and POP3S, respectively.)

18.6 MAIL ALIASES

Another concept that is common to all MTAs is the use of aliases. Aliases allow mail to be rerouted either by the system administrator or by individual users.

Aliases can define mailing lists, forward mail among machines, or allow users to be referred to by more than one name. Alias processing is recursive, so it's legal for an alias to point to other destinations that are themselves aliases.

Technically, aliases are configured only by sysadmins. A user's control of mail routing through the use of a **.forward** file is not really aliasing, but we have lumped them together here.

Sysadmins often use role or functional aliases (e.g., printers@example.com) to route email about a particular issue to whatever person is currently handling that issue. Other examples might include an alias that receives the results of a nightly security scan or an alias for the postmaster in charge of email.

The most common method for configuring aliases is to use a simple flat file such as the **/etc/mail/aliases** file discussed later in this section. This method was originally introduced by **sendmail**, but Exim and Postfix support it, too.

Most user agents also provide some sort of "aliasing" feature (usually called "my groups," "my mailing lists," or something of that nature). However, the user agent expands such aliases before mail ever reaches an MSA or MTA. These aliases are internal to the user agent and don't require support from the rest of the mail system.

Aliases can also be defined in a forwarding file in the home directory of each user, usually **~/.forward**. These aliases, which use a slightly nonstandard syntax, apply to all mail delivered to that particular user. They're often used to forward mail to a different account or to implement automatic "I'm on vacation" responses.

MTAs look for aliases in the global **aliases** file (**/etc/mail/aliases** or **/etc/aliases**) and then in recipients' forwarding files. Aliasing is applied only to messages that the transport agent considers to be local.

The format of an entry in the **aliases** file is

```
local-name: recipient1,recipient2,...
```

where *local-name* is the original address to be matched against incoming messages and the recipient list contains either recipient addresses or the names of other aliases. Indented lines are considered continuations of the preceding lines.

From mail's point of view, the **aliases** file supersedes **/etc/passwd**, so the entry

```
david: david@somewhere-else.edu
```

would prevent the local user david from ever receiving any mail. Therefore, administrators and **adduser** tools should check both the **passwd** file and the **aliases** file when selecting new usernames.

The **aliases** file should always contain an alias named “postmaster” that forwards mail to whoever maintains the mail system. Similarly, an alias for “abuse” is appropriate in case someone outside your organization needs to contact you regarding spam or suspicious network behavior that originates at your site. An alias for automatic messages from the MTA must also be present; it’s usually called Mailer-Daemon and is often aliased to postmaster.

Sadly, the mail system is so commonly abused these days that some sites configure their standard contact addresses to throw mail away instead of forwarding it to a human user. Entries such as

```
# Basic system aliases -- these MUST be present
mailer-daemon:  postmaster
postmaster:     "/dev/null"
```

are common. We don’t recommend this practice, because humans who are having trouble reaching your site by email do sometimes write to the postmaster address.

A better paradigm might be

```
# Basic system aliases -- these MUST be present
mailer-daemon:  "/dev/null"
postmaster:     root
```

You should redirect root’s mail to your site’s sysadmins or to someone who logs in every day. The bin, sys, daemon, nobody, and hostmaster accounts (and any other site-specific pseudo-user accounts you set up) should all have similar aliases.

In addition to a list of users, aliases can refer to

- A file containing a list of addresses
- A file to which messages should be appended
- A command to which messages should be given as input

These last two targets should push your “What about security?” button, because the sender of a message totally determines its content. Being able to append that content to a file or deliver it as input to a command sounds pretty scary. Many MTAs either disallow these alias targets or severely limit the commands and file permissions that are acceptable.

Aliases can cause mail loops. MTAs try to detect loops that would cause mail to be forwarded back and forth forever and return the errant messages to the sender. To determine when mail is

looping, an MTA can count the number of Received lines in a message's header and stop forwarding it when the count reaches a preset limit (usually 25). Each visit to a new machine is called a "hop" in email jargon; returning a message to the sender is known as "bouncing" it. So a more typically jargonized summary of loop handling would be, "Mail bounces after 25 hops."

Another way MTAs can detect mail loops is by adding a Delivered-To header for each host to which a message is forwarded. If an MTA finds itself wanting to send a message to a host that's already mentioned in a Delivered-To header, it knows the message has traveled in a loop.

In this chapter, we sometimes call a returned message a "bounce" and sometimes call it an "error." What we really mean is that a delivery status notification (DSN, a specially formatted email message) has been generated. Such a notification usually means that a message was undeliverable and is therefore being returned to the sender.

Getting aliases from files

The :include: directive in the **aliases** file (or a user's **.forward** file) allows the list of targets for the alias to be taken from the specified file. It is a great way to let users manage their own local mailing lists. The included file can be owned by the user and changed without involving a system administrator. However, such an alias can also become a tasty and effective spam expander, so don't let email from outside your site be directed there.

When setting up a list to use :include:, the sysadmin must enter the alias into the global **aliases** file, create the included file, and **chown** the included file to the user that is maintaining the mailing list. For example, the **aliases** file might contain

```
sa-book: :include:/usr/local/mail/ulsah.authors
```

The file **ulsah.authors** should be on a local filesystem and should be writable only by its owner. To be complete, we should also include aliases for the mailing list's owner so that errors (bounces) are sent to the owner of the list and not to the sender of a message addressed to the list:

```
owner-sa-book: evi
```

Mailing to files

If the target of an alias is an absolute pathname, messages are appended to the specified file. The file must already exist. For example:

```
cron-status: /usr/local/admin/cron-status-messages
```

If the pathname includes special characters, it must be enclosed in double quotes.

It's useful to be able to send mail to files, but this feature arouses the interest of the security police and is therefore restricted. This syntax is only valid in the **aliases** file and in a user's **.forward** file (or in a file that's interpolated into one of these files with the :include: directive). A filename is not understood as a normal address, so mail addressed to /etc/passwd@example.com would bounce.

If the destination file is referenced from the **aliases** file, it must be world-writable (not advisable), setuid but not executable, or owned by the MTA's default user. The identity of the default user is set in the MTA's configuration file.

If the file is referenced in a **.forward** file, it must be owned and writable by the original message recipient, who must be a valid user with an entry in the **passwd** file and a valid shell that's listed in **/etc/shells**. For files owned by root, use mode 4644 or 4600, setuid but not executable.

Mailing to programs

An alias can also route mail to the standard input of a program. This behavior is specified with a line such as

```
autolog: "|/usr/local/bin/autologger"
```

It’s even easier to create security holes with this feature than with mailing to a file, so once again it is only permitted in **aliases**, **.forward**, or :include: files, and often requires the use of a restricted shell.

Building the hashed alias database

Since entries in the **aliases** file are unordered, it would be inefficient for the MTA to search this file directly. Instead, a hashed version is constructed with the Berkeley DB system. Hashing significantly speeds alias lookups, especially when the file gets large.

The file derived from **/etc/mail/aliases** is called **aliases.db**. If you are running Postfix or **sendmail**, you must rebuild the hashed database with the **newaliases** command every time you change the **aliases** file. Exim detects changes to the **aliases** file automatically. Save the error output if you run **newaliases** automatically—you might have introduced formatting errors in the **aliases** file.

18.7 EMAIL CONFIGURATION

The heart of an email system is its MTA, or mail transport agent. **sendmail** is the original UNIX MTA, written by Eric Allman while he was a graduate student many years ago. Since then, a host of other MTAs have been developed. Some of them are commercial products and some are open source implementations. In this chapter, we cover three open source mail-transport agents: **sendmail**, Postfix by Wietse Venema of IBM Research, and Exim by Philip Hazel of the University of Cambridge.

Configuration of the MTA can be a significant sysadmin chore. Fortunately, the default or sample configurations that ship with MTAs are often close to what the average site needs. You need not start from scratch when configuring your MTA.

SecuritySpace (securityspace.com) does a survey monthly to determine the market share of the various MTAs. In their June 2017 survey, 1.7 million out of 2 million MTAs surveyed replied with a banner that identified the MTA software in use. Table 18.3 shows these results, as well as the SecuritySpace results for 2009 and some 2001 values from a different survey.

Table 18.3: Mail transport agent market share

			Market share		
MTA	**Source**	**Default MTA on**	**2017**	**2009**	**2001**
Exim	exim.org	Debian	56%	30%	8%
Postfix	postfix.org	Red Hat, Ubuntu	33%	20%	2%
Exchange	microsoft.com/exchange	–	1%	20%	4%
sendmail	sendmail.org	FreeBSD	5%	19%	60%
All others	–	–	<3% ea	<3% ea	< 3% ea

The trend is clearly away from **sendmail** and toward Exim and Postfix, with Microsoft dropping to almost nothing. Keep in mind that this data includes only MTAs that are directly exposed to the Internet.

For each of the MTAs we cover, we include details on the common areas of interest:

- Configuration of simple clients
- Configuration of an Internet-facing mail server
- Control of both inbound and outbound mail routing
- Stamping of mail as coming from a central server or the domain itself
- Security

- Debugging

If you are implementing a mail system from scratch and have no site politics or biases to deal with, you may find it hard to choose an MTA. **sendmail** is largely out of vogue, with the possible exception of pure FreeBSD sites. Exim is powerful and highly configurable but suffers in complexity. Postfix is simpler, faster, and was designed with security as a primary goal. If your site or your sysadmins have a history with a particular MTA, it's probably not worth switching unless you need features that are not available from your old MTA.

sendmail configuration is covered in the next section. Exim configuration begins here, and Postfix configuration here.

18.8 SENDMAIL

The **sendmail** distribution is available in source form from sendmail.org, but it's rarely necessary to build **sendmail** from scratch these days. If you must do so, refer to the top-level **INSTALL** file for instructions. To tweak some of the build defaults, look up **sendmail**'s assumptions in **devtools/OS/***your-OS-name*. Add features by editing **devtools/Site/site.config.m4**. As of October 2013, **sendmail** is supported and distributed by Proofpoint, Inc., a public company.

sendmail uses the **m4** macro preprocessor not only for compilation but also for configuration. An **m4** configuration file is usually named *hostname***.mc** and is then translated from a slightly user-friendly syntax into a totally inscrutable low-level language in the file *hostname***.cf**, which is in turn installed as **/etc/mail/sendmail.cf**.

To see what version of **sendmail** is installed on your system and how it was compiled, try the following command:

```
linux$ /usr/sbin/sendmail -d0.1 -bt < /dev/null
Version 8.13.8
 Compiled with: DNSMAP HESIOD HES_GETMAILHOST LDAPMAP LOG
    MAP_REGEX MATCHGECOS MILTER MIME7TO8 MIME8TO7 NAMED_BIND
    NETINET NETINET6 NETUNIX NEWDB NIS PIPELINING SASLv2 SCANF
    SOCKETMAP STARTTLS TCPWRAPPERS USERDB USE_LDAP_INIT
============ SYSTEM IDENTITY (after readcf) ============
      (short domain name) $w = ross
  (canonical domain name) $j = ross.atrust.com
         (subdomain name) $m = atrust.com
              (node name) $k = ross.atrust.com
========================================================
```

This command puts **sendmail** in address test mode (**-bt**) and debug mode (**-d0.1**) but gives it no addresses to test (**</dev/null**). A side effect is that **sendmail** tells us its version and the compiler flags it was built with. Once you know the version number, you can look at the sendmail.org web site to see if any known security vulnerabilities are associated with that release.

To find the **sendmail** files on your system, look at the beginning of the installed **/etc/mail/sendmail.cf** file. The comments there mention the directory in which the configuration was built. That directory should in turn lead you to the **.mc** file that is the original source of the configuration.

Most vendors that ship **sendmail** include not only the binary but also the **cf** directory from the distribution tree, which they hide somewhere among the operating system files. Table 18.4 will help you find it.

Table 18.4: Config directory locations

System	Directory
Ubuntu	**/usr/share/sendmail**
Debian	**/usr/share/sendmail**
Red Hat	**/etc/mail**
CentOS	**/etc/mail**
FreeBSD	**/etc/mail**

The switch file

The service switch is covered in more detail starting here.

Most systems have a “service switch” configuration file, **/etc/nsswitch.conf**, that enumerates the methods that can satisfy various standard queries such as user and host lookups. If more than one resolution method is listed for a given type of query, the service switch file also determines the order in which the various methods are consulted.

The existence of the service switch is normally transparent to software. However, **sendmail** likes to exert fine-grained control over its lookups, so it currently ignores the system switch file and instead uses its own internal service configuration file (**/etc/mail/service.switch**).

Two fields in the switch file impact the mail system: aliases and hosts. The possible values for the hosts service are dns, nis, nisplus, and files. For aliases, the possible values are files, nis, nisplus, and ldap. Support for the mechanisms you use (except files) must be compiled into **sendmail** before the service can be used.

Starting sendmail

sendmail should not be controlled by **inetd** or **systemd**, so it must be explicitly started at boot time. See Chapter 2, *Booting and System Management Daemons*, for startup details.

The flags that **sendmail** is started with determine its behavior. You can run it in several different modes, selected with the **-b** flag. **-b** stands for "be" or "become" and is always used with another flag that determines the role **sendmail** will play. Table 18.5 lists the legal values and also includes the **-A** flag, which selects between MTA and MSA behavior.

Table 18.5: Command-line flags for sendmail's major modes

Flag	Meaning
-Ac	Uses the **submit.cf** config file and acts as an MSA
-Am	Uses the **sendmail.cf** config file and acts as an MTA
-ba	Runs in ARPANET mode (expects CR/LF at the ends of lines)
-bd	Runs in daemon mode and listens for connections on port 25
-bD	Runs in daemon mode, but in the foreground rather than the background[a]
-bh	Views recent connection info (same as **hoststat**)
-bH	Purges disk copy of outdated connection info (same as **purgestat**)
-bi	Initializes hashed aliases (same as **newaliases**)
-bm	Runs as a mailer, delivers mail in the usual way (default)
-bp	Prints the mail queue (same as **mailq**)
-bP	Prints the number of entries in queues via shared memory
-bs	Enters SMTP server mode (on standard input, not port 25)
-bt	Enters address test mode
-bv	Verifies mail addresses only; doesn't send mail

a. Use this mode for debugging so you can see the error and debugging messages.

If you are configuring a server that will accept incoming mail from the Internet, run **sendmail** in daemon mode (**-bd**). In this mode, **sendmail** listens on network port 25 and waits for work. (The ports that **sendmail** listens on are determined by DAEMON_OPTIONS; port 25 is the default.)

You will usually specify the **-q** flag, too—it sets the interval at which **sendmail** processes the mail queue. For example, **-q30m** runs the queue every thirty minutes and **-q1h** runs it every hour.

sendmail normally tries to deliver messages immediately, saving them in the queue only momentarily to guarantee reliability. But if your host is too busy or the destination machine is unreachable, **sendmail** queues messages and tries to send them again later. **sendmail** uses persistent queue runners that are usually started at boot time. It does locking, so multiple, simultaneous queue runs are safe. You can use the "queue groups" configuration feature to facilitate delivery of large mailing lists and queues.

sendmail reads its configuration file, **sendmail.cf**, only when it starts up. Therefore, you must either kill and restart **sendmail** or send it a HUP signal when you change the config file. **sendmail** creates a **sendmail.pid** file that contains its process ID and the command that started it. You should start **sendmail** with an absolute path because it re-**exec**s itself on receipt of the HUP signal. The **sendmail.pid** file allows the process to be HUPed with the command

```
$ sudo kill -HUP `head -1 sendmail.pid`
```

The location of the PID file is OS dependent. It's usually **/var/run/sendmail.pid** or **/etc/mail/sendmail.pid** but can be set in the config file with the confPID_FILE option:

```
define(confPID_FILE, '/var/run/sendmail.pid')
```

Mail queues

sendmail uses at least two queues: **/var/spool/mqueue** when acting as an MTA on port 25, and **/var/spool/clientmqueue** when acting as an MSA on port 587. **sendmail** can use multiple queues beneath **mqueue** to increase performance. All messages make at least a brief stop in the queue before being sent on their way.

A queued message is saved in pieces in several different files. Table 18.6 shows the six possible pieces. Each filename has a two-letter prefix that identifies the piece, followed by a random ID built from **sendmail**'s process ID.

Table 18.6: Prefixes for files in the mail queue

Prefix	File contents
qf	The message header and control file
df	The body of the message
tf	A temporary version of the **qf** file while the **qf** file is being updated
Tf	A notice that 32 or more failed locking attempts have occurred
Qf	A notice that the message bounced and could not be returned
xf	Temporary transcript file of error messages from mailers

If subdirectories **qf**, **df**, or **xf** exist in a queue directory, then those pieces of the message are put in the proper subdirectory. The **qf** file contains not only the message header but also the envelope addresses, the date at which the message should be returned as undeliverable, the message's priority in the queue, and the reason the message is in the queue. Each line begins with a single-letter code that identifies the rest of the line.

Each message that is queued must have a **qf** and **df** file. All the other prefixes are used by **sendmail** during attempted delivery. When a machine crashes and reboots, the startup sequence for **sendmail** should delete the **tf**, **xf**, and **Tf** files from each queue. If you are the sysadmin responsible for mail, check occasionally for **Qf** files in case local configuration is causing the bounces. An occasional glance at the queue directories lets you spot problems before they become disasters.

The mail queue opens up several opportunities for things to go wrong. For example, the filesystem can fill up (avoid putting **/var/spool/mqueue** and **/var/log** on the same partition), the queue can become clogged, or orphaned mail messages can get stuck in the queue. **sendmail** has configuration options to help with performance on busy machines.

sendmail configuration

sendmail is controlled by a single configuration file, typically called **/etc/mail/sendmail.cf** for a **sendmail** running as an MTA or **/etc/mail/submit.cf** for a **sendmail** acting as an MSA. The flags with which **sendmail** is started determine which config file it uses: **-bm**, **-bs**, and **-bt** use **submit.cf** if it exists, and all other modes use **sendmail.cf**. You can change these names with command-line flags or config file options, but it is best not to.

The raw config file format was designed to be easy to parse by machines, not humans. The **m4** source (**.mc**) file from which the **.cf** file is generated is an improvement, but its picky and rigid syntax isn't going to win any awards for user friendliness either. Fortunately, many of the paradigms you might want to set up have already been hammered out by others with similar needs and are supplied in the distribution as prepackaged features.

sendmail configuration involves several steps:

1. Determine the role of the machine you are configuring: client, server, Internet-facing mail receiver, etc.
2. Choose the features needed to implement that role and build an **.mc** file for the configuration
3. Compile the **.mc** file with **m4** to produce a **.cf** config file

We cover the features commonly used for site-wide, Internet-facing servers and for little desktop clients. For more detailed coverage, we refer you to two key pieces of documentation on the care and feeding of **sendmail**: the O'Reilly book *sendmail* by Bryan Costales et al. and the file **cf/README** from the distribution.

The m4 preprocessor

m4, originally intended as a front end for programming languages, lets users write more readable (or perhaps more cryptic) programs. **m4** is powerful enough to be useful in many input transformation situations, and it works nicely for **sendmail** configuration files.

m4 macros have the form

```
name(arg1, arg2, ..., argn)
```

There cannot be any space between the name and the opening parenthesis. Left and right single quotes (that is, backticks and "normal" single quotes) designate strings as arguments. **m4**'s quote conventions are weird, since the left and right quotes are different characters. Quotes nest, too.

m4 has some built-in macros, and users can also define their own. Table 18.7 lists the most common built-in macros that are used in **sendmail** configuration.

Table 18.7: m4 macros commonly used with sendmail

Macro	Function
`define`	Defines a macro named *arg1* with value *arg2*
`divert`	Manages output streams
`dnl`	Discards characters up to and including the next newline
`include`	Includes (interpolates) the file named *arg1*
`undefine`	Discards a previous definition of macro named *arg1*

The sendmail configuration pieces

The **sendmail** distribution includes a **cf** subdirectory beneath which are all the pieces necessary for **m4** configuration. Table 18.4 shows the location of the **cf** directory if you did not install the **sendmail** source but relied on your vendor. The **README** file found in the **cf** directory is **sendmail**'s configuration documentation. The subdirectories, listed in Table 18.8, contain examples and snippets you can include in your own configuration.

Table 18.8: sendmail configuration subdirectories

Directory	Contents
cf	Sample **.mc** (master configuration) files
domain	Sample **m4** files for various domains at Berkeley
feature	Fragments that implement various features
hack	Special features of dubious value or implementation
m4	The basic config file and other core files
mailer	**m4** files that describe common mailers (delivery agents)
ostype	OS-dependent file locations and quirks
sh	Shell scripts used by **m4**

The **cf/cf** directory contains examples of **.mc** files. In fact, it contains so many examples that yours may get lost in the clutter. We recommend that you keep your own **.mc** files separate from those in the distributed **cf** directory. Either create a new directory named for your site (**cf**/*sitename*) or move the **cf** directory aside to **cf.examples** and create a new **cf** directory. If you do this, copy the **Makefile** and **Build** script over to your new directory so the instructions in the **README** file still work. Alternatively, you can copy all your own configuration **.mc** files to a central location rather than leaving them inside the **sendmail** distribution. The **Build** script uses relative pathnames, so you'll have to modify it if you want to build a **.cf** file from an **.mc** file and are not in the **sendmail** distribution hierarchy.

The files in the **cf/ostype** directory configure **sendmail** for each specific operating system. Many are predefined, but if you have moved things around on your system, you might have to modify one or create a new one. Copy one that is close to reality for your system and give it a new name.

The **cf/feature** directory is where you shop for any configuration pieces you might need. There is a feature for just about anything that any site running **sendmail** has found useful.

The other directories beneath **cf** are pretty much boilerplate and do not need to be tweaked or even understood—just use them.

A configuration file built from a sample .mc file

Before we take off into the wilds of the various configuration macros, features, and options you might use in a **sendmail** configuration, we shall put the cart before the horse and devise a "no frills" configuration to illustrate the general process. Our example is for a leaf node, myhost.example.com; the master configuration file is called **myhost.mc**. Here's the complete **.mc** file:

```
divert(-1)
#### basic .mc file for example.com
divert(0)
VERSIONID('$Id$')
OSTYPE('linux')
MAILER('local')
MAILER('smtp')
```

Except for the diversions and comments, each line invokes a prepackaged macro. The first four lines are boilerplate; they insert comments in the compiled file to note the version of **sendmail**, the directory the configuration was built in, etc. The OSTYPE macro includes the **../ostype/linux.m4** file. The MAILER lines allow for local delivery (to users with accounts on myhost.example.com) and for delivery to Internet sites.

To build the real configuration file, just run the **Build** command you copied over to the new **cf** directory:

```
$ ./Build myhost.cf
```

Finally, install **myhost.cf** in the right spot—normally **/etc/mail/sendmail.cf**, but some vendors move it. Favorite vendor hiding places are **/etc** and **/usr/lib**.

At a larger site, you might want to create a separate **m4** file to hold site-wide defaults; put it in the **cf/domain** directory. Individual hosts can then include the contents of this file with the DOMAIN macro. Not every host needs a separate config file, but each group of similar hosts (same architecture and same role: server, client, etc.) will probably need its own configuration.

The order of the macros in the **.mc** file is not arbitrary. It should be

```
VERSIONID
OSTYPE
DOMAIN
FEATURE
local macro definitions
MAILER
```

Even with **sendmail**'s easy **m4** configuration system, you still have to make several configuration decisions for your site. As you read about the features described below, think about

how they might fit into your site's organization. A small site will probably have only a hub node and leaf nodes and thus will need only two versions of the config file. A larger site might need separate hubs for incoming and outgoing mail and, perhaps, a separate POP/IMAP server.

Whatever the complexity of your site and whatever face it shows to the outside world (exposed, behind a firewall, or on a virtual private network, for example), it's likely that the **cf** directory contains some appropriate ready-made configuration snippets just waiting to be customized and put to work.

Configuration primitives

sendmail configuration commands are case sensitive. By convention, the names of predefined macros are all caps (e.g., OSTYPE), **m4** commands are all lower case (e.g., define), and configurable option names usually start with lowercase conf and end with an all-caps variable name (e.g., confFAST_SPLIT). Macros usually refer to an **m4** file called **../***macroname***/***arg1***.m4**. For example, the reference OSTYPE(`linux') causes the file **../ostype/linux.m4** to be included.

Tables and databases

Before we plunge into specific configuration primitives, we must first discuss tables (sometimes called maps or databases), which **sendmail** can use to perform mail routing or address rewriting. Most are used in conjunction with the FEATURE macro.

A table is a cache (usually a text file) of routing, aliasing, policy, or other information that is converted to a database format with the **makemap** command and then used as an information source for one or more of **sendmail**'s various lookup operations. Although the data usually starts as a text file, data for **sendmail** tables can come from DNS, LDAP, or other sources. The use of a centralized IMAP server relieves **sendmail** of the chore of chasing down users and obsoletes some of its tables.

sendmail defines three database map types:

- dbm – legacy; uses an extensible hashing algorithm (**dbm**/**ndbm**)
- hash – uses a standard hashing scheme (DB)
- btree – uses a B-tree data structure (DB)

For most table applications in **sendmail**, the hash database type—the default—is the best. Use the **makemap** command to build the database file from a text file; you specify the database type and the output file base name. The text version of the database should appear on **makemap**'s standard input. For example:

```
$ sudo makemap hash /etc/mail/access < /etc/mail/access
```

At first glance this command looks like a mistake that would cause the input file to be overwritten by an empty output file. However, **makemap** tacks on an appropriate suffix, so the actual output file is **/etc/mail/access.db** and in fact no conflict occurs. Each time the text file is changed, the database file must be rebuilt with **makemap** (but **sendmail** need not be HUP'd).

Comments can appear in the text files from which maps are produced. They begin with # and continue until the end of the line.

In most circumstances, the longest possible match is used for database keys. As with any hashed data structure, the order of entries in the input text file is not significant. Some FEATUREs expect a database file as a parameter; they default to hash as the database type and **/etc/mail/***tablename***.db** as the filename for the database.

Generic macros and features

Table 18.9 lists common configuration primitives, whether they are typically used (yes, no, maybe), and a brief description of what they do.

Table 18.9: sendmail generic configuration primitives

Primitive	Used?[a]	Description
OSTYPE	Yes	Includes OS-specific paths and mailer flags
DOMAIN	No	Includes site-specific configuration details
MAILER	Yes	Enables mailers, typically `smtp` and `local`
FEATURE	Maybe	Enables a variety of **sendmail** features
use_cw_file	Yes (S)	Lists hosts for which you accept mail
redirect	Maybe (S)	Bounces mail nicely when users move
always_add_domain	Yes	Fully qualifies hostnames if UA didn't
access_db	Maybe (S)	Sets database of hosts to relay mail for
virtusertable	Maybe (S)	Turns on domain aliasing (virtual domains)
ldap_routing	Maybe (S)	Routes incoming mail using LDAP
MASQUERADE_AS	Yes	Makes all mail seem to come from one place
EXPOSED_USER	Yes	Lists users who shouldn't be masqueraded
MAIL_HUB	Yes (S)	Specifies mail server for incoming mail
SMART_HOST	Yes (C)	Specifies mail server for outgoing mail

a. S = servers, C = clients

OSTYPE macro

An OSTYPE file packages a variety of vendor-specific information, such as the expected locations of mail-related files, paths to commands that **sendmail** needs, flags to mailer programs, etc. See **cf/README** for a list of all the variables that can be defined in an OSTYPE file.

So where is the OSTYPE macro itself defined? In a file in the **cf/m4** directory, which is magically prepended to your config file when you run the **Build** script.

DOMAIN macro

The DOMAIN directive lets you specify site-wide generic information in one place (**cf/domain/***filename***.m4**) and then include it in each host's config file with

```
DOMAIN(`filename')
```

MAILER macro

You must include a MAILER macro for every delivery agent you want to enable. You'll find a complete list of supported mailers in the directory **cf/mailers**, but typically you need only local and smtp. MAILER lines are generally the last thing in the **.mc** file.

FEATURE macro

The FEATURE macro enables a whole host of common scenarios (56 at last count!) by including **m4** files from the **feature** directory. The syntax is

```
FEATURE(keyword, arg, arg, ...)
```

where *keyword* corresponds to a file *keyword***.m4** in the **cf/feature** directory and the *args* are passed to it. There can be at most nine arguments to a feature.

use_cw_file feature

The **sendmail** internal class w (hence the name **cw**) contains the names of all local hosts for which this host accepts and delivers mail. This feature specifies that mail be accepted for the hosts listed, one per line, in **/etc/mail/local-host-names**. The configuration line

```
FEATURE('use_cw_file')
```

invokes the feature. A client machine does not really need this feature unless it has nicknames, but your incoming mail hub machine does. The **local-host-names** file should include any local hosts and virtual domains for which you accept email, including sites whose backup MX records (see this page) point to you.

Without this feature, **sendmail** delivers mail locally only if it is addressed to the machine on which **sendmail** is running.

If you add a new host at your site, you must add it to the **local-host-names** file and send a HUP signal to **sendmail** to make your changes take effect.

redirect feature

When people leave your organization, you usually either forward their mail or let mail to them bounce back to the sender with an error. The redirect feature provides support for a more elegant way of bouncing mail.

If Joe Smith has graduated from oldsite.edu (login smithj) to newsite.com (login joe), then enabling redirect with

```
FEATURE('redirect')
```

and adding the line

```
smithj: joe@newsite.com.REDIRECT
```

to the **aliases** file at oldsite.edu causes mail to smithj to be returned to the sender with an error message suggesting that the sender try the address joe@newsite.com instead. The message itself is not automatically forwarded.

always_add_domain feature

The always_add_domain feature makes all email addresses fully qualified. It should always be used.

access_db feature

The access_db feature controls relaying and other policy issues. Typically, the raw data that drives this feature either comes from LDAP or is kept in a text file called **/etc/mail/access**. In the latter case, the text file must be converted to some kind of indexed format with the **makemap** command, as described here. To use the flat file, use FEATURE(`access_db') in the configuration file; for the LDAP version, use FEATURE(`access_db', `LDAP'). The LDAP form uses the default schema defined in the file **cf/sendmail.schema**; if you want a different schema file, use additional arguments in your FEATURE statement.

The key field in the access database is an IP network or a domain name with an optional tag such as Connect:, To:, or From:. The value field specifies what to do with the message.

The most common values are OK to accept the message, RELAY to allow it to be relayed, REJECT to reject it with a generic error indication, or ERROR:"*error code and message*" to reject it with a specific message. Other possible values allow for finer-grained control. Here is a snippet from a sample **/etc/mail/access** file:

```
localhost       RELAY
127.0.0.1       RELAY
192.168.1.1     RELAY
192.168.1.17    RELAY
66.77.123.1     OK
fax.com         OK
61              ERROR:"550 We don't accept mail from spammers"
67.106.63       ERROR:"550 We don't accept mail from spammers"
```

virtusertable feature

The virtusertable feature supports domain aliasing for incoming mail through a map stored in **/etc/mail/virtusertable**. This feature lets one machine host multiple virtual domains and is used frequently at web-hosting sites. The key field of the table contains either an email address (*user@host.domain*) or a domain specification (*@domain*). The value field is a local or external email address. If the key is a domain, the value can either pass the *user* field along as the variable %1 or route the mail to a different user. Here are some examples:

```
@appliedtrust.com        %1@atrust.com
unix@book.admin.com      sa-book-authors@atrust.com
linux@book.admin.com     sa-book-authors@atrust.com
webmaster@example.com    billy.q.zakowski@colorado.edu
info@testdomain.net      ausername@hotmail.com
```

All the host keys on the left side of the data mappings must be listed in the **cw** file, **/etc/mail/local-host-names**, or be included in the VIRTUSER_DOMAIN list. If they are not, **sendmail** will not know to accept the mail locally and will try to find the destination host on the Internet. But DNS MX records will point **sendmail** back to this same server and you will get a "local configuration error" message in the resulting bounce message. Unfortunately, **sendmail** cannot tell that the error message for this instance should in fact be "virtusertable key not in cw file."

ldap_routing feature

LDAP, the Lightweight Directory Access Protocol, can be a source of data for aliases or mail routing information as well as general tabular data as described earlier. The **cf/README** file has a long section on LDAP with lots of examples.

See this page for general information about LDAP.

To use LDAP in this way, you must have built **sendmail** to include LDAP support. In your **.mc** file, add the lines

```
define('confLDAP_DEFAULT_SPEC', '-h server -b searchbase')
FEATURE('ldap_routing')
LDAPROUTE_DOMAIN('my_domain')
```

Those lines tell **sendmail** that you want to use an LDAP database to route incoming mail addressed to the specified domain. The LDAP_DEFAULT_SPEC option identifies the LDAP server and the LDAP basename for searches. LDAP uses port 389 unless you specify a different port by adding -p *ldap_port* to the define.

sendmail uses the values of two tags in the LDAP database:

- mailLocalAddress for the addressee on incoming mail
- mailRoutingAddress for the destination to which email should be sent

sendmail also supports the tag mailHost, which if present routes mail to the MX-designated mail handler for the specified host. The recipient address remains the value of the mailRoutingAddress tag.

LDAP database entries support a wild card entry, *@domain*, that reroutes mail addressed to anyone at the specified domain (as was done in the virtusertable).

By default, mail addressed to user@host1.mydomain would first trigger a lookup on user@host1.mydomain. If that failed, **sendmail** would try @host1.mydomain but not user@mydomain. Including the line

```
LDAPROUTE_EQUIVALENT('host1.mydomain')
```

would also try the keys user@mydomain and @mydomain. This feature enables a single database to route mail at a complex site. You can also take the entries for the LDAPROUTE_EQUIVALENT clauses from a file, which makes the feature quite usable. The syntax for that form is

```
LDAPROUTE_EQUIVALENT_FILE('filename')
```

Additional arguments to the ldap_routing feature let you specify more details about the LDAP schema and control the handling of addressee names that have a *+detail* part. As always, see the **cf/README** file for exact details.

Masquerading features

An email address is usually made up of a username, a host, and a domain, but many sites do not want the names of their internal hosts exposed on the Internet. The MASQUERADE_AS macro lets you specify a single identity for other machines to hide behind. All mail appears to emanate from the designated machine or domain. This is fine for regular users, but for debugging purposes, system users such as root should be excluded from the masquerade.

For example, the sequence

```
MASQUERADE_AS('atrust.com')
EXPOSED_USER('root')
EXPOSED_USER('Mailer-Daemon')
```

would stamp mail as coming from user@atrust.com unless it was sent by root or the mail system; in these cases, the mail would carry the name of the originating host.

MASQUERADE_AS is actually just the tip of a vast masquerading iceberg that extends downward through a dozen variations and exceptions. The allmasquerade and masquerade_envelope features (in combination with MASQUERADE_AS) hide just the right amount of local info. See the **cf/README** for details.

MAIL_HUB and SMART_HOST macros

Masquerading makes all mail *appear* to come from a single host or domain by rewriting the headers and, optionally, the envelope. But most sites want all mail to *actually* come from (or go to) a single machine so that they can control the flow of viruses, spam, and company secrets. You can achieve this control with a combination of MX records in DNS, the MAIL_HUB macro for incoming mail, and the SMART_HOST macro for outgoing mail.

See this page for more information about DNS MX records.

For example, in a structured email implementation, MX records would direct incoming email from the Internet to an MTA in the network's demilitarized zone. After verification that the received email was free of viruses and spam and was directed to valid local users, the mail could be relayed, with the following define, to the internal routing MTA for delivery:

```
define('MAIL_HUB', 'smtp:routingMTA.mydomain')
```

See the next section for more about nullclient.

Likewise, client machines would relay their mail to the SMART_HOST designated in the nullclient feature in their configuration. The SMART_HOST could then filter for viruses and spam so that mail from your site did not pollute the Internet.

The syntax of SMART_HOST parallels that of MAIL_HUB, and the default delivery agent is again relay. For example:

```
define('SMART_HOST', 'smtp:outgoingMTA.mydomain')
```

You can use the same machine as the server for both incoming and outgoing mail. Both the SMART_HOST and the MAIL_HUB must allow relaying, the first from clients inside your domain and the second from the MTA in the DMZ.

Client configuration

Most of your site's machines should be configured as clients who just submit outgoing mail generated by users and don't receive mail at all. One of **sendmail's** FEATURES, nullclient, is just right for this situation. It creates a config file that forwards all mail to a central hub over SMTP. The entire config file, after the VERSIONID and OSTYPE lines, would be simply

```
FEATURE(`nocanonify')
FEATURE(`nullclient', `mailserver')
EXPOSED_USER(`root')
```

where *mailserver* is the name of your central hub. The nocanonify feature tells **sendmail** not to do DNS lookups or rewrite addresses with fully qualified domain names. All that work will be done by the *mailserver* host. This feature is similar to SMART_HOST and assumes that the client will MASQUERADE_AS *mailserver*. The EXPOSED_USER clause exempts root from the masquerading and so facilitates debugging.

The *mailserver* machine must allow relaying from its null clients. That permission is granted in the access_db, described here. The null client must have an associated MX record that points to *mailserver* and must also be included in the *mailserver*'s **cw** file (usually **/etc/mail/local-host-names**). These settings allow the *mailserver* to accept mail for the client.

sendmail should run as an MSA (without the **-bd** flag) if the user agents on the client machine can be taught to use port 587 for submitting mail. If not, you can run **sendmail** in daemon mode (**-bd**), but set the DAEMON_OPTIONS configuration option to listen for connections only on the loopback interface.

m4 configuration options

You set config file options with the **m4** define command. A complete list of options that are accessible as **m4** variables (along with their default values) is given in the **cf/README** file.

The defaults are OK for a typical site that is not too paranoid about security and not too concerned with performance. The defaults try to protect you from spam by turning off relaying, by requiring addresses to be fully qualified, and by requiring that senders' domains resolve to an IP address. If your mail hub machine is busy and services a lot of mailing lists, you might need to tweak some of the performance values.

Table 18.10 lists some options that you might need to adjust (about 10% of over 175 configuration options). Their default values are shown in parentheses. To save space, the option names are shown without their conf prefix; for example, the FAST_SPLIT option is actually named confFAST_SPLIT. We divided the table into subsections that identify the kind of issue the variable addresses: resource management, performance, security and spam abatement, and miscellaneous options. Some options fit in more than one category, but we have listed them only once.

Table 18.10: Basic sendmail configuration options

	Option name	Description (default value)
Resources	MAX_DAEMON_CHILDREN	Max number of child processes[Table 18.10] (no limit)
	MAX_MESSAGE_SIZE	Max size in bytes of a single message (infinite)
	MIN_FREE_BLOCKS	Min filesystem space to accept mail (100)
	TO_*lots_of_stuff*	Timeouts for all kinds of things (various)
Performance	DELAY_LA	Load avg. to slow deliveries (0 = no limit)
	FAST_SPLIT	Suppresses MX lookups as recipients are sorted and split across queues (1 = true)
	MCI_CACHE_SIZE	# of open outgoing TCP connections cached (2)
	MCI_CACHE_TIMEOUT	Time to keep cached connections open (5m)
	MIN_QUEUE_AGE	Minimum time jobs must stay in queue (0)
	QUEUE_LA	Load average at which mail should be queued instead of delivered immediately (8 * #CPUs)
	REFUSE_LA	Load avg. at which to refuse mail (12 * #CPUs)
Security and spam	AUTH_MECHANISMS	SMTP auth mechanisms[Table 18.10]
	CONNECTION_RATE_THROTTLE	Limits connection acceptance rate (no limit)
	DONT_BLAME_SENDMAIL	Overrides security and file checking (safe)[c]
	MAX_MIME_HEADER_LENGTH	Sets max size of MIME headers (no limit)[d]
	MAX_RCPTS_PER_MESSAGE	Slows spam delivery; defers extra recipients and sends a temporary error msg (infinite)
	PRIVACY_FLAGS	Limits info given out by SMTP (authwarnings)
Misc	DOUBLE_BOUNCE_ADDRESS	Catches lots of spam; some sites use **/dev/null**, which can hide serious problems (postmaster)
	LDAP_DEFAULT_SPEC	Map spec for LDAP database, including the host/port the server is running on (undefined)

a. More specifically, the maximum number of child processes that can run at once. When the limit is reached, **sendmail** refuses connections. This option can prevent (or create) denial of service attacks.

b. The default value is EXTERNAL GSSAPI KERBEROS_V4 DIGEST-MD5 CRAM-MD5; don't add PLAIN LOGIN, because the password is transmitted as cleartext. That might be OK internally, but not on the Internet unless the connection is also secured through the use of SSL.

c. Don't change this setting casually!

d. This option can prevent user agent buffer overflows. "256/128" is a good value to use—it means 256 bytes per header and 128 bytes per parameter to that header.

Spam-related features in sendmail

sendmail has a variety of features and configuration options that can help you control spam and viruses:

- Rules that control third party (aka promiscuous, aka open) relaying; that is, the use of your mail server by one off-site user to send mail to another off-site user. Spammers often use relaying to mask the true source of their mail and thereby avoid detection by ISPs. Relaying also lets spammers use *your* cycles and save their own.
- The access database for filtering recipient addresses. This feature is rather like a firewall for email.
- Blacklists that catalog open relays and known spam-friendly sites that **sendmail** can check against.
- Throttles that can slow down mail acceptance when certain types of bad behavior are detected.
- Header checking and input mail filtering by means of a generic mail filtering interface called **libmilter**. It allows arbitrary scanning of message headers and content and lets you reject messages that match a particular profile. Milters are plentiful and powerful; see milter.org.

Relay control

sendmail accepts incoming mail, looks at the envelope addresses, decides where the mail should go, and then passes the message along to an appropriate destination. That destination can be local or it can be another transport agent farther along in the delivery chain. When an incoming message has no local recipients, the transport agent that handles it is said to be acting as a relay.

Only hosts that are tagged with RELAY in the access database (see this page) or that are listed in **/etc/mail/relay-domains** are allowed to submit mail for relaying. Some types of relaying are useful and legitimate. How can you tell which messages to relay and which to reject? Relaying is actually necessary in only three situations:

- When the transport agent acts as a gateway for hosts that are not reachable in any other way; for example, hosts that are not always turned on (laptops, Windows PCs) and virtual hosts. In this situation, all the recipients for which you want to relay lie within the same domain.
- When the transport agent is the outgoing mail server for other, not-so-smart hosts. In this case, all the senders' hostnames or IP addresses are local (or at least enumerable).
- When you have agreed to be a backup MX destination for another site.

Any other situation that appears to require relaying is probably just an indication of bad design (with the possible exception of support for mobile users). You can obviate the first use of relaying (above) by designating a centralized server to receive mail, with POP or IMAP being used for client access. The second case should always be allowed, but only for your own hosts. You can check IP addresses or hostnames. In the third case, you can list the other site in your access database and allow relaying just for that site's IP address blocks.

Although **sendmail** comes with relaying turned off by default, several features can turn relaying back on, either fully or in a limited and controlled way. These features are listed below for completeness, but our recommendation is that you be careful about opening things up too much. The access_db feature is the safest way to allow limited relaying.

- FEATURE(`relay_entire_domain') – allows relaying for just your domain
- RELAY_DOMAIN(`*domain*, ...') – adds more domains to be relayed
- RELAY_DOMAIN_FILE(`*filename*') – same; takes domain list from a file
- FEATURE(`relay_hosts_only') – affects RELAY_DOMAIN, accessdb

You need to make an exception if you use the SMART_HOST or MAIL_HUB designations to route mail through a particular mail server machine. That server must be set up to relay mail from local hosts. Configure it with

```
FEATURE('relay_entire_domain')
```

If you consider turning on relaying in some form, consult the **sendmail** documentation in **cf/README** to be sure you don't inadvertently become a friend of spammers. When you are done, have one of the relay-checking sites verify that you did not inadvertently create an open relay—try spamhelp.org.

User or site blacklisting

If you have local users or hosts to which you want to block mail, use

```
FEATURE('blacklist_recipients')
```

It supports the following types of entries in your access file:

```
To:nobody@              ERROR:550 Mailbox disabled for this user
To:printer.mydomain     ERROR:550 This host does not accept mail
To:user@host.mydomain   ERROR:550 Mailbox disabled for this user
```

These lines block incoming mail to user nobody on any host, to host printer, and to a particular user's address on one machine. The use of the To: tag lets these users send messages, just not receive them; some printers have that capability.

To include a DNS-style blacklist for incoming email, use the dnsbl feature:

```
FEATURE(`dnsbl', `zen.spamhaus.org')
```

This feature makes **sendmail** reject mail from any site whose IP address is in any of the three blacklists of known spammers (SBL, XBL, and PBL) maintained at spamhaus.org. Other lists catalog sites that run open relays and blocks of addresses that are known to be havens for spammers. These blacklists are distributed through a clever tweak of the DNS system; hence the name dnsbl.

You can pass a third argument to the dnsbl feature to specify the error message you would like returned. If you omit this argument, **sendmail** returns a fixed error message from the DNS database that contains the records.

You can include the dnsbl feature several times to check multiple lists of abusers.

Throttles, rates, and connection limits

Table 18.11 lists several **sendmail** controls that can slow down mail processing when clients' behavior appears suspicious.

Table 18.11: sendmail's "slow down" configuration primitives

Primitive	Description
BAD_RCPT_THROTTLE	Slows down spammers collecting addresses
MAX_RCPTS_PER_MESSAGE	Defers delivery if a message has too many recipients
`ratecontrol` feature	Limits the rate of incoming connections
`conncontrol` feature	Limits the number of simultaneous connections
`greet_pause` feature	Delays HELO response, forces strict SMTP compliance

After the no-such-login count reaches the limit set in the BAD_RCPT_THROTTLE option, **sendmail** sleeps for one second after each rejected RCPT command, slowing a spammer's address harvesting to a crawl. To set that threshold to 3, use

```
define(`confBAD_RCPT_THROTTLE', `3')
```

Setting the MAX_RCPTS_PER_MESSAGE option causes the sender to queue extra recipients for later. This is a cheap form of greylisting for messages that have a suspiciously large number of recipients.

The ratecontrol and conncontrol features allow per-host or per-net limits on the rate at which incoming connections are accepted and the number of simultaneous connections, respectively. Both use the **/etc/mail/access** file to specify the limits and the domains to which they should apply, the first

with the tag ClientRate: in the key field and the second with tag ClientConn:. To enable rate controls, insert lines like these in your **.mc** file:

```
FEATURE(`access_db')
FEATURE(`ratecontrol', `nodelay',`terminate')
FEATURE(`conncontrol', `nodelay',`terminate')
```

Then, add to your **/etc/mail/access** file the list of hosts or nets to be controlled and their restriction thresholds. For example, the lines

```
ClientRate:192.168.6.17     2
ClientRate:170.65.3.4       10
```

limit the hosts 192.168.6.17 and 170.65.3.4 to two new connections per minute and ten new connections per minute, respectively. The lines

```
ClientConn:192.168.2.8      2
ClientConn:175.14.4.1       7
ClientConn:                 10
```

set limits of two simultaneous connections for 192.168.2.8, seven for 175.14.4.1, and ten simultaneous connections for all other hosts.

Another nifty feature is greet_pause. When a remote transport agent connects to your **sendmail** server, the SMTP protocol mandates that it wait for your server's welcome greeting before speaking. However, it's common for spam mailers to blurt out an EHLO/HELO command immediately. This behavior is partially explainable as poor implementation of the SMTP protocol in spam-sending tools, but it may also be a feature that aims to save time on the spammer's behalf. Whatever the cause, this behavior is suspicious and is known as "slamming."

The greet_pause feature makes **sendmail** wait for a specified period of time at the beginning of the connection before greeting its newfound friend. If the remote MTA does not wait to be properly greeted and proceeds with an EHLO or HELO command during the planned awkward moment, **sendmail** logs an error and refuses subsequent commands from the remote MTA.

You can enable greeting pauses with this entry in the **.mc** file:

```
FEATURE(`greet_pause', `700')
```

This line causes a 700 millisecond delay at the beginning of every new connection. You can set per-host or per-net delays with a GreetPause: prefix in the access database, but most sites use a blanket value for this feature.

Security and sendmail

With the explosive growth of the Internet, programs such as **sendmail** that accept arbitrary user-supplied input and deliver it to local users, files, or shells have frequently provided an avenue of attack for hackers. **sendmail**, along with DNS and even IP, is flirting with authentication and encryption as a built-in solution to some of these fundamental security issues.

sendmail supports both SMTP authentication and encryption with TLS, Transport Layer Security (formerly known as SSL, the Secure Sockets Layer). TLS brought with it six new configuration options for certificate files and key files. New actions for access database matches can require that authentication must have succeeded.

sendmail carefully inspects file permissions before it believes the contents of, say, a **.forward** or an **aliases** file. Although this tightening of security is generally welcome, it's sometimes necessary to relax the tough policies. To this end, **sendmail** introduced the DontBlameSendmail option, so named in hopes that the name might suggest to sysadmins that what they are doing is unsafe.

This option has many possible values—55 at last count. The default is safe, the strictest possible. For a complete list of values, see **doc/op/op.ps** in the **sendmail** distribution or the O'Reilly **sendmail** book. Or just leave the option set to safe.

Ownerships

Three user accounts are important in the **sendmail** universe: the DefaultUser, the RunAsUser, and the TrustedUser.

By default, all of **sendmail**'s mailers run as the DefaultUser unless the mailer's flags specify otherwise. If a user mailnull, sendmail, or daemon exists in the **passwd** file, DefaultUser will be that. Otherwise, it defaults to UID 1 and GID 1. We recommend the use of the mailnull account and a mailnull group. Add it to **/etc/passwd** with a star as the password, no valid shell, no home directory, and a default group of mailnull. You'll have to add the mailnull entry to the **group** file, too. The mailnull account should not own any files. If **sendmail** is not running as root, the mailers must be setuid.

If RunAsUser is set, **sendmail** ignores the value of DefaultUser and does everything as RunAsUser. If you are running **sendmail** setgid, then the submission **sendmail** just passes messages to the real **sendmail** through SMTP. The real **sendmail** does not have its setuid bit set, but it runs as root from the startup files.

The RunAsUser is the UID that **sendmail** runs under after opening its socket connection to port 25. Ports numbered less than 1,024 can be opened only by the superuser; therefore, **sendmail** must initially run as root. However, after performing this operation, **sendmail** can switch to a different UID. Such a switch reduces the risk of damage or access if **sendmail** is tricked into doing something bad. Don't use the RunAsUser feature on machines that support user accounts or other

services; it is meant for use only on firewalls or bastion hosts (specially hardened hosts intended to withstand attack when placed in a DMZ or outside a firewall).

By default, **sendmail** does not switch identities and continues to run as root. If you change the RunAsUser to something other than root, you must change several other things as well. The RunAsUser must own the mail queue, be able to read all maps and include files, be able to run programs, etc. Expect to spend a few hours discovering all the file and directory ownerships that must be changed.

sendmail's TrustedUser can own maps and alias files. The TrustedUser is allowed to start the daemon or rebuild the **aliases** file. This facility exists mostly to support GUI interfaces to **sendmail** that need to provide limited administrative control to certain users. If you set TrustedUser, be sure to guard the account that it points to because this account can easily be exploited to gain root access. The TrustedUser is different from the TRUSTED_USERS class, which determines who can rewrite the From line of messages. (The TRUSTED_USERS feature is typically used to support mailing list software.)

Permissions

File and directory permissions are important to **sendmail** security. Use the settings listed in Table 18.12 to be safe.

Table 18.12: Owner and permissions for sendmail-related directories

Path	Owner	Mode	What it contains
/var/spool/clientmqueue	smmsp:smmsp	770	Queue for initial submissions
/var/spool/mqueue	RunAsUser	700	Mail queue directory
/, /var, /var/spool	root	755	Path to **mqueue**
/etc/mail/*	TrustedUser	644	Maps, the config file, aliases
/etc/mail	TrustedUser	755	Parent directory for maps
/etc	root	755	Path to **mail** directory

sendmail no longer reads **.forward** files that have link counts greater than 1 if the directory paths that lead to them have lax permissions. This rule bit Evi when one of her **.forward** files, which she usually hard-linked to either **.forward.to.boulder** or **.forward.to.sandiego**, silently failed to forward her mail from a small site at which she did not receive much mail. It was months before she realized that "I never got your mail" was her own fault and not a valid excuse.

You can turn off many of the restrictive file access policies mentioned above with the DontBlameSendmail option. But don't do that.

Safer mail to files and programs

We recommend that you use **smrsh** instead of **/bin/sh** as your program mailer and that you use **mail.local** instead of **/bin/mail** as your local mailer. Both programs are included in the **sendmail** distribution. To incorporate them into your configuration, add the lines

```
FEATURE(`smrsh', `path-to-smrsh')
FEATURE(`local_lmtp', `path-to-mail.local')
```

to your **.mc** file. If you omit the explicit paths, the commands are assumed to live in **/usr/libexec**. You can use **sendmail**'s confEBINDIR option to change the default location of the binaries to whatever you want. Table 18.13 helps you find where our friendly vendors have stashed things.

Table 18.13: Location of sendmail's restricted delivery agents

OS	smrsh	mail.local	sm.bin
Ubuntu	/usr/lib/sm.bin	/usr/lib/sm.bin	/usr/adm
Debian	/usr/lib/sm.bin	/usr/lib/sm.bin	/usr/adm
Red Hat	/usr/sbin	–	/etc/smrsh
CentOS	/usr/sbin	–	/etc/smrsh
FreeBSD	/usr/libexec	/usr/libexec	/usr/adm

smrsh is a restricted shell that executes only the programs contained in one directory (**/usr/adm/sm.bin** by default). **smrsh** ignores user-specified paths and tries to find any requested commands in its own known-safe directory. **smrsh** also blocks the use of certain shell metacharacters such as <, the input redirection symbol. Symbolic links are allowed in **sm.bin**, so you need not make duplicate copies of the programs you allow. The **vacation** program is a good candidate for **sm.bin**. Don't put **procmail** there; it's insecure.

Here are some example shell commands and their possible **smrsh** interpretations:

```
vacation eric                # Executes /usr/adm/sm.bin/vacation eric
cat /etc/passwd              # Rejected, cat not in sm.bin
vacation eric < /etc/passwd  # Rejected, no < allowed
```

sendmail's SafeFileEnvironment option controls where files can be written when email is redirected to a file by **aliases** or a **.forward** file. It causes **sendmail** to execute a **chroot** system call, making the root of the filesystem no longer / but rather **/safe** or whatever path you specified in the SafeFileEnvironment option. An alias that directed mail to the **/etc/passwd** file, for example, would actually be written to **/safe/etc/passwd**.

The SafeFileEnvironment option also protects device files, directories, and other special files by allowing writes only to regular files. Besides increasing security, this option ameliorates the effects of user mistakes. Some sites set the option to **/home** to allow access to home directories while keeping system files off-limits.

Mailers can also be run in a **chroot**ed directory.

Privacy options

sendmail privacy options also control

- What external folks can determine about your site through SMTP
- What you require of the host on the other end of an SMTP connection
- Whether your users can see or run the mail queue

Table 18.14 lists the possible values for the privacy options as of this writing; see the file **doc/op/op.ps** in the distribution for current information.

Table 18.14: Values of the PrivacyOption variable

Value	Meaning
`authwarnings`	Adds warning header if an outgoing message seems forged
`goaway`	Disables all SMTP status queries (EXPN, VRFY, etc.)
`needexpnhelo`	Does not expand addresses (EXPN) without a HELO
`needmailhelo`	Requires SMTP HELO (identifies remote host)
`needvrfyhelo`	Does not verify addresses (VRFY) without a HELO
`nobodyreturn`	Does not return the message body in a DSN
`noetrn`[a]	Disallows asynchronous queue runs
`noexpn`	Disallows the SMTP EXPN command
`noreceipts`	Turns off delivery status notification for success return receipts
`noverb`[b]	Disallows verbose mode for EXPN
`novrfy`	Disallows the SMTP VRFY command
`public`	Does no privacy/security checking
`restrictexpand`	Restricts info displayed by the **-bv** and **-v** flags[c]
`restrictmailq`	Allows only **mqueue** directory's group to see the queue
`restrictqrun`	Allows only **mqueue** directory's owner to run the queue

a. ETRN is an ESMTP command for use by dial-up hosts. It requests that the queue be run just for messages to that host.
b. Verbose mode follows **.forward** files when an EXPN command is given and reports more information on the whereabouts of a user's mail. Use `noverb` or, better yet, `noexpn`, on any machine exposed to the outside world.
c. Unless executed by root or the `TrustedUser`

We recommend conservatism; in your **.mc** file, use

```
define(`confPRIVACY_OPTIONS', ``goaway, authwarnings, restrictmailq,
    restrictqrun'')
```

sendmail's default value for the privacy options is authwarnings; the above line would reset that value. Notice the double sets of quotes; some versions of **m4** require them to protect the commas in the list of privacy option values.

*Running a **chroot**ed **sendmail** (for the truly paranoid)*

If you are worried about the access that **sendmail** has to your filesystem, you can start it in a **chroot**ed jail. Create a minimal filesystem in your jail, including things like **/dev/null**, **/etc** essentials (**passwd**, **group**, **resolv.conf**, **sendmail.cf**, any map files, **mail/***), the shared libraries that **sendmail** needs, the **sendmail** binary, the mail queue directory, and any log files. You will probably have to fiddle with the list to get it just right. Use the **chroot** command to start a jailed **sendmail**. For example:

```
$ sudo chroot /jail /usr/sbin/sendmail -bd -q30m
```

Denial of service attacks

Denial of service attacks are difficult to prevent because no a priori method can determine that a message is an attack rather than a valid piece of email. Attackers can try various nasty things, including flooding the SMTP port with bogus connections, filling disk partitions with giant messages, clogging outgoing connections, and mail bombing. **sendmail** has some configuration parameters that can help slow down or limit the impact of a denial of service attack, but these parameters can also interfere with the delivery of legitimate mail.

The MaxDaemonChildren option limits the number of **sendmail** processes. It prevents the system from being overwhelmed with **sendmail** work. However, it also allows an attacker to easily shut down SMTP service.

The MaxMessageSize option can help prevent the mail queue directory from filling. But if you set it too low, legitimate mail will bounce. You might mention your limit to users so that they aren't surprised when their mail bounces. We recommend a fairly high limit (such as 50MB) anyway, since some legitimate mail is huge.

The ConnectionRateThrottle option, which limits the number of permitted connections per second, can slow things down a bit. Finally, setting MaxRcptsPerMessage, which controls the maximum number of recipients allowed on a single message, may also help.

sendmail has always been able to refuse connections (option REFUSE_LA) or queue email (QUEUE_LA) according to the system load average. A variation, DELAY_LA, keeps the mail flowing, but at a reduced rate.

In spite of all these protections for your mail system, someone mail bombing you will still interfere with legitimate mail. Mail bombing can be quite nasty.

TLS: Transport Layer Security

See this page for general information about TLS.

TLS, a encryption/authentication system, is specified in RFC3207. It is implemented in **sendmail** as an extension to SMTP called STARTTLS.

Strong authentication can replace a hostname or IP address as the authorization token for relaying mail or for accepting a connection from a host in the first place. An entry such as

```
TLS_Srv:secure.example.com   ENCR:112
TLS_Clt:laptop.example.com   PERM+VERIFY:112
```

in the access_db indicates that STARTTLS is in use and that email to the domain secure.example.com must be encrypted with at least 112-bit encryption keys. Email from a host in the laptop.example.com domain should be accepted only if the client has authenticated itself.

Although STARTTLS provides strong encryption, note that its protection covers only the journey to the “next hop” MTA. Once the message arrives at the next hop, it might be forwarded to another MTA that does not use a secure transport method. If you have control of all possible MTAs in the path, you can create a secure mail transport network. If not, you will need to rely on a UA-based encryption package (such as PGP/GPG) or a centralized email encryption service (see this page).

Greg Shapiro and Claus Assmann of Sendmail, Inc., have stashed some (slightly dated) extra documentation about security and **sendmail** on the web. It’s available from sendmail.org/~gshapiro and sendmail.org/~ca. The **index** link in ~ca is especially useful.

sendmail testing and debugging

m4-based configurations are to some extent pretested. You probably won't need to do low-level debugging if you use them. But one thing the debugging flags cannot test is your design.

While researching this chapter, we found errors in several of the configuration files and designs that we examined. The errors ranged from invoking a feature without the prerequisite macro (e.g., enabling masquerade_envelope without having turned on masquerading with MASQUERADE_AS) to total conflict between the design of the **sendmail** configuration and the firewall that controlled whether and under what conditions mail was allowed in.

You cannot design a mail system in a vacuum. You must synchronize it with (or at least not be in conflict with) your DNS MX records and your firewall policy.

Queue monitoring

You can use the **mailq** command (which is equivalent to **sendmail -bp**) to view the status of queued messages. Messages are queued while they are being delivered or when delivery has been attempted but has failed.

mailq prints a human-readable summary of the files in **/var/spool/mqueue** at any given moment. The output is useful for determining why a message may have been delayed. If it appears that a mail backlog is developing, you can monitor the status of **sendmail**'s attempts to clear the jam.

There are two default queues: one for messages received on port 25 and another for messages received on port 587 (the client submission queue). You can invoke **mailq -Ac** to see the client queue.

Below, some typical output from **mailq** shows three messages waiting to be delivered.

```
$ sudo mailq
/var/spool/mqueue (3 requests)
-----Q-ID-----  -Size-  ---Q-Time---      -------Sender/Recipient------
k623gYYk008732  23217   Sat Jul 1 21:42   MAILER-DAEMON
    8BITMIME (Deferred: Connection refused by agribusinessonline.com.)
                                    <Nimtz@agribusinessonline.com>
k5ULkAHB032374  279     Fri Jun 30 15:46  <randy@atrust.com>
    (Deferred: Name server: k2wireless.com.: host name lookup fa)
                                    <relder@k2wireless.com>
k5UJDm72023576  2485    Fri Jun 30 13:13  MAILER-DAEMON
    (reply: read error from mx4.level3.com.)
                                    <lfinist@bbnplanet.com>
```

If you think you understand the situation better than **sendmail** or you just want **sendmail** to try to redeliver the queued messages immediately, you can force a queue run with **sendmail -q**. If you use **sendmail -q -v**, **sendmail** shows the play-by-play results of each delivery attempt,

information that is often useful for debugging. Left to its own devices, **sendmail** retries delivery every queue run interval (typically every 30 minutes).

Logging

See Chapter 10 for more information about syslog.

---

sendmail uses syslog to log error and status messages with the syslog facility "mail" and levels "debug" through "crit"; messages are tagged with the string "sendmail." You can override the logging string "sendmail" with the **-L** command-line option; this capability is handy if you are debugging one copy of **sendmail** while other copies are doing regular email chores.

The confLOG_LEVEL option, specified on the command line or in the config file, determines the severity level that **sendmail** uses as a threshold for logging. High values of the log level imply low severity levels and cause more info to be logged.

Table 18.15 gives an approximate mapping between **sendmail** log levels and syslog severity levels.

Table 18.15: sendmail log levels (L) vs. syslog levels

L	Syslog levels	L	Syslog levels
0	No logging	4	notice
1	alert or crit	5–11	info
2	crit	≥ 12	debug
3	err or warning		

Recall that a message logged to syslog at a particular level is reported to that level and all those above it. The **/etc/syslog.conf** or **/etc/rsyslog.conf** file determines the eventual destination of each message. Table 18.16 shows their default locations.

Table 18.16: Default sendmail log locations

System	Log file location
Debian	**/var/log/mail.log**
Ubuntu	**/var/log/mail.log**
Red Hat	**/var/log/maillog**
CentOS	**/var/log/maillog**
FreeBSD	**/var/log/maillog**

Several programs can summarize **sendmail** log files, with the end products ranging from simple counts and text tables (**mreport**) to fancy web pages (Yasma). You might need to limit access to this data or at least inform your users that you are collecting it.

18.9 Exim

The Exim mail transport and submission agent was written in 1995 by Philip Hazel of the University of Cambridge and is distributed under the GNU General Public License. The current release, Exim version 4.89, came out in spring 2017. Tons of Exim documentation are available on-line, as are a couple of books by the author of the software.

Googling for Exim questions often seems to lead to old, undated, and sometimes inappropriate materials, so check the official documentation first. A 400+ page specification and configuration document (**doc/spec.txt**) is included in the distribution. This document is also available from exim.org as a PDF file. It's the definitive reference work for Exim and is updated religiously with each new release.

There are two cultures with respect to Exim configuration: Debian's and the rest of the world's. Debian runs its own set of mailing lists to support users; we do not cover the Debian-specific configuration extensions here.

Exim is like **sendmail** in that it is implemented as a single process that performs essentially all the ongoing chores associated with email. However, Exim does not carry all **sendmail**'s historical baggage (support for ancient address formats, needing to get mail to hosts not on the Internet, etc.). Many aspects of Exim's behavior are specified at compile time, the chief examples being Exim's database and message store formats.

The workhorses in the Exim system are called routers and transports. Both are included in the general category of "drivers." Routers decide how messages should be delivered, and transports decide on the mechanics of making deliveries. Routers are an ordered list of things to try, whereas transports are an unordered set of delivery methods.

Exim installation

You can download the latest distribution from exim.org or from your favorite package repository. Refer to the top-level **README** file and the file **src/EDITME**, in which you must set installation locations, user IDs, and other compile-time parameters. **EDITME** is over 1,000 lines long, but it's mostly comments that lead you through the compilation process; required changes are well labeled. After your edits, save the file as **../Local/Makefile** or **../Local/Makefile**-*osname* (if you are building configurations for several different operating systems from the same distribution directory) before you run **make**.

Here are a few of the important variables (our opinion) and suggested values (Exim developers' opinion) from the **EDITME** file. The first five are required, and the rest are recommended.

```
BIN_DIRECTORY=/usr/exim/bin     # Where the exim binary should live
SPOOL_DIRECTORY=/var/spool/exim # Mail spool directory
CONFIGURE_FILE=/usr/exim/configure  # Exim's configuration file
SYSTEM_ALIASES_FILE=/etc/aliases    # Location of aliases file
EXIM_USER=ref:exim              # User to run as after rootly chores

ROUTER_ACCEPT=yes               # Router drivers to include
ROUTER_DNSLOOKUP=yes
ROUTER_IPLITERAL=yes
ROUTER_MANUALROUTE=yes
ROUTER_QUERYPROGRAM=yes
ROUTER_REDIRECT=yes

TRANSPORT_APPENDFILE=yes        # Transport drivers to include
TRANSPORT_AUTOREPLY=yes
TRANSPORT_PIPE=yes
TRANSPORT_SMTP=yes

SUPPORT_MAILDIR=yes             # Mailbox formats to understand
SUPPORT_MAILSTORE=yes
SUPPORT_MBX=yes

LOOKUP_DBM=yes                  # DB lookup methods to include
LOOKUP_LSEARCH=yes              # Linear search lookup
LOOKUP_DNSDB=yes                # Allow near-arbitrary DNS lookups
USE_DB=yes                      # Use Berkeley DB (from README)
DBMLIB=-ldb                     # (from README)
WITH_CONTENT_SCAN=yes           # Include content scanning via ACLs

EXPERIMENTAL_SPF=yes            # Include SPF support, needs libspf2
CFLAGS += -I/usr/local/include  # From www.libspf2.org
LDFLAGS += -lspf2

LOG_FILE_PATH=/var/log/exim_%slog   # Log files: file, syslog, or both
LOG_FILE_PATH=syslog
LOG_FILE_PATH=syslog:/var/log/exim_%slog
EXICYCLOG_MAX=10                # Compress/cycle log files, keep 10
```

Routers and transports must be compiled into the code if you intend to use them. In these days of large memories, you might as well leave them all in. Some default paths are certainly nonstandard: for example, the binary in **/usr/exim/bin** and the PID file in **/var/spool/exim**. You might want to tweak these values to match your other installed software.

About ten database lookup methods are available, including MySQL, Oracle, and LDAP. If you include LDAP, you must specify the LDAP_LIB_TYPE variable to tell Exim which LDAP library you are using. You may also need to specify the path to LDAP include files and libraries.

The **EDITME** file does a good job of telling you about any dependencies your database choices might entail. Any entries above that have "(from README)" in their comment line were not listed in **src/EDITME** but rather in the **README**.

EDITME has many additional security options that you might want to include, such as support for SMTP AUTH, TLS, PAM, and options for controlling file ownerships and permissions. You can disable certain Exim options at compile time to limit the damage a hacker might cause if the software is compromised.

It's advisable to read the entire **EDITME** file before you complete the installation. It gives you a good feel for what you can control at run time through the configuration file. The top-level **README** file has lots of detail about OS-specific quirks that you migh need to add to the **EDITME** file as well.

Once you have modified **EDITME** and installed it as **Local/Makefile**, run **make** at the top of the distribution tree followed by **sudo make install**. The next step is to test your shiny new **exim** binary and see if it delivers mail as expected. The **doc/spec.txt** file contains good testing documentation.

Once you are satisfied that Exim is working properly, link **/usr/sbin/sendmail** to **exim** so that Exim can emulate the traditional command-line interface to the mail system used by many user agents. You must also arrange for **exim** to be started at boot time.

Exim startup

On a mail hub machine, **exim** typically starts at boot time in daemon mode and runs continuously, listening on port 25 and accepting messages through SMTP. See Chapter 2, *Booting and System Management Daemons*, for startup details for your operating system.

Like **sendmail**, Exim can wear several hats, and if started with specific flags or alternative command names, it performs different functions. Exim's mode flags are similar to those understood by **sendmail** because **exim** works hard to maintain compatibility when called by user agents and other tools. Table 18.17 lists a few common flags.

Table 18.17: Common exim command-line flags

Flag	Meaning
-bd	Runs in daemon mode and listens for connections on port 25
-bf or **-bF**	Runs in user or system filter test mode
-bi	Rebuilds hashed aliases (same as **newaliases**)
-bp	Prints the mail queue (same as **mailq**)
-bt	Enters address test mode
-bV	Checks for syntax errors in the configuration file
-d+-*category*	Runs in debug mode, flexible category-based configuration
-q	Starts a queue runner (same as **runq**)

Any errors in the config file that can be detected at parse time are caught by **exim -bV**, but some errors can only be caught at run time. Misplaced braces are a common mistake.

The **exim** man page gives lots of detail on all the nooks and crannies of **exim**'s command-line flags and options, including extensive debugging information.

Exim utilities

The Exim distribution includes a bunch of utilities to help you monitor, debug, and sanity-check your installation. Below is the current list along with a brief description of each. See the documentation from the distribution for more detail.

- **exicyclog** – rotates log files
- **exigrep** – searches the main log
- **exilog** – visualizes log files across multiple servers
- **exim_checkaccess** – checks address acceptance from a given IP address
- **exim_dbmbuild** – builds a DBM file
- **exim_dumpdb** – dumps a hints database
- **exim_fixdb** – patches a hints database
- **exim_lock** – locks a mailbox file
- **exim_tidydb** – cleans up a hints database
- **eximstats** – extracts statistics from the log
- **exinext** – extracts retry information
- **exipick** – selects messages according to various criteria
- **exiqgrep** – searches the queue
- **exiqsumm** – summarizes the queue
- **exiwhat** – lists what Exim processes are doing

Another utility that is part of the Exim suite is **eximon**, an X Windows application that displays Exim's state, the state of Exim's queue, and the tail of the log file. As with the main distribution, you build it by editing a well-commented **EDITME** file in the **exim_monitor** directory and running **make**. However, in the case of **eximon** the defaults are usually fine, so you should not have to do much configuration to build the application. Some configuration and queue management can be done from the **eximon** GUI as well.

Exim configuration language

The Exim configuration language (or more accurately, languages: one for filters, one for regular expressions, etc.) feels a bit like the ancient (1970s) language Forth. When first reading an Exim configuration, you might find it hard to distinguish between keywords and option names (which are fixed by Exim) and variable names (which are defined by sysadmins through configuration statements).

Although Exim is advertised as being easy to configure and is extensively documented, there can be quite a learning curve for new users. The section “How Exim receives and delivers mail” in the specification document is essential reading for newcomers. It gives a good feel for the underlying concepts of the system.

When assigned a value, the Exim language’s predefined options sometimes cause an action. The values of about 120 predefined variables may also change in response to an action. These variables can be included in conditional statements.

The language for evaluating `if` statements and the like may remind you of the reverse Polish notation used during the heyday of Hewlett-Packard calculators. Let’s look at a simple example. In the line

```
acl_smtp_rcpt = ${if ={25}{$interface_port} \
    {acl_check_rcpt} {acl_check_rcpt_submit} }
```

the `acl_smtp_rcpt` option, when set, causes an ACL to be implemented for each recipient (SMTP RCPT command) in the SMTP exchange. The value assigned to this option is either `acl_check_rcpt` or `acl_check_rcpt_submit`, depending on whether or not the Exim variable `$interface_port` has value 25.

We do not detail the Exim configuration language in this chapter, but refer you instead to the extensive documentation. In particular, pay close attention to the string expansion section of the Exim specification.

Exim configuration file

Exim's run-time behavior is controlled by a single configuration file, usually called **/usr/exim/configure**. Its name is one of the required variables specified in the **EDITME** file and compiled into the binary.

The supplied default configuration file, **src/configure.default**, is well commented and is a good starting place for sites just getting set up with Exim. In fact, we recommend that you don't stray too far from it until you thoroughly understand the Exim paradigm and need to elaborate on the default configuration for a specific purpose. Exim works hard to support common situations and has sensible defaults.

It's also helpful to stick with the variable names used in the default config file. These naming conventions are assumed by folks on the exim-users mailing list. Those people are also a good resource to consult regarding your configuration questions.

exim prints a message to stderr and exits if you have a syntax error in your configuration file. It doesn't catch all syntax errors immediately, however, because it does not expand variables until it needs to.

The order of entries in the configuration file is not quite arbitrary: the global configuration options section must be first and must exist. All other sections are optional and can appear in any order.

Possible sections include

- Global configuration options (mandatory)
- acl – access control lists that filter addresses and messages
- authenticators – for SMTP AUTH or TLS authentication
- routers – ordered sequence to determine where a message should go
- transports – definitions of the drivers that do the actual delivery
- retry – policy settings for dealing with problem messages
- rewrite – global address rewriting rules
- local_scan – a hook for fancy flexibility

Each section except the first starts with a begin *section-name* statement—for example, begin acl. There is no end *section-name* statement; the end is signaled by the next section's begin statement. Indentation to show subordination makes the config file easier to read for humans, but it is not meaningful to Exim.

Some configuration statements name objects that will later be used to control the flow of messages. Those names must begin with a letter and contain only letters, numbers, and the underscore character. If the first non-whitespace character on a line is `#`, the rest of the line is treated as a comment. Note that this means you cannot put a comment on the same line as a statement; it will not be recognized as a comment because the first character is not `#`.

Exim lets you include files anywhere in the configuration file. Two forms of include are used:

```
.include absolute-path
.include_if_exists absolute-path
```

The first form generates an error if the file does not exist. Although include files keep your config file tidy, they are read several times during the life of a message, so it might be best just to include their contents directly into your configuration.

Global options

Lots of stuff is specified in the global options section, including operating parameters (limits, sizes, timeouts, properties of the mail server on this host), list definitions (local hosts, local hosts to relay for, remote domains to relay for), and macros (hostname, contact, location, error messages, SMTP banner).

Options

Options are set with the basic syntax

```
option_name = value[s]
```

where the *values* can be Booleans, strings, integers, decimal numbers, or time intervals. Multivalued options are allowed, in which case the various values are separated by colons.

Use of the colon as a value separator presents a problem when you express IPv6 addresses, which use colons as part of the address. You can escape the colons by doubling them, but the easiest and most readable fix is to redefine the separator character with the < character as you assign values to the option. For example, both of the following two lines set the value of the localhost_interfaces option, which contains the IPv4 and IPv6 localhost addresses:

```
local_interfaces = 127.0.0.1 : ::::1
local_interfaces = <; 127.0.0.1 ; ::1
```

The second form, in which the semicolon has been defined as the separator, is more readable and less fragile.

There are a zillion options—more than 500 in the options index of the documentation. And we said **sendmail** was complicated! Most options have sensible defaults, and all have descriptive names. It's handy to have a copy of the **doc/spec.txt** file from the distribution in your favorite text editor when you are researching a new option. We don't cover all the options below, just the ones that occur in our example configuration bits.

Lists

Exim has four kinds of lists, introduced by the keywords hostlist, domainlist, addresslist, and localpartslist. Here are two examples that use hostlist:

```
hostlist my_relay_list = 192.168.1.0/24 : myfriend.example.com
hostlist my_relay_list = /usr/local/exim/relay_hosts.txt
```

Members can be listed in-line or taken from a file. If in-line, they are separated by colons. There can be up to 16 named lists of each type. In the in-line example above, we included all machines on a local /24 network and a specific hostname.

The symbol @ can be a member of a list; it means the name of the local host and helps you write a single generic configuration file that works for most nonhub machines at your site. The notation @[] is also useful and means all IP addresses on which Exim is listening; that is, all the IP addresses of the local host.

Lists can include references to other lists and the ! character to indicate negation. Lists that include references to variables (e.g., $variable_name) make processing slower because Exim cannot cache the results of evaluating the list, which it otherwise does by default.

To reference a list, just put + in front of its name to match members of the list or !+ to match nonmembers; for example, +my_relay_list. Omit space between the + sign and the name of the list.

Macros

You can use macros to define parameters, error messages, etc. The parsing is primitive, so you cannot define a macro whose name is a subset of another macro without unpredictable results.

The syntax is

```
MACRO_NAME = rest of the line
```

For example, the first of the following lines defines a macro named ALIAS_QUERY that looks up a user's alias entry in a MySQL database. The second line shows the use of the macro to perform an actual lookup, with the result being stored in the variable called data.

```
ALIAS_QUERY = \
    select mailbox from user where login = '${quote_mysql:$local_part}';
data = ${lookup mysql{ALIAS_QUERY}}
```

Macro names are not required to be all caps, but they must begin with a capital letter. However, the all-caps convention aids clarity. The configuration file can include ifdefs that evaluate a macro and use it to determine whether or not to include a portion of the config file. Every imaginable form of ifdef is supported; they all begin with a dot.

Access control lists (ACLs)

Access control lists filter the addresses of incoming messages and either accept or deny them. Exim divides incoming addresses into a local part that represents the user and a domain part that is the recipient's domain.

ACLs can be applied at any of the various stages of an SMTP conversation: HELO, MAIL, RCPT, DATA, etc. Typically, an ACL enforces strict adherence to the SMTP protocol at the HELO stage, checks the sender and the sender's domain at the MAIL stage, checks the recipients at the RCPT stage, and scans the message content at the DATA stage.

A slew of options named acl_smtp_*command* specify which ACL should be applied after each *command* in the SMTP protocol. For example, the acl_smtp_rcpt option directs the ACL to run on each address that is a recipient of the message. Another commonly used checkpoint is acl_smtp_data, which checks the ACL against the message after it has been received, for example, to scan content.

You can define ACLs in the acl section of the config file, in a file that is referenced by the acl_smtp_*command* option or in-line when the option is defined.

A sample ACL called my_acl_check_rcpt is defined below. We would invoke it by assigning its name to the acl_smtp_rcpt option in the global options section of the config file. (If this ACL denies an address at the level of the RCPT command, the sending server should give up and not try the address again.)

This is a long ACL specification, so we break it up into digestible pieces that we can decode individually.

The first portion:

```
begin acl
    my_acl_check_rcpt:
        accept  hosts = :
                control = dkim_disable_verify
```

The default name for this access control list is acl_check_rcpt; you probably should not change its name as we did here. We used a nonstandard name simply to emphasize that the name is something you specify, not a keyword that's special to Exim.

The first accept line, containing just a colon, is an empty list. The empty list of remote hosts matches cases in which a local MUA submitted a message on the MTA's standard input. If the address being tested meets this condition, the ACL accepts the address and disables DKIM signature validation, which is turned on by default. If the address does not match this address clause, control drops through to the next clause in the ACL definition:

```
deny    message = Restricted characters in address
        domains = +local_domains
        local_parts = ^[.] : ^.*[@%!/|]

deny    message = Restricted characters in address
        domains = !+local_domains
        local_parts = ^[./|] : ^.*[@%!] : ^.*/\\.\\./
```

The first deny stanza is intended for messages coming into your local domains. It rejects any address whose local part (the username) starts with a dot or contains the special characters @, %, !, /, or |. The second deny applies to messages being sent out by your users. It, too, disallows certain special characters and sequences in the local parts of addresses, in case your users' machines have been infected with a virus or other malware. In the past, such addresses have been used by spammers to confuse ACLs or have been associated with other security problems.

In general, if you are intending to use $local_parts (supposedly, the recipient's username) in a directory path (to store mail or look for a vacation file, for example) be careful that your ACLs have filtered out any special characters that could cause unwanted behavior. (The example looks for the sequence /../, which could be problematic if the username were inserted into a path.)

```
accept  local_parts = postmaster
        domains = +local_domains
```

This accept stanza guarantees that mail to postmaster always gets through if it's sent to a local domain, and that can help with debugging.

```
require  verify = sender
```

The require line checks to see if a bounce message can be returned; however, it checks only the sender's domain. (require means "deny if not matched.") If the sender's username has been forged, a bounce message could still fail; that is, the bounce message itself could bounce.You can add more extensive checking here by calling another program, but some sites consider such callouts abusive and might add your mail server to a blacklist or bad-reputation list.

```
accept  hosts = +relay_from_hosts
        control = submission
        control = dkim_disable_verify
```

The above accept stanza checks for hosts that are allowed to relay through this host, namely, local hosts that are submitting mail into the system. The control line specifies that Exim should act as a mail submission agent and fix up any header deficiencies as the message arrives from the user agent. The recipient's address is not checked because many user agents are confused by error returns. (This part of the configuration is appropriate only for local machines that relay to a smart host, not for any external domains you might be willing to relay for.) DKIM verification is disabled because these messages are outbound from your users or relay friends.

```
accept  authenticated = *
        control = submission
        control = dkim_disable_verify
```

The last accept stanza deals with local hosts that authenticate through SMTP AUTH. Once again, these messages are treated as submissions from user agents.

```
require  message = Relay not permitted
         domains = +local_domains : +relay_to_domains

require  verify = recipient
```

Here, we check the destination domain to which the message is headed and require that it be either in our list of local_domains or in our list of domains to which we allow relaying, relay_to_domains. (These domain lists are defined outside the context of the ACL.) Any destinations not in one of those lists are refused with a customized error message.

```
accept
```

Finally, given that all previous requirements have been met but that no more-specific accept or deny rule has been triggered, we verify the recipient and accept the message. Most Internet messages to local users fall into this category.

We haven't included any blacklist scanning in the example above. To access a blacklist, use one of the examples in the default config file or something like this:

```
deny     condition = ${if isip4{$sender_host_address}}
         !authenticated = *
         !hosts = +my_whitelist_ips
         !dnslists = list.dnswl.org
         domains = +local_domains
         verify = recipient
         message = You are on RBL $dnslist_domain: $dnslist_text
         dnslists = zen.spamhaus.org
         logwrite = Blacklisted sender [$sender_host_address] \
             $dnslist_domain: $dnslist_text
```

Translated to English, the code specifies that if a message matches *all* of the following criteria, it is rejected with a custom error message and logged (also with a custom message):

- It's from an IPv4 address (some lists don't handle IPv6 correctly).
- It's not associated with an authenticated SMTP session.
- It's from a sender not in the local whitelist.
- It's from a sender not in the global (Internet) whitelist.

- It’s addressed to a valid local recipient.
- The sending host is on the zen.spamhaus.org blacklist.

The variables dnslist_text and dnslist_domain are set by the assignment to dnslists, which triggers the blacklist lookup. This deny clause could be placed right after your checks for unusual characters in addresses.

Here’s another example ACL that rejects mail if the remote side does not say HELO properly:

```
acl_check_mail:
    deny    message = 503 Bad command - must send HELO/EHLO first
            condition = ${if !def:sender_helo_name}
    accept
```

Exim solves the early talker problem (a more specific case of “not saying HELO properly”) with the smtp_enforce_sync option, which is turned on by default.

Content scanning at ACL time

Exim supports powerful content scanning at several points in a message's traversal of the mail system: at ACL time (after the SMTP DATA command); at delivery time through the transport_filter option; or with a local_scan function after all ACL checks have been completed. You must compile support for content scanning into Exim by setting the WITH_CONTENT_SCAN variable in the **EDITME** file; it is commented out by default. This option endows ACLs with extra power and flexibility and adds two new configuration options: spamd_address and av_scanner.

Scanning at ACL time allows a message to be rejected in-line with the MTA's conversation with the sending host. The message is never accepted for delivery, so it need not be bounced. This way of rejecting the message is nice because it avoids backscatter spam caused by bounce messages to forged sender addresses.

Authenticators

Authenticators are drivers that interact with the SMTP AUTH command's challenge-and-response sequence and identify an authentication mechanism acceptable to both client and server. Exim supports the following mechanisms:

- AUTH_CRAM_MD5 (RFC2195)
- AUTH_PLAINTEXT, which includes both PLAIN and LOGIN
- AUTH_SPA, which supports Microsoft's Secure Password Authentication

If Exim is receiving email, it is acting as an SMTP AUTH server. If it is sending mail, it is a client. Options that appear in the definitions of authenticator instances are tagged with a prefix of either server_ or client_ to allow for different configurations depending on the role Exim is playing.

Authenticators are used in access control lists, as in the following clause in the ACL example from this page:

```
        deny    !authenticated = *
```

Below is an example that shows both the client-side and server-side LOGIN mechanisms. This simple example uses a fixed username and password, which is OK for small sites but probably inadvisable for larger installations.

```
begin authenticators

    my_client_fixed_login:
        driver = plaintext
        public_name = LOGIN
        client_send = : myusername : mypasswd

    my_server_fixed_login:
        driver = plaintext
        public_name = LOGIN
        server_advertise_condition = ${if def:tis_cipher}
        server_prompts = User Name : Password
        server_condition = ${if and {{eq{$auth1}{username}} \
            {eq{$auth2}{mypasswd}}}}
        server_set_id = $auth1
```

Authentication data can come from many sources: LDAP, PAM, **/etc/passwd**, etc. The server_advertise_condition clause above prevents mail clients from sending passwords in the clear by requiring TLS security (through STARTTLS or SSL) on connection. If you want the same behavior when Exim acts as the client system, use the client_condition option in the client clause, too, again with tis_cipher.

Refer to the Exim documentation for details of all possible authentication options and for examples.

Routers

Routers work on recipient email addresses, either by rewriting them or by assigning them to a transport and sending them on their way. A particular router can have multiple instances, each with different options.

You specify a sequence of routers. A message starts with the first router and progresses through the list until the message is either accepted or rejected. The accepting router typically hands the message to a transport driver. Routers handle both incoming and outgoing messages. They feel a bit like subroutines in a programming language.

A router can return any of the dispositions shown in Table 18.18 for a message.

Table 18.18: Exim router statuses

Status	Meaning
`accept`	The router accepts the address and hands it to a transport driver.
`pass`	The router can't handle the address; go on to the next router.
`decline`	The router chooses not to handle the address; next router, please!
`fail`	The address is invalid; the router queues it for a bounce message.
`defer`	The message is left in the queue to be dealt with later.
`error`	There is an error in the router specification; the message is deferred.

If a message receives a `pass` or `decline` from all the routers in the sequence, it is unroutable. Exim bounces or rejects such messages, depending on the context.

If a message meets the preconditions for a router and the router ends with a `no_more` statement, then that message will not be presented to any additional routers, regardless of its disposition by the current router. For example, if your remote SMTP router has the precondition `domains = !+local_domains` and has `no_more` set, then only messages to local users (that is, those that would fail the `domains` precondition) will continue to the next router in the sequence.

Routers have many possible options; some common examples are preconditions, acceptance or failure conditions, error messages to return, and transport drivers to use.

The next few sections detail the routers called `accept`, `dnslookup`, `manualroute`, and `redirect`. The example configuration snippets assume that Exim is running on a local machine in the example.com domain. They're all pretty straightforward; refer to the documentation if you want to use some of the fancier routers.

The accept router

The `accept` router labels an address as OK and passes the associated message to a transport driver. Below are examples of `accept` router instances called `localusers`, for delivering local mail, and

save_to_file, for appending to an archive.

```
localusers:
    driver = accept
    domains = example.com
    check_local_user
    transport = my_local_delivery

save_to_file:
    driver = accept
    domains = dialup.example.com
    transport = batchsmtp_appendfile
```

The localusers router instance checks that the domain part of the destination address is example.com and that the local part of the address is the login name of a local user. If both conditions are met, the router hands the message to the transport driver instance called my_local_delivery, which is defined in the transports section. The save_to_file instance is designed for dial-up users; it appends the message to a file specified in the batchsmtp_appendfile transport definition.

The dnslookup router

The dnslookup router typically handles outgoing messages. It looks up the MX record of the recipient's domain and hands the message to an SMTP transport driver for delivery. Here is an instance called remoteusers:

```
remoteusers:
    driver = dnslookup
    domains = !+example.com
    transport = my_remote_delivery
```

See this page for more information about RFC1918 private address spaces.

The dnslookup code looks up the MX records for the addressee. If no MX records exist, it tries the A record. A common extension to this router instance prohibits delivery to certain IP addresses; a prime example is the RFC1918 private addresses that cannot be routed on the Internet. See the ignore_target_hosts option for more information.

The manualroute router

The flexible manualroute driver can pretty much route email in whatever way you want. The routing information can be a table of rules that match by recipient domain (route_list) or a single rule that applies to all domains (route_data).

Below are two examples of manualroute instances. The first example implements the "smart host" concept, in which all outgoing nonlocal mail is sent to a central ("smart") host for processing.

This instance is called smarthost and applies to all recipients' domains that are not (the ! character) in the local_domains list.

```
smarthost:
    driver = manualroute
    domains = !+local_domains
    transport = remote_smtp
    route_data = smarthost.example.com
```

The router instance below, firewall, speaks SMTP to send incoming messages to hosts inside the firewall (perhaps after scanning them for spam and viruses). It looks up the routing data for each recipient domain in a DBM database that contains the names of local hosts.

```
firewall:
    driver = manualroute
    transport = remote-smtp
    route_data = ${lookup{$domain} dbm {/internal/host/routes}}
```

The redirect router

The redirect driver does address rewriting, such as that called for in the system-wide **aliases** file or in a user's **~/.forward** file. It usually does not assign the rewritten address to a transport; that task is left to other routers in the chain.

The first instance shown below, system_aliases, looks up aliases with a linear search (lsearch) of the **/etc/aliases** file. That's fine for a small **aliases** file, but if yours is huge, replace that linear search with a database lookup. The second instance, user_forward, first verifies that mail is addressed to a local user, then checks that user's **.forward** file.

```
system_aliases:
    driver = redirect
    data = ${lookup{$local_part} lsearch {/etc/aliases}}

user_forward:
    driver = redirect
    check_local_user
    file = $home/.forward
    no_verify
```

The check_local_user option ensures that the recipient is a valid local user. The no_verify says not to verify the validity of the address to which the forward file redirects the message; just ship it.

*Per-user filtering through **.forward** files*

Exim not only allows forwarding through **.forward** files but also allows filtering. It supports its own filtering system as well as the Sieve filtering that is being standardized by the IETF. If the first line of a user's **.forward** file is

```
#Exim filter
```

or

```
#Sieve filter
```

then the subsequent filtering commands (there are about 15 of them) can determine where the message should be delivered. Filtering does not actually deliver messages—it just meddles with the destination. For example:

```
#Exim filter
if  $header_subject: contains sysadmin
then
    save $home/mail/sysadmin
endif
```

Lots of options are available to control what users can and cannot do in their **.forward** files. The option names begin with forbid_ or allow_. They're important because they can prevent users from running shells, loading libraries into binaries, or accessing the embedded Perl interpreter when they shouldn't. Check for new forbid_* options when you upgrade to be sure your users can't get too fancy in their **.forward** files.

Transports

Routers decide where messages should go, and transports actually take them there. Local transports typically append to a file, pipe to a local program, or speak the LMTP protocol to an IMAP server. Remote transports speak SMTP to their counterparts across the Internet.

There are five Exim transports: appendfile, lmtp, smtp, autoreply, and pipe; we detail appendfile and smtp. The autoreply transport is typically used to send vacation messages, and the pipe transport hands messages as input to a command through a UNIX pipe. As with routers, you must define instances of transports, and it's OK to have multiple instances of the same type of transport. Order is significant for routers, but not for transports.

The appendfile transport

The appendfile driver stores messages in **mbox**, **mbx**, **Maildir**, or **mailstore** format in a specified file or directory. You must have included the appropriate mailbox formats when you compiled Exim; they are commented out of the **EDITME** file by default.

The following example defines the my_local_delivery transport (an instance of the appendfile transport) referred to in the localusers router instance definition on this page.

```
my_local_delivery:
    driver = appendfile
    file = /var/mail/$local_part
    delivery_date_add
    envelope_to_add
    return_path_add
    group = mail
    mode = 0660
```

The various *_add lines add headers to the message. The group and mode clauses ensure that the transport agent can write to the file.

The smtp transport

The smtp transport is the workhorse of any mail system. Here, we define two instances, one for the standard SMTP port (25) and one for the mail submission port (587).

```
my_remote_delivery:
    driver = smtp

my_remote_delivery_port587:
    driver = smtp
    port = 587
    headers_add = X-processed-by: MACRO_HEADER port 587
```

The second instance, my_remote_delivery_port587, specifies the port and also a header to be added to the message that includes an indication of the outgoing port. MACRO_HEADER would be defined elsewhere in the configuration file.

Retry configuration

The `retry` section of the configuration file must exist or Exim will never attempt redelivery of messages that could not be delivered on the first attempt. You can specify three time intervals, each less frequent than the previous one. After the last interval has expired, messages bounce back to the sender as undeliverable. `retry` statements understand the suffixes `m`, `h`, `d`, and `w` to indicate minutes, hours, days, and weeks. You can specify different intervals for different hosts or domains.

Here's what a `retry` section looks like:

```
begin retry
    *   *   F, 2h, 15m;  F, 24h, 1h;  F, 4d, 6h
```

This example means, "For any domain, an address that fails temporarily should be retried every 15 minutes for 2 hours, then every hour for the next 24 hours, then every 6 hours for 4 days, and finally, bounced as undeliverable."

Rewriting configuration

The rewriting section of the configuration file starts with `begin rewrite`. It's used to fix up addresses, not to reroute messages. For example, you could use it on your outgoing addresses

- To make mail appear to be from your domain, not from individual hosts
- To map usernames to a standard format such as First.Last

Do not apply rewriting to addresses in incoming mail.

Local scan function

To further customize Exim, for example, to filter for the latest and greatest virus, you could write a C function that does your scanning and install it in the local_scan section of the config file. Refer to the Exim documentation for details and examples of how to do this.

Logging

Exim by default writes three different log files: a main log, a reject log, and a panic log. Each log entry includes the time the message was written. You specify the location of the log files in the **EDITME** file (before building Exim) or in the run-time config file in the value of the log_file_path option. By default, logs are kept in the **/var/spool/exim/log** directory.

The log_file_path option accepts up to two colon-separated values. Each value must be either the keyword syslog or an absolute path with a %s embedded where the names **main**, **reject**, and **panic** can be substituted. For example,

```
log_file_path = syslog : /var/log/exim_%s
```

would log both to syslog (with facility "mail") and to the separate files **exim_main**, **exim_reject**, and **exim_panic** in the **/var/log** directory. Exim submits the **main** log entries to syslog at priority info, the **reject** entries at priority notice, and the **panic** entries at priority alert.

The **main** log contains one line for the arrival and delivery of each message. It can be summarized by the Perl script **eximstats**, which is included in the Exim distribution.

The **reject** log records information about messages that have been rejected for policy reasons: malware, spam, etc. It includes the summary line for the message from the **main** log and also the original headers of the message that was rejected. If you change your policies, check the **reject** log to make sure that all is still well.

The **panic** log is for serious errors in the software; **exim** writes here just before it gives up. The **panic** log should not exist in the absence of problems. Ask **cron** to check it for you and if it exists, fix the problem that caused the panic and then delete the file. **exim** will re-create it when the next panic-worthy situation arises.

When debugging, you can increase the amount and type of data logged. Invoke the log_selector option. For example:

```
log_selector = +smtp_connection +smtp_incomplete_transaction +...
```

The logging categories that can be included or excluded by the log_selector mechanism are listed in the Exim specification, in the section called "Log files" toward the end. About 35 categories are defined, including +all, which will really fill your disks!

exim also keeps a temporary log for each message it handles. It is named with the message ID and lives in **/var/spool/exim/msglog**. If you are having trouble with a particular destination, check there.

Debugging

Exim has powerful debugging aids. You can configure the amount of information you want to see about each potential debugging topic. **exim -d** tells **exim** to go into debugging mode, in which it stays in the foreground and does not detach from the terminal. You can add specific debugging categories to **-d** with a + or - in front of them to verbosify or eliminate a category. For example, **-d+expand+acl** requests regular debugging output plus extra details regarding string expansions and ACL interpretation. (These two categories are common problem spots.) You can tune more than 30 categories of debugging information; see the man page for a list.

A common technique when debugging mail systems is to start the MTA on a nonstandard port and then talk to it through **telnet**. For example, to start **exim** in daemon mode, listening on port 26, with debugging info turned on, run

```
$ sudo exim -d -oX 26 -bd
```

You can then **telnet** to port 26 and type SMTP commands in an attempt to reproduce the problem you are debugging.

Alternatively, you can have **swaks** do your SMTP talking for you. It's a Perl script that makes SMTP debugging faster and easier. **swaks --help** gets you some documentation, and jetmore.org/john/code/swaks supplies complete details.

If your log files show timeouts of around 30 seconds, that's suggestive of a DNS issue.

18.10 POSTFIX

Postfix is another popular alternative to **sendmail**. Wietse Venema started the Postfix project when he spent a sabbatical year at IBM's T. J. Watson Research Center in 1996, and he is still actively developing it. Postfix's design goals included not only security (first and foremost!), but also an open source distribution policy, speedy performance, robustness, and flexibility. All major Linux distributions include Postfix, and since version 10.3, macOS has shipped Postfix instead of **sendmail** as its default mail system.

See this page for more information about regular expressions.

The most important things to know about Postfix are, first, that it works almost out of the box (the simplest config files are only a line or two long), and second, that it leverages regular expression maps to filter email effectively, especially in conjunction with the PCRE (Perl-Compatible Regular Expression) library. Postfix is compatible with **sendmail** in the sense that Postfix's **aliases** and **.forward** files have the same format and semantics as those of **sendmail**.

Postfix speaks ESMTP. Virtual domains and spam filtering are both supported. For address rewriting, Postfix relies on table lookups from flat files, Berkeley DB, DBM, LDAP, NetInfo, or SQL databases.

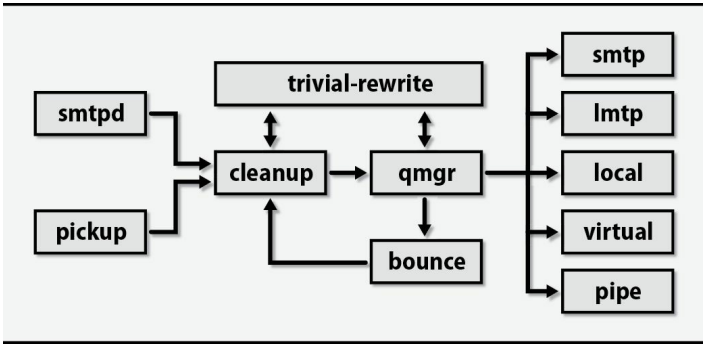
Postfix architecture

Postfix comprises several small, cooperating programs that send network messages, receive messages, deliver email locally, etc. Communication among them is performed through local domain sockets or FIFOs. This architecture is quite different from that of **sendmail** and Exim, wherein a single large program does most of the work.

The **master** program starts and monitors all Postfix processes. Its configuration file, **master.cf**, lists the subsidiary programs along with information about how they should be started. The default values set in that file cover most needs; in general, no tweaking is necessary. One common change is to comment out a program, for example, **smtpd**, when a client should not listen on the SMTP port.

The most important server programs involved in the delivery of email are shown in Exhibit B.

Exhibit B: Postfix server programs



Receiving mail

smtpd receives mail entering the system through SMTP. It also verifies that the connecting clients are authorized to send the mail they are trying to deliver. When email is sent locally through the **/usr/lib/sendmail** compatibility program, a file is written to the **/var/spool/postfix/maildrop** directory. That directory is periodically scanned by the **pickup** program, which processes any new files it finds.

All incoming email passes through **cleanup**, which adds missing headers and rewrites addresses according to the canonical and virtual maps. Before inserting mail into the incoming queue, **cleanup** passes it through **trivial-rewrite**, which does minor fixing of the addresses, such as appending a mail domain to addresses that are not fully qualified.

Managing mail-waiting queues

qmgr manages five queues that contain mail waiting to be delivered:

- incoming – mail that is arriving
- active – mail that is being delivered
- deferred – mail for which delivery has failed in the past
- hold – mail blocked in the queue by the administrator
- corrupt – mail that can't be read or parsed

The queue manager generally follows a simple FIFO strategy to select the next message to process, but it also supports a complex preemption algorithm that prefers messages with few recipients over bulk mail.

To avoid overwhelming a receiving host, especially one that has been down, Postfix uses a slow-start algorithm to control how fast it tries to deliver email. Deferred messages are given a try-again time stamp that exponentially backs off so as not to waste resources on undeliverable messages. A status cache of unreachable destinations avoids unnecessary delivery attempts.

Sending mail

qmgr, aided by **trivial-rewrite**, decides where a message should be sent. The routing decision made by **trivial-rewrite** can be overridden through lookup tables (transport_maps).

Delivery to remote hosts via the SMTP protocol is performed by the **smtp** program. **lmtp** delivers mail with LMTP, the Local Mail Transfer Protocol defined in RFC2033. LMTP is derived from SMTP, but the protocol has been modified so that the mail server is not required to manage a mail queue. This mailer is particularly useful for delivering email to mailbox servers such as the Cyrus IMAP suite.

local's job is to deliver email locally. It resolves addresses in the **aliases** table and follows instructions found in recipients' **.forward** files. Messages are forwarded to another address, passed to an external program for processing, or stored in users' mail folders.

The **virtual** program delivers email to "virtual mailboxes"; that is, mailboxes that are not related to a local UNIX account but that still represent valid email destinations. Finally, **pipe** implements delivery through external programs.

Security

Postfix implements security at several levels. Most of the Postfix server programs can run in a **chroot**ed environment. They are separate programs with no parent/child relationship. None of them are setuid. The mail drop directory is group-writable by the postdrop group, to which the **postdrop** program is setgid.

Postfix commands and documentation

Several command-line utilities permit user interaction with the mail system:

- **postalias** – builds, modifies, and queries alias tables
- **postcat** – prints the contents of queue files
- **postconf** – displays and edits the main configuration file, **main.cf**
- **postfix** – starts and stops the mail system (must be run as root)
- **postmap** – builds, modifies, or queries lookup tables
- **postsuper** – manages mail queues
- **sendmail**, **mailq**, **newaliases** – are **sendmail**-compatible replacements

The Postfix distribution includes a set of man pages that describe all the programs and their options. On-line documents at postfix.org explain how to configure and manage various aspects of Postfix. These documents are also included in the Postfix distribution in the **README_FILES** directory.

Postfix configuration

The **main.cf** file is Postfix's principal configuration file. The **master.cf** file configures the server programs. It also defines various lookup tables that are referenced from **main.cf** and that provide different types of service mappings.

The **postconf**(5) man page describes every parameter you can set in the **main.cf** file. There is also a **postconf** program, so if you just type **man postconf**, you'll get the man page for that instead of **postconf**(5). Use **man -s 5 postconf** to get the right version.

The Postfix configuration language looks a bit like a series of **sh** comments and assignment statements. Variables can be referenced in the definition of other variables by being prefixed with a `$`. Variable definitions are stored just as they appear in the config file; they are not expanded until they are used, and any substitutions occur at that time.

You can create new variables by assigning values to them. Be careful to choose names that do not conflict with existing configuration variables.

All Postfix configuration files, including the lookup tables, consider lines starting with whitespace to be continuation lines. This convention results in readable configuration files, but you must start new lines in column one.

*What to put in **main.cf***

More than 500 parameters can be specified in the **main.cf** file. However, just a few of them need to be set at an average site. The author of Postfix strongly recommends that only parameters with nondefault values be included in your configuration. That way, if the default value of a parameter changes in the future, your configuration will automatically adopt the new value.

The sample **main.cf** file that comes with the distribution includes many commented-out example parameters, along with some brief documentation. The original version is best left alone as a reference. Start with an empty file for your own configuration so that your settings do not become lost in a sea of comments.

Basic settings

The simplest possible Postfix configuration is an empty file. Surprisingly, this is a perfectly reasonable setup. It results in a mail server that delivers email locally within the same domain as the local hostname and that sends any messages directed to nonlocal addresses directly to the appropriate remote servers.

Null client

Another simple configuration is a "null client"; that is, a system that doesn't deliver email locally but rather forwards outbound mail to a designated central server. To implement this configuration, you define several parameters, starting with `mydomain`, which defines the domain

part of the hostname, and myorigin, which is the mail domain appended to unqualified email addresses. If these two parameters are the same, you can write something like this:

```
mydomain = cs.colorado.edu
myorigin = $mydomain
```

Another parameter you should set is mydestination, which specifies the mail domains that are local. If the recipient address of a message has mydestination as its mail domain, the message is delivered through the **local** program to the corresponding user (assuming that no relevant alias or **.forward** file is found). If more than one mail domain is included in mydestination, these domains are all considered aliases for the same domain.

For a null client, you want no local delivery, so leave this parameter empty:

```
mydestination =
```

Finally, the relayhost parameter tells Postfix to send all nonlocal messages to a specified host instead of sending them directly to their apparent destinations:

```
relayhost = [mail.cs.colorado.edu]
```

The square brackets tell Postfix to treat the specified string as a hostname (DNS A record) instead of a mail domain name (DNS MX record).

Since null clients should not receive mail from other systems, the last thing to do in a null client configuration is to comment out the **smtpd** line in the **master.cf** file. This change prevents Postfix from running **smtpd** at all. With just these few lines, you've defined a fully functional null client!

For a "real" mail server, you'll need a few more configuration options as well as some mapping tables. We cover these in the next few sections.

*Use of **postconf***

postconf is a handy tool that helps you configure Postfix. When run without arguments, it prints all the parameters as they are currently configured. If you name a specific parameter as an argument, **postconf** prints the value of that parameter. The **-d** option makes **postconf** print the defaults instead of the currently configured values. For example:

```
$ postconf mydestination
mydestination =
$ postconf -d mydestination
mydestination = $myhostname, localhost.$mydomain, localhost
```

Another useful option is **-n**, which tells **postconf** to print only the parameters that differ from the default. If you ask for help on the Postfix mailing list, that's the configuration information you should put in your email.

Lookup tables

Many aspects of Postfix's behavior are shaped through the use of lookup tables, which can map keys to values or implement simple lists. For example, the default setting for the alias_maps table is

```
alias_maps = dbm:/etc/mail/aliases
```

Data sources are specified with the notation *type:path*. Multiple values can be separated by commas, spaces, or both. Table 18.19 lists the available data sources; **postconf -m** shows this information as well.

Table 18.19: Information sources for Postfix lookup tables

Type	Description
dbm/sdbm	Legacy **dbm** or **gdbm** database file
cidr	Network addresses in CIDR form
hash/btree	Berkeley DB hash table or B-tree file (replaces **dbm**)
ldap	LDAP directory service
mysql	MySQL database
pcre	Perl-compatible regular expressions
pgsql	PostgreSQL database
proxy	Access through **proxymap**, e.g., to escape a **chroot**
regexp	POSIX regular expressions
static	Return of value specified as *path* regardless of the key
unix	The **/etc/passwd** and **/etc/group** files[a]

a. unix:passwd.byname is the **passwd** file, and unix:group.byname is the **group** file.

The dbm and sdbm types are only for compatibility with the traditional **sendmail** alias table. Berkeley DB (hash) is a more modern implementation; it's safer and faster. If compatibility is not a problem, then go with

```
alias_database = hash:/etc/mail/aliases
alias_maps = hash:/etc/mail/aliases
```

The alias_database specifies the table that is rebuilt by **newaliases** and should correspond to the table that you specify in alias_maps. The two parameters are separate because alias_maps might include non-DB sources such as mysql that never need to be rebuilt.

All DB-class tables (dbm, sdbm, hash, and btree) compile a text file to an efficiently searchable binary format. The syntax for these text files is similar to that of the configuration files with respect to comments and continuation lines. Entries are specified as simple key/value pairs separated by whitespace, except for alias tables, which use a colon after the key to retain **sendmail** compatibility. For example, the following lines are appropriate for an alias table:

```
postmaster:  david, tobias
webmaster:   evi
```

As another example, here's an access table for relaying mail from any client with a hostname ending in cs.colorado.edu.

```
.cs.colorado.edu     OK
```

Text files are compiled to their binary formats with the **postmap** command for normal tables and the **postalias** command for alias tables. The table specification (including the type) must be given as the first argument. For example:

```
$ sudo postmap hash:/etc/postfix/access
```

postmap can also query values in a lookup table (no match = no output):

```
$ postmap -q blabla hash:/etc/postfix/access
$ postmap -q .cs.colorado.edu hash:/etc/postfix/access
OK
```

Local delivery

The **local** program delivers mail to local recipients. It also handles local aliasing. For example, if mydestination is set to cs.colorado.edu and email arrives for the recipient evi@cs.colorado.edu, **local** first consults the alias_maps tables and then substitutes any matching entries recursively.

If no aliases match, **local** looks for a **.forward** file in user evi's home directory and follows the instructions in this file if it exists. (The syntax is the same as for the right side of an alias map.) Finally, if no **.forward** file is found, the email is delivered to evi's local mailbox.

By default, **local** writes to standard **mbox**-format files under **/var/mail**. You can change that behavior with the parameters shown in Table 18.20.

Table 18.20: Parameters for local mailbox delivery (set in main.cf)

Parameter	Description
home_mailbox	Delivers mail to ~*user* under the specified relative path
mail_spool_directory	Delivers mail to a central directory that serves all users
mailbox_command	Delivers mail with an external program, typically **procmail**
mailbox_transport	Delivers mail through a service as defined in **master.cf**[a]
recipient_delimiter	Allows extended usernames (see description below)

a. This option interfaces with mailbox servers such as the Cyrus **imapd**.

The mail_spool_directory and home_mailbox options normally generate **mbox**-format mailboxes, but they can also produce **Maildir** mailboxes. To request this behavior, add a slash to the end of the

pathname.

If recipient_delimiter is +, mail addressed to evi+*whatever*@cs.colorado.edu is accepted for delivery to the evi account. With this facility, users can create special-purpose addresses and sort their mail by destination address. Postfix first attempts lookups on the full address, and only if that fails does it strip the extended components and fall back to the base address. Postfix also looks for a corresponding forwarding file, **.forward+***whatever*, for further aliasing.

Virtual domains

To host a mail domain on your Postfix mail server, you have three choices:

- List the domain in mydestination. Delivery is performed as described above: aliases are expanded and mail is delivered to the corresponding accounts.
- List the domain in the virtual_alias_domains parameter. This option gives the domain its own addressing namespace that is independent of the system's user accounts. All addresses within the domain must be resolvable (through mapping) to real addresses outside of it.
- List the domain in the virtual_mailbox_domains parameter. As with the virtual_alias_domains option, the domain has its own namespace. All mailboxes must live beneath a specified directory.

List the domain in only one of these three places. Choose carefully, because many configuration elements depend on that choice. We have already reviewed the handling of the mydestination method. The other options are discussed below.

Virtual alias domains

If a domain is listed as a value of the virtual_alias_domains parameter, mail to that domain is accepted by Postfix and must be forwarded to an actual recipient either on the local machine or elsewhere.

The forwarding for addresses in the virtual domain must be defined in a lookup table included in the virtual_alias_maps parameter. Entries in the table have the address in the virtual domain on the left side and the actual destination address on the right. An unqualified name on the right is interpreted as a local username.

Consider the following example from **main.cf**:

```
myorigin = cs.colorado.edu
mydestination = cs.colorado.edu
virtual_alias_domains = admin.com
virtual_alias_maps = hash:/etc/mail/admin.com/virtual
```

In **/etc/mail/admin.com/virtual** we could then have the lines

```
postmaster@admin.com    evi, david@admin.com
david@admin.com         david@schweikert.ch
evi@admin.com           evi
```

Mail for evi@admin.com would be redirected to evi@cs.colorado.edu (myorigin is appended) and would ultimately be delivered to the mailbox of user evi because cs.colorado.edu is included in mydestination.

Definitions can be recursive: the right hand side can contain addresses that are further defined on the left hand side. Note that the right hand side can only be a list of addresses. To execute an external program or to use :include: files, redirect the email to an alias, which can then be expanded according to your needs.

To keep everything in one file, set virtual_alias_domains to the same lookup table as virtual_alias_maps and put a special entry in the table to mark it as a virtual alias domain. In **main.cf**:

```
virtual_alias_domains = $virtual_alias_maps
virtual_alias_maps = hash:/etc/mail/admin.com/virtual
```

In **/etc/mail/admin.com/virtual**:

```
admin.com               notused
postmaster@admin.com    evi, david@admin.com
...
```

The right hand side of the entry for the mail domain (admin.com) is never actually used; admin.com's existence in the table as an independent entry is enough to make Postfix consider it a virtual alias domain.

Virtual mailbox domains

Domains listed under virtual_mailbox_domains are similar to local domains, but the list of users and their corresponding mailboxes must be managed independently of the system's user accounts.

The parameter virtual_mailbox_maps points to a table that lists all valid users in the domain. The map format is

```
user@domain     /path/to/mailbox
```

If the path ends with a slash, the mailboxes are stored in **Maildir** format. The value of virtual_mailbox_base is always prefixed to the specified paths.

You often want to alias some of the addresses in the virtual mailbox domain. A virtual_alias_map will do that for you. Here is a complete example. In **main.cf**:

```
virtual_mailbox_domains = admin.com
virtual_mailbox_base = /var/mail/virtual
virtual_mailbox_maps = hash:/etc/mail/admin.com/vmailboxes
virtual_alias_maps = hash:/etc/mail/admin.com/valiases
```

/etc/mail/admin.com/vmailboxes might contain entries like these:

```
evi@admin.com           nemeth/evi/
```

/etc/mail/admin.com/valiases might contain:

```
postmaster@admin.com    evi@admin.com
```

You can use virtual alias maps even on addresses that are not within virtual alias domains. Virtual alias maps let you redirect any address from any domain, independently of the type of the domain (canonical, virtual alias, or virtual mailbox). Since mailbox paths can only be put on the right hand side of the virtual mailbox map, this mechanism is the only way to set up aliases in that domain.

Access control

Mail servers should relay mail for third parties only on behalf of trusted clients. If a mail server forwards mail from unknown clients to other servers, it is a so-called open relay, which is bad. See this page for more details.

Fortunately, Postfix doesn't act as an open relay by default. In fact, its defaults are quite restrictive; you are more likely to need to liberalize the permissions than to tighten them. Access control for SMTP transactions is configured in Postfix through "access restriction lists." The parameters shown in Table 18.21 control what should be checked during the different phases of an SMTP session.

Table 18.21: Postfix parameters for SMTP access restriction

Parameter	When applied
`smtpd_client_restrictions`	On connection request
`smtpd_data_restrictions`	On DATA command (mail body)
`smtpd_etrn_restrictions`	On ETRN command[a]
`smtpd_helo_restrictions`	On HELO/EHLO command (start of the session)
`smtpd_recipient_restrictions`	On RCPT TO command (recipient specification)
`smtpd_relay_restrictions`	On relay attempt to a third party domain
`smtpd_sender_restrictions`	On MAIL FROM command (sender specification)

a. This is a special command used for resending messages in the queue.

The most important parameter is `smtpd_recipient_restrictions`. That's because access control is most easily performed when the recipient address is known and can be identified as being local or not. All other parameters in Table 18.21 are empty in the default configuration. The default value is

```
smtpd_recipient_restrictions = permit_mynetworks,
    reject_unauth_destination
```

Each of the specified restrictions is tested in turn until a definitive decision about what to do with the mail is reached. Table 18.22 shows the common restrictions.

Table 18.22: Common Postfix access restrictions

Restriction	Function
`check_client_access`	Checks client host address through a lookup table
`check_recipient_access`	Checks recipient mail address through a lookup table
`permit_mynetworks`	Grants access to addresses listed in `mynetworks`
`reject_unauth_destination`	Rejects mail for nonlocal recipients; no relaying

Everything can be tested in these restrictions, not just specific information like the sender address in the smtpd_sender_restrictions. Therefore, for simplicity, you might want to put all the restrictions under a single parameter. Make that smtpd_recipient_restrictions because it is the only one that can test everything (except the DATA part).

smtpd_recipient_restrictions and smtpd_relay_restrictions are where mail relaying is tested. Keep the reject_unauth_destination restriction and carefully choose the "permit" restrictions before it.

Access tables

Each restriction returns one of the actions shown in Table 18.23. Access tables are used in restrictions such as check_client_access and check_recipient_access to select an action according to the client host address or recipient address, respectively.

Table 18.23: Actions for access tables

Action	Meaning
4*nn* *text*	Returns temporary error code *4nn* and message *text*
5*nn* text	Returns permanent error code *5nn* and message *text*
DEFER_IF_PERMIT	If restrictions result in PERMIT, changes it to a temp error
DEFER_IF_REJECT	If restrictions result in REJECT, changes it to a temp error
DISCARD	Accepts the message but silently discards it
DUNNO	Pretends the key was not found; tests further restrictions
FILTER *transport:dest*	Passes the mail through the filter *transport:dest*
HOLD	Blocks the mail in the queue
OK	Accepts the mail
PREPEND *header*	Adds a header to the message
REDIRECT *addr*	Forwards the mail to a specified address
REJECT	Rejects the mail
WARN *message*	Enters the given warning *message* in the logs

For example, suppose you wanted to allow relaying for all machines within the cs.colorado.edu domain and that you wanted to allow only trusted clients to post to the internal mailing list newsletter@cs.colorado.edu. You could implement these policies with the following lines in **main.cf**:

```
smtpd_recipient_restrictions =
    permit_mynetworks
    check_client_access hash:/etc/postfix/relaying_access
    reject_unauth_destination
    check_recipient_access hash:/etc/postfix/restricted_recipients
```

Note that commas are optional when the list of values for a parameter is specified.

In **/etc/postfix/relaying_access**:

```
.cs.colorado.edu              OK
```

In **/etc/postfix/restricted_recipients**:

```
newsletter@cs.colorado.edu   REJECT Internal list
```

The text after REJECT is an optional string that is sent to the client along with the error code. It tells the sender why the mail was rejected.

Authentication of clients and encryption

For users sending mail from home, it is usually easiest to route outgoing mail through the home ISP's mail server, regardless of the sender address that appears on that mail. Most ISPs trust their direct clients and allow relaying. If this configuration isn't possible or if you are using a system such as Sender ID or SPF, ensure that mobile users outside your network can be authorized to submit messages to your **smtpd**.

The solution to this problem is to have the SMTP AUTH mechanism authenticate directly at the SMTP level. Postfix must be compiled with support for the SASL library to make this work. You can then configure the feature like this:

```
smtpd_sasl_auth_enable = yes
smtpd_recipient_restrictions =
    permit_mynetworks
    permit_sasl_authenticated
    ...
```

You also need to support encrypted connections to avoid sending passwords in clear text. Add lines like the following to **main.cf**:

```
smtpd_tls_security_level = may
smtpd_tls_auth_only = yes
smtpd_tls_loglevel = 1
smtpd_tls_received_header = yes
smtpd_tls_cert_file = /etc/certs/smtp.pem
smtpd_tls_key_file = $smtpd_tls_cert_file
smtpd_tls_protocols = !SSLv2
```

You need to put a properly signed certificate in **/etc/certs/smtp.pem**. It's also a good idea to turn on encryption on outgoing SMTP connections:

```
smtp_tls_security_level = may
smtp_tls_loglevel = 1
```

Debugging

When you have a problem with Postfix, first check the log files. The answers to your questions are most likely there; it's just a question of finding them. Every Postfix program normally issues a log entry for every message it processes. For example, the trail of an outbound message might look like this:

```
Aug 18 22:41:33 nova postfix/pickup: 0E4A93688: uid=506
    from=<dws@ee.ethz.ch>
Aug 18 22:41:33 nova postfix/cleanup: 0E4A93688:
    message-id= <20040818204132.GA11444@ee.ethz.ch>
Aug 18 22:41:33 nova postfix/qmgr: 0E4A93688:
    from=<dws@ee.ethz.ch>, size=577,nrcpt=1 (queue active)
Aug 18 22:41:33 nova postfix/smtp: 0E4A93688:
    to=<evi@ee.ethz.ch>,relay=tardis.ee.ethz.ch[129.132.2.217],
    delay=0, status=sent (250 Ok: queued as 154D4D930B)
Aug 18 22:41:33 nova postfix/qmgr: 0E4A93688: removed
```

As you can see, the interesting information is spread over many lines. Note that the identifier 0E4A93688 is common to every line: Postfix assigns a queue ID as soon as a message enters the mail system and never changes it. Therefore, when searching the logs for the history of a message, first concentrate on determining the message's queue ID. Once you know that, it's easy to **grep** the logs for all the relevant entries.

Postfix is good at logging helpful messages about problems that it notices. However, it's sometimes difficult to spot the important lines among the thousands of normal status messages. This is a good place to consider using some of the tools discussed in the section *Management of logs at scale*.

Looking at the queue

Another place to look for problems is the mail queue. As in the **sendmail** system, a **mailq** command prints the contents of a queue. You can use it to see if and why a message has become stuck.

Another helpful tool is the **qshape** script that's shipped with recent versions of Postfix. It shows summary statistics about the contents of a queue. The output looks like this:

```
$ sudo qshape deferred
                  T  5  10  20  40  80  160  320  640  1280  1280+
TOTAL            78  0   0   0   7   3    3    2   12     2     49
expn.com         34  0   0   0   0   0    0    0    9     0     25
chinabank.ph      5  0   0   0   1   1    1    2    0     0      0
prob-helper.biz   3  0   0   0   0   0    0    0    0     0      3
```

qshape summarizes the given queue (here, the deferred queue), sorted by recipient domain. The columns report the number of minutes the relevant messages have been in the queue. For

example, you can see that 25 messages bound for expn.com have been in the queue longer than 1,280 minutes. All the destinations in this example are suggestive of messages having been sent from vacation scripts in response to spam.

qshape can also summarize by sender domain with the **-s** flag.

Soft-bouncing

If soft_bounce is set to yes, Postfix sends temporary error messages whenever it would normally send permanent error messages such as “user unknown” or “relaying denied.” This is a great testing feature; it lets you monitor the disposition of messages after a configuration change without the risk of permanently losing legitimate email. Anything you reject will eventually come back for another try. Don’t forget to turn off this feature when you are done testing or you will have to deal with every rejected message over and over again.

18.11 Recommended reading

Rather than jumble together the references listed here, we've sorted them by MTA and topic.

sendmail references

COSTALES, BRYAN, CLAUS ASSMANN, GEORGE JANSEN, AND GREGORY NEIL SHAPIRO. *sendmail, 4th Edition.* Sebastopol, CA: O'Reilly Media, 2007.

This book is the definitive tome for **sendmail** configuration—1,300 pages' worth. It includes a sysadmin guide as well as a complete reference section. An electronic edition is available, too. The author mix includes two key **sendmail** developers (Claus and Greg) who enforce technical correctness and add insight to the mix.

Installation instructions and a good description of the configuration file are covered in the *Sendmail Installation and Operation Guide*, which can be found in the **doc/op** subdirectory of the **sendmail** distribution. This document is quite complete, and in conjunction with the **README** file in the **cf** directory, gives a good nuts-and-bolts view of the **sendmail** system.

sendmail.org, sendmail.org/~ca, and sendmail.org/~gshapiro all contain documents, HOWTOs, and tutorials related to **sendmail**.

Exim references

HAZEL, PHILIP. *The Exim SMTP Mail Server: Official Guide for Release 4, 2nd Edition*. Cambridge, UK: User Interface Technologies, Ltd., 2007.

HAZEL, PHILIP. *Exim: The Mail Transfer Agent*. Sebastopol, CA: O'Reilly Media, 2001.

The Exim specification is the defining document for Exim configuration. It is quite complete and is updated with each new distribution. A text version is included in the file **doc/spec.txt** in the distribution, and a PDF version is available from exim.org. The web site also includes several how-to documents.

Postfix references

DENT, KYLE D. *Postfix: The Definitive Guide*. Sebastopol, CA: O'Reilly Media, 2003.

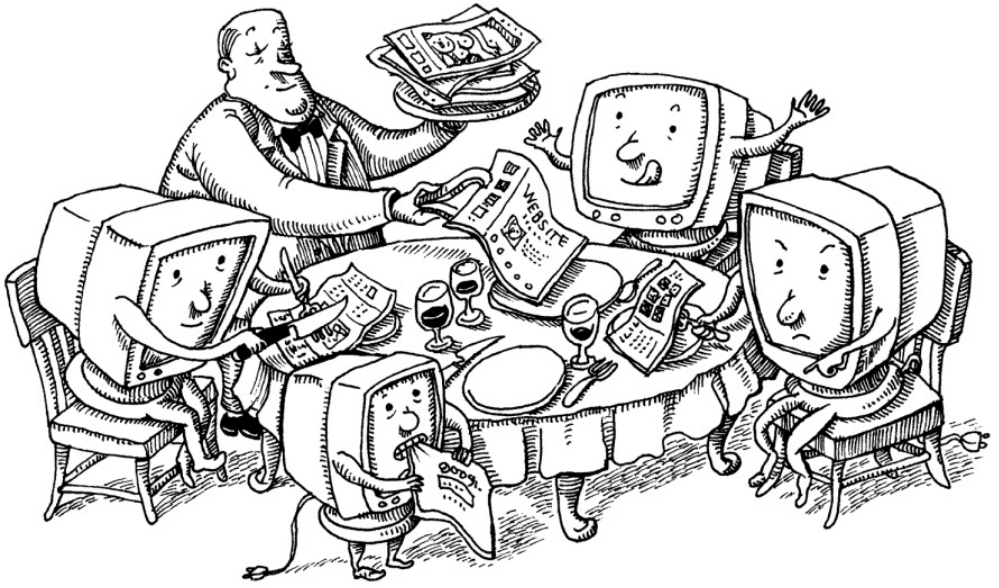
HILDEBRANDT, RALF, AND PATRICK KOETTER. *The Book of Postfix: State of the Art Message Transport.* San Francisco, CA: No Starch Press, 2005.

This book is the best; it guides you through all the details of Postfix configuration, even for complex environments. The authors are active in the Postfix community and participate regularly on the postfix-users mailing list. The book is unfortunately out of print, but used copies are readily available.

RFCs

RFCs 5321 (updated by 7504) and 5322 (updated by 6854) are the current versions of RFCs 821 and 822. They define the SMTP protocol and the formats of messages and addresses for Internet email. RFCs 6531 and 6532 cover extensions for internationalized email addresses. There are currently almost 90 email-related RFCs, too many to list here. See the general RFC search engine at rfc-editor.org for more.

19 Web Hosting



UNIX and Linux are the predominant platforms for serving web applications. According to data from w3techs.com, 67% of the top one million web sites are served by either Linux or FreeBSD. Above the OS level, open source web server software commands more than 80% of the market.

At scale, web applications do not run on a single system. Instead, a collection of software components distributed through a meshwork of systems cooperate to answer requests as quickly and as flexibly as possible. Each piece of this architecture must be resilient to server failures, load spikes, network partitions, and targeted attacks.

Cloud infrastructure helps address these needs. Its ability to provision capacity quickly in response to demand is an ideal match for the sudden and sometimes unexpected tidal waves of users that materialize on the web. In addition, cloud providers' add-on services include a variety of convenient recipes that meet common requirements, greatly simplifying the design, deployment, and operation of web systems.

19.1 HTTP: the Hypertext Transfer Protocol

HTTP is the core network protocol for communication on the web. Lurking beneath a deceptively simple facade of stateless requests and responses lie layers of refinements that bring both flexibility and complexity. A well-rounded understanding of HTTP is a core competency for all system administrators.

In its simplest form, HTTP is a client/server, one-request/one-response protocol. Clients, also called user agents, submit requests for resources to an HTTP server. Servers receive incoming requests and process them by retrieving files from local disks, resubmitting them to other servers, querying databases, or performing any number of other possible computations. A typical page view on the web entails dozens or hundreds of such exchanges.

As with most Internet protocols, HTTP has adapted over time, albeit slowly. The centrality of the protocol to the modern Internet makes updates a high-stakes proposition. Official revisions are a slog of committee meetings, mailing list negotiations, public review periods, and maneuvering by stakeholders with vested and conflicting interests. During the long gaps between official revisions documented in RFCs, unofficial protocol extensions are born from necessity, become ubiquitous, and are eventually included as features in the next specification.

HTTP versions 1.0 and 1.1 are sent over the wire in plain text. Adventurous administrators can interact with servers directly by running **telnet** or **netcat**. They can also observe and collect HTTP exchanges by using protocol-agnostic packet capture software such as **tcpdump**.

See this page for general information about TLS.

The web is in the process of adopting HTTP/2, a major protocol revision that preserves compatibility with previous versions but introduces a variety of performance improvements. In an effort to promote the universal use of HTTPS (secure, encrypted HTTP) for the next generation of the web, major browsers such as Firefox and Chrome have elected to support HTTP/2 only over TLS-encrypted connections.

HTTP/2 moves from plain text to binary format in an effort to simplify parsing and improve network efficiency. HTTP's semantics remain the same, but because the transmitted data is no longer directly legible to humans, generic tools such as **telnet** are no longer useful. The handy **h2i** command-line utility, part of the Go language networking repository at github.com/golang/net, helps restore some interactivity and debuggability to HTTP/2 connections. Many HTTP-specific tools such as **curl** also support HTTP/2 natively.

Uniform Resource Locators (URLs)

A URL is an identifier that specifies how and where to access a resource. URLs are not HTTP-specific; they are used for other protocols as well. For example, mobile operating systems use URLs to facilitate communication among apps.

You may sometimes see the acronyms URI (Uniform Resource Identifier) and URN (Uniform Resource Name) used as well. The exact distinctions and taxonomic relationships among URLs, URIs, and URNs are vague and unimportant. Stick with "URL."

The general pattern for URLs is *scheme:address*, where *scheme* identifies the protocol or system being targeted and *address* is some string that's meaningful within that scheme. For example, the URL mailto:ulsah@admin.com encapsulates an email address. If it's invoked as a link target on the web, most browsers will bring up a preaddressed window for sending mail.

For the web, the relevant schemes are http and https. In the wild, you might also see the schemes ws (WebSockets), wss (WebSockets over TLS), ftp, ldap, and many others.

The address portion of a web URL allows quite a bit of interior structure. Here's the overall pattern:

scheme://[*username:password*@]*hostname*[:*port*][/*path*][?*query*][#*anchor*]

All the elements are optional except *scheme* and *hostname*.

See this page for more details about HTTP basic authentication.

The use of a *username* and *password* in the URL enables "HTTP basic authentication," which is supported by most user agents and servers. In general, it's a bad idea to embed passwords into URLs because URLs are apt to be logged, shared, bookmarked, visible in **ps** output, etc. User agents can get their credentials from a source other than the URL, and that is typically a better option. In a web browser, just leave the credentials out and let the browser prompt you for them separately.

HTTP basic authentication is not self-securing, which means that the password is accessible to anyone who listens in on the transaction. Therefore, basic authentication should really only be used over secure HTTPS connections.

The *hostname* can be a domain name or IP address as well as an actual hostname. The *port* is the TCP port number to connect to. The http and https schemes default to ports 80 and 443, respectively.

The *query* section can include multiple parameters separated by ampersands. Each parameter is a *key=value* pair. For example, Adobe InDesign users may find the following URL eerily familiar:

http://adobe.com/search/index.cfm?term=indesign+crash&loc=en_us

As with passwords, sensitive data should never appear as a URL query parameter because URL paths are often logged as plain text. The alternative is to transmit parameters as part of the request body. (You can't really control this in other people's web software, but you can make sure your own site behaves properly.)

The *anchor* component identifies a subtarget of a specific URL. For example, Wikipedia uses named anchors extensively as section headings, allowing specific parts of an entry to be linked to directly.

Structure of an HTTP transaction

HTTP requests and responses are similar in structure. After an initial line, both include a sequence of headers, a blank line, and finally, the body of the message, called the payload.

HTTP requests

The first line of a request specifies an action for the server to perform. It consists of a request method (also known as the verb), a path on which to perform the action, and the HTTP version to use. For example, a request to retrieve a top-level HTML page might look like this:

```
GET /index.html HTTP/1.1
```

Table 19.1 shows the common HTTP request methods. Verbs marked as "safe" should not change the server's state. However, this is more a convention than a mandate. It's ultimately up to the software that handles the request to decide how to interpret the verb.

Table 19.1: HTTP request methods

Verb	Safe?	Purpose
GET	Yes	Retrieves the specified resource
HEAD	Yes	Like GET, but requests no payload; retrieves metadata only
DELETE	No	Deletes the specified resource
POST	No	Applies request data to the given resource
PUT	No	Similar to POST, but implies replacement of existing contents
OPTIONS	Yes	Shows what methods the server supports for the specified path

GET is by far the most commonly used HTTP verb, followed by POST. REST APIs, discussed in *Application programming interfaces (APIs)*, are more likely to employ the more exotic verbs such as PUT and DELETE.

The distinction between POST and PUT is subtle and largely of concern to web API developers. PUTs should be idempotent, meaning that a PUT can be repeated without causing ill effects. For example, a transaction that causes the server to send email should not be represented as a PUT. The rules for HTTP caching also differ significantly between PUT and POST. See RFC2616 for more details.

HTTP responses

The initial line in a response, called the status line, indicates the disposition of the request. It looks like this:

```
HTTP/1.1 200 OK
```

The important part is the three-digit numeric status code. The phrase that follows it is a helpful English translation that software ignores.

The first digit in the code determines its class; that is, the general nature of the result. Table 19.2 shows the five defined classes. Within a class, additional detail is provided by the remaining two digits. More than 60 status codes are defined, but only a few of these are commonly seen in the wild.

Table 19.2: HTTP response classes

Code	General indication	Examples
1xx	Request received; processing continues	101 Switching Protocols
2xx	Success	200 OK 201 Created
3xx	Further action needed	301 Moved Permanently 302 Found [a]
4xx	Unsatisfiable request	403 Forbidden 404 Not Found
5xx	Server or environment failure	503 Service Unavailable

a. Most often used (inappropriately, according to the spec) for temporary redirects

Headers and the message body

Headers specify metadata about a request or response, such as whether to allow compression; what types of content are accepted, expected, or provided; and how intermediate caches should handle the data. For requests, the only required header is Host, which is used by web server software to determine which site is being contacted.

Table 19.3 shows some common headers.

Table 19.3: Commonly encountered HTTP headers

Name: example value	Dir[a]	Content
Host: www.admin.com	→	Domain name and port being requested
Content-Type: application/json	↔	Data format wanted or contained
Authorization: Basic QWx...FtZ==	→	Credentials for HTTP basic authentication
Last-Modified: Wed, Sep 7 2016...	←	Object's last known modification date
Cookie: flavor=oatmeal	→	Cookie returned from a user agent
Content-Length: 423	↔	Length of the body in bytes
Set-Cookie: flavor=oatmeal	←	Cookie to be stored by the user agent
User-Agent: curl/7.37.1	→	User agent submitting the request
Server: nginx/1.6.2	←	Server software responding to the request
Upgrade: HTTP/2.0	↔	Request to change to another protocol
Expires: Sat, 15 Oct 2016 14:02:...	←	Length of time the response can be cached
Cache-Control: max-age=7200	↔	Like Expires, but allows more control

a. Direction: → request-only, ← response-only, or ↔ both

Table 19.3 is by no means a definitive list. In fact, both sides of the transaction can include any headers they wish. Both sides must ignore headers they don't understand. By convention, custom and experimental headers were originally prefixed with "X-". But some X- headers (such as X-Forwarded-For) became de facto standards, and it was then infeasible to remove the prefix because that would break compatibility. The use of X- is now deprecated by RFC6648.

Headers are separated from the message body by a blank line. For requests, the body can include parameters (for POST or PUT requests) or the contents of a file to upload. For responses, the message body is the payload of the resource being requested (e.g., HTML, image data, or query results). The message body is not necessarily human-readable, since it can contain images or other binary data. The body can also be empty, as for GET requests or most error responses.

curl: HTTP from the command line

curl (cURL) is a handy command-line HTTP client that's available for most platforms. (Administrators will also encounter **libcurl**, a client library that developers can use to give their own software **curl**-like superpowers.) Here, we use **curl** to explore an HTTP exchange.

Below is an invocation of **curl** that requests the root of the web site admin.com on TCP port 80, which is the default for unencrypted (non-HTTPS) requests. The response payload (i.e., the admin.com homepage) and some informative messages from **curl** itself have been hidden by the **-o /dev/null** and **-s** flags. We also include the **-v** flag to request that **curl** display verbose output, which includes headers.

```
$ curl -s -v -o /dev/null http://admin.com
* Rebuilt URL to: http://admin.com/
* Hostname was NOT found in DNS cache
*   Trying 54.84.253.153...
* Connected to admin.com (54.84.253.153) port 80 (#0)
> GET / HTTP/1.1
> User-Agent: curl/7.37.1
> Host: admin.com
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Mon, 27 Apr 2015 18:17:08 GMT
* Server Apache/2.4.7 (Ubuntu) is not blacklisted
< Server: Apache/2.4.7 (Ubuntu)
< Last-Modified: Sat, 02 Feb 2013 03:08:20 GMT
< ETag: "66d3-4d4b52c0c1100"
< Accept-Ranges: bytes
< Content-Length: 26323
< Vary: Accept-Encoding
< Content-Type: text/html
<
{ [2642 bytes data]
* Connection #0 to host admin.com left intact
```

Lines starting with > and < denote the request and response, respectively. In the request, the client tells the server that the user agent is **curl**, that it's looking for host admin.com, and that it will accept any type of content as a response. The server identifies itself as Apache 2.4.7 and replies with contents of type HTML, along with a variety of other metadata.

We can set headers explicitly with **curl**'s **-H** argument. This feature is especially handy for making requests directly against IP addresses, bypassing DNS. For example, we could check that the server for www.admin.com responds identically to requests targeted at admin.com by setting the Host header, which informs the remote server of the domain the user agent is attempting to contact:

```
$ curl -H "Host: www.admin.com" -s -v -o /dev/null 54.84.253.153
<same output as the previous example, but with a different Host header>
```

We use the **-O** argument to download a file. This example downloads a tarball of the **curl** source code to the current directory:

```
$ curl -O http://curl.haxx.se/snapshots/curl-7.46.0-20151105.tar.gz
```

We've only scratched the surface of **curl**'s capabilities. It can handle other request methods such as POST and DELETE, store and submit cookies, download files, and assist with many different debugging scenarios.

Google's Chrome browser offers a feature called "Copy as cURL" that creates a **curl** command to simulate the browser's own behavior, including headers, cookies, and other details. You can easily retry requests with various adjustments and see the results exactly as the browser would. (Right-click a resource name in the Network tab of the developer tools panel to uncover this option.)

TCP connection reuse

TCP connections are expensive. In addition to the memory needed to maintain them, the three-way handshake used to establish each new connection adds latency equivalent to a full round trip before an HTTP request can even begin. (TCP Fast Open is a proposal that aims to improve this situation by allowing the SYN and SYN-ACK packets of TCP's three-way handshake to also carry data. See RFC7413.)

The HTTP Archive, a project that tracks web statistics, estimates that the average site incurs requests for 99 resources per page load. If each resource required a new TCP connection, the performance of the web would be atrocious indeed. This was in fact the case early in the life of the web.

The original HTTP/1.0 specification did not include any provisions for connection reuse, but some adventurous developers added experimental support as an extension. The Connection: Keep-Alive header was added informally to clients and servers, then improved and made the default in HTTP/1.1. With keep-alive (also known as persistent) connections, HTTP clients and servers send multiple requests over a single connection, thus saving some of the cost and latency of initiating and tearing down multiple connections.

TCP overhead turns out to be nontrivial even when HTTP/1.1 persistent connections are enabled. Most browsers open as many as six parallel connections to the server to improve performance. Busy servers in turn must maintain many thousands of TCP connections in various states, resulting in network congestion and wasted resources.

HTTP/2 introduces multiplexing as a solution, allowing several transactions to be interleaved on a single connection. HTTP/2 servers can therefore support more clients per system, since each client imposes lower overhead.

HTTP over TLS

On its own, HTTP provides no network-level security. The URL, headers, and payload are open to inspection and modification at any point between the client and server. A malevolent infiltrator can intercept messages, alter their contents, or redirect requests to servers of its choice.

Enter Transport Layer Security (TLS), which runs as a separate layer between TCP and HTTP. TLS supplies only the security and encryption for the connection; it does not involve itself at the HTTP layer.

The precursor of TLS was known as SSL, the Secure Sockets Layer. All versions of SSL are obsolete and formally deprecated, but the name SSL remains in wide colloquial use. Outside of cryptographic contexts, assume that references to SSL really mean TLS.

The user agent verifies the server's identity as part of the TLS connection process, eliminating the possibility of spoofing by counterfeit servers. Once the connection is established, its contents are protected against snooping and modification for the duration of the exchange. Attackers can still see the host and port used at the TCP layer, but they cannot access HTTP details such as the URL of a request or the headers that accompany it.

See *Transport Layer Security* for more details on TLS cryptography.

Virtual hosts

In the early days of the web, a server typically hosted only a single web site. When admin.com was requested, for example, clients performed a DNS lookup to find the IP address associated with that name and then sent an HTTP request to port 80 at that address. The server at that address knew that it was dedicated to hosting admin.com and served results accordingly.

As web use increased, administrators realized that they could achieve economies of scale if a single server could host more than one site at once. But how do you distinguish requests bound for admin.com from those bound for example.com if both kinds of traffic end up at the same network port?

One possibility is to define virtual network interfaces, effectively permitting several different IP addresses to be bound to a single physical connection. Most systems allow this, and it works fine, but the scheme is fiddly and requires management at several different layers.

A better solution, virtual hosts, was provided by HTTP 1.1 in RFC2616. This scheme defines a Host HTTP header that user agents set explicitly to indicate what site they're attempting to contact. Servers examine the header and behave accordingly. This convention conserves IP addresses and simplifies management, especially for sites that have hundreds or thousands of web sites on a single server.

HTTP 1.1 *requires* user agents to provide a Host header, so virtual hosts are now the standard way that web servers and administrators handle server consolidation.

The use of name-based virtual hosts in combination with TLS is a bit tricky under the hood. TLS certificates are issued to specific hostnames, which are chosen when the certificate is generated. A TLS connection must be established before the server can read the Host header from the HTTP request, but without that header, the server does not know which virtual host it should be impersonating, and hence, which certificate to select.

The solution is SNI, Server Name Indication, in which the client submits the hostname that it's requesting as part of the initial TLS connection message. Modern servers and clients all handle SNI automatically.

19.2 Web software basics

A rich library of open source software facilitates the construction of flexible, resilient web applications. Table 19.4 lists a few general categories of services that speak HTTP and perform specific functions within the web application stack.

Table 19.4: Partial list of HTTP server types

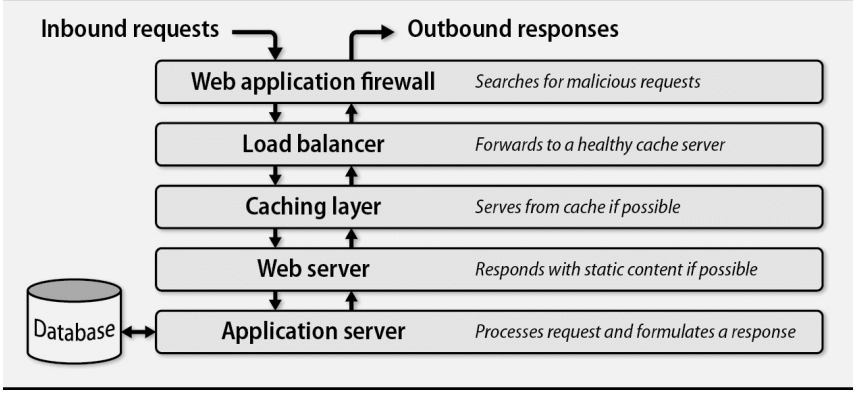
Type	Purpose	Examples
Application server	Runs web app code, interfaces to web servers	Unicorn, Tomcat
Cache	Speeds access to frequently requested content	Varnish, Squid
Load balancer	Relays requests to downstream systems	Pound, HAProxy
Web app firewall [a]	Inspects HTTP traffic for common attacks	ModSecurity
Web server	Serves static content, couples to other servers	Apache, NGINX

a. Often abbreviated WAF

A web proxy is an intermediary that receives HTTP requests from clients, optionally performs some processing, and relays the requests to their ultimate destination. Proxies are normally transparent to clients. Load balancers, web application firewalls, and cache servers are all specialized types of proxy servers. A web server also acts as a sort of proxy if it relays requests to application servers.

Exhibit A illustrates the role that each service plays in an HTTP exchange. Requests can be fulfilled higher in the stack if the requested resource can be satisfied, or rejected with a 4xx or 5xx code if a problem occurs. Requests that require a query to the database traverse every layer.

Exhibit A: Components of a web application stack



To maximize availability, each layer should run on more than one node simultaneously. Ideally, redundancy should span geographical regions so that the overall design is not dependent on any single physical data center. This goal is a lot easier to achieve when you build on a cloud platform that offers well-defined geographic regions as a fundamental building block (as most do).

Real-world architectures aren't usually as straightforward as Exhibit A suggests. In addition, most web software components perform functions in more than one area. NGINX is best known as a web server, for example, but it's also a highly capable cache and load balancer. An NGINX web server with caching features enabled is more efficient than a stack of separate servers running on individual virtual machines.

Web servers and HTTP proxy software

Most sites use web servers either to proxy HTTP connections to application servers or to serve static content directly. A few of the features provided by web servers include

- Virtual hosts, allowing many sites to coexist peacefully within a single server
- Handling of TLS connections
- Configurable logging that tracks requests and responses
- HTTP basic authentication
- Routing to different downstream systems according to requested URLs
- Execution of dynamic content through application servers

The leading open source web servers are the Apache HTTP Server, known colloquially as **httpd**, and NGINX, which is pronounced “engine X.”

Netcraft, an English Internet research and security company, publishes monthly market share statistics for web servers. As of June 2017, Netcraft shows that around 46% of active web sites run Apache. NGINX accounts for 20% and has been steadily rising since 2008.

Apache **httpd** is the original project from the Apache Software Foundation, now known for supporting a variety of excellent open source projects. **httpd** has been under active development since 1995 and is widely regarded as the reference HTTP server implementation.

NGINX is a versatile server designed for speed and efficiency. Like **httpd**, NGINX supports service of static web content, load balancing, monitoring of downstream servers, proxying, caching, and other related functions.

Some development systems, notably Node.js and the language Go, implement web servers internally and can handle many HTTP workflows without the need for a separate web server. These systems incorporate sophisticated connection management features and are robust enough for production workloads.

The H2O server (h2o.examp1e.net; note the numeral one in “examp1e”) is a newer web server project that takes full advantage of HTTP/2 features and achieves even better performance than NGINX. Because it was first released in 2014, it can’t claim the track record of Apache or NGINX. On the other hand, neither is it constrained by historical implementation decisions as is **httpd**. It’s certainly worth a look for new deployments.

It’s hard to make strong recommendations among these options because they’re all quite good. That said, for mainstream production use, we suggest NGINX. It offers exceptional performance and a relatively simple and modern configuration system.

Load balancers

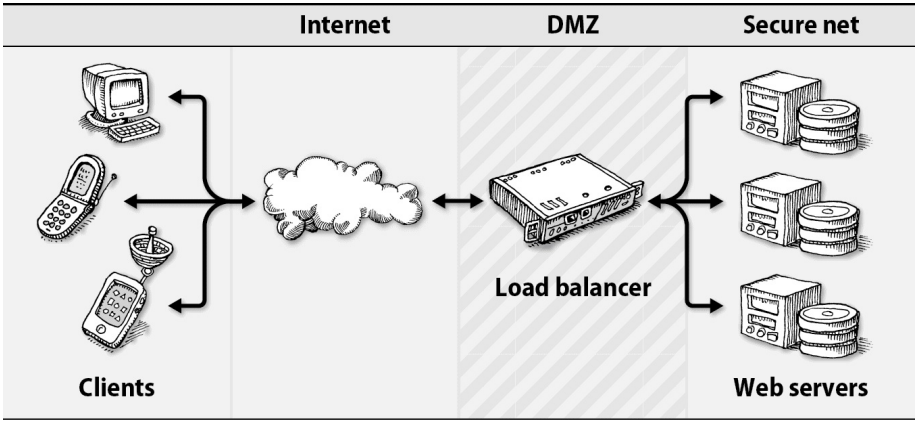
You can't run a highly available web site on a single server. Not only does this configuration expose your users to every potential hiccup experienced by the server, but it also gives you no way to update the software, operating system, or configuration without downtime.

Single servers are also exquisitely vulnerable to load spikes and intentional attacks. The more overloaded a server becomes, the more time it spends thrashing instead of getting useful work done. Past a certain load threshold (one that you will have to discover through bitter experience!), performance craters abruptly rather than degrading gracefully.

To avoid these problems, you can use a load balancer, which is a type of proxy server that distributes incoming requests among a set of downstream web servers. Load balancers also monitor the status of those servers to ensure that they are providing timely and correct responses.

Exhibit B shows the placement of load balancers in an architecture diagram.

Exhibit B: The role of a load balancer



Load balancers solve many of the problems inherent in a single-system design:

- Load balancers do not process requests but merely route them to other systems. As a result, they can handle many more concurrent requests than does a typical web server.
- When a web server needs a software upgrade or has to be taken off-line for any other reason, it can easily be removed from the rotation.
- If one of the servers experiences a problem, the health-check mechanism on the load balancer detects the problem and removes the errant system from the server pool until it becomes healthy again.

To avoid becoming single points of failure themselves, load balancers usually run in pairs. Depending on the configuration, one balancer might act as a passive backup while the other serves live traffic, or both balancers might serve requests simultaneously.

The way that requests are distributed is usually configurable. Here are a few common algorithms:

- Round robin, in which requests are distributed among the active servers in a fixed rotation order
- Load equalization, in which new requests go to the downstream server that's currently handling the smallest number of connections or requests
- Partitioning, in which the load balancer selects a server according to a hash of the client's IP address. This method ensures that requests from the same client always reach the same web server.

Load balancers normally operate at layer four of the OSI model, in which they route requests based just on the IP address and port of the request. However, they can also operate at layer seven by inspecting requests and routing them according to their target URL, cookie values, or other HTTP headers. For example, requests for example.com/linux might route to a separate set of servers than do requests for example.com/bsd.

See this page for more information about network DMZs (demilitarized zones).

As an added bonus, load balancers can improve security. They usually reside in the DMZ portion of a network and proxy requests to web servers behind an internal firewall. If HTTPS is in use, they also perform TLS termination: the connection from the client to the load balancer uses TLS, but the connection from the load balancer to the web server can be vanilla HTTP. This arrangement offloads some processing overhead from the web servers.

Load balancers can distribute other kinds of traffic in addition to (or instead of) HTTP. A common use is to add a load balancer that distributes requests to databases such as MySQL or Redis.

The most common open source load balancers for UNIX and Linux are NGINX, already introduced as a web server, and HAProxy, a high-performance TCP and HTTP proxy beloved by veteran administrators for its flexible configuration, stability, and robust performance. Both are excellent and well documented, and both have large user communities. (Apache **httpd** also has a load-balancing module, though we haven't seen it used as widely.)

Commercial load balancers such as those from F5 and Citrix are available both as hardware devices to be installed in a data center and as software solutions. They typically offer a graphical configuration interface, more features than open source tools, extra functions in addition to straightforward load balancing, and hefty price tags.

Amazon offers a dedicated load-balancing service, the Elastic Load Balancer (ELB), for use with EC2 virtual machines. ELB is a completely managed service; no virtual machine is required for the load balancer itself. ELB handles an extremely large number of concurrent connections and can balance traffic among multiple availability zones.

In ELB terminology, a “listener” accepts connections from clients and proxies them to back-end EC2 instances that do the actual work. Listeners can proxy TCP, HTTP, or HTTPS traffic. The load is distributed according to the “least connections” algorithm.

ELB is not the most fully featured load balancer, but it is our recommended solution for AWS-hosted systems because it requires virtually no administrative attention.

Caches

Web caches were born from the observation that clients often repeatedly access the same content within a short time. Caches live between clients and web servers and store the results of the most frequent requests, sometimes in memory. They can then intervene to answer requests for which they know the correct response, reducing load on the authoritative web servers and improving response times for users.

In caching jargon, an origin is the original content provider, the source of truth about the content. Caches get their content directly from the origin or from another upstream cache.

Several factors determine caching behavior:

- The values of HTTP headers, including Cache-Control, ETag, and Expires
- Whether the request is served by HTTPS, for which caching is more nuanced
- The response status code; some are not cacheable (see RFC2616)
- The contents of HTML <meta> tags (not respected by all caches)

Static blobs such as images, videos, CSS stylesheets, and JavaScript files are well suited to caching because they rarely change. Dynamic content loaded from a database or another system in real time is more difficult—but not necessarily impossible—to cache.

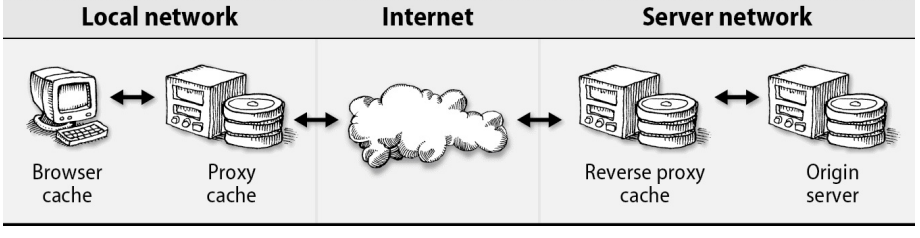
Because HTTPS payloads are encrypted, responses cannot be cached unless the cache server terminates the TLS connection, decrypting the payload. The connection from the cache server to the origin may then require a separately encrypted TLS connection (or not, depending on whether the connection between the two is trusted).

For highly variable pages that should never be cached, developers set the following HTTP header:

```
Cache-Control: no-cache, no-store
```

Exhibit C shows the placement of several potential cache layers in an HTTP request.

Exhibit C: Caching players involved in handling an HTTP request



Browser caches

All modern web browsers save recently used resources (images, stylesheets, JavaScript files, and some HTML pages) locally to speed up backtracking and return visits.

In theory, browser caches should follow exactly the same caching rules as does any other HTTP cache. This is why your browser's Back button usually evinces a slight lag instead of zipping you instantly to the previous page. Even though most of the resources needed to render the page are cached locally, the page's top-level HTML wrapper is typically dynamic and uncacheable, so a round trip to the server is still required. The browser *could* simply rerender the materials on hand from the previous visit—and one or two used to do that—but this shortcut breaks the correctness of caching and leads to a variety of subtle problems.

Proxy cache

You can install a proxy cache at the edge of an organization's network to speed up access for all users. When a user loads a web site, the requests are first received by the proxy cache. If a requested resource is cached, that resource is immediately returned to the user without the remote site being consulted.

You can configure a proxy cache in two ways: actively, by changing users' browser settings to point to the proxy; or passively, by having a network router send all web traffic through the cache server. The latter configuration is known as an intercepting proxy. There are also methods by which user agents can automatically discover the relevant proxies.

Reverse proxy cache

Web site operators configure a "reverse proxy" cache to offload traffic from their web and application servers. Incoming requests are first routed to the reverse proxy cache, from which they may be served immediately if the requested resources are available. The reverse proxy passes requests for uncached resources along to the appropriate web server.

Server sites use reverse proxy caches primarily because they reduce load on the origin servers. They may also have the beneficial side effect of speeding response times for clients.

Cache problems

Web caches are tremendously important to the performance of the web, but they also introduce complexity. A problem at any caching layer can introduce stale content that is out of date with respect to the origin server. Cache problems can befuddle both users and administrators and are sometimes difficult to debug.

Stale cache entries are best detected by a direct query at each hop along the path. If you are the site operator, try using **curl** to request a problematic page directly from the origin, then from the reverse proxy cache, and if applicable, from the proxy cache and from any other caches in the request path.

You can use **curl -H "Cache-Control: no-cache"** to politely request a cache refresh. This is the same as invoking <Shift-Reload> in a web browser. Conformant caches will obey, but if you're still seeing old data, don't assume that your reload request has been honored unless you can prove it on the server.

Cache software

Table 19.5 lists a few of the open source caching software implementations. Of these, we find ourselves using NGINX most frequently. Its caching is easy to configure, and NGINX is often already in use as a proxy or web server.

Table 19.5: Open source caching software

Server	Notes
Squid	One of the first open source cache implementations Normally used as a proxy cache Includes important features like antivirus and TLS
Varnish	Exceptional configuration language Multithreaded Modular and extensible
Apache mod_cache	Good choice for sites already running **httpd**
NGINX	Good choice for sites already running NGINX Has a reputation for good performance
Apache Traffic Server	Runs at extremely high-traffic sites Supports HTTP/2 Donated to the Apache Foundation by Yahoo!

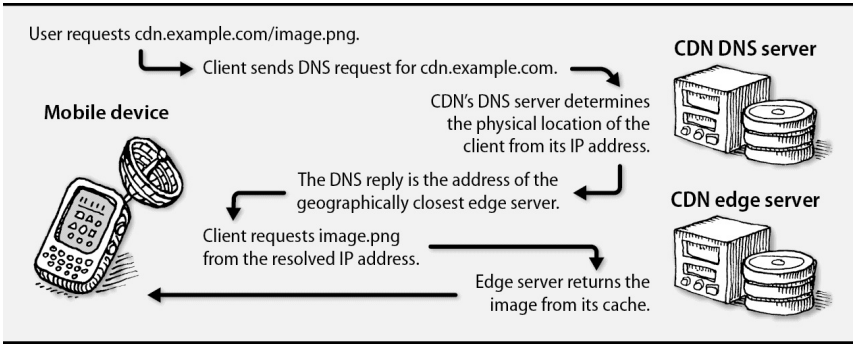
Content delivery networks

A content delivery network (CDN) is a globally distributed system that improves web performance by moving content closer to users. Nodes in a CDN are dispersed geographically to hundreds or thousands of locations. When clients request content from a site that uses a CDN, they are routed to the closest node (called an edge server), thereby decreasing latency and reducing congestion for the origin.

Edge servers are similar to proxy caches. They store copies of content locally. If they don't have a local copy of a requested resource or if their version of the content has expired, they retrieve the resource from the origin, respond to the client, and update their cache.

CDNs use DNS to redirect clients to the geographically nearest host. Exhibit D explains how it works.

Exhibit D: The role of DNS in a content delivery network



CDNs can now host dynamic content, but traditionally they have been best suited to static content such as images, stylesheets, JavaScript libraries, HTML files, and downloadable objects. Streaming services like Netflix and YouTube use CDNs to serve large media files.

CDNs also offer value beyond performance improvements. Most CDNs provide security services such as denial-of-service attack prevention and web application firewalls. Some specialty CDNs offer other innovations that optimize page rendering and reduce the load on origin servers.

A substantial portion of content on the web today is served by CDNs. If you're at a large site, expect to pull out your pocketbook to pay for the privilege of fast performance. If you run a smaller service, optimize your local caching layers before turning to a CDN.

One of the oldest and most prestigious (read: expensive) CDNs is Akamai, headquartered in Massachusetts. Akamai counts some of the world's largest governments and businesses among its customers. It has the largest global network as well as some of the most advanced CDN features. Nobody was ever fired for choosing Akamai.

CloudFlare is another popular CDN. Unlike Akamai, CloudFlare has a history of selling to smaller customers, although their target market has more recently shifted to enterprise. Pricing is listed clearly on their web site, and they offer some of the best security features available. CloudFlare was one of the first large-scale providers to deploy HTTP/2 for all its customers.

Amazon's CDN service is called CloudFront. It integrates with other AWS services such as S3, EC2, and ELB, but can also work for sites hosted outside of Amazon's cloud. As with other AWS products, pricing is competitive and metered by usage.

Languages of the web

The web has evolved from being mostly static to a rich, interactive experience. The web apps that enable this bounty are coded in a variety of programming languages, each with associated tooling and unique quirks. Administrators need to manage software libraries, install application servers, and configure web applications according to the standards established for each language's ecosystem.

All the languages mentioned in the following sections are in common use on the web today. They all feature engaged communities of developers, extensive support libraries, and well-documented best practices. Sites typically choose whichever languages and frameworks their teams are most comfortable with.

Ruby

See the section starting here for a short summary of Ruby.

Ruby is well known in DevOps and system administration circles for its use in Chef and Puppet. It's also the language of Ruby on Rails, a widely used web framework. Rails is a good choice for rapid development processes and is often used for prototyping new ideas.

Rails has a reputation for mediocre performance and for fostering monolithic applications. Over time, many Rails applications tend to accumulate baggage that makes them harder to modify; the end result is often a gradual performance sag over time.

Ruby features a large collection of libraries called gems that developers can use to simplify their projects. Most are hosted at rubygems.org. They are curated by the community, but many are of marginal quality. Managing a system's installed versions of Ruby and its various gem dependencies can be both tedious and troublesome.

Python

See the section starting here for a short summary of Python.

Python is a general-purpose language used not only in web development but also in a wide swath of scientific disciplines. It's easy to read and to learn. The most widely deployed web framework for Python is Django, which has many of the same benefits and drawbacks as Ruby on Rails.

Java

Java, now controlled by Oracle, is used most often in enterprise environments with slower development workflows. Java offers fast performance at the expense of complex, clunky tooling

and many layers of abstraction. Java's challenging license requirements and obtuse conventions can frustrate neophytes.

Node.js

JavaScript is known first and foremost as a client-side scripting language that runs within web browsers. As a language, it has been ridiculed as hastily designed, difficult to read, and frequently counterintuitive. Now, Node.js—an engine for executing JavaScript on servers—brings JavaScript to the data center as well.

To be fair, Node.js boasts high concurrency and native real-time messaging capabilities. As a newer language, it has so far avoided much of the cruft built up over the years in other systems.

PHP

PHP is simple to get started with, and for that reason it tends to attract new and inexperienced programmers. PHP applications are notorious for being difficult to maintain. Past versions of PHP made it far too easy for developers to create large security holes in their applications, but recent versions have made improvements in this area. PHP is the language used by WordPress, Drupal, and several other content management systems.

Go

Go is a lower-level language from Google. It has gained popularity in recent years through its use in major open source projects such as Docker. It's excellent for systems programming but is also well suited for web applications because of its powerful native concurrency primitives. One benefit for administrators is that Go software usually compiles to a stand-alone binary, making it simple to deploy.

Application programming interfaces (APIs)

Web APIs are application interfaces intended for use not by humans but by software agents. An API defines a set of methods through which a remote system can make use of an application's data and services. APIs have become ubiquitous on the web because they promote cooperation among many diverse clients.

APIs are nothing new. Operating systems define APIs to allow user-space applications to interact with the kernel. Nearly all software packages use defined interfaces to facilitate modularity and separation of functions within the code base. However, web APIs are a bit special because they are exposed to the world on the public web with the intention of promoting use by outside developers.

Web API calls are normal HTTP requests. They're only "APIs" because the client and server have agreed, by convention, that certain URLs and verbs have specific meanings and effects within the domain of their interaction.

Web APIs commonly use some kind of text-based serialization format to encode data for exchange. These formats are relatively simple and can be parsed by applications written in any programming language. Many formats exist, but by far the most common are JavaScript Object Notation (JSON) and Extensible Markup Language (XML). (Douglas Crockford, who named and promoted the JSON format, says it's pronounced like the name Jason. But somehow, "JAY-sawn" seems to have become more common in the technical community.)

HTTP APIs are perhaps easiest to explain by example. The Spotify music service exposes an API that represents its music library. A client of the API can request information about albums, artists, and tracks; execute searches; and perform other related actions. This API is used both by Spotify's own client applications (its browser, desktop, and mobile clients) and by third parties that want to incorporate Spotify's services.

Because web APIs consist of HTTP requests, you can interact with them with all the normal HTTP tools, including web browsers and **curl**. For example, we can obtain Spotify's JSON record for The Beatles:

```
$ curl https://api.spotify.com/v1/artists/3WrFJ7ztbogyGnTHbHJFl2 | jq '.'
{
  "external_urls": {
    "spotify": "https://open.spotify.com/artist/3WrFJ7ztbogyGnTHbHJFl2"
  },
  "followers": {
    "href": null,
    "total": 1566620
  },
  "genres": [ "british invasion" ],
  "href": "https://api.spotify.com/v1/artists/3WrFJ7ztbogyGnTHbHJFl2",
  "id": "3WrFJ7ztbogyGnTHbHJFl2",
  "images": [ <removed for concision> ],
  "name": "The Beatles",
  "popularity": 91,
  "type": "artist",
  "uri": "spotify:artist:3WrFJ7ztbogyGnTHbHJFl2"
}
```

Here, we've piped the JSON output through **jq** to clean up the formatting a bit. On a terminal, **jq** also colorizes the output. **jq** does far more than just formatting and is highly recommended for parsing and filtering JSON from the command line. Find it at stedolan.github.io/jq.

How did we know that 3WrFJ7ztbogyGnTHbHJFl2 is the Spotify ID for The Beatles? Try searching through the API: https://api.spotify.com/v1/search?type=artist&q=beatles.

Spotify's API is also an example of a "RESTful" API, which is the predominant approach today. REST (Representational State Transfer) is an architectural style of API design introduced by Roy Fielding in his doctoral dissertation; Fielding is also a primary author of the HTTP specification. The term was originally quite specific but is now more loosely applied to web services that 1) explicitly use HTTP verbs to communicate intent, and 2) use a directory-like path structure to locate resources. Most REST APIs use JSON as their underlying representation for data.

REST contrasts starkly with SOAP (Simple Object Access Protocol), an earlier system for implementing HTTP APIs that defines strict and elaborate multilevel guidelines for interactions among systems. SOAP APIs use a complex XML-based format that funnels all calls through a few specific URLs, resulting in large HTTP payloads, poor performance, and endless difficulties in development, debugging, and deployment.

The development of the SOAP ecosystem is an interesting case study of the ways that technical initiatives can go awry. In particular, it illustrates the risks of attempting to design systems for a hazy and uncertain future. SOAP put a lot of effort into remaining platform-, language-, data-, and transport-neutral, and indeed it largely achieved these goals—even basic data types such as integers were left open to definition. Unfortunately, the resulting system was complex and didn't fit well with real-world needs.

19.3 Web hosting in the cloud

Cloud providers offer dozens of services for hosting web applications, and the landscape changes weekly. We can’t possibly cover everything, but a few points stand out as being of particular interest to web administrators.

Small sites with few users and a tolerance for occasional outages can get away with a single virtual cloud instance as a web server (or possibly two instances behind a load balancer for improved reliability). But the cloud offers many opportunities to improve these simple configurations without significant increases in cost and complexity of administration.

Build versus buy

Administrators working on a cloud platform can build custom, self-managed web applications out of “raw” virtual machines. Alternatively, they can farm out parts of the design to off-the-shelf cloud services, thus reducing the labor involved in designing, configuring, and maintaining everything by hand. For the sake of efficiency, we prefer to rely on vendor services when possible.

Load balancers are a good example of this tradeoff. On AWS, for example, you can either run an EC2 instance with open source load-balancing software or sign up for an AWS-provided Elastic Load Balancer. The former offers greater customization but requires you to manage the load balancer’s operating system, configure the load-balancing software, tune performance, and promptly install security patches as they are released in the future. In addition, the glue needed to gracefully handle failures of the software or the instance will be somewhat more complex.

An ELB, on the other hand, can be created in a matter of seconds and requires no further administrative action. AWS handles everything behind the scenes. Unless the ELB lacks a specific feature that you need, it is clearly the expedient choice.

Ultimately, this is a decision between building a service or outsourcing it to the vendor. For the sake of your own sanity, avoid the building option unless the function in question is a core competence for your business.

Platform-as-a-Service

The PaaS concept simplifies web hosting for developers by eliminating infrastructure as a concern. Developers package their code in a specific format and upload it to the PaaS provider, which provisions appropriate systems and runs it automatically. The provider issues a DNS endpoint connected to the client's running application, which the client can then customize by using a DNS CNAME record.

Although PaaS offerings greatly simplify infrastructure management, they sacrifice flexibility and customization. Most offerings either do not allow administrative access to a shell or they actively discourage it. Users of a PaaS must accept certain design decisions made by the vendor. Users' ability to implement some features may be constrained.

Google App Engine pioneered the PaaS concept and remains one of the most prominent products. App Engine supports Python, Java, PHP, and Go, and has many supporting features such as **cron**-like scheduled job execution, programmatic access to logs, real-time messaging, and access to various databases. It is considered the Cadillac of PaaS offerings.

The competing product from AWS is called Elastic Beanstalk. In addition to all the languages supported by App Engine, it supports Ruby, Node.js, Microsoft .NET, and Docker containers. It integrates with Elastic Load Balancers and AWS's Auto Scaling feature, leveraging the power of the AWS ecosystem.

In practice, we've found Elastic Beanstalk to be a mixed bag. Customization is possible through an extension framework that is proprietary and tedious. Users are still responsible for running the EC2 instances that host the application. So although Elastic Beanstalk might be a fine fit for prototyping, we believe that the system is not a good choice for production workloads consisting of many services.

Heroku is another respected vendor in this space. An application on Heroku is deployed to a dyno, Heroku's word for a lightweight Linux container. Users control the dyno deployments. Heroku has a strong network of partnerships that offer databases, load balancing, and other integrations. Heroku's pricing is higher than some other offerings in part because its own infrastructure runs on AWS under the hood.

Static content hosting

See this page for more information about content delivery networks.

It seems like overkill to run an operating system just for the sake of hosting static web sites. Fortunately, the cloud providers can host them for you. In AWS S3, you create a bucket for your content, then configure a CNAME record from your domain to an endpoint within the provider. In Google Firebase, you use a command-line tool to copy your local content to Google, which provisions an SSL certificate and hosts your files. In both cases you can serve your content from a CDN for better performance.

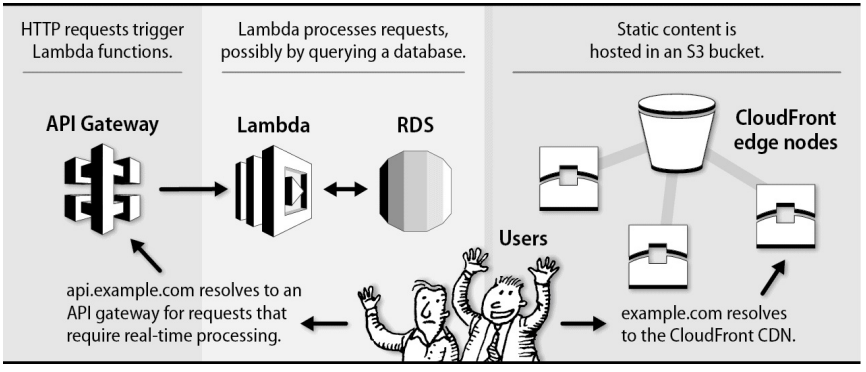
Serverless web applications

AWS Lambda is an event-based computing service. Developers who use Lambda write code that runs in response to an event such as the arrival of a message in a queue, a new object in a bucket, or even an HTTP request. Lambda feeds the event payload and metadata as inputs to a user-defined function, which performs processing and returns a response. There are no instances or operating systems to manage.

To process HTTP requests, Lambda is used in conjunction with another AWS service called API Gateway, a proxy that can scale to hundreds of thousands of simultaneous requests. API Gateway is interposed in front of an origin to add features such as access control, rate limiting, and caching. HTTP requests are received by the API Gateway, and when a request arrives, the gateway triggers a Lambda function.

In combination with static hosting on S3, Lambda and API Gateway can lead to a fully serverless platform for running web applications, as illustrated in Exhibit E.

Exhibit E: Serverless web hosting with AWS Lambda, API Gateway, and S3



This technology is still in its youth, but it's already altering the mechanics of hosting web applications. We expect enhancements, frameworks, competing services, and best practices to mature rapidly in the coming years.

19.4 APACHE HTTPD

The **httpd** web server is ubiquitous among the many flavors of UNIX and Linux. It is portable across many architectures, and prebuilt packages exist for all major systems. Unfortunately, OS vendors have varied and highly opinionated approaches to **httpd** configuration.

A modular architecture has been fundamental to Apache's adoption. Dynamic modules can be turned on through configuration, offering alternative authentication options, improved security, support for running code written in most languages, URL-rewriting superpowers, and many other features.

For largely historical reasons, Apache has a pluggable connection handling system called multi-processing modules (MPMs) that determines how HTTP connections are managed at the network I/O layer. The event MPM is the modern choice and is recommended over the worker and prefork alternatives. (Some legacy software that is not considered thread-safe, such as mod_php, should use the prefork MPM. It uses processes rather than threads for each connection.)

To bind to privileged ports (those below 1024, such as HTTP port 80 and HTTPS port 443), the initial **httpd** process must run as root. That process then forks additional workers under a local account with lower privileges to handle actual requests. Sites that do not need to listen on port 80 or 443 can be run entirely without root privileges.

httpd is configured through directives (Apache-speak for configuration options) in plain text files that use a distinctive Apache-style syntax. Though hundreds of directives exist, administrators usually need to tweak only a few. The directives and their values are documented directly in the default files that ship with the OS as well as on Apache's web site.

httpd in use

httpd is both the name given to the daemon's binary and to the project. Ubuntu has taken the liberty of renaming **httpd** to **apache2**, which matches the name of the **apt** package but otherwise does little more than create confusion.

System V **init**, BSD **init**, and **systemd** can all manage **httpd**. Whichever option is standard for your system is the one you should default to. For debugging and configuration, however, you can interact with the daemon independently of the startup scripts.

Administrators can either run **httpd** directly or use **apachectl**. Invoking **httpd** offers direct control over the server daemon, but remembering (and typing!) all the options is a challenge. **apachectl** is a shell script wrapper around **httpd**. Each operating system vendor customizes **apachectl** to conform to the conventions of its **init** process. It can start, stop, reload, and show the status of Apache.

For example, here's how to start the server with the default configuration:

```
# apachectl start
Performing sanity check on apache24 configuration:
Syntax OK
Starting apache24.
# apachectl status
apache24 is running as pid 1337.
```

In this output from a FreeBSD system, **apachectl** first performs a **lint**-like configuration check by running **httpd -t**, then starts the daemon. (**lint** is a UNIX program that evaluates C code for potential bugs. The term is now applied more broadly to any tool that inspects software and configuration files for errors, bugs, or other problems.)

apachectl graceful waits for any currently open connections to conclude and then restarts the server. This feature is handy for updating without interrupting active connections. It's available through the system start and stop scripts as well.

Use **apachectl**'s **-f** flag to start Apache with a custom configuration, e.g.:

```
# apachectl -f /etc/httpd/conf/custom-config.conf -k start
```

Some vendors deprecate this use of **apachectl** in favor of running **httpd** directly.

Refer to Chapter 2, *Booting and System Management Daemons*, to learn how to configure **httpd** to start automatically at boot time.

httpd configuration logistics

Although an entire **httpd** configuration can be contained in a single file, OS maintainers typically use the Include directive to split the default configuration into multiple files and directories. This architecture simplifies site management and is better suited to automation. Predictably, the specifics of the configuration hierarchy differ by system. Table 19.6 lists the Apache configuration defaults for each of our example platforms.

Table 19.6: Apache configuration details by platform

	RHEL/CentOS	Debian/Ubuntu	FreeBSD
Package name	httpd	apache2	apache24
Config root	/etc/httpd	/etc/apache2	/usr/local/etc/apache24
Primary config file	conf/httpd.conf	apache2.conf	httpd.conf
Module config	conf.modules.d/	mods-available/ mods-enabled/	modules.d/
Virtual host config	conf.d/	sites-available/ sites-enabled/	Includes/
Log location	/var/log/httpd	/var/log/apache2	/var/log/httpd-*.log
User	apache	www-data	www

When **httpd** starts, it consults a primary configuration file, usually **httpd.conf**, and incorporates any additional files as referenced by Include directives. The default **httpd.conf** is heavily commented and serves as a quick reference. Configuration options in this file can be grouped into three categories:

- Global settings such as the path to **httpd**'s configuration root, the user and group as which to run, the modules to activate, and the network interfaces and ports to listen on
- VirtualHost sections that define how to provide service for a given domain (usually delegated to subdirectories and Included in the main configuration)
- Instructions for answering requests that don't match any VirtualHost definition

Many admins will be satisfied with the global settings and need only manage individual VirtualHosts.

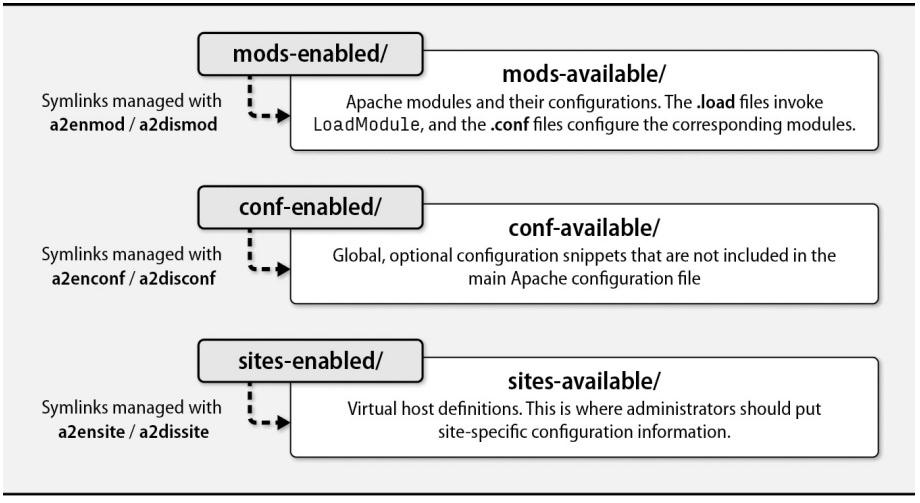
Modules exist independently of the **httpd** core and often have their own configuration options. Most OS vendors choose to separate out module configuration into subdirectories.

Debian and Ubuntu approach Apache configuration idiosyncratically. A structure of subdirectories, configuration files, and symlinks creates a more flexible system for managing the

server, at least in theory.

Exhibit F attempts to clarify this puzzle. The master **apache2.conf** file includes all files from the ***-enabled** subdirectories in **/etc/apache2**. These files are in fact symbolic links to files in the ***-available** subdirectories. A pair of configuration commands that create and remove symlinks is provided for each set of subdirectories.

Exhibit F: Subdirectories of /etc/apache2 on Debian-based systems



In our experience, the Debian system is unnecessary and overly complex. A simple **site-configuration** subdirectory usually provides sufficient structure. If you're running Debian or Ubuntu, though, it makes sense to stick with their defaults.

Virtual host configuration

The lion's share of **httpd** configuration lies in virtual host definitions. It's generally a good idea to create a file for each site.

When an HTTP request arrives, **httpd** identifies which virtual host to select by consulting the HTTP Host header and network port. It then matches the path portion of the requested URL to a Files, Directory, or Location directive to determine how to serve the requested content. This mapping process is known as request routing.

The following sample shows the HTTP and HTTPS configuration for admin.com.

```
<VirtualHost *:80>
    ServerName  admin.com
    ServerAlias  www.admin.com
    ServerAlias  ulsah.admin.com
    Redirect /   https://admin.com/
</VirtualHost>

<VirtualHost *:443>
    ServerName           admin.com
    ServerAlias          www.admin.com
    ServerAlias          ulsah.admin.com
    DocumentRoot         /var/www/admin.com/
    CustomLog            /var/log/apache2/admin_com_access combined
    ErrorLog             /var/log/apache2/admin_com_error
    SSLEngine            on
    SSLCertificateFile   "/etc/ssl/certs/admin.com.crt"
    SSLCertificateKeyFile"/etc/ssl/private/admin.com.key"
    <Directory "/var/www/admin.com">
        Require all granted
    </Directory>
    <Directory "/var/www/admin.com/photos">
        Options +Indexes
    </Directory>
    <IfModule mod_rewrite.c>
        RewriteEngine on
        RewriteRule ^/(usah|lsah)$ /ulsah
    </IfModule>
    ExtendedStatus On
    <Location /server-status>
        SetHandler server-status
        Require ip 10.0.10.10/32
    </Location>
</VirtualHost>
```

Much of this is self explanatory, but a few details are worth noting:

- The first VirtualHost answers on port 80 and redirects all HTTP requests for admin.com, www.admin.com, and ulsah.admin.com to use HTTPS.
- Requests for admin.com/photos receive an index of all files in that directory.
- Requests for /usah or /lsah are rewritten to /ulsah.

Server status, accessible in this configuration at www.admin.com/server-status, is a module that shows useful runtime performance information, including statistics about the daemon's CPU and memory usage, request status, the average number of requests per second, and more. Monitoring systems can use this feature to collect data about the web server for alerting, reporting, and visualization of HTTP traffic. Here, access to server status is restricted to a single IP address, 10.0.10.10.

HTTP basic authentication

In the HTTP basic authentication scheme, clients pass a base-64-encoded username and password in the Authorization HTTP header. If a user includes a name and password in a URL (e.g., https://user:pass@www.admin.com/server-status), the browser performs the encoding and transfers the value to the Authorization header automatically.

The username and password are not encrypted, so basic authentication does not provide any confidentiality. Thus, it is safe to use only in combination with HTTPS.

Basic authentication in Apache is configured in Location or Directory blocks. For example, the following snippet requires authentication to access /server-status (a best practice) and limits access to a subnet:

```
<Location /server-status>
    SetHandler server-status
    Require ip 10.0.10.0/24
    AuthType Basic
    AuthName "Restricted"
    AuthUserFile /var/www/.htpasswd
    Require valid-user
</Location>
```

Note that the account information is stored externally to the configuration file. Use **htpasswd** to create the account entries:

```
# htpasswd -c /var/www/.htpasswd ben
New password: <password>
Re-type new password: <password>
Adding password for user ben
# cat /var/www/.htpasswd
ben:$apr1$mPh0x0Cj$hfqMavkdHfVRVscE678Sp0
# chown www-data /var/www/.htpasswd  # Set ownership
# chmod 600 /var/www/.htpasswd       # Restrictive permissions
```

Password files are conventionally hidden files called **.htpasswd**, but they can be named anything. Even though the passwords are encrypted, set the permissions on **.htpasswd** files to be readable only by the web-server user. This precaution limits attackers' ability to see usernames and to run passwords through cracking software.

Configuring TLS

SSL might have changed its name to TLS, but in the interest of backward compatibility, Apache retains the SSL name for its configuration options (as do many other software packages). Just a few lines are needed to set up TLS:

```
SSLEngine               on
SSLProtocol             all -SSLv2 -SSLv3
SSLCertificateFile      "/etc/ssl/certs/admin.com.crt"
SSLCertificateKeyFile   "/etc/ssl/private/admin.com.key"
```

Here, the TLS certificate and key are located in Linux's central system location, **/etc/ssl**. The public certificates can be readable by anyone, but the key should be accessible only to the Apache master-process user, typically root. We prefer to set permissions to 444 for the certificate and 400 for the key.

All versions of the actual SSL protocol (precursor to TLS) are known to be insecure and should be disabled with the SSLProtocol directive, shown above.

See this page for a complete citation for the Server Side TLS guide.

A few ciphers have known weaknesses. You can configure the web server's supported ciphers with the SSLCipherSuite directive. The best practices for precisely which settings to use are constantly in flux. The Mozilla *Server Side TLS* guide is the best resource that we are aware of for staying current on best practices for TLS. It also has a handy configuration syntax reference for Apache, NGINX, and HAProxy.

Running web applications within Apache

httpd can be extended to run programs written in Python, Ruby, Perl, PHP, and other languages from within the module system. Modules run inside Apache processes and have access to the full

HTTP request/response life cycle.

Modules provide additional configuration directives that let administrators control the runtime characteristics of applications. Table 19.7 lists some common application server modules.

Table 19.7: Application server modules for httpd

Module	Lang	
mod_php	PHP	Deprecated; use only with the prefork MPM
mod_wsgi	Python	The Web Server Gateway Interface, a Python standard
mod_passenger	Multiple	Flexible, commercially supported application server for multiple languages, including Ruby, Python, and Node.js
mod_proxy_fcgi	Any	A standard server interface usable from any language
mod_perl	Perl	A Perl interpreter that lives within **httpd**

The following example (to configure a Python Django application for api.admin.com) uses mod_wsgi:

```
LoadModule wsgi_module mod_wsgi.so

<VirtualHost *:443>
    ServerName api.admin.com

    CustomLog /var/log/apache2/api_admin_com_access combined
    ErrorLog /var/log/apache2/api_admin_com_error

    SSLEngine on
    SSLCertificateFile "/etc/ssl/certs/api_admin.com.crt"
    SSLCertificateKeyFile "/etc/ssl/private/api_admin.com.key"

    WSGIDaemonProcess admin_api user=user1 group=group1 threads=5
    WSGIScriptAlias / /var/www/api.admin.com/admin_api.wsgi

    <Directory /var/www/api.admin.com>
        WSGIProcessGroup admin_api
        WSGIApplicationGroup %{GLOBAL}
        Require all granted
    </Directory>
</VirtualHost>
```

Once **mod_wsgi.so** has been loaded by Apache, several WSGI configuration directives become available. The `WSGIScriptAlias` file in the configuration above, **admin_api.wsgi**, contains Python code that is needed by the WSGI module.

Logging

httpd offers best-in-class logging capabilities, with fine-grained control over the data that is logged and the ability to separate log data by virtual host. Administrators use these logs to debug configuration problems, detect potential security threats, and analyze usage information.

A sample log message from **admin.com.access.log** looks like this:

```
127.0.0.1 - - [19/Jun/2015:15:21:06 +0000] "GET /search HTTP/1.1"
    200 20892 "-" "curl/7.38.0"
```

The message shows:

- The source of the request; in this case, 127.0.0.1, the local host
- A time stamp
- The path of the requested resource (/search) and the HTTP method (GET)
- The response status code (200)
- The size of the response
- The user agent (the **curl** command-line tool)

The documentation for mod_log_config has all the details on how to customize the log format.

A busy web site generates a large number of request logs that can quickly fill up the disk. Administrators are responsible for ensuring that this never happens. Keep web server logs on a dedicated partition to prevent a large log file from affecting the rest of the system.

---

*See this page for more information about **logrotate**.*

On most Linux distributions, the default package installation of Apache includes an appropriate **logrotate** configuration. FreeBSD comes with no such default, and administrators should instead add an entry in **/etc/newsyslog.conf** for Apache's logs.

The log directory and the files within should be writable only by the user of the master **httpd** process, which is normally root. If nonroot users have write access, they can create a symlink to another file, causing it to be overwritten with bogus data. The system defaults are set safely, so avoid customizing the owner and group.

19.5 NGINX

A busy web server must respond to many thousands of concurrent requests. Most of the time needed to handle each request is spent waiting for data to arrive from the network or disk. The time spent actively processing the request is short by comparison.

To handle this workload efficiently, NGINX uses an event-based system in which just a few worker processes handle many requests simultaneously. When a request or response (an event) is ready for servicing, a worker process quickly completes processing before returning to handle the next event. Above all, NGINX aims to avoid blocking on network or disk I/O.

The event MPM included in newer releases of Apache uses a similar architecture, but for high-volume and performance-sensitive sites, NGINX remains the software of choice.

Administrators running NGINX will notice at least two processes: a master and a worker. The master performs housekeeping duties such as opening sockets, reading the configuration, and keeping the other NGINX processes running. Workers do most of the heavy lifting by handling and processing requests. Some configurations use additional processes dedicated to caching. As in Apache, the master process runs as root so that it can open sockets for any ports below 1024. The other processes run as a less privileged user.

The number of worker processes is configurable. A good rule of thumb is to run as many worker processes as the system has CPU cores. Debian and Ubuntu configure NGINX this way by default if it’s installed from a package. FreeBSD and RHEL default to a single worker process.

Installing and running NGINX

Although NGINX continues to grow in popularity and is a staple among some of the world's busiest web sites, OS distributions still lag on NGINX support. The versions available in the official repositories for Debian and RHEL are usually out of date, though FreeBSD is typically more current. NGINX is open source, so it can be built and installed manually. The project's web page, nginx.org, offers packages for **apt** and **yum** than are generally more current than those supplied by the distributions.

See this page for more information about signals.

The system's normal service management is appropriate for day-to-day wrangling of **nginx**. You can also run the **nginx** daemon during development and debugging. Use the **-c** argument to specify a custom configuration file. The **-t** option performs a check of the configuration file syntax.

nginx uses signals to trigger various maintenance actions; Table 19.8 lists these. Make sure you target the master **nginx** process (usually the one with the lowest PID).

Table 19.8: Signals understood by the nginx daemon

Signal	Function
TERM or INT	Shuts down immediately
QUIT	Completes and closes all current connections, then shuts down
USR1	Reopens log files (used to facilitate log rotations)
HUP	Reloads the configuration[a]
USR2	Gracefully replaces the server binary without interrupting service[b]

a. This option tests the syntax of the new configuration, and if the syntax is valid, starts new workers with the new configuration. It then gracefully shuts down the old workers.
b. See the **nginx** command-line documentation for details on how this works.

Configuring NGINX

The NGINX configuration style is generally C-like; it uses curly braces to distinguish blocks of configuration lines and semicolons to separate lines. The main configuration file is called **nginx.conf** by default. Table 19.9 summarizes the most important system-specific aspects of NGINX configuration.

Table 19.9: NGINX configuration details by platform

	RHEL/CentOS	Debian/Ubuntu	FreeBSD
Package name	nginx [a]	nginx	nginx
Daemon path	**/sbin/nginx**	**/usr/sbin/nginx**	**/usr/local/sbin/nginx**
Configuration root	**/etc/nginx**	**/etc/nginx**	**/usr/local/etc/nginx**
Virtual host config [b]	**conf.d/**	**sites-available/ sites-enabled/**	*No prescribed location*
Default user	nginx	www-data	nobody

a. You must enable the EPEL software repository; see fedoraproject.org/wiki/EPEL.
b. Relative to the configuration root directory

Within the **nginx.conf** file, blocks of configuration directives surrounded by curly braces are called contexts. A context contains directives specific to that block of configuration. For example, here's a minimal (but complete) NGINX configuration that shows three contexts:

```
events { }

http {
    server {
        server_name www.admin.com;
        root /var/www/admin.com;
    }
}
```

The outermost context (called `main`) is implicit and configures the core functionality. The `events` and `http` contexts live within `main`. `events` is a required context that configures connection handling. Since it's blank in this example, default values are implied. Fortunately, the defaults are sensible:

- Run one worker process (use the unprivileged user account).
- Listen on port 80 if started as root or port 8000 otherwise.
- Write logs to **/var/log/nginx** (chosen at compile time).

The `http` context contains all directives relating to web and HTTP proxy services. `server` contexts, which define virtual hosts, are nested within `http`. Multiple `server` contexts within `http` would configure multiple virtual hosts.

Aliases can be included in server_name to match the Host header against a group of subdomains:

```
http {
    server {
        server_name admin.com www.admin.com;
        root /var/www/admin.com;
    }
    server {
        server_name example.com www.example.com;
        root /var/www/example.com;
    }
}
```

See this page for an overview of regular expressions.

The value for server_name can also be a regular expression, and the match can even be captured and named as a variable for use later in configuration. By using this feature, you can refactor the previous configuration to

```
http {
    server {
        server_name ~^(www\.)?(?<domain>(example|admin).com)$;
        root /var/www/$domain;
    }
}
```

The regular expression, which starts with a tilde, matches either example.com or admin.com, optionally preceded by www. The value of the matched domain is stored in the $domain variable, which is then used to determine which server root to select.

Be aware that use of this syntax commits NGINX to performing a regular expression match on every HTTP request. We use it here to demonstrate NGINX's flexibility, but in practice you would probably want to just list all possible hostnames in plain text. It's perfectly reasonable to use regular expressions in **nginx.conf**, but make sure they're delivering actual value, and try to keep them low in the configuration hierarchy so that they activate only in specific situations.

Name-based virtual hosts can be distinguished from IP-based hosts by using the listen and server_name directives together.

```
server {
    listen 10.0.10.10:80
    server_name admin.com www.admin.com;
    root /var/www/admin.com/site1;
}
server {
    listen 10.0.10.11:80
    server_name admin.com www.admin.com;
    root /var/www/admin.com/site2;
}
```

This configuration shows two versions of admin.com being served from different web roots. The IP address of the interface on which the request was received determines which version of the site the client sees.

The root is the base directory where HTML, images, stylesheets, scripts, and other files for the virtual host are stored. By default, NGINX just serves files out of the root, but you can use the location directive to do more sophisticated request routing. If a given path isn't matched by a location directive, NGINX automatically falls back to the root.

The following example uses location in combination with the proxy_pass directive. It instructs NGINX to serve most requests from the web root but forward requests for http://www.admin.com/nginx to nginx.org.

```
server {
    server_name admin.com www.admin.com;
    root /var/www/admin.com;
    location /nginx/ {
        proxy_pass http://nginx.org/;
    }
}
```

proxy_pass instructs NGINX to act as a proxy and replay requests from clients to another downstream server. We revisit the proxy_pass directive when we describe how to use NGINX as a load balancer here.

location can use regular expressions to perform powerful path-based routing to different sources based on the requested content. The official NGINX documentation analyzes how NGINX evaluates the server_name, listen, and location directives to route requests.

A common pattern among distributions is to set sensible defaults for many directives in the global http context, then use the include directive to add site-specific virtual hosts to the final configuration. For example, the default **nginx.conf** file for Ubuntu includes the line

```
include /etc/nginx/conf.d/*.conf;
```

This architecture helps eliminate redundancy since all children inherit the global settings. Administrators in straightforward environments may not need to do anything more than write virtual host configurations expressed as `server` contexts.

Configuring TLS for NGINX

Although NGINX didn't borrow much from Apache's configuration style, its TLS configuration is one area in which it's strikingly similar. As in Apache, the configuration keywords all refer to SSL, TLS's earlier name.

Enable TLS and point to the certificate and private key file like so:

```
server {
    listen 443;
    ssl on;
    ssl_certificate /etc/ssl/certs/admin.com.crt;
    ssl_certificate_key /etc/ssl/private/admin.com.crt;
    ssl_protocols TLSv1 TLSv1.1 TLSv1.2;
    ssl_ciphers ECDHE-RSA-AES128-GCM-SHA256:ECDHE... # truncated
    ssl_prefer_server_ciphers on;

    server_name admin.com www.admin.com;
    root /var/www/admin.com/site1;
}
```

Only the actual TLS protocols (not the older SSL versions) should be enabled; all SSL protocols have been deprecated. Permissions on the certificate and key should follow the recommendations outlined in the Apache TLS section here.

Use the ssl_ciphers directive to require cryptographically strong cipher suites and to disable weaker ciphers. The ssl_prefer_server_ciphers option in conjunction with ssl_ciphers instructs NGINX to choose from the server's list rather than from the client's; otherwise, the client could suggest any cipher it pleased. (The previous example does not show a full list of ciphers because the appropriate list is quite long; refer to the Mozilla *Server Side TLS* guide cited here for recommended values. If you prefer a shorter cipher list, try the one at cipherli.st.)

Load balancing with NGINX

In addition to being a web and cache server, NGINX is also a capable load balancer. Its configuration style is flexible but somewhat nonobvious.

Use the upstream module to create named groups of servers. For example, the following clause defines admin-servers as a collection of two servers:

```
upstream admin-servers {
    server web1.admin.com:8080 max_fails=2;
    server web2.admin.com:8080 max_fails=2;
}
```

upstream groups can be referenced from virtual host definitions. In particular, they can be used as proxying destinations, just like hostnames:

```
http {
    server {
        server_name admin.com www.admin.com;
        location / {
            proxy_pass http://admin-servers;
            health_check interval=30 fails=3 passes=1 uri=/health_check
                match=admin-health  # line not split in original file
        }
    }
    match admin-health {
        status 200;
        header Content-Type = text/html;
        body ~ "Red Leader, Standing By";
    }
}
```

Here, traffic for admin.com and www.admin.com is farmed out to the web1 and web2 servers in round robin order (the default).

This configuration also sets up health checks for the back-end servers. Checks are performed every 30 seconds (interval=30) against each server at the /health_check endpoint (uri=/health_check). NGINX will mark the server down if the health check fails on three consecutive attempts (fails=3), but will add the server back to the rotation if it succeeds just once (passes=1).

The match keyword is peculiar to NGINX. It dictates the conditions under which the health check is considered successful. In this case, NGINX must receive a 200 response code, the Content-Type header must be set to text/html, and the body of the response must contain the phrase “Red Leader, Standing By.”

We’ve added an additional condition within the upstream context that sets the maximum number of connection attempt failures to two. That is, if NGINX cannot connect to the server at all within

two attempts, it gives up and removes that server from the pool. This is an additive connectivity check that complements the more structured checks from the `health_check` clause.

19.6 HAProxy

HAProxy is the most widely used open source load-balancing software. It proxies HTTP and TCP, supports sticky sessions to pin a given client to a specific web server, and offers advanced health-checking capabilities. Recent versions also support TLS, IPv6, and HTTP compression. Support for HTTP/2 is a work in progress and is expected to mature quickly beginning with HAProxy version 1.7.

HAProxy's configuration is usually contained in a single file, **haproxy.cfg**. It's so simple that OS vendors generally don't overcomplicate things and instead embrace the default directory structure recommended by the project.

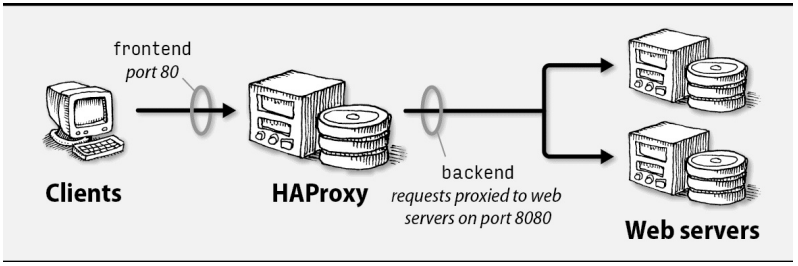
On Debian and RHEL systems, the configuration is in **/etc/haproxy/haproxy.cfg**. FreeBSD doesn't provide a default, as there really is no sensible one for load balancing; it's entirely dependent on your setup. You can find an example configuration on FreeBSD in **/usr/local/share/examples/haproxy** after the HAProxy package has been installed.

The following simple example configuration sets HAProxy to listen on port 80 and distribute requests in a round robin fashion between two web servers, web1 and web2, on port 8080.

```
global
    daemon
    maxconn 5000
defaults
    mode http
    timeout connect  5000    # Milliseconds
    timeout client   10000
    timeout server   10000
frontend http-in
    bind *:80
    default_backend  webservers
backend webservers
    balance          roundrobin
    server   web1    10.0.0.10:8080
    server   web2    10.0.0.11:8080
```

This example introduces HAProxy's frontend and backend keywords, illustrated in Exhibit G.

Exhibit G: HAProxy frontend and backend specifications



frontend dictates how HAProxy will receive requests from clients: which addresses and ports to use, what types of traffic to serve, and other client-facing considerations. backend configures the set of servers that actually process requests. Multiple frontend/backend pairs can exist in a single configuration, allowing a single HAProxy to service multiple sites.

The timeout settings allow fine-grained control over how long a system should wait when trying to open a new connection to a server and how long to keep connections open once they have been established. Fine-tuning these values is important on busy web servers. On local networks, the timeout connect value can be quite low (500ms or less) because new connections should be established quickly.

Health checks

Although the previous configuration provides basic functionality, it doesn't check the status of downstream web servers. If web1 or web2 goes off-line, half of incoming requests would begin to fail.

HAProxy's status-check feature performs regular HTTP requests to determine the health of each server. As long as servers respond with an HTTP 200 response code, they remain in service and continue to receive requests from the load balancer.

If a server fails a status check (by returning anything other than status 200), then HAProxy removes the errant server from the pool. However, HAProxy continues to perform health checks on the server. If it starts to respond successfully once again, HAProxy will return it to the pool.

The specifics of the health check, such as what request method to use, the interval between checks, and the path to request, can all be adjusted. In this example, HAProxy performs a GET request for / on each server every 30 seconds:

```
backend webservers
    balance roundrobin
    option httpchk GET /
    server web1 10.0.0.10:8080 check inter 30000
    server web2 10.0.0.11:8080 check inter 30000
```

It's reassuring to know that you can contact a machine's web server, but that's hardly the last word on server health. Well-constructed web applications commonly expose a health-check endpoint that performs a thorough probe of the application to determine its true health. These checks may include verification of database or cache connectivity as well as performance monitoring. Use these more sophisticated checks if they are available.

Server statistics

HAProxy offers a convenient web interface that displays server stats, much like mod_status in Apache. HAProxy's version shows the state of each server in the pool and lets you manually enable and disable servers as needed.

The syntax is straightforward:

```
listen stats :8000
    mode http
    stats enable
    stats hide-version
    stats realm HAProxy\ Statistics
    stats uri /
    stats auth myuser:mypass
    stats admin if TRUE
```

Server stats can be configured either within a specific listener or within a `backend` or `frontend` block, to limit the feature to that configuration alone.

Sticky sessions

HTTP is a stateless protocol, so each transaction is an independent session. From the perspective of the protocol, requests from the same client are unrelated.

At the same time, most web applications need state to track user behavior over time. The classic example of state is a shopping cart. Users browse a store, add items to the cart, and when ready to check out, submit their payment information. The web application needs some way to track the contents of the cart across multiple page views.

Most web applications use cookies to track state. The web application generates a session for a user and puts the session ID in a cookie that is sent back to the user in the response header. Each time a client submits a request to the server, the cookie is sent with the request. The server uses the cookie to recover the client's context.

Ideally, web applications should store their state information in a persistent and shared medium such as a database. However, some poorly behaved web applications keep their session data locally, in the server's memory or on its local disk. When placed behind a load balancer, these applications break because a single client's requests might be routed to multiple servers, depending on the vagaries of the load balancer's scheduling algorithm.

To address this issue, HAProxy can insert a cookie of its own into responses, a feature known as sticky sessions. Any future requests from the same client will include the cookie. HAProxy can use the value of the cookie to route the request back to the same server.

A version of the previous configuration modified to support sticky sessions looks like the following. Note the addition of the `cookie` directive.

```
backend webservers
    balance roundrobin
    option httpchk GET /
    cookie SERVERNAME insert httponly secure
    server web1 10.0.0.10:8080 cookie web1 check inter 30000
    server web2 10.0.0.11:8080 cookie web2 check inter 30000
```

In this configuration, HAProxy maintains a SERVERNAME cookie to track the server that a client is dealing with. The `secure` keyword specifies that the cookie should only be sent over TLS connections, and `httponly` informs browsers to use the cookie only over HTTP. Refer to RFC6265 for further information on these attributes.

TLS termination

HAProxy versions 1.5 and later include TLS support. A common configuration is to terminate TLS connections at the HAProxy server and communicate with back-end servers over plain HTTP. This approach offloads the cryptographic overhead from the back-end servers and reduces the number of systems that need a private key.

For particularly security-conscious sites, it's also possible to use HTTPS from HAProxy to the back-end servers. You can use the same TLS certificate or a different one; either way, you will still need to terminate and reinitiate TLS at the proxy.

Since HAProxy terminates the TLS connection from clients, you'll need to add the pertinent configuration to the frontend configuration block.

```
frontend https-in
    bind *:443 ssl crt /etc/ssl/private/admin.com.pem
    default_backend webservers
```

Apache and NGINX require the private key and certificate to be in separate files in PEM format, but HAProxy expects both components to be present in the same file. You can simply concatenate the separate files to create a composite file:

```
# cat /etc/ssl/{private/admin.com.key,certs/admin.com.crt} >
    /etc/ssl/private/admin.com.pem
# chmod 400 /etc/ssl/private/admin.com.pem
# ls -l /etc/ssl/private/admin.com.pem
-r-------- 1 root root 3660 Jun 18 17:46 /etc/ssl/private/admin.com.pem
```

Since the private key is part of the composite file, ensure that the file is owned by root and is not readable by any other user. (If you do not run HAProxy as root because you are not accessing any privileged ports, make sure the ownership of the key file matches the identity under which HAProxy runs.)

All usual best practices for TLS apply to HAProxy: disable SSL-era protocols and explicitly configure the acceptable cipher suites.

19.7 Recommended reading

Adrian, David, et al. *Weak Diffie-Hellman and the Logjam Attack*. weakdh.org. This page describes the Logjam attack on the Diffie-Hellman key exchange protocol and suggests ways to secure systems properly.

CloudFlare, Inc. blog.cloudflare.com. This is the corporate blog of content delivery network CloudFlare. Some posts are just marketing information, but many include insights on the latest web trends and technologies.

Google, Inc. *Web Fundamentals.* developers.google.com/web/fundamentals. This is a useful guide to various best practices for web development, including sections on site design, user interfaces, security, performance, and other topics of interest to both developers and administrators. The caching discussion is particularly good.

Grigorik, Ilya. *High Performance Browser Networking.* O'Reilly Media. 2013. An exceptional guide to the protocols, strengths, limitations, and performance aspects of the web. Useful for developers and system administrators alike.

IANA. *Index of HTTP Status Codes.* www.iana.org/assignments/http-status-codes.

International Engineering Task Force. *Hypertext Transfer Protocol Version 2*. http2.github.io/http2-spec. The working draft of the HTTP2 specification.

Mozilla. *Security/Server Side TLS*. wiki.mozilla.org/Security/Server_Side_TLS. An excellent resource that documents best practices for TLS configuration across many platforms.

Stenberg, Daniel. daniel.haxx.se/blog. This is the blog of Daniel Stenberg, the author of **curl** and a prolific HTTP expert.

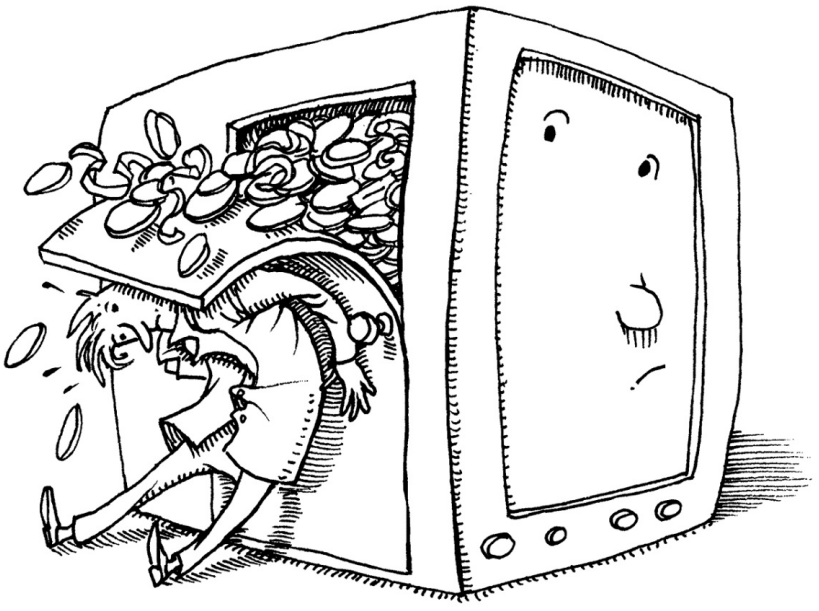
van Elst, Remy. *Strong Ciphers for Apache, nginx, and Lighthttpd.* cipherli.st. Correct and secure cipher configuration for Apache **httpd**, NGINX, and the **lighttpd** web servers, as well as a TLS configuration tester.

SECTION THREE

STORAGE



20 Storage



Data storage systems are looking more and more like a giant set of Lego blocks that you can assemble in an infinite variety of configurations. You can build anything from a lightning-fast storage space for a mission-critical database to a vast, archival vault that stores three copies of all data and can be rewound to any point in the past.

Mechanical hard drives remain a popular storage medium when capacity is the most important consideration, but solid state drives (SSDs) are preferred for performance-sensitive applications. Caching systems, both software and hardware, help combine the best features these storage types.

On cloud servers, you usually have a choice of storage hardware, but you'll pay more for SSD-backed virtual disks. You can also choose from a variety of purpose-specific storage types, such as object stores, infinitely expandable network drives, and relational databases-as-a-service.

Running on top of this real and virtual hardware are a variety of software components that mediate between the raw storage devices and the filesystem hierarchy seen by users. These components include device drivers, partitioning conventions, RAID implementations, logical volume managers, systems for virtualizing disks over a network, and the filesystem implementations themselves.

In this chapter, we discuss the administrative tasks and decisions that occur at each of these layers. We begin with "fast path" instructions for adding a basic disk to Linux or FreeBSD. We

then review storage-related hardware technologies and look at the general architecture of storage software. We then work our way up the storage stack from low-level formatting to the filesystem level. Along the way, we cover disk partitioning, RAID systems, and logical volume managers.

Above the level of individual machines lie a variety of schemes for sharing data on a network. Chapters 21 and 22 describe two common file sharing systems: NFS for native sharing among UNIX and Linux systems, and SMB for interoperability with Windows and macOS systems.

20.1 I JUST WANT TO ADD A DISK!

Before we launch into many pages of storage architecture and theory, we first address the most common scenario: you want to install a hard disk and make it accessible through the filesystem. Nothing fancy: no RAID, all the drive's space in a single volume, and the default filesystem type.

Step one is to attach the drive. If the machine in question is a cloud server, you generally provision a virtual drive of the desired size within the provider's administrative GUI (or through their API) and then attach it to an existing virtual server as a separate step. It's normally unnecessary to reboot the server because cloud (and virtual) kernels recognize such hardware changes on the fly.

In the case of physical hardware, drives that communicate through a USB port can simply be powered on and plugged in. SATA and SAS drives need to be mounted in a bay, enclosure, or cradle. Although some hardware and drivers are designed to permit hot-addition of SATA drives, that feature requires hardware support and is uncommon in mass-market hardware. Reboot the system to make sure the OS is in a configuration that's reasonably reproducible at boot time.

If you're running a desktop machine with a window system and all the stars align, the system might offer to format a new disk for you when you plug it in. That's particularly likely if you're plugging in an external USB disk or thumb drive. The autoformat option usually works fine; use it if it's offered. However, check the mount details afterwards (by running the **mount** command in a terminal window) to make sure the drive hasn't been mounted with restrictions you don't want (e.g., with execution or normal ownerships disabled).

If you set up the disk by hand, it's critically important to identify and format the right disk device. A newly added drive is not necessarily represented by the highest-numbered device file, and on some systems, the addition of a new drive can change the device names of existing drives (after a reboot, usually). Double-check the identity of the new drive by reviewing its manufacturer, size, and model number before you do anything that's potentially destructive! Use the commands mentioned in the next two sections.

Linux recipe

First, run **lsblk** to list the system's disks and identify the new drive. If that output doesn't give you enough information to conclusively identify the new drive, you can list model and serial numbers with **lsblk -o +MODEL,SERIAL**.



See this page for an explanation of GPT partition tables.

Once you know which device file refers to the new disk (assume it's **/dev/sdb**), install a partition table on the disk. Several commands and utilities can do this, including **parted**, **gparted**, **fdisk**, **cfdisk**, and **sfdisk**; it doesn't matter which one you use, as long as it understands GPT-style partition tables. **gparted** is probably the easiest option on a system with a graphical user interface. Below, we show the **fdisk** recipe, which works on all Linux systems. (Some systems still ship a version of **parted** that doesn't understand GPT.)

```
$ sudo fdisk /dev/sdb
Welcome to fdisk (util-linux 2.23.2).

Changes will remain in memory only, until you decide to write them.
Be careful before using the write command.

Command (m for help): g
Building a new GPT disklabel (GUID:
    AB780438-DA90-42AD-8538-EEC9626228C7)

Command (m for help): n
Partition number (1-128, default 1): <Return>
First sector (2048-1048575966, default 2048): <Return>
Last sector, +sectors or +size{K,M,G,T,P} (2048-1048575966,
    default 1048575966): <Return>
Created partition 1

Command (m for help): w
The partition table has been altered!

Calling ioctl() to re-read partition table.
Syncing disks.
```

The **g** subcommand creates a GPT partition table. The **n** subcommand creates a new partition; pressing <Return> in response to all **fdisk**'s questions allocates all free space to the new partition (partition 1). Finally, the **w** subcommand writes the new partition table to the disk.

The device file for the newly created partition is the same as the device file for the disk as a whole with a **1** appended to it. In the example above, the partition is **/dev/sdb1**.

You can now create a filesystem on **/dev/sdb1** with the **mkfs** command. The **-L** option gives the filesystem a shorthand label (here, "spare"). The label stays the same even if the disk that contains the filesystem is assigned a different device name during a subsequent boot.

```
$ sudo mkfs -t ext4 -L spare /dev/sdb1
mke2fs 1.42.9 (28-Dec-2013)
Discarding device blocks: done
Filesystem label=spare
OS type: Linux
Block size=4096 (log=2)
...
```

Next, create a mount point and mount the new filesystem:

```
$ sudo mkdir /spare
$ sudo mount LABEL=spare /spare
```

You could equivalently specify **/dev/sdb1** instead of **LABEL=spare** as a way of identifying the partition, but that name won't necessarily work in the future.

To have the filesystem automatically mounted at boot time, edit the **/etc/fstab** file and duplicate one of the existing entries. Change the device name and mount point to match those shown in the **mount** command above. For example,

```
LABEL=spare  /spare         ext4      errors=remount-ro      0   0
```

You can also use a UUID to identify the filesystem; see this page.

See this page for more details on Linux device files for disks. See this page for partitioning information. See this page for information about the ext4 filesystem.

FreeBSD recipe



Run **geom disk list** to list the disk devices that the kernel is aware of. Unfortunately, FreeBSD doesn't divulge much information beyond device names and sizes. You can resolve any ambiguity as to which disk is which by running **geom part list** to see which devices have existing partitions. An unformatted disk should have no partitions.

Once you know the disk name, you can install a partition table and create a filesystem. In this example, we assume that the disk name is **ada1** and that you want to mount the new filesystem as **/spare**.

```
$ sudo gpart create -s GPT ada   # Create GPT partition table
ada1 created

$ sudo gpart add -l spare -t freebsd-ufs -a 1M ada1   # Create partition
ada1p1 added

$ sudo newfs -L spare /dev/ada1p1   # Create filesystem
/dev/ada1p1: 5120.0MB (10485680 sectors) block size 32768, fragment
    size 4096
    using 9 cylinder groups of 626.09MB, 20035 blks, 80256 inodes.
super-block backups (for fsck_ffs -b #) at:
 192, 1282432, 2564672, 3846912, 5129152, 6411392, 7693632,
    8975872, 10258112
...
```

The **-l** option to **gpart add** applies a text label to the new partition. The label makes the partition accessible through the path **/dev/gpt/spare** regardless of what device name the kernel assigns to the underlying disk device. The **-L** option to **newfs** applies a similar (but distinct) label to the new filesystem to make the partition accessible as **/dev/ufs/spare**.

Mount the filesystems with the following commands:

```
$ sudo mkdir /spare
$ sudo mount /dev/ufs/spare /spare
```

To have the filesystem automatically mounted at boot time, add it to the **/etc/fstab** file (see this page).

20.2 STORAGE HARDWARE

Even in today's post-Internet world, computer data can be stored in only a few basic ways: hard disks, flash memory, magnetic tapes, and optical media. The last two technologies have significant limitations that disqualify them from use as a system's primary filesystem. However, they're still sometimes used for backups and for "near line" storage—cases in which instant access and rewritability are not of primary concern.

After 40 years of traditional magnetic disk technology, performance-minded system builders finally received a practical alternative in the form of solid state disks (SSDs). These flash-memory-based devices offer a different set of tradeoffs from those of a standard disk, and they will be influencing the architectures of databases, filesystems, and operating systems for years to come.

At the same time, traditional hard disks are continuing their exponential increases in capacity. Thirty years ago, at the dawn of the 5.25" form factor that remains in use today, a 60MB hard disk cost $1,000. Today, a garden-variety 4TB drive runs $125 or so. That's more than 500,000 times more storage for the money, or double the TB/$ every 1.6 years. During that same period, the sequential throughput of mass-market drives has increased from 500 kB/s to 200 MB/s, a comparatively paltry factor of 400. And random-access seek times have hardly budged. The more things change, the more they stay the same.

See this page for more information on IEC units (gibibytes, etc.).

Disk sizes are specified in gigabytes that are billions of bytes, as opposed to memory, which is specified in "gigabytes" (gibibytes, really) of 2^{30} (1,073,741,824) bytes. The difference is about 7%. Be sure to check your units when estimating and comparing capacities.

Hard disks and SSDs are enough alike that they can act as drop-in replacements for each other, at least at the hardware level. They use the same hardware interfaces and interface protocols. And yet they have very different strengths, as Table 20.1 summarizes.

Table 20.1: Comparison of HDD and SSD technology

Characteristic	HDD	SSD
Typical size	< 16TB	< 2TB
Random access time [a]	8ms	0.25ms
Sequential read	200 MB/s	450 MB/s
Random read	2 MB/s	450 MB/s
IOPS [b]	150 ops/s	100,000 ops/s
Cost	$0.03/GB	$0.26/GB
Reliability	Poor	Poor [c]
Limited writes	No	In theory

a. Performance and cost values are as of mid 2017
b. I/O operations per second
c. Fewer whole-device failures than HDD, but more data loss

In the next sections, we take a closer look at each of these technologies along with a more recent category of storage devices: hybrid drives.

Hard disks

A typical hard drive contains several rotating platters coated with magnetic film. They are read and written by tiny skating heads mounted on a metal arm that swings back and forth to position them. The heads float close to the surface of the platters but don't actually touch them.

Reading from a platter is quick; it's the mechanical maneuvering needed to address a particular sector that drives down random-access throughput. Delays come from two main sources.

First, the head armature must swing into position over the appropriate track. This part is called seek delay. Second, the system must wait for the right sector to pass underneath the head as the platter rotates. That part is rotational latency. Disks can stream data at hundreds of MB/s if reads are optimally sequenced, but random reads are unlikely to achieve more than a few MB/s.

A set of tracks on different platters that are all the same distance from the spindle is called a cylinder. The cylinder's data can be read without any additional movement of the arm. Although heads move amazingly fast, they still move much more slowly than the disks spin around. Therefore, any disk access that does not require the heads to seek to a new position will be faster.

Spindle speeds vary. 7,200 RPM remains the mass-market standard for enterprise and performance-oriented drives. A few 10,000 RPM and 15,000 RPM drives remain available at the high end, but the advent of inexpensive SSDs now limits these drives to a small and shrinking niche market. Higher rotational speeds decrease latency and increase the bandwidth of data transfers, but the drives tend to run hot.

Hard disk reliability

Hard disks fail frequently. A 2007 Google Labs study of 100,000 drives surprised the tech world with the news that hard disks more than two years old had an average annual failure rate (AFR) of more than 6%, much higher than the failure rates manufacturers predicted from extrapolating their short-term testing. The overall pattern was a few months of infant mortality, a two-year honeymoon of annual failure rates of a few percent, and then a jump up to the 6%–8% AFR range. Overall, hard disks in the Google study had less than a 75% chance of surviving a five-year tour of duty.

Interestingly, Google found no correlation between failure rate and two environmental factors that were formerly thought to be important: operating temperature and drive activity. The complete paper can be found at goo.gl/Y7Senk.

More recently, Backblaze, a cloud storage provider, has posted regular updates about its experience with various hard disk models at backblaze.com/blog. This data is 10 years more recent than the original Google study but suggests the same basic pattern: high infant mortality followed by a two- or three-year honeymoon and then a precipitous rise in annual failure rate. The absolute numbers are pretty close, too.

Failure modes and metrics

Hard disk failures typically stem from either platter surface defects (bad blocks) or mechanical failures. Drives attempt to transparently correct errors in the former category and remap the recovered data to a different portion of the disk. When block errors become visible at the operating system level (i.e., in the logs), that means data has already been lost. It's a bad prognostic sign; pull the drive from service and replace it.

A disk's firmware and hardware interface usually remain operable after a failure, and it can be entertaining to attempt to query the disk for details about what's going on (see this page). However, disks are so cheap that it's rarely worth your time to do this except perhaps as a learning exercise.

Drive reliability is often quoted by manufacturers in terms of mean time between failures (MTBF), denominated in hours. A typical value for an enterprise drive is around 1.2 million hours. However, MTBF is a statistical measure and should not be read to imply that an individual drive will run for 140 years before failing.

MTBF is defined as the inverse of AFR in the drive's steady-state period—that is, after break-in but before wear-out. A manufacturer's MTBF of 1.2 million hours corresponds to an AFR of 0.7% per year. This value is almost, but not quite, concordant with the AFR range observed by Google and Backblaze (1%–2%) during the honeymoon years of their sample drives' lives.

Manufacturers' MTBF values are probably accurate, but they are cherry-picked from the most reliable phase of each drive's life. MTBF values should therefore be regarded as an upper bound on reliability; they do not predict your actual expected failure rate over the long term. (Our technical reviewer Jon Corbet refers to these as "reliability guaranteed not to exceed" values.) Based on the limited data quoted above, you might consider dividing manufacturers' MTBFs by a factor of 7.5 or so to arrive at a more realistic estimate of five-year failure rates.

Drive types

Only two manufacturers of hard drives remain: Seagate and Western Digital. You may see a few other brands for sale, but they're all ultimately made by these same two companies, both of which have been on decade-long acquisition binges.

Brands segment their hard disk offerings into a few general categories:

- **Value drives:** These products offer lots of storage at the lowest possible price point. Performance isn't a priority, but it's usually decent. Today's low-end drives are often faster than the high-performance drives of five or ten years ago.
- **Mass-market performance drives:** These step-up products targeted at end users (often gamers) have higher spindle speeds and larger caches than those of their value equivalents. They perform notably better than value drives on most benchmarks. As

with value drives, firmware tuning emphasizes single-user access patterns such as large sequential reads and writes. The drives often run hot.

- **NAS drives:** NAS stands for "network-attached storage," but these drives are intended for use in all sorts of servers, RAID systems, and arrays—anywhere that multiple drives are housed and accessed together. They're designed to be constantly on and working, and to balance performance, reliability, and low heat-emission.

  Benchmarks that replicate stand-alone access patterns may not reveal much performance difference from value drives, but NAS drives typically handle multiple streams of independent operations more intelligently because of firmware tuning. NAS drives often have a longer warranty than value drives; their pricing is somewhere between that for value and performance drives.

- **Enterprise drives:** "Enterprise" can mean a lot of things in the context of hard disks, but most commonly it means "expensive." Here's where you'll find drives with non-SATA interfaces and uncommon features such as 10,000+ RPM spindle speeds. These are generally premium drives with long (often, five-year) warranties.

The differences among these drive categories are about half real and half marketing. All classes of drives work fine in all applications, but performance and reliability may vary. NAS drives are probably the best all-around choice for drives to keep on hand to fill a variety of potential needs.

Hard disks are commodity products, and one brand's model of a given size, class, and spindle speed is much like another's. These days, you need a dedicated qualification laboratory to make fine distinctions among competing drives, at least in terms of performance.

Reliability is another matter. The Google and Backblaze data demonstrate significant differences among models. The least reliable are an order of magnitude more likely to fail than the best. Unfortunately, there's really no way to identify the turkeys until they've been sold for a year or two and have established a reputation in the real world.

That said, Hitachi (HGST, now part of Western Digital) deserves recognition as a particularly high-reliability brand. Over the last decade, its drives have consistently led the reliability charts. However, HGST-branded drives command a significant price premium over their competitors' offerings.

No matter; even the best drives are relatively failure prone. There's no escaping the need for backups and redundant storage when important data is at stake. Design your infrastructure with the assumption that drives will fail, then figure out how much an incrementally more reliable drive is worth within this context.

Warranties and retirement

Because hard drives are more likely to require warranty service than are other types of hardware, warranty length is an important purchasing consideration. The industry standard has shrunk to a

paltry two years, suspiciously close to the length of the average hard drive's honeymoon period. The three-year warranty offered on many NAS drives is a significant advantage.

Hard disk exchanges under warranty are straightforward if you can demonstrate that drives fail a diagnostic test supplied by the manufacturer. Test programs typically run only under Windows and are intolerant of virtualization environments and of intervening connection hardware such as USB cradles. If your operations entail frequent drive exchanges, you may find it worthwhile to maintain a dedicated Windows machine as a drive testing station.

It usually pays to be aggressive in taking drives out of service, even if you can't quite document that they are broken enough to be eligible for exchange under warranty. Even seemingly insignificant signs (e.g., funny noises or block errors within temporary files) are likely indications that a drive is nearing the end of its life.

Solid state disks

SSDs spread reads and writes across banks of flash memory cells, which are individually rather slow in comparison to modern hard disks. But because of parallelism, the SSD as a whole meets or exceeds the bandwidth of a traditional disk. The great strength of SSDs is that they continue to perform well when data is read or written at random, an access pattern that's predominant in real-world use.

Storage device manufacturers like to quote sequential transfer rates for their products because the numbers are impressively high. But for traditional hard disks, these sequential numbers have almost no relationship to the throughput observed with random reads and writes. It pays to know your workloads. For access patterns that are in fact heavily sequential, hard disks can still be competitive with SSDs, especially when hardware costs are taken into consideration.

SSDs' performance comes at a cost, however. Not only are they more expensive per gigabyte of storage than are hard disks, but they also introduce several new wrinkles and uncertainties into the storage equation. Anand Shimpi's March 2009 article on SSD technology is a superb introduction to the promise and perils of the SSD. It can be found at tinyurl.com/dexnbt.

Rewritability limits

Each page of flash memory in an SSD (typically 4KiB on current products) can be rewritten only a limited number of times (usually about 100,000, depending on the underlying technology). To limit the wear on any given page, the SSD firmware maintains a mapping table and distributes writes across all the drive's pages. This remapping is invisible to the operating system, which sees the drive as a linear series of blocks. Think of it as virtual memory for storage.

The theoretical limits on the rewritability of flash memory are probably less an issue than they might initially seem. Just as a matter of arithmetic, you would have to stream 100 MB/s of data to a 500GB SSD for more than 15 continuous years to start running up against the rewrite limit. The more general question of long-term SSD reliability is as yet unanswered, however. We have a pretty good idea of how SSDs manufactured five years ago held up over time, but today's products will no doubt behave differently.

Flash memory and controller types

SSDs are constructed from several types of flash memory. The main difference among the types has to do with how many bits of information are stored in each individual flash memory location. Single-level cells (SLC memories) store a single bit; they're the fastest but most expensive option. Also common in the mix are multilevel cells (MLC) and triple-level cells (TLC).

SSD reviews lovingly describe these implementation details as a matter of course, but it's not clear why buyers should care. Some SSDs are faster than others, but no particular hardware-related insight is needed to appreciate this fact. Standard benchmarks capture the performance differences quite well.

In theory, SLC flash memory has a reliability advantage over other types. In practice, reliability seems to have more to do with how well a drive's firmware manages the memory and with how much memory the manufacturer has set aside for replacing cells that develop problems.

The controllers that coordinate SSD components are still evolving. Some are better than others, but these days all mainstream offerings tend to be respectable. If you want to invest time in scrutinizing SSD hardware, it's usually more efficient to research the reputations of the flash memory controllers used to implement SSDs than to investigate the individual brands and models of SSD. SSD manufacturers are usually pretty open about the controllers they're using. If they won't tell you, reviewers certainly will.

Page clusters and pre-erasing

A further complication is that flash memory pages must be erased before they can be rewritten. SSDs handle this detail for you. However, erasing is a separate operation that is slower than writing. It's also impossible to erase individual pages—clusters of adjacent pages (typically 128 pages or 512KiB) must be erased together. The write performance of an SSD can drop substantially when the pool of pre-erased pages is exhausted and the drive must recover pages on-the-fly to service ongoing writes.

Rebuilding a buffer of erased pages is harder than it might seem because filesystems designed for traditional hard disks do not actually erase data blocks they are no longer using. A storage device doesn't know that the filesystem now considers a given block to be free; it knows only that long ago someone gave it data to store there. For an SSD to maintain its cache of pre-erased pages (and thus, its write performance), the filesystem must be capable of informing the SSD that certain pages are no longer needed. Support for this operation, known as TRIM, has finally become widespread among filesystems. On our example systems, the only filesystem that does not yet support TRIM is ZFS on Linux.

SSD reliability

A 2016 paper by Bianca Schroeder et al. (goo.gl/lzuX6c) summarized a vast set of SSD-related data from Google's data centers. The main conclusions:

- Memory technology has no relationship to reliability. Reliability varies widely among models, but as with hard disks, it can be assessed only retrospectively.
- Most read errors occur at the bit level and are corrected through redundant storage coding. These "raw" (but correctable) read errors are common and expected. They occur on most SSD drives on most days of operation.
- The most common failure mode is to discover more bad bits in a block than can be fixed by the coding system. These errors are detectable but uncorrectable; they necessarily entail data loss.

- Even among the most reliable SSD models, 20% of drives experienced at least one uncorrectable read error. Among the least reliable models, 63%.
- Although both drive age and workload correlate with uncorrectable error rates, the correspondence is weak. In particular, the study found no evidence for the notion that older SSDs are ticking time bombs that asymptotically approach certain failure.
- Because uncorrectable errors are only marginally correlated to workload, the standard reliability figure quoted by manufacturers—the uncorrectable bit error rate, or UBER—is meaningless. Workload has little effect on the number of errors observed, so reliability should not be characterized as a rate.

The most notable of these findings is that unreadable blocks are common and that they typically occur in isolation. The usual scenario is for an SSD to report a block error but then continue to function normally.

See Chapter 28 for more information about setting up a comprehensive surveillance program.

Of course, unreliable storage devices are nothing new; backups and redundancy remain essential no matter what hardware you're using. However, SSD failures are sneakier than those you might be accustomed to from dealing with hard disks. Unlike a hard disk, an SSD will rarely demand your attention by failing in some obvious and unambiguous way. SSDs need structured and systematic monitoring.

Errors develop over time regardless of a drive's duty cycle, so SSDs are probably not a good choice for archival storage. And conversely, an isolated bad block is not an indication that an SSD has gone bad or is nearing the end of its useful life. In the absence of a larger pattern of failures, it's fine to reformat such a drive and return it to service.

Hybrid drives

After spending many years in the vaporware category, SSHDs—hard disks with built-in flash memory caches—have become increasingly available. Current products are pitched at consumers.

The initialism SSHD stands for “solid state hybrid drive” and is something of a triumph of marketing, designed as it is to encourage confusion with SSDs. SSHDs are just traditional hard disks with some extras on the logic board; in reality, they’re about as “solid state” as the average dishwasher.

Benchmarks of current SSHD products have generally been unimpressive, even when the benchmarks attempt to emulate real-world access patterns. In large part, that’s because the current products often include only a token amount of flash memory cache.

Despite current SSHDs’ lackluster performance, the basic idea of multilevel caching is sound and has been well exploited in systems such as ZFS and Apple’s Fusion Drive. As the price of flash memory continues to fall, we anticipate that platter-based drives will continue to include more and more cache. Those products may or may not be sold explicitly as SSHDs.

Advanced Format and 4KiB blocks

For decades, the standard size of a disk block was fixed at 512 bytes. That's too small to be practical from the perspective of most filesystems, so the filesystems themselves have long aggregated 512-byte sectors into page clusters of 1KiB to 8KiB that are read and written together.

Since no software that communicates with storage hardware actually has an interest in reading and writing data at 512-byte granularity, it's inefficient and wasteful for the hardware to maintain such tiny sectors. Over the last decade, the storage industry has migrated to a new standard block size of 4KiB, known as Advanced Format. All modern storage devices use 4KiB sectors internally, although most of them continue to emulate 512-byte blocks from the perspective of clients.

There are currently three different "worlds" that a storage device can live in:

- 512n (or 512-native) devices are the old ones that actually have 512-byte sectors. These devices are no longer manufactured, but of course there are still plenty of them out there in the real world. These drives know nothing about Advanced Format.
- 4Kn (or 4K-native) devices are Advanced Format devices that have 4KiB sectors (or pages, in the case of SSDs) and that report their block size as 4KiB to the host computer. All interfacing hardware and all software that deals directly with the device must be aware of, and prepared to deal with, 4KiB blocks.

  4Kn is the wave of the future, but because it demands both hardware and software support, its adoption will be gradual. Enterprise drives with 4Kn interfaces started becoming available in 2014, but at this point you're in no danger of encountering a 4Kn drive unless you explicitly order one.

- 512e (or 512-emulated) devices use 4KiB blocks internally, but they report their sector size as 512 bytes to the host computer. Firmware in the device aggregates 512-byte block operations into operations on the actual 4KiB storage blocks.

The transition from 512n to 512e was completed in 2011. These two systems look essentially identical from the perspective of the host computer, so 512e devices work fine with old computers and old operating systems.

The one thing to know about 512e is that it's sensitive to misalignment between filesystem page clusters and hardware disk blocks. Because the disk can only read or write 4KiB pages (despite its emulation of traditional 512-byte blocks), filesystem cluster boundaries and hard disk block boundaries should coincide. You wouldn't want a 4KiB logical cluster to correspond to half of one 4KiB disk block and half of another—with that layout, the disk might have to read or write twice as many physical pages as it should to service a given number of logical clusters.

Since filesystems usually count off their clusters starting at the beginning of whatever storage is allocated to them, you can finesse the alignment issue by aligning disk partitions to a power-of-2 boundary that is large in comparison with the likely size of disk and filesystem pages (e.g., 64KiB). Partitioning tools on modern versions of Windows, Linux, and BSD automatically enforce such alignment. However, 512e disks that were mispartitioned on legacy systems can't be transparently corrected; you'll need to run an alignment utility to adjust the partition boundaries and physically move the data. Or, you can simply erase the device entirely and start over.

20.3 Storage hardware interfaces

These days, only a few interface standards are in common use. If a system supports several different interfaces, use the one that best meets your requirements for speed, redundancy, mobility, and price.

The SATA interface

Serial ATA, SATA, is the predominant hardware interface for storage. In addition to supporting high transfer rates (currently 6 Gb/s), SATA has native support for hot-swapping and (optional) command queueing, two features that finally make ATA a viable alternative to SAS in server environments.

SATA cables slide easily onto their mating connectors, but they can just as easily slide off. Cables with locking catches are available, but they're a mixed blessing. On motherboards with six or eight SATA connectors packed together, it can be hard to disengage the locking connectors without a pair of needle-nosed pliers.

SATA also introduces an external cabling standard called eSATA. The cables are electrically identical to standard SATA, but the connectors are slightly different. You can add an eSATA port to a system that has only internal SATA connectors by installing an inexpensive converter bracket.

Be leery of external multidrive enclosures that have only a single eSATA port—some of these are smart (RAID) enclosures that require a proprietary driver, and the drivers rarely support UNIX or Linux. Others are dumb enclosures that have a SATA port multiplier built in. These are potentially usable on UNIX systems, but since not all SATA host adapters support port expanders, pay close attention to the compatibility information. Enclosures with multiple eSATA ports—one per drive bay—are always safe.

The PCI Express interface

The PCI Express (Peripheral Component Interconnect Express, abbreviated PCIe) backplane bus has been used on PC motherboards for more than a decade. It's now the predominant standard for connecting all kinds of add-on circuit boards, even video cards.

As the SSD market developed, it became clear that even at 6 Gb/s, the speed of SATA interfaces would soon become inadequate to handle the fastest storage devices. Rather than assuming the traditional shape of a 2.5" laptop hard disk, high-end SSDs began to take the form of circuit boards that plugged directly into the system's PCIe bus.

PCIe was attractive because of its flexible architecture and fast signaling rate. The version that is now mainstream, PCIe 3.0, has a signaling rate of 8 gigatransfers per second (GT/s). The actual throughput depends on how many signaling channels a device has; there can be as few as 1 or as many as 16. The widest devices can achieve more than 15 GB/s of throughput. (It's not quite 16 GB/s because some of the bandwidth is consumed by signaling overhead. However, the amount of overhead is so small—about 1.5%—that it can safely be ignored.) The soon-to-debut PCIe 4.0 standard doubles the basic signaling rate to 16 GT/s.

When comparing PCIe to SATA, keep in mind that SATA's speed of 6 Gb/s is quoted in giga*bits* per second. Full-width PCIe is actually more than 20 times faster than SATA.

The SATA standard is feeling the pressure. Unfortunately, the SATA ecosystem is constrained by past design choices and by the need to support existing cabling and connectors. It's unlikely that the speed of SATA interfaces can be meaningfully improved over the next few years.

Instead, recent work has focused on attempting to unify SATA and PCIe at the level of interconnections. The M.2 standard for plug-in cards routes SATA, PCIe (with up to four data lanes), and USB 3.0 connectivity over a standard connector. One or two of these slots are now standard on laptop computers, and they can also be found on desktop systems.

M.2 cards are about an inch wide and can be up to about four inches long. They are thin, with only a few millimeters allowed on both sides for components.

U.2 is more recent tweak to the M.2 approach; it's just starting to become available. Instead of USB, U.2 feature SAS connectivity in addition to SATA and PCIe.

The SAS interface

SAS stands for Serial Attached SCSI, the SCSI portion of which denotes the Small Computer System Interface, a generic data pipe that once connected many different types of peripherals. These days, USB has captured the market for peripheral connections and SCSI is found only in the form of SAS, an enterprise-level interface used to connect large numbers of storage devices.

Now that SAS and SCSI are largely synonymous, the vast history of different SCSI technologies dating back to 1986 serves mostly to create confusion. Operating systems further muddy the waters by filtering all disk access through a "SCSI subsystem" regardless of whether an actual SCSI device is involved or not. Our advice is to ignore all this history and consider SAS as its own system.

Like SATA, SAS is a point-to-point system: you plug a drive into a SAS port through a cable or direct-mount backplane. However, SAS allows "expanders" to connect multiple devices to a single host port. They're analogous to SATA port multipliers, but whereas support for port multipliers is hit or miss, SAS expanders are always supported.

SAS currently operates at 12 Gb/s, twice the speed of SATA.

In past editions of this book, SCSI was the obvious interface choice for server applications. It offered the highest available bandwidth, out-of-order command execution (aka tagged command queueing), lower CPU utilization, easier handling of large numbers of storage devices, and access to the market's most advanced hard drives.

The advent of SATA has removed or minimized most of these advantages, so SAS simply does not deliver the clear advantages that SCSI used to. SATA drives compete with (and in some cases, outperform) equivalent SAS disks in nearly every category. At the same time, both SATA devices and the interfaces and cabling used to connect them are cheaper and far more widely available.

SAS still holds a few trump cards:

- Manufacturers continue to use the SATA/SAS divide to stratify the storage market. To help justify premium pricing, the fastest and most reliable drives are still available only with SAS interfaces.
- SATA is limited to a queue depth of 32 pending operations. SAS can handle thousands.
- SAS can handle many storage devices (hundreds or thousands) on a single host interface. But keep in mind that all those devices share a single pipe to the host; you are still limited to 12 Gb/s of aggregate bandwidth.

The SAS vs. SATA debate may ultimately be moot because the SAS standard includes support for SATA drives. SAS and SATA connectors are similar enough that a single SAS backplane can

accommodate drives of either type. At the logical layer, SATA commands are simply tunneled over the SAS bus.

This convergence is an amazing technical feat, but the economic argument for it is less clear. The expense of a SAS installation is mostly in the host adapter, backplane, and infrastructure; the SAS drives themselves aren't outrageously priced. Once you've invested in a SAS setup, you might as well stick with SAS from end to end. (On the other hand, perhaps the modest price premiums for SAS drives are a *result* of the fact that SATA drives can easily be substituted for them.)

USB

The Universal Serial Bus (USB) is a popular option for connecting external hard disks. Current speeds are 4 Gb/s for USB 3.0 and up to 10 GB/s for USB 3.1. (The speed of USB 3.0 is often cited as 5 Gb/s, but because of mandatory encoding overhead, the actual transfer rate is more like 4 Gb/s.)

Both USB 3.0 and USB 3.1 are fast enough to accommodate all but the fastest SSDs streaming data at full speed. Watch out for USB 2.0, however; it tops out at 480 Mb/s, which is too slow to keep up with even a mechanical hard drive.

Storage devices themselves never come with native USB interfaces. External drives sold with these interfaces are invariably SATA drives with a protocol converter built into the enclosure. You can also buy these enclosures separately and install your choice of hard disks.

USB adapters are also available in the form of cradles and cable dongles. Cradles are particularly helpful when disks must be swapped out frequently: just yank out the old disk and pop in a new one.

USB thumb drives are perfectly legitimate storage devices. They present a block interface similar to that of any other disk, although throughput is typically mediocre. The underlying technology is similar to that of an SSD, but without some of the flourishes that give SSDs their superior speed and robustness.

20.4 Attachment and low-level management of drives

The way a disk is attached to the system depends on the interface. The rest is all mounting brackets and cabling. Fortunately, modern connection schemes are all pretty much idiot-proof.

See this page for more information about dynamic handling of devices.

SAS is a hot-pluggable interface, so it's fine to plug in new drives without powering off the system or restarting it. The kernel should automatically recognize new devices and create device files for them. SATA interfaces can also theoretically support hot-plugging. However, the SATA specification does not require support for this feature, and most mass-market hardware does not implement it.

It's fine to attempt hot-plugging a SATA drive to find out if hot-plugging works on a particular system. You won't hurt anything. The worst that can happen is that the system ignores the drive.

Hot-plugging might seem like a neat trick that creates all sorts of options, such as the ability to swap out a bad drive with little or no software-side wrangling. However, it's tricky to get the higher layers of the storage stack tuned to achieve these feats safely and reliably. We don't describe the management of hot-plugging in this book.

Installation verification at the hardware level

After you install a new disk, check to make sure that the system acknowledges its existence at the lowest possible level. On a physical PC this is easy: the BIOS shows you a list of SATA and USB disks connected to the system. SAS disks may be included here as well if the motherboard supports them directly. If the system has a separate SAS interface card, you might need to invoke the BIOS setup for that card to see the disk inventory.

On cloud servers and systems that support hot-pluggable drives, you might have to do some sleuthing. Check the diagnostic output from the kernel as it probes for devices. For example, one of our test systems showed the following messages for an older SCSI disk attached to a BusLogic SCSI host adapter.

```
scsi0 : BusLogic BT-948
scsi : 1 host.
  Vendor: SEAGATE   Model: ST446452W       Rev: 0001
  Type:   Direct-Access                    ANSI SCSI revision: 02
Detected scsi disk sda at scsi0, channel 0, id 3, lun 0
scsi0: Target 3: Queue Depth 28, Asynchronous
SCSI device sda: hdwr sector=512 bytes. Sectors=91923356
    [44884 MB] [44.9 GB]
```

You may be able to review this information after the system has finished booting: look in your system log files. See the material starting on this page for more information about the handling of boot-time messages from the kernel.

Several commands can print out a list of the disks that the system is aware of. On Linux systems, the best option is usually **lsblk**, which is standard on all distributions. For more information, ask for model and serial numbers:

lsblk -o +MODEL,SERIAL

On FreeBSD, use **geom disk list**.

Disk device files

A newly added disk is represented by device files in **/dev**. See this page for general information about device files.

All our example systems automatically create these files for you, but you still need to know where to look for the device files and how to identify the ones that correspond to your new device. Formatting the wrong device file is a rapid route to disaster.

Table 20.2 summarizes the device naming conventions for disks on our example systems. Instead of showing the abstract pattern according to which devices are named, Table 20.2 simply shows a typical example for the name of the system's first disk.

Table 20.2: Device naming standards for disks

System	Whole disk	Partition
Linux	**/dev/sda**	**/dev/sda1**
FreeBSD	**/dev/ada0**	**/dev/ada0p1**

Device names for whole disks comprise a basename that depends on the device driver and a sequence number or letter that differentiates disks from each other. For example, **/dev/sda** on Linux is the first drive managed by the sd driver. The next drive would be **/dev/sdb**, and so on. FreeBSD has different driver names and uses numbers instead of letters, but the pattern is the same.

Don't ascribe too much significance to the driver names that show up in disk device files. Modern kernels funnel both SATA and SAS management through a generic SCSI layer, so don't be surprised to see SATA disks masquerading as SCSI devices. Driver names also vary on cloud and virtualized systems; a virtual SATA disk may or may not have the same driver name as an actual SATA disk.

Device files for partitions add an additional decoration to the device file to indicate the partition number. Partition numbering normally starts at 1 rather than 0.

Ephemeral device names

Disk names are assigned in sequence as the kernel enumerates the various interfaces and devices on the system. Adding a disk can cause existing disks to change their names. In fact, even rebooting the system can sometimes cause name changes.

These facts suggest a couple of good rules for system administrators to follow:

- Never make changes to disks, partitions, or filesystems without verifying the identity of the disk you're working on, even on a stable system.
- Never mention a disk device in any sort of configuration file, lest it change out from under you at some point in the future.

The latter issue is most notable when you are setting up the **/etc/fstab** file, which lists filesystems for the system to mount at boot time. It was once common to identify disk partitions by their device files in **/etc/fstab**, but this is no longer safe. See this page for some alternative approaches.



Linux has a couple of general ways around the "ephemeral names" issue. Subdirectories under **/dev/disk** list disks by various stable characteristics such as their manufacturer ID or connection information. These device names (which are really just links back to the top-level files in **/dev**) are stable, but they're long and awkward.

At the level of filesystems and disk arrays, Linux uses both unique ID strings and text labels to persistently identify objects. In many cases, the existence of these long IDs is cleverly concealed so that you don't have to deal with them directly.

parted -l lists the sizes, partition tables, model numbers, and manufacturers of every disk on the system.

Formatting and bad block management

People sometimes use the word “formatting” to mean “writing a partition table on a disk and setting up filesystems in the partitions.” But in this section, we use the word “formatting” to mean the more fundamental operation of setting up a disk’s media at the hardware level. We’d prefer to call the former operation “initializing,” but in the real world the terms are used more or less interchangeably, so you have to decode the meaning through context.

The formatting process writes address information and timing marks on the platters to delineate each sector. It also identifies bad blocks, imperfections in the media that result in areas that cannot be reliably read or written. All modern disks have bad block management built in, so neither you nor the driver need worry about managing defects. The drive firmware substitutes known-good blocks from an area of backup storage on the disk that is reserved for this purpose.

All hard disks come preformatted, and the factory formatting is at least as good as any formatting you can do in the field. It is best to avoid doing a low-level format if it’s not required. Don’t reformat new drives as a matter of course.

If you encounter read or write errors on a disk, first check for cabling, termination, and address problems, all of which can cause symptoms similar to those of a bad block. If after this procedure you are still convinced that the disk has defects, you might be better off replacing it with a new one rather than waiting long hours for a format to complete and hoping the problem doesn’t come back.

Bad blocks that manifest themselves after a disk has been formatted may or may not be automatically handled. If the drive is sure that the affected data can be reliably reconstructed, the newly discovered defect might be mapped out on the fly and the data rewritten to a new location. For more serious or less clearly recoverable errors, the drive aborts the read or write operation and reports the error back to the host operating system.

SATA disks are usually not designed to be formatted outside the factory. However, you might be able to obtain formatting software from the manufacturer, usually for Windows. Make sure the software matches the drive you plan to format and follow the manufacturer’s directions carefully. (On the other hand, at $100 for a 4TB drive, why bother?)

SAS disks format themselves in response to a standard command that you send from the host computer. The procedure for sending this command varies from system to system. On PCs, you can often send the command from the SAS controller’s BIOS. To issue the format command from within the operating system, use the **sg_format** command on Linux and the **camcontrol** command on FreeBSD.

Various utilities let you verify the integrity of a disk by writing random patterns to it and then reading them back. Thorough tests take a long time (hours) and unfortunately seem to be of little prognostic value. Unless you suspect that a disk is bad and are unable to simply replace it (or you bill by the hour), you can skip these tests. Barring that, let the tests run overnight. Don’t be

concerned about “wearing out” a disk with overuse or aggressive testing. Enterprise-class disks are designed for constant activity.

ATA secure erase

Since 2000, PATA and SATA disks have implemented a "secure erase" command that overwrites the data on the disk according to a method the manufacturer has determined to be secure against recovery efforts. Secure erase is NIST-certified for most needs. Under the U.S. Department of Defense categorization, it's approved for use at security levels less than "secret."

Why is this feature even needed? First, filesystems generally do no erasing of their own, so an **rm -rf *** of a disk's data leaves everything intact and recoverable with software tools. It's critically important to remember this fact when disposing of disks, whether their destination is eBay or the trash. (Now that most filesystems support the TRIM command to inform SSDs of blocks that are no longer needed by the system, this statement is not quite as true as it used to be. However, TRIM is advisory; an SSD is not required to erase anything in response.)

Second, even a manual rewrite of every sector on a traditional hard disk can leave magnetic traces that are recoverable by a determined attacker with access to a laboratory. Secure erase performs as many overwrites as are needed to eliminate these shadow signals. Magnetic remnants won't be a serious concern for most sites, but it's always nice to know that you're not exporting your organization's confidential data to the world at large. Some sites may have regulatory or business requirements that dictate how data is to be erased.

Finally, secure erase has the effect of resetting SSDs to their fully erased state. This reset can improve performance in cases in which the ATA TRIM command (the command to erase a block) cannot be issued, either because the filesystem used on the SSD does not know to issue it or because the SSD is connected through a host adapter or RAID interface that does not propagate TRIM.

The ATA secure erase command is password-protected at the drive level to reduce the risk of inadvertent activation. Therefore, you must set the password on a drive before invoking the command. Don't bother to record the password, however; you can reset it at will. There is no danger of locking the drive.



Under Linux, you can use the **hdparm** command to activate secure erase:

```
$ sudo hdparm --user-master u --security-set-pass password /dev/disk
$ sudo hdparm --user-master u --security-erase password /dev/disk
```



The analogous FreeBSD command is **camcontrol**:

```
$ sudo camcontrol security disk -U user -s password -e password
```

The SAS world has no analog to ATA's secure erase command, but the SCSI "format unit" command described under *Formatting and bad block management* is a reasonable alternative.

Many systems have a **shred** utility that attempts to securely erase the contents of individual files. Unfortunately, it relies on the assumption that a file's blocks can be overwritten in place. This assumption is invalid in so many circumstances (any filesystem on any SSD, any logical volume that has snapshots, anything on ZFS or Btrfs) that **shred**'s general utility is questionable.

For sanitizing an entire PC system at once, another option is Darik's Boot and Nuke (dban.org). This tool runs from its own boot disk, so it's not a tool you'll use every day. It is quite handy for decommissioning old hardware, however.

hdparm and camcontrol: set disk and interface parameters

The **hdparm** (Linux) and **camcontrol** (FreeBSD) commands can do more than just send secure erase commands. They give you a general way to interact with the firmware of SATA and SAS hard disks.

As tools that operate close to the hardware layer, these commands work properly only on nonvirtualized systems. On a traditional physical server, they are actually the best way to get information about the system's disk devices (**hdparm -I** and **camcontrol devlist**); we don't mention them elsewhere (e.g., in the "adding a disk" recipes at the start of this chapter) only because they don't work on virtual systems.

hdparm comes from the prehistoric world of IDE and has gradually grown to include coverage of SATA and SCSI features. **camcontrol** started as a SCSI wrangling tool and has been extended to cover some SATA features. The syntaxes are different, but the tools cover approximately the same territory these days.

Among other things, these tools can set drive power options, enable or disable noise reduction options, set the read-only flag, and print detailed drive information.

Hard disk monitoring with SMART

Hard disks are fault-tolerant systems that use error-correction coding and intelligent firmware to hide their imperfections from the host operating system. In some cases, an uncorrectable error that the drive is forced to report to the OS is merely the latest event in a long crescendo of correctable but inauspicious problems. It would be nice to know about those omens before the crisis occurs.

SATA devices implement a detailed form of status reporting that is sometimes predictive of drive failures. This standard, called SMART, for "self-monitoring, analysis, and reporting technology," exposes more than 50 operational parameters for investigation by the host computer.

The Google disk drive study mentioned on this page has been widely summarized in media reports as concluding that SMART data is not predictive of drive failure. That summary is not accurate. In fact, Google found that four SMART parameters were highly predictive of failure but that failure was not consistently preceded by changes in SMART values. Of failed drives in the study, 56% showed no change in the four most predictive parameters. On the other hand, predicting nearly half of failures sounds pretty good to us!

Those four sensitive SMART parameters are

- Scan error count
- Reallocation count
- Off-line reallocation count
- Number of sectors on probation

Those values should all be zero. According to the Google Labs study, a nonzero value in these fields raises the likelihood of failure within 60 days by a factor of 39, 14, 21, or 16, respectively.

To take advantage of SMART data, you need software that queries your drives to obtain it and then judges whether the current readings are sufficiently ominous to warrant administrator notification. Unfortunately, reporting standards vary by drive manufacturer, so decoding isn't necessarily straightforward. Most SMART monitors collect baseline data and then look for sudden changes in the "bad" direction rather than interpreting absolute values. (According to the Google study, taking account of these "soft" SMART indicators in addition to the Big Four predicts 64% of all failures.)

The standard software for SMART wrangling is the smartmontools package from smartmontools.org. It's installed by default on Red Hat, CentOS, and FreeBSD systems and is usually in the default package repository on other systems.

The smartmontools package consists of a **smartd** daemon that monitors drives continuously and a **smartctl** command you can use for interactive queries or for scripting. The daemon has a single configuration file, normally **/etc/smartd.conf**, which is extensively commented and includes plenty of examples.

SCSI has its own system for out-of-band status reporting, but unfortunately the standard is much less granular in this respect than is SMART. The smartmontools attempt to include SCSI devices in their schema, but the predictive value of the SCSI data is less clear.

20.5 The software side of storage: peeling the onion

If you're accustomed to plugging in a disk and having your Windows system ask if you want to format it, you may be a bit taken aback by the apparent complexity of storage management on UNIX and Linux systems. Why is it all so complicated?

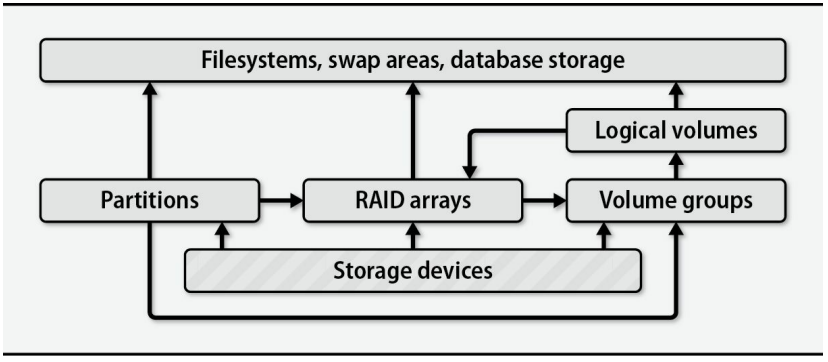
To begin with, much of the complexity is optional. On UNIX and Linux systems with a window manager, you can log in to your system's desktop, connect that same USB drive, and have much the same experience as on Windows. You'll get a simple setup for personal data storage. If that's all you need, you're good to go.

As usual in this book, we're primarily interested in enterprise-class storage systems: filesystems that are accessed by many users or processes (both local and remote) and that are reliable, high-performance, easy to back up, and easy to adapt to future needs. These systems require a bit more thought, and UNIX and Linux give you plenty to think about.

Elements of a storage system

Exhibit A shows a typical set of software components that can mediate between a raw storage device and its end users. The architecture shown in Exhibit A is for Linux, but other systems include similar features, although not necessarily in the same packages.

Exhibit A: Storage management layers



The arrows in Exhibit A mean "can be built on." For example, a Linux filesystem can be built on top of a partition, a RAID array, or a logical volume. It's up to the administrator to construct a stack of modules that connect each storage device to its final application.

Sharp-eyed readers will note that the graph has a cycle, but real-world configurations should not loop. Linux allows RAID and logical volumes to be stacked in either order, but neither component should be used more than once (though it is technically possible to do this).

Here's what the pieces in Exhibit A represent:

- A **storage device** is anything that looks like a disk. It can be a hard disk, a flash drive, an SSD, an external RAID array implemented in hardware, or even a network service that gives block-level access to a remote device. The exact hardware doesn't matter, as long as the device allows random access, handles block I/O, and is represented by a device file.
- A **partition** is a fixed-size subsection of a storage device. Each partition has its own device file and acts much like an independent storage device. For efficiency, the same driver that handles the underlying device usually implements partitioning. Partitioning schemes consume a few blocks at the start of the device to record the ranges of blocks in each partition.
- **Volume groups** and **logical volumes** are associated with logical volume managers (LVMs). These systems aggregate physical devices to form pools of storage called volume groups. An administrator can then subdivide this pool into logical volumes in

much the same way that disks can be divided into partitions. For example, a 6TB disk and a 2TB disk could be aggregated into an 8TB volume group and then split into two 4TB logical volumes. At least one volume would include data blocks from both hard disks.

Since the LVM adds a layer of indirection between logical and physical blocks, it can freeze the logical state of a volume simply by making a copy of the mapping table. Therefore, logical volume managers often have some kind of a "snapshot" feature. Writes to the volume are then directed to new blocks, and the LVM keeps both the old and new mapping tables. Of course, the LVM has to store both the original image and all modified blocks, so it can eventually run out of space if a snapshot is never deleted.

- A **RAID array** (a redundant array of inexpensive/independent disks) combines multiple storage devices into one virtualized device. Depending on how you set up the array, this configuration can increase performance (by reading or writing disks in parallel), increase reliability (by duplicating or parity-checking data across multiple disks), or both. RAID can be implemented by the operating system or by various types of hardware.

  As the name suggests, RAID is typically conceived of as an aggregation of bare drives, but modern implementations let you use as a component of a RAID array anything that acts like a disk.

- A **filesystem** mediates between the raw bag of blocks presented by a partition, RAID array, or logical volume and the standard filesystem interface expected by programs: paths such as **/var/spool/mail**, UNIX file types, UNIX permissions, etc. The filesystem determines where and how the contents of files are stored, how the filesystem namespace is represented and searched on disk, and how the system is made resistant to (or recoverable from) corruption.

  Most storage space ends up as part of a filesystem, but on some systems (not current versions of Linux), swap space and database storage can potentially be slightly more efficient without "help" from a filesystem. The kernel or database imposes its own structure on the storage, rendering the filesystem unnecessary.

If it seems to you that this taxonomy has a few too many little components that simply implement one block storage device in terms of another, you're in good company. The trend over the last few years has been toward consolidating these components to increase efficiency and remove duplication. Although logical volume managers did not originally function as RAID controllers, most have absorbed some RAID-like features (notably, striping and mirroring).

On the cutting edge today are systems that combine a filesystem, a RAID controller, and an LVM system all in one tightly integrated package. ZFS was the earliest example, but the Btrfs

filesystem for Linux has similar design goals. We have lots more to say about ZFS and Btrfs starting on this page. (Spoiler alert: if you can use one of these systems, you probably should.)

The Linux device mapper



For simplicity, we omitted a central component of the Linux storage stack from Exhibit A: the device mapper. This is a protean little beastie that has fingers inserted in multiple contexts, prime examples being the implementation of LVM2, the implementation of filesystem layers for containerization (see Chapter 25), and the implementation of whole-disk encryption (search the web for LUKS).

The device mapper abstracts the idea of one block device being built on a collection of other block devices. Given a mapping table of devices, it implements the ongoing translation among them and routes each block to its appropriate home.

For the most part, the device mapper is part of the implementation of Linux storage and not something you'll deal with directly. However, you'll see its traces whenever you access devices under **/dev/mapper**. You can also set up your own mapping tables with the **dmsetup** command, although cases in which you might need to do that are relatively rare.

In the next sections, we look in more detail at the layers involved in storage configuration: partitioning, RAID, logical volume management, and filesystems.

20.6 DISK PARTITIONING

Partitioning and logical volume management are both ways of dividing up a disk (or pool of disks, in the case of LVM) into separate chunks of known size. Linux and FreeBSD support both of these methods.

Traditionally, partitioning was the lowest possible level of disk management, and only disks could be partitioned. You could put individual disk partitions under the control of a RAID controller or logical volume manager, for example, but you couldn't then partition the resulting logical volumes or RAID volumes.

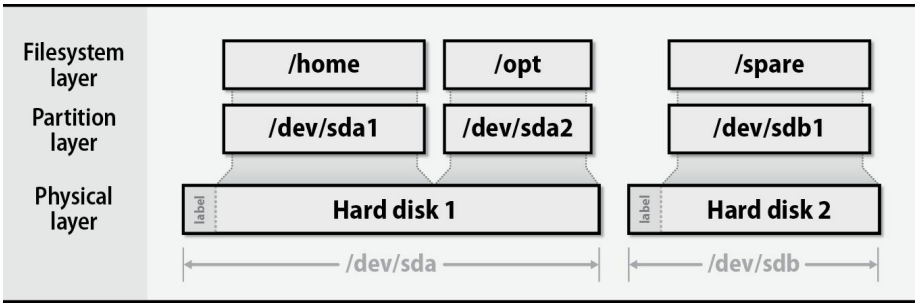
The rule that only disks can be partitioned is increasingly being waived in favor of a more general model in which disks, partitions, LVM pools, and RAID arrays can be derived from one another in any order or combination. From the standpoint of software architecture, this is beautiful and elegant. But from the standpoint of practicality, it has the unfortunate side effect of implying that there's some valid reason to partition entities other than disks.

In fact, partitioning is less desirable than logical volume management in most respects. It's coarse and brittle and lacks features such as snapshot management. Partitioning decisions are difficult to revise later. The only notable advantages of partitioning over logical volume management are its simplicity and the fact that Windows and PC BIOSs understand and expect it. A few versions of UNIX that run on proprietary hardware have done away with partitioning altogether, and nobody on those systems seems to miss it.

Both partitions and logical volumes make backups easier, prevent users from poaching each other's disk space, and confine potential damage from runaway programs. All systems have a root "partition" that includes / and most of the local host's configuration data. In theory, everything needed to bring the system up to single-user mode is part of the root partition. Various subdirectories (most commonly **/var**, **/usr**, **/tmp**, **/share**, and **/home**) can be broken out into their own partitions or volumes. Most systems also have at least one swap area.

Opinions differ on the best way to divide up disks, as do the defaults used by various systems. Most setups are relatively simple. Exhibit B illustrates a traditional partitions-and-filesystems schema as it might be found on a couple of data disks on a Linux system. (The boot disk is not shown.)

Exhibit B: Traditional data disk partitioning scheme (Linux device names)



Here are some general points to guide you:

- In the distant past, it was sometimes useful to have a backup root device that you could boot to if something went wrong with the normal root partition. These days, a bootable USB thumb drive or an OS installation DVD is a better recovery option for most systems. Backup root partitions are more trouble than they're worth.
- Putting **/tmp** on a separate filesystem limits temporary files to a finite size and saves you from having to back them up. Some systems use a memory-based filesystem to hold **/tmp** for performance reasons. The memory-based filesystems are still backed by swap space, so they work well in a broad range of situations.
- Since log files are kept in **/var/log**, it's a good idea for either **/var** or **/var/log** to be a separate disk partition. Leaving **/var** as part of a small root partition makes it easy to fill the root and bring the machine to a halt.
- It's useful to put users' home directories on a separate partition or volume. Even if the root partition is corrupted or destroyed, user data has a good chance of remaining intact. Conversely, the system can continue to operate even after a user's misguided shell script fills up **/home**.
- Splitting swap space among several physical disks can potentially increase performance, although with today's cheap RAM it's usually better not to swap at all. This technique works for filesystems, too; put the busy ones on different disks. See this page for notes on this subject.
- As you add memory to your machine, also add swap space. See this page for more information about virtual memory.
- Try to cluster quickly changing information on a few partitions that are backed up frequently.
- The Center for Internet Security publishes configuration guidelines for a variety of operating systems at www.cisecurity.org/cis-benchmarks. They are "benchmarks" in the sense of being best practices. The documents include helpful recommendations for partitioning and filesystem layout.

Traditional partitioning

Systems that support partitions implement them by writing a “label” at the beginning of the disk to define the range of blocks included in each partition. The exact details vary; the label must often coexist with other startup information (such as a boot block), and it often contains extra information such as a name or unique ID that identifies the disk as a whole.

The device driver responsible for representing the disk reads the label and uses the partition table to calculate the physical location of each partition. Typically, one device file represents each partition and an additional device file represents the disk as a whole.

Despite the universal availability of logical volume managers, some situations still require or benefit from traditional partitioning.

- Only two partitioning schemes are used these days: MBR and GPT. We discuss the details of both schemes in the next sections.
- On PC hardware, the boot disk must have a partition table. Systems manufactured before 2012 usually require MBR, and some new systems require GPT. Most new systems support both.
- Installing an MBR or GPT partition table makes a disk comprehensible to Windows, even if the contents of the individual partitions are not. Though you may have no particular plans to interoperate with Windows, consider the ubiquity of Windows, the prevalence of virtual machines, and the portability of hard disks.
- Partitions have a defined location on the disk, so they guarantee locality of reference. Logical volumes do not (at least, not by default). In most cases, this fact isn’t terribly important. However, short seeks are faster than long seeks on mechanical hard disks, and the throughput of a disk’s outer cylinders (those containing the lowest-numbered blocks) can exceed the throughput of its inner cylinders by 30% or more.
- RAID systems (see this page) use disks or partitions of matched size. A given RAID implementation might accept entities of different sizes, but it will probably use only the block ranges that all devices have in common. Rather than letting extra space go to waste, you can isolate it in a separate partition. If you do this, however, use the spare partition for data that is infrequently accessed; otherwise, traffic on the partition will degrade the performance of the RAID array.

MBR partitioning

MBR (Master Boot Record) partitioning is an old Microsoft standard that dates back to the 1980s. It's a cramped and ill-conceived format that can't support disks larger than 2TB. Who knew disks could ever get that big?

MBR offers no advantages over GPT except that it's the only format from which old PC hardware can boot Windows. Unless you're forced by circumstances to use MBR partitions, you typically don't want them. Unfortunately, MBR is still a common default setup for many distributions' installers.

The MBR label occupies a single 512-byte disk block, most of which is consumed by boot code. Only enough space remains to define four partitions. These are termed "primary" partitions because they are defined directly in the MBR.

You can, in theory, define one of the primary partitions to be an "extended" partition, which means that it contains its own subsidiary partition table. Unfortunately, extended partitions have been known to cause a variety of subtle problems. It's best to avoid them in these twilight years of MBR.

The Windows partitioning system lets one partition be marked "active." Boot loaders look for the active partition and try to load the operating system from it.

Each partition also has a one-byte type attribute that is supposed to signal the partition's contents. Generally, the codes represent either filesystem types or operating systems. These codes are not centrally assigned, but some common conventions have evolved. They are summarized by Andries E. Brouwer at goo.gl/ATi3.

The MS-DOS command that partitioned hard disks was called **fdisk**. Most operating systems that support MBR-style partitions have adopted this name for their own partitioning commands, but there are many variations among **fdisk**s. Windows itself has moved on: the command-line tool in recent versions is called **diskpart**. Windows also has a partitioning GUI that's available through the Disk Management plug-in of **mmc**.

It does not matter whether you partition a disk with Windows or some other operating system. The end result is the same.

GPT: GUID partition tables

Intel's extensible firmware interface (EFI) project replaced the rickety conventions of PC BIOSs with a more modern and functional architecture. EFI firmware is now standard for new PC hardware, and EFI's partitioning scheme has gained universal support among operating systems.

EFI has more recently become UEFI, a "unified" EFI effort supported by multiple vendors. However, EFI remains the more common term in general use. UEFI and EFI are essentially interchangeable.

The EFI partitioning scheme, known as a "GUID partition table" or GPT, removes the obvious weaknesses of MBR. It defines only one kind of partition (no more "logical partitions in the extended partition"), and you can create arbitrarily many of them. Each partition has a type specified by a 16-byte ID code (a globally unique ID, or GUID) that requires no central arbitration.

Significantly, GPT retains primitive compatibility with MBR-based systems by dragging along an MBR as the first block of the partition table. This "fakie" MBR makes the disk look like it's occupied by one large MBR partition (at least, up to the 2TB limit of MBR). It isn't useful per se, but the hope is that the decoy MBR will at least prevent naïve systems from attempting to reformat the disk.

Versions of Windows from the Vista era forward support GPT disks for data, but only systems with EFI firmware can boot Windows from them. Linux and its GRUB boot loader have fared better: GPT disks are supported by the OS and bootable on any system. Intel-based macOS systems use both EFI and GPT partitioning.

Although GPT has already been well accepted by operating system kernels, many disk management utilities are unmaintained and lack support for it. Make sure that any utility you run on a GPT disk actually supports GPT.

Linux partitioning

Linux systems give you several options for partitioning, which makes for treacherous terrain, given that some of the offerings are not GPT-aware. Default to **parted**, a command-line tool that understands several label formats (including Solaris's native one) and can move and resize partitions in addition to simply creating and deleting them. A GUI version, **gparted**, runs under GNOME.



In general, we recommend **gparted** over **parted**. Both are simple, but with **gparted** you can specify the size of the partitions you want instead of specifying the starting and ending block ranges. For partitioning the boot disk, most distributions' graphical installers are the best option since they typically suggest a partitioning plan that works well with that particular distribution's layout.

FreeBSD partitioning



Like Linux, FreeBSD has several partitioning tools. Ignore all except **gpart**. The others exist only to lure you into making some kind of terrible mistake.

The mysterious “geoms” you’ll see referred to in the **gpart** man page (and in other storage-related contexts on FreeBSD) are FreeBSD’s abstraction of storage devices. Not all geoms are disk drives, but all disk drives are geoms, so you can use a generic disk name such as ada0 wherever a geom is called for.

The FreeBSD “adding a disk” recipe on this page uses **gpart** to configure the partition table on a new disk.

20.7 LOGICAL VOLUME MANAGEMENT

Imagine a world in which you don't know exactly how large a partition needs to be. Six months after creating the partition, you discover that it is much too large, but that a neighboring partition doesn't have enough space. Sound familiar? A logical volume manager lets you reallocate space dynamically from the greedy partition to the needy partition.

Logical volume management is essentially a supercharged and abstracted version of disk partitioning. It groups individual storage devices into "volume groups." The blocks in a volume group can then be allocated to "logical volumes," which are represented by block device files and act like disk partitions.

However, logical volumes are more flexible and powerful than disk partitions. Here are some of the magical operations a volume manager lets you carry out:

- Move logical volumes among different physical devices
- Grow and shrink logical volumes on the fly
- Take copy-on-write "snapshots" of logical volumes
- Replace on-line drives without interrupting service
- Incorporate mirroring or striping in your logical volumes

The components of a logical volume can be put together in various ways. Concatenation keeps each device's physical blocks together and lines the devices up one after another. Striping interleaves the components so that adjacent virtual blocks are actually spread over multiple physical disks. By reducing single-disk bottlenecks, striping can often result in higher bandwidth and lower latency.

If you've had some prior exposure to RAID (see the section starting on this page), you might find striping reminiscent of RAID 0. LVM implementations of striping tend to be more flexible than RAID, though. For example, they may automatically optimize striping or allow devices of different sizes to be striped, even if striping won't actually happen 100% of the time. The line between LVM and RAID has become blurry indeed, and even parity schemes like RAID 5 and RAID 6 are making regular appearances in volume managers.

Linux logical volume management



Linux's volume manager, called LVM2, is essentially a clone of HP-UX's volume manager, which is itself based on software by Veritas. The commands for the two systems are essentially identical. Table 20.3 summarizes the LVM command set.

Table 20.3: LVM commands in Linux

Entity	Operation	Command
Physical volume	Create	**pvcreate**
	Inspect	**pvdisplay**
	Modify	**pvchange**
	Check	**pvck**
Volume group	Create	**vgcreate**
	Modify	**vgchange**
	Extend	**vgextend**
	Inspect	**vgdisplay**
	Check	**vgck**
	Enable	**vgscan**
Logical volume	Create	**lvcreate**
	Modify	**lvchange**
	Resize	**lvresize**
	Inspect	**lvdisplay**

The top-level architecture of LVM is that individual disks and partitions (physical volumes) are gathered into storage pools called volume groups. Volume groups are then subdivided into logical volumes, which are the block devices that hold filesystems.

A physical volume needs to have an LVM label applied with **pvcreate**. Applying such a label is the first step to accessing the device through the LVM. In addition to bookkeeping information, the label includes a unique ID to identify the device.

"Physical volume" is a somewhat misleading term because physical volumes need not have a direct correspondence to physical devices. They *can* be disks, but they can also be disk partitions or RAID arrays. LVM doesn't care.

You can control LVM with either a large group of simple commands (the ones listed in Table 20.3) or with the single **lvm** command and its various subcommands. These options are essentially identical; in fact, the individual commands are just links to **lvm**, which looks to see how it's been called to know how to behave. **man lvm** is a good introduction to the system and its tools.

A Linux LVM configuration proceeds in a few distinct phases:

- Creating (defining, really) and initializing physical volumes
- Adding the physical volumes to a volume group
- Creating logical volumes on the volume group

LVM commands start with letters that make it clear at which level of abstraction they operate: **pv** commands manipulate physical volumes, **vg** commands manipulate volume groups, and **lv** commands manipulate logical volumes. A few commands with the prefix **lvm** (e.g., **lvmchange**) operate on the system as a whole.

In the following example, we set up a 1TB hard disk (**/dev/sdb**) for use with LVM and create a logical volume. We assume that the disk has been partitioned as described on this page, with all space being assigned to a single partition, **/dev/sdb1**. We could omit the partitioning step entirely and just use the raw disk as our physical device, but there is no performance benefit to doing so. Partitioning makes the disk comprehensible to the broadest variety of software and operating systems.

The first step is to label the **sdb1** partition as an LVM physical volume:

```
$ sudo pvcreate /dev/sdb1
Physical volume "/dev/sdb1" successfully created
```

Our physical device is now ready to be added to a volume group:

```
$ sudo vgcreate DEMO /dev/sdb1
Volume group "DEMO" successfully created
```

Although we're using only a single physical device in this example, we could of course add additional devices. To step back and examine our handiwork, we use the **vgdisplay** command:

```
$ sudo vgdisplay DEMO
--- Volume group ---
VG Name               DEMO
System ID
Format                lvm2
Metadata Areas        1
Metadata Sequence No  1
VG Access             read/write
VG Status             resizable
Open LV               0
Max PV                0
Cur PV                1
Act PV                1
VG Size               1000.00 GiB
PE Size               4.00 MiB
Total PE              255999
Alloc PE / Size       0 / 0
Free  PE / Size       255999 / 1000.00 GiB
VG UUID               n26rxj-X5HN-x4nv-rdnM-7AWe-OQ21-EdDwEO
```

A PE is a physical extent, the allocation unit according to which the volume group is subdivided.

The final steps are to create the logical volume within DEMO and then to create a filesystem within that volume. We make the logical volume 100GB in size:

```
$ sudo lvcreate -L 100G -n web1 DEMO
Logical volume "web1" created
```

Most of LVM's interesting options live at the logical volume level. That's where striping, mirroring, and contiguous allocation would be requested if we were using those features.

We can now access the volume through the device **/dev/DEMO/web1**. We discuss filesystems in general starting on this page, but here is a quick overview of creating an ext4 filesystem so that we can demonstrate a few additional LVM tricks.

```
$ sudo mkfs /dev/DEMO/web1
...
$ sudo mkdir /mnt/web1
$ sudo mount /dev/DEMO/web1 /mnt/web1
```

Volume snapshots

You can create copy-on-write duplicates of any LVM logical volume, whether or not it contains a filesystem. This feature is handy for creating a quiescent image of a filesystem to be backed up elsewhere, but unlike ZFS and Btrfs snapshots, LVM2 snapshots are unfortunately not very useful as a general method of version control.

The problem is that logical volumes are of fixed size. When you create one, storage space is allocated for it up front from the volume group. A copy-on-write duplicate initially consumes no space, but as blocks are modified, the volume manager must find space in which to store both the old and new versions. This space for modified blocks must be set aside when you create the snapshot, and like any LVM volume, the allocated storage is of fixed size.

Note that it does not matter whether you modify the original volume or the snapshot (which by default is writable). Either way, the cost of duplicating the blocks is charged to the snapshot. Snapshots' allocations can be pared away by activity on the source volume even when the snapshots themselves are idle.

If you do not allocate as much space for a snapshot as is consumed by the volume of which it is an image, you can potentially run out of space in the snapshot. That's more catastrophic than it sounds because the volume manager then has no way to maintain a coherent image of the snapshot; additional storage space is required *just to keep the snapshot the same*. The result of running out of space is that LVM stops maintaining the snapshot, and the snapshot becomes corrupted.

So, as a matter of practice, LVM snapshots should be either short-lived or as large as their source volumes. So much for "lots of cheap virtual copies."

To create **/dev/DEMO/web1-snap** as a snapshot of **/dev/DEMO/web1**, we would use the following command:

```
$ sudo lvcreate -L 100G -s -n web1-snap DEMO/web1
Logical volume "web1-snap" created.
```

Note that the snapshot has its own name and that the source of the snapshot must be specified as *volume_group/volume*.

In theory, **/mnt/web1** should really be unmounted first to ensure the consistency of the filesystem. In practice, ext4 protects us against filesystem corruption, although we might lose a few of the most recent data block updates. This is a perfectly reasonable compromise for a snapshot used as a backup source.

To check on the status of your snapshots, run **lvdisplay**. If **lvdisplay** tells you that a snapshot is "inactive," that means it has run out of space and should be deleted. There's little you can do with a snapshot once it reaches this point.

Filesystem resizing

Filesystem overflows are more common than disk crashes, and one advantage of logical volumes is that they're much easier to juggle and resize than are hard partitions. We have experienced everything from servers used for personal video storage to departments full of email pack rats.

The logical volume manager doesn't know anything about the contents of its volumes, so you must do your resizing at both the volume and filesystem levels. The order depends on the specific

operation. Reductions must be filesystem-first, and enlargements must be volume-first. Don't memorize these rules: just think about what's actually happening and use common sense.

Suppose that in our example, **/mnt/web1** has grown more than we predicted and needs another 100GB of space. We first check the volume group to be sure additional space is available.

```
$ sudo vgdisplay DEMO
--- Volume group ---
VG Name               DEMO
System ID
Format                lvm2
Metadata Areas        1
Metadata Sequence No  4
VG Access             read/write
VG Status             resizable
Open LV               1
Max PV                0
Cur PV                1
Act PV                1
VG Size               1000.00 GiB
PE Size               4.00 MiB
Total PE              255999
Alloc PE / Size       51200 / 200.00 GiB
Free  PE / Size       204799 / 800.00 GiB
VG UUID               n26rxj-X5HN-x4nv-rdnM-7AWe-OQ21-EdDwEO
```

Note that 200GB of space has been consumed, 100GB for the original filesystem and 100GB for the snapshot. However, plenty of space is still available. We unmount the filesystem and use **lvresize** to add space to the logical volume.

```
$ sudo umount /mnt/web1
$ sudo lvchange -an DEMO/web1
$ sudo lvresize -L +100G DEMO/web1
Size of logical volume DEMO/web1 changed from 100.00 GiB
    (25600 extents) to 200.00 GiB (51200 extents).
Logical volume DEMO/web1 successfully resized.
$ sudo lvchange -ay DEMO/web1
```

The **lvchange** commands are needed to deactivate the volume for resizing and to reactivate it afterwards. This part is needed only because an existing snapshot of web1 remains from our previous example. After the resize operation, the snapshot will "see" the additional 100GB of allocated space, but since the filesystem it contains is only 100GB in size, the snapshot will still be usable.

We can now resize the filesystem with **resize2fs**. (The **2** comes from the original ext2 filesystem, but the command supports all versions of ext.) Since **resize2fs** can determine the size of the new filesystem from the volume, we don't need to specify the new size explicitly. We would have to do so when shrinking the filesystem:

```
$ sudo resize2fs /dev/DEMO/web1
resize2fs 1.43.3 (04-Sep-2016)
Resizing the filesystem on /dev/DEMO/web1 to 52428800 (4k) blocks.
The filesystem on /dev/DEMO/web1 is now 52428800 (4k) blocks long.
```

That's it! Examining the output of **df** again shows the changes:

```
$ sudo mount /dev/DEMO/web1 /mnt/web1
$ df -h /mnt/web1
Filesystem             Size  Used Avail  Use%  Mounted on
/dev/mapper/DEMO-web1  197G   60M  187G    1%  /mnt/web1
```

Commands for resizing other filesystems work similarly. For XFS filesystems (the default on Red Hat and CentOS systems), use **xfs_growfs**; for UFS filesystems (the default on FreeBSD), use **growfs**. XFS filesystems must be mounted to be expanded. As the names of these commands suggest, XFS and UFS filesystems can be expanded but not made smaller. If you need to remove space, you'll need to copy the filesystem's contents to a new, smaller filesystem.

It's worth noting that "disks" you allocate and attach to virtual machines in the cloud are essentially logical volumes, although the volume manager itself lives elsewhere in the cloud. These volumes are usually resizable through the cloud provider's management console or command-line utility.

The procedure for resizing cloud filesystems is much the same as the one outlined above, but keep in mind that because these virtual devices impersonate disk drives, they probably have partition tables. You'll need to resize on three separate layers: at the cloud provider level, at the partition level, and at the filesystem level.

FreeBSD logical volume management



FreeBSD has a full-fledged logical volume manager of its own. Previous versions were known by the name Vinum, but now that the system has been rewritten to conform to FreeBSD's generalized geom architecture for storage devices, the name has been changed to GVinum. Like LVM2, GVinum implements a variety of RAID types.

FreeBSD has recently put a lot of effort into ZFS support, and although GVinum has not been officially deprecated, developers' public comments suggest that ZFS is the recommended approach for logical volume management and RAID going forward. Accordingly, we do not discuss GVinum here. ZFS is covered starting on this page.

20.8 RAID: REDUNDANT ARRAYS OF INEXPENSIVE DISKS

Even with backups, the consequences of a disk failure on a server can be disastrous. RAID, “redundant arrays of inexpensive disks,” is a system that distributes or replicates data across multiple disks. RAID is sometimes glossed as “redundant arrays of independent disks,” too. Both versions are historically accurate.

RAID not only helps avoid data loss but also minimizes the downtime associated with hardware failures (often to zero) and potentially increases performance.

RAID can be implemented by dedicated hardware that presents a group of hard disks to the operating system as a single composite drive. It can also be implemented simply by the operating system’s reading or writing multiple disks according to the rules of RAID.

Software vs. hardware RAID

Because the disks themselves are always the most significant bottleneck in a RAID implementation, there is no reason to assume that a hardware-based implementation of RAID will necessarily be faster than a software- or OS-based implementation. Hardware RAID has been predominant in the past for two main reasons: lack of software alternatives (no direct OS support for RAID) and hardware's ability to buffer writes in some form of nonvolatile memory.

The latter feature does improve performance because it makes writes appear to complete instantaneously. It also protects against a potential corruption issue called the "RAID 5 write hole," which we describe in more detail starting on this page. But beware: many of the common "RAID cards" sold for PCs have no nonvolatile memory at all; they are just glorified SATA interfaces with some RAID software on-board. RAID implementations on PC motherboards fall into this category as well. You're better off using the RAID features in Linux or FreeBSD on these systems. (Or better yet, use ZFS or Btrfs.)

We have experienced a disk controller failure on an important production server. Although the data was replicated across several physical drives, a faulty hardware RAID controller destroyed the data on all disks. A lengthy and ugly restore process ensued. The rebuilt server now relies on the kernel's software to manage its RAID environment, removing the possibility of another RAID controller failure.

RAID levels

RAID can do two basic things. First, it can improve performance by "striping" data across multiple drives, thus allowing several drives to work simultaneously to supply or absorb a single data stream. Second, it can replicate data across multiple drives, decreasing the risk associated with a single failed disk.

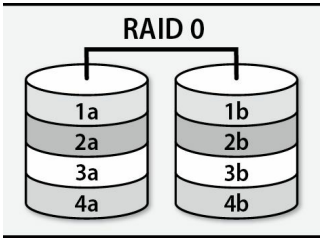
Replication assumes two basic forms: mirroring, in which data blocks are reproduced bit-for-bit on several different drives, and parity schemes, in which one or more drives contain an error-correcting checksum of the blocks on the remaining data drives. Mirroring is faster but consumes more disk space. Parity schemes are more disk-space-efficient but have lower performance.

RAID is traditionally described in terms of "levels" that specify the exact details of the parallelism and redundancy implemented by an array. The term is perhaps misleading because "higher" levels are not necessarily "better." The levels are simply different configurations; use whichever versions suit your needs.

In the following illustrations, numbers identify stripes and the letters a, b, and c identify data blocks within a stripe. Blocks marked p and q are parity blocks.

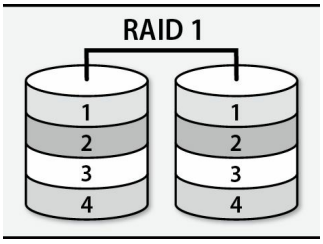
"Linear mode," also known as JBOD (for "just a bunch of disks") is not even a real RAID level. And yet, every RAID controller seems to implement it. JBOD concatenates the block addresses of multiple drives to create a single, larger virtual drive. It has no data redundancy or performance benefit. These days, JBOD functionality is best achieved through a logical volume manager rather than a RAID system.

RAID level 0 increases performance. It combines two or more drives of equal size, but instead of stacking them end-to-end, it stripes data alternately among the disks in the pool. Sequential reads and writes are therefore spread among several disks, decreasing write and access times.

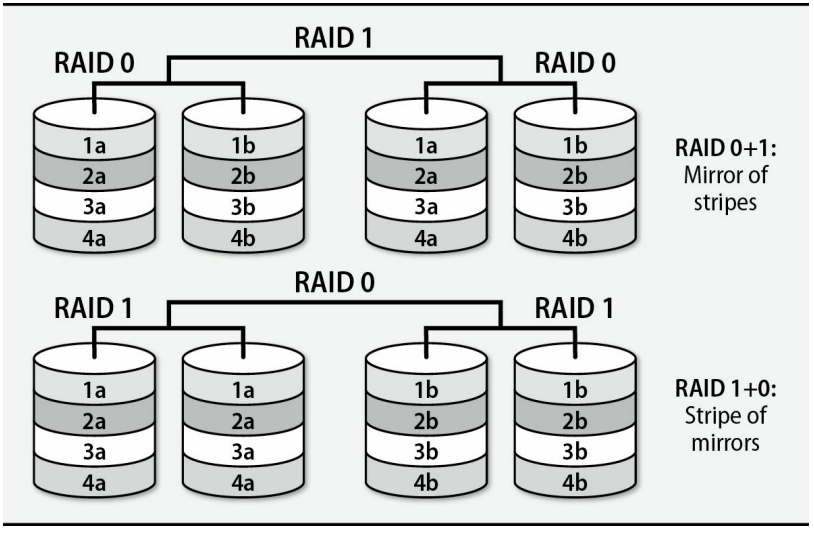


Note that RAID 0 has reliability characteristics that are significantly *inferior* to separate disks. A two-drive array has roughly double the annual failure rate of a single drive, and so on.

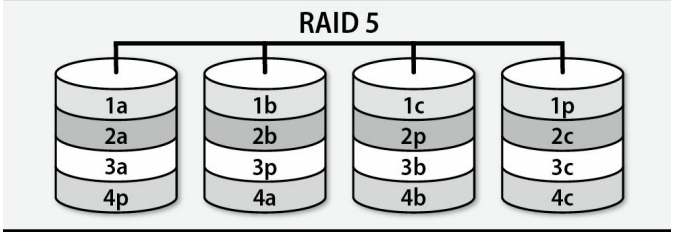
RAID level 1 is colloquially known as mirroring. Writes are duplicated to two or more drives simultaneously. This arrangement makes writes slightly slower than they would be on a single drive. However, it offers read speeds comparable to RAID 0 because reads can be farmed out among the several duplicate disk drives.



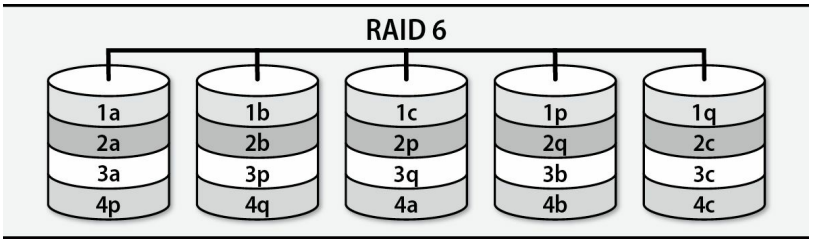
RAID levels 1+0 and 0+1 are stripes of mirrors or mirrors of stripes. Logically, they are concatenations of RAID 0 and RAID 1, but many controllers and software implementations support them directly. The goal of both modes is to simultaneously obtain the performance of RAID 0 and the redundancy of RAID 1. These configurations need at least four devices.



RAID level 5 stripes both data and parity information, adding redundancy while simultaneously improving read performance. In addition, RAID 5 is more efficient in its use of disk space than is RAID 1. If an array has N drives (at least three are required), N–1 of them can store data. The space-efficiency of RAID 5 is therefore at least 67%, whereas that of mirroring cannot be higher than 50%.



RAID level 6 is similar to RAID 5 with two parity disks. A RAID 6 array can withstand the complete failure of two drives without losing data. RAID 6 requires at least four devices.



RAID levels 2, 3, and 4 are defined but rarely deployed. Logical volume managers usually include both striping (RAID 0) and mirroring (RAID 1) features.

As RAID systems, logical volume managers, and filesystem all rolled into one, ZFS and Btrfs support striping, mirroring, and configurations similar to RAID 5 and RAID 6. See this page for more details on these options.

Linux supports both ZFS and Btrfs, though you might have to install ZFS separately. Btrfs's RAID 5 and RAID 6 support is not officially ready for production use.

For simple striped and mirrored configurations outside the context of one of these filesystems, Linux gives you a choice between a dedicated RAID system (**md**; see this page) and the logical volume manager, LVM. The LVM approach is perhaps more flexible, but the **md** approach may be a bit more rigorously predictable. If you opt for **md**, you can still use LVM to manage the space on the RAID volume. For RAID 5 and RAID 6, you must use **md** to implement software RAID.

ZFS is the preferred RAID implementation for FreeBSD. However, two additional implementations are available.

At the disk driver level, FreeBSD's geom system can combine disks into RAID arrays with support for RAID 0, RAID 1, and RAID 3. (RAID 3 is similar to RAID 5 but uses a dedicated parity disk instead of distributing parity among all disks in a pool.) You can stack geoms, so RAID 1+0 and RAID 0+1 are possible as well.

FreeBSD also includes support for RAID 0, RAID 1, and RAID 5 in its logical volume manager, GVinum. However, with the advent of full support for ZFS on FreeBSD, the future of GVinum appears to be in question. It is not yet officially deprecated but no longer seems to be actively maintained.

Disk failure recovery

The Google disk failure study cited on this page should be pretty convincing evidence of the need for some form of storage redundancy in most production environments. At an 8% annual failure rate, your organization needs only 150 hard disks in service to expect an average of one disk failure per month.

JBOD and RAID 0 modes are of no help when hardware problems occur; you must recover your data manually from backups. Other forms of RAID enter a degraded mode in which the offending devices are marked as faulty. The RAID arrays continue to function normally from the perspective of storage clients, although perhaps at reduced performance.

Bad disks must be swapped out for new ones as soon as possible to restore redundancy to the array. A RAID 5 array or two-disk RAID 1 array can tolerate the failure of only a single device. Once that failure has occurred, the array is vulnerable to a second failure.

The specifics of the process are usually pretty simple. You replace the failed disk with another of similar or greater size, then tell the RAID implementation to replace the old disk with the new one. What follows is an extended period during which the parity or mirror information is rewritten to the new, blank disk. This is typically an overnight operation. The array remains available to clients during this phase, but performance is likely to be poor.

To limit downtime and the vulnerability of the array to a second failure, most RAID implementations let you designate one or more disks as “hot” spares. When a failure occurs, the faulted disk is automatically swapped for a spare, and the process of resynchronizing the array begins immediately. Where supported, hot spares should be used as a matter of course.

Drawbacks of RAID 5

RAID 5 is a popular configuration, but it has some weaknesses, too. The following issues apply to RAID 6 also, but for simplicity we frame the discussion in terms of RAID 5.

First, it's critically important to note that RAID 5 does not replace regular off-line backups. It protects the system against the failure of one disk—that's it. It does not protect against the accidental deletion of files. It does not protect against controller failures, fires, hackers, or any number of other hazards.

Second, RAID 5 isn't known for its great write performance. RAID 5 writes data blocks to N–1 disks and parity blocks to the N^{th} disk. Parity data is distributed among all the drives in the array; each stripe has its parity stored on a different drive. Since there's no dedicated parity disk, it's unlikely that any single disk will act as a bottleneck.

Whenever a random block is written, at least one data block and the parity block for that stripe must be updated. Furthermore, the RAID system doesn't know what the new parity block ought to contain until it has read the old parity block and the old data. Each random write therefore expands into four operations: two reads and two writes. (Sequential writes may fare better if the implementation is smart.)

Finally, RAID 5 is vulnerable to corruption in certain circumstances. Its incremental updating of parity data is more efficient than reading the entire stripe and recalculating the stripe's parity from the original data. On the other hand, it means that at no point is parity data ever validated or recalculated. If any block in a stripe should fall out of sync with the parity block, that fact will never become evident in normal use; reads of the data blocks will still return the correct data.

Only when a disk fails does the problem become apparent. The parity block will likely have been rewritten many times since the occurrence of the original desynchronization. Therefore, the reconstructed data block on the replacement disk will consist of essentially random data.

This kind of desynchronization between data and parity blocks isn't all that unlikely, either. Disk drives are not transactional devices. Without an additional layer of safeguards, there is no simple way to guarantee that either two blocks or zero blocks on two different disks will be properly updated. It's quite possible for a crash, power failure, or communication problem at the wrong moment to create data/parity skew.

This problem is known as the RAID 5 "write hole," and it has received increasing attention over the last ten years or so. The implementors of the ZFS filesystem claim that because ZFS uses variable-width stripes, it is immune to the RAID 5 write hole. That's also why ZFS calls its RAID implementation RAID-Z instead of RAID 5, though in practice the concept is similar.

Another potential solution is "scrubbing," validating parity blocks one by one while the array is relatively idle. Most RAID implementations include some form of scrubbing function. You just have to remember to activate it regularly (by initiating it from **cron** or a **systemd** timer).

mdadm: Linux software RAID



The standard software RAID implementation for Linux is called **md**, the "multiple disks" driver. It's front-ended by the **mdadm** command. **md** supports all the RAID configurations listed above as well as RAID 4. An earlier system known as **raidtools** is no longer used.

You can also implement RAID on Linux through the logical volume manager (LVM2) or through Btrfs or another filesystem with built-in volume management and RAID features. We address LVM2 starting on this page and next-generation filesystems starting on this page. Generally, these multiple implementations represent different epochs of software development, with **mdadm** being the earliest and ZFS/Btrfs the most recent.

All these systems are actively maintained, so choose whichever you prefer. Sites without an installed base are best off jumping directly to an all-in-one system like Btrfs.

Creating an array

The following scenario configures a RAID 5 array composed of three identical 1TB hard disks. Although **md** can use raw disks as components, we prefer to give every disk a partition table for consistency, so we start by running **gparted**, creating a GPT partition table on each disk and assigning all the disk's space to a single partition of type "Linux RAID." It's not strictly necessary to set the partition type, but it's a useful reminder to anyone who might inspect the table later.

The following command builds a RAID 5 array from three whole-disk partitions:

```
$ sudo mdadm --create /dev/md/extra --level=5 --raid-devices=3
    /dev/sdf1 /dev/sdg1 /dev/sdh1
mdadm: Defaulting to version 1.2 metadata
mdadm: array /dev/md/extra started.
```

The virtual file **/proc/mdstat** always contains a summary of **md**'s status and the status of all the system's RAID arrays. It is especially useful to keep an eye on the **/proc/mdstat** file after adding a new disk or replacing a faulty drive. (**watch cat /proc/mdstat** is a handy idiom.)

```
$ cat /proc/mdstat
Personalities : [linear] [multipath] [raid0] [raid1] [raid6] [raid5]
    [raid4] [raid10]
md127 : active raid5 sdh1[3] sdg1[1] sdf1[0]
    2096886784 blocks super 1.2 level 5, 512k chunk, algo 2 [3/2] [UU_]
    [>....................]  recovery =  0.0% (945840/1048443392)
        finish=535.2min speed=32615K/sec
    bitmap: 0/8 pages [0KB], 65536KB chunk

unused devices: <none>
```

The **md** system does not keep track of which blocks in an array have been used, so it must manually synchronize all the parity blocks with their corresponding data blocks. **md** calls the operation a “recovery” since it’s essentially the same procedure used when you swap out a bad hard disk. It can take many hours on a large array.

Some helpful notifications appear in the system logs, too (usually **/var/log/messages** or **/var/log/syslog**):

```
kernel: md: bind<sdf1>
kernel: md: bind<sdg1>
kernel: md: bind<sdh1>
kernel: md/raid:md127: device sdg1 operational as raid disk 1
kernel: md/raid:md127: device sdf1 operational as raid disk 0
kernel: md/raid:md127: allocated 3316kB
kernel: md/raid:md127: raid level 5 active with 2 out of 3 devices,
    algorithm 2
kernel: RAID conf printout:
kernel:  --- level:5 rd:3 wd:2
kernel:  disk 0, o:1, dev:sdf1
kernel:  disk 1, o:1, dev:sdg1
kernel: created bitmap (8 pages) for device md127
mdadm[1174]: NewArray event detected on md device /dev/md127
mdadm[1174]: DegradedArray event detected on md device /dev/md127
kernel: md127: bitmap initialized from disk: read 1 pages, set 15998
    of 15998 bits
kernel: md127: detected capacity change from 0 to 2147212066816
kernel: RAID conf printout:
kernel:  --- level:5 rd:3 wd:2
kernel:  disk 0, o:1, dev:sdf1
kernel:  disk 1, o:1, dev:sdg1
kernel:  disk 2, o:1, dev:sdh1
kernel: md: recovery of RAID array md127
kernel: md: minimum _guaranteed_  speed: 1000 KB/sec/disk.
kernel: md: using maximum available idle IO bandwidth (but not more
    than 200000 KB/sec) for recovery.
kernel: md: using 128k window, over a total of 1048443392k.
mdadm[1174]: RebuildStarted event detected on md device /dev/md127
```

The initial creation command also serves to “activate” the array (make it available for use). On subsequent reboots, most distributions (including all our examples) automatically discover and activate any existing arrays.

Note that you specify a device pathname for the composite array when you run **mdadm --create**. Old-style **md** device paths looked like **/dev/md0**, but when you specify a path under the **/dev/md** directory, as was done in this example, **mdadm** writes your chosen name into the array’s superblock. This measure ensures that you can always locate the array by its logical path, even when the array is autostarted and might be assigned a different array number. As you can see

from the log entries above, the array also has a traditional name (here, **/dev/md127**). **/dev/md/extra** is just a symbolic link to the actual array device.

***mdadm.conf**: document array configuration*

mdadm does not technically require a configuration file, but it will use a configuration file if you supply one, typically **/etc/mdadm/mdadm.conf** or **/etc/mdadm.conf**. We recommend that you add ARRAY entries to the configuration file as you create new arrays. Doing so documents the RAID configuration in a standard place and gives administrators an obvious place to look for information when problems occur.

mdadm --detail --scan dumps the current RAID setup in the format required for inclusion in **mdadm.conf**. For example,

```
$ sudo mdadm --detail --scan
ARRAY /dev/md/extra metadata=1.2 name=ubuntu:extra UUID=b72de2fb:60b3
    03af:3c176048:dc5b6c8b
```

With the addition of this line, **mdadm** can now read **mdadm.conf** at startup or shutdown to easily manage the array. For example, to take down the array created above, we could run

```
$ sudo mdadm -S /dev/md/extra
```

And to start it up again, run

```
$ sudo mdadm -As /dev/md/extra
```

The first of these commands would work even without the **mdadm.conf** file, but the second would not.

We formerly recommended that you add DEVICE entries for the components of each array to **mdadm.conf**, too. We take that back. Device names are more ephemeral these days and **mdadm** is better at finding and identifying array components than it used to be. We don't think DEVICE entries are a best practice anymore.

mdadm has a **--monitor** mode in which it runs continuously as a daemon process and raises an alarm when problems are detected on a RAID array. Use this feature! To set it up, add a MAILADDR or PROGRAM line to your **mdadm.conf** file. A MAILADDR notifies you of issues by email, and a PROGRAM configuration runs an external reporting tool that you supply (as is useful for integrating with monitoring systems; see Chapter 28).

You also need to arrange for the monitor daemon to run at boot time. All our example distributions have an **init** script that does this for you, but the names and procedures for enabling it are slightly different.

```
debian$ sudo update-rc.d mdadm enable
ubuntu$ sudo update-rc.d mdadm enable

redhat$ sudo systemctl enable mdmonitor
centos$ sudo systemctl enable mdmonitor
```

Simulating a failure

What happens when a disk actually fails? Let's find out! **mdadm** offers the handy option to simulate a failed disk.

```
$ sudo mdadm /dev/md/extra -f /dev/sdg1
mdadm: set /dev/sdg1 faulty in /dev/md/extra

$ sudo tail -1 /var/log/messages
Apr 10 16:18:39 ubuntu kernel: md/raid:md127: Disk failure on sdg1,
    disabling device.#012md/raid:md127: Operation continuing on 2
    devices.

$ cat /proc/mdstat
Personalities : [linear] [multipath] [raid0] [raid1] [raid6] [raid5]
    [raid4] [raid10]
md127 : active raid5 sdh1[3] sdf1[0] sdg1[1](F)
    2096886784 blocks super 1.2 level 5, 512k chunk, algo 2 [3/2] [UU_]

unused devices: <none>
```

Because RAID 5 is a redundant configuration, the array continues to function in degraded mode, so users will not necessarily be aware of the problem.

To remove the drive from the RAID configuration, use **mdadm -r**:

```
$ sudo mdadm /dev/md/extra -r /dev/sdg1
mdadm: hot removed /dev/sdg1 from /dev/md/extra
```

Once the disk has been logically removed, you can shut down the system and replace the drive. Hot-swappable drive hardware lets you make the change without turning off the system or rebooting.

If your RAID components are raw disks, replace them only with an identical drive. You can replace partition-based components with any partition of similar size, which is a good reason to build your arrays on top of partitions rather than raw disks. Still, for bandwidth matching it's best if the underlying drive hardware is similar. (Of course, if your RAID configuration is built on top of partitions, you must run a partitioning utility to define the partitions appropriately before adding the replacement disk to the array.)

In our example, the failure is just simulated, so we can add the drive back to the array without replacing any hardware:

```
$ sudo mdadm /dev/md/extra -a /dev/sdg1
mdadm: hot added /dev/sdc1
```

md immediately starts to rebuild the array. As always, you can see its progress in **/proc/mdstat**. A rebuild can take hours, so consider this fact in your disaster recovery (and testing!) plans.

20.9 FILESYSTEMS

Even after a hard disk has been conceptually divided into partitions or logical volumes, it is still not ready to hold files. All the abstractions and goodies described in Chapter 5, *The Filesystem*, must be implemented in terms of raw disk blocks. The filesystem is the code that implements these, and it needs to add a bit of its own overhead and data.

Early systems bundled the filesystem implementation into the kernel, but it soon became apparent that support for multiple filesystem types was an important design goal. UNIX systems developed a well-defined kernel interface that allowed multiple types of filesystems to be active at once. The filesystem interface also abstracted the underlying hardware, so filesystems see approximately the same interface to storage devices as do other UNIX programs that access the disks through device files in **/dev**.

Support for multiple filesystem types was initially motivated by the need to support NFS and filesystems for removable media. But once the floodgates were opened, the “what if” era began; many different groups started to work on improved filesystems. Some were system-specific, and others (such as ReiserFS) were not tied to any particular OS.

Most systems have settled on one or two filesystems as mainstream defaults. These filesystems are rigorously tested along with the rest of the system before stable releases are issued.

The predominant pattern is for systems to officially support one traditional-style filesystem (UFS, ext4, or XFS) and one next-generation filesystem that includes volume management and RAID features (ZFS or Btrfs). Support for the latter options is usually best on physical hardware; cloud systems can use them for data partitions, but sometimes not for the boot disk.

Although other filesystem implementations are often just a package installation away, add-on filesystems do bring risk and potential instability. Filesystems are foundational, so they need to be 100% stable and reliable under all use scenarios. Filesystem developers work hard to achieve this level of robustness, but the risk can’t be entirely eliminated.

Unless you’re setting up a storage pool or data disk for a specific application, we recommend against straying from your systems’ supported filesystems. That’s what the documentation and administrative tools most likely assume.

The upcoming sections describe the most common filesystems and their management in a bit more detail. We first describe the traditional filesystems UFS, ext4, and XFS, then move on to the next-generation systems ZFS (this page) and Btrfs (this page).

20.10 Traditional filesystems: UFS, ext4, and XFS

UFS, ext4, and XFS have separate code bases and histories, but over time they've become eerily similar to one another from an administrative perspective.

These filesystems exemplify the old school approach in which volume management and RAID are implemented separately from the filesystem itself. The filesystems limit themselves to plain-vanilla file storage on block devices of defined size. Their features are more or less limited to those outlined in Chapter 5.

Older filesystems in this category were subject to subtle corruption if power was interrupted in the middle of a write operation, because then disk blocks could contain inconsistent data structures. The **fsck** command was used at boot time to check filesystems for this kind of problem and to automatically patch the most common issues.

Modern filesystems include a feature called journaling that averts the possibility of this type of corruption. When a filesystem operation occurs, the required modifications are first written to the journal. Once the journal update is complete, a "commit record" is written to mark the end of the entry. Only then is the normal filesystem modified. If a crash occurs during the update, the filesystem can later replay the journal log to reconstruct a perfectly consistent filesystem.

In most cases, only metadata changes are journaled. The actual data to be stored is written directly to the filesystem. Some filesystems can use the journal for data too, but at a significant performance cost.

Journaling reduces the time needed to perform filesystem consistency checks (see the **fsck** section on this page) to approximately one second per filesystem. Barring some type of hardware failure, the state of a filesystem can almost instantly be assessed and restored.

The Berkeley Fast File System implemented by McKusick et al. in the 1980s was an early standard that spread to many UNIX systems. With some small adjustments, it eventually became known as the UNIX File System (UFS) and formed the basis of several other filesystem implementations, including Linux's ext series. UFS remains the default filesystem used by FreeBSD.

The "second extended filesystem," ext2, was for a long time the mainstream Linux standard. It was designed and implemented primarily by Rémy Card, Theodore Ts'o, and Stephen Tweedie. Although the code for ext2 was written specifically for Linux, it is functionally similar to the Berkeley Fast File System.

Ext3 added journaling, and ext4 is a comparatively modest update that raises a few size limits, increases the performance of certain operations, and allows the use of "extents" (disk block ranges) for storage allocation rather than just individual disk blocks. Ext4 is the default filesystem on Debian and Ubuntu.

XFS was developed by Silicon Graphics, Inc., later known as SGI. It was the default filesystem for IRIX, SGI's version of UNIX, and was one of the first extent-based filesystems. That made it particularly suitable for sites that processed large media files, as many SGI customers did. XFS is the default filesystem on Red Hat and CentOS.

Filesystem terminology

Largely because of their common history, many filesystems share some descriptive terminology. The implementations of the underlying objects have often changed, but the terms are still widely used by administrators as labels for fundamental concepts.

“Inodes” are fixed-length table entries, each of which holds information about one file. The term is probably short for “index nodes,” although its exact etymology is unclear. Inodes were originally preallocated at the time a filesystem was created, but some filesystems now create them dynamically as they are needed. Either way, an inode usually has an identifying number, which you can see with **ls -i**.

Inodes are the “things” pointed to by directory entries. When you create a hard link to an existing file, you create a new directory entry, but you do not create a new inode.

A superblock is a record that describes the characteristics of the filesystem. It contains information about the length of a disk block, the size and location of the inode tables, the disk block map and usage information, the size of the block groups, and a few other important parameters of the filesystem. Because damage to the superblock could erase crucial information, several copies of it are maintained in scattered locations.

The kernel caches disk blocks to increase efficiency. All types of blocks can be cached, including superblocks, inode blocks, and directory information. Caches are normally not “write through,” so there might be some delay between the point at which an application thinks it has written a block and the point at which the block is actually saved to disk. Applications can request more predictable behavior for a file, but this option lowers throughput.

The **sync** system call flushes modified blocks to their permanent homes on disk, possibly making the on-disk filesystem fully consistent for a split second. This periodic save minimizes the amount of data loss that might occur if the machine were to crash with many unsaved blocks. Filesystems can do syncs on their own schedule or leave this up to the OS. Modern filesystems have journaling mechanisms that minimize or eliminate the possibility of structural corruption caused by a crash, so sync frequency now mostly has to do with how many data blocks might be lost in a crash.

A filesystem’s disk block map is a table of the free blocks it contains. When new files are written, this map is examined to devise an efficient layout scheme. The block usage summary records basic information about the blocks that are already in use.

Filesystem polymorphism

Filesystems are software packages with multiple components. One part lives in the kernel (or even potentially in user space under Linux; search for “FUSE”) and implements the nuts and bolts of translating the standard filesystem API into reads and writes of disk blocks. Other parts are user-level commands that initialize new volumes to the standard format, check filesystems for corruption, and perform other format-specific tasks.

Long ago, the standard user-level commands knew about “the filesystem” that the system used, and they simply implemented the appropriate functionality. **mkfs** or **newfs** created new filesystems, **fsck** fixed problems, and **mount** mostly just invoked the appropriate underlying system calls.

These days, many more filesystems exist, so systems have had to decide how to address this cornucopia of options. For a long time, Linux tried to fit all filesystems into the standard mold of **mkfs** and **fsck** by making those commands be wrappers. The wrappers called discrete commands named, e.g., **mkfs.***fsname* or **fsck.***fsname* depending on the type of filesystem being manipulated. These days, the pretense of homogeneity among filesystems has been stretched past the breaking point, and most systems now advise you to call the filesystem-specific commands directly.

Filesystem formatting



The general recipe for creating a new Linux filesystem is

```
mkfs.fstype [ -L label ] [ other_options ] device
```

On FreeBSD, the process for creating a UFS filesystem is similar, but with **newfs**:

```
newfs [ -L label ] [ other_options ] device
```

The **-L** option to both **mkfs** and **newfs** sets a volume label for the filesystem such as “spare,” “home,” or “extra.” This is just one option among many, but it’s an option that we recommend you use on every filesystem. Labeling the filesystem frees you from having to track the device on which it’s been installed. It’s particularly handy given that disk device names can change whenever hardware is adjusted.

The available *other_options* are filesystem-specific, but their use is uncommon.

fsck: check and repair filesystems

Because of block buffering and the fact that disk drives are not really transactional devices, filesystem data structures can potentially become self-inconsistent. If these problems are not corrected quickly, they propagate and snowball.

The original fix for corruption was a command called **fsck** ("filesystem consistency check," spelled aloud or pronounced "FS check" or "fisk") that carefully inspected all data structures and walked the allocation tree for every file. It relied on a set of heuristic rules about what the filesystem state might look like after failures at various points during an update.

The original **fsck** scheme worked surprisingly well, but because it involved reading all the data on a disk, it could take hours on a large drive. An early optimization was a "filesystem clean" bit that could be set in the superblock when the filesystem was properly unmounted. When the system restarted, it would see the clean bit and know to skip the **fsck** check.

Now, filesystem journals let **fsck** pinpoint the activity that was occurring at the time of a failure. **fsck** can simply rewind the filesystem to the last known consistent state.

Disks are normally **fsck**ed automatically at boot time if they are listed in the system's **/etc/fstab** file. The **fstab** file has legacy "**fsck** sequence" fields that ordered and parallelized filesystem checks. But now that **fsck**s are fast, the only thing that matters is that the root filesystem be checked first.

You can run **fsck** by hand to perform an in-depth examination more akin to the original **fsck** procedure, but be aware of the time required.



Linux ext-family filesystems can be set to force a recheck after they have been remounted a certain number of times or after a certain period of time, even if all the unmounts were "clean." This precaution is probably good hygiene, and in most cases the default value (usually around 20 mounts) is acceptable. However, on systems that mount filesystems frequently, such as desktop workstations, even that frequency of **fsck**s can become tiresome. To increase the interval to 50 mounts, use the **tune2fs** command:

```
$ sudo tune2fs -c 50 /dev/sda3
tune2fs 1.43.3 (04-Sep-2016)
Setting maximal mount count to 50
```

If a filesystem appears damaged and **fsck** cannot automatically repair it, *do not* experiment with it before making an ironclad backup. The best insurance policy is to **dd** the entire disk to a backup file or backup disk.

Most filesystems create a **lost+found** directory at the root of each filesystem in which **fsck** can deposit files whose parent directory cannot be determined. The **lost+found** directory has some extra space preallocated so that **fsck** can store orphaned files there without having to allocate

additional directory entries on an unstable filesystem. Don’t delete this directory. (Some systems have a **mklost+found** command you can use to re-create this directory if it is deleted.)

Since the name given to a file is recorded only in the file’s parent directory, names for orphan files are not available and so the files placed in **lost+found** are named with their inode numbers. The inode table does record the UID of the file’s owner, however, so getting a file back to its original owner is relatively easy.

Filesystem mounting

A filesystem must be mounted before it becomes visible to processes. The mount point for a filesystem can be any directory, but the files and subdirectories beneath it are not accessible while a filesystem is mounted there. See *Filesystem mounting and unmounting* for more information.

After installing a new disk, mount new filesystems by hand to be sure that everything is working correctly. For example, the command

```
$ sudo mount /dev/sda1 /mnt/temp
```

mounts the filesystem in the partition represented by the device file **/dev/sda1** (device names will vary among systems) on a subdirectory of **/mnt**, which is a traditional path used to contain temporary mounts.

You can verify the size of a filesystem with the **df** command. The example below uses the Linux **-h** flag to request "human readable" output. Unfortunately, most systems' **df** defaults to an unhelpful unit such as "disk blocks," but there is usually a flag to make **df** report something specific such as kibibytes or gibibytes.

```
$ df -h /mnt/web1
Filesystem             Size  Used  Avail  Use%  Mounted on
/dev/mapper/DEMO-web1  197G   60M   187G    1%  /mnt/web1
```

Setup for automatic mounting

You will generally want to configure the system to mount local filesystems at boot time. The **/etc/fstab** file lists the device names and mount points of all the system's disks (among other things).

mount, **umount**, **swapon**, and **fsck** all read the **fstab** file, so it's helpful if the data presented there is correct and complete. **mount** and **umount** use the catalog to figure out what you want done if you specify only a partition name or mount point on the command line. For example, with the Linux **fstab** configuration shown on this page, the command

```
$ sudo mount /media/cdrom0
```

would have the same effect as typing

```
$ sudo mount -t udf -o user,noauto,exec,utf8 /dev/scd0 /media/cdrom0
```

The command **mount -a** mounts all regular filesystems listed in the filesystem catalog; it is usually executed from the startup scripts at boot time. (The noauto mount option excludes a given filesystem from automatic mounting by **mount -a**.)

The **-t** *fstype* argument constrains the operation to filesystems of a certain type. For example,

```
$ sudo mount -at ext4
```

mounts all local ext4 filesystems. The **mount** command reads **fstab** sequentially. Therefore, filesystems that are mounted beneath other filesystems must follow their parent partitions in the **fstab** file. For example, the line for **/var/log** must follow the line for **/var** if **/var** is a separate filesystem.

The **umount** command for unmounting filesystems accepts a similar syntax. You cannot unmount a filesystem that a process is using as its current directory or on which files are open. Several commands can identify the processes that are interfering with your **umount** attempt; see this page.



The FreeBSD **fstab** file is the most traditional of our example systems. Here's a sample from a system with only one real filesystem beyond the root (**/spare**):

```
# Device          Mountpoint  FStype    Options   Dump  Pass#
/dev/gpt/rootfs   /           ufs       rw        1     1
/dev/gpt/swap-a   none        swap      sw        0     0
/dev/gpt/swap-b   none        swap      sw        0     0
fdesc             /dev/fd     fdescfs   rw        0     0
proc              /proc       procfs    rw        0     0
/dev/gpt/spare    /spare      ufs       rw        0     0
```

Each line holds six fields separated by whitespace. Each line describes a single filesystem. The fields are traditionally aligned for readability, but alignment is not required.

See Chapter 21 for more information about NFS.

The first field gives the device name. The **fstab** file can include mounts from remote systems, in which case the first field contains an NFS path. The notation *server: /export* denotes the */export* directory on the machine named *server*.

The second field specifies the mount point, and the third field names the type of filesystem. The exact type name used to identify local filesystems varies among machines.

The fourth field specifies **mount** options to be applied by default. There are many possibilities; see the man page for **mount** for the ones that are common to all filesystem types. Individual filesystems usually introduce options of their own.

The fifth and sixth fields are vestigial. They are supposedly a “dump frequency” column and a column that controls **fsck** parallelism. Neither is important on contemporary systems.

The devices listed for **/dev/fd** and **/proc** are dummy entries. These virtual filesystems are task-specific and don’t require any additional information to be mounted. The other devices are identified by their GPT partition labels, which is a more robust option than using actual device names. To find out the label of an existing partition, run

gpart show -l *disk*

to print the partition table of the appropriate disk. To set the label on a partition, use

gpart modify -i *index* **-l** *label disk*

A cautionary note: partition tables are sometimes referred to as “disk labels.” Make sure when reading documentation that you distinguish between the label of an individual partition and the “label” of the disk itself. Overwriting a disk’s partition table is potentially disastrous.

UFS filesystems also have labels of their own, and these show up beneath the **/dev/ufs** directory. The UFS labels and partition labels are separate, but they can be (and probably should be) set to the same value. In this example, **/dev/ufs/spare** would work just as well as **/dev/gpt/spare**. To find a filesystem’s current label, run

tunefs -p *device*

To set the label, run

tunefs -L *label device.*

Unmount the filesystem before setting the label.

Below are some additional examples culled from an Ubuntu system's **fstab**. The general format is the same, but Linux systems use a different way to avoid naming disk devices.



```
# <file system> <mount point>  <type>       <options>             <d> <p>
proc            /proc          proc         defaults               0   0
UUID=a8e3…8f8a  /              ext4         errors=remount-ro      0   1
UUID=13e9…b8d2  none           swap         sw                     0   0
/dev/scd0       /media/cdrom0  udf,iso9660  user,noauto,exec,utf8  0   0
```

The first line addresses the **/proc** filesystem, which in fact is presented by a kernel driver and has no actual backing store. As in the FreeBSD example above, the proc device listed in the first column is just a placeholder.

The second and third lines use filesystem IDs (UUIDs, which we've truncated to make the excerpt more readable) instead of device names to identify volumes. This system is similar to the UFS label system used by FreeBSD, except that the identifiers are long random numbers instead of text strings. Use the **blkid** command to discover the UUID of a particular filesystem.

Filesystems can also have administratively assigned labels; use **e2label** or **xfs_admin** to read or set them. If you want to use labels in **fstab** (which is tidier), just substitute LABEL=*label* for UUID=*long-random-number*.

GPT disk partitions can have UUIDs and labels of their own that are independent of the UUIDs and labels of the filesystems they contain. For use of these options to identify partitions in the **fstab** file, the incantations are PARTUUID= and PARTLABEL=. However, common practice seems to have converged on the use of filesystem UUIDs.

You can also identify devices with pathnames beneath the **/dev/disk** directory. Subdirectories such as **/dev/disk/by-uuid** and **/dev/disk/by-partuuid** are automatically maintained by udev.

USB drive mounting

USB storage devices come in many flavors: personal “thumb” drives, digital cameras, and large external disks, to name a few. Most of these are supported by UNIX systems as data storage devices.

In the past, special tricks were necessary to manage USB devices. But now that operating systems have embraced dynamic device management as a fundamental requirement, USB drives are just one more type of device that shows up or disappears without warning.

From the perspective of storage management, the issues are two-fold:

- Getting the kernel to recognize a device and to assign a device file to it
- Finding out what assignment has been made

The first step usually happens automatically. Once a device file has been assigned, you can use the normal procedures described in *Disk device files* to find out what it is.For additional information about dynamic device management, see Chapter 11, *Drivers and the Kernel*.

Swapping recommendations

Raw partitions or logical volumes, rather than structured filesystems, are normally used for swap space. Instead of using a filesystem to keep track of the swap area's contents, the kernel maintains its own simplified mapping from memory blocks to swap space blocks.

On some systems, it's also possible to swap to a file in a filesystem partition. With older kernels this configuration can be slower than using a dedicated partition, but it's still handy in a pinch. In any event, logical volume managers eliminate most of the reasons you might want to use a swap file rather than a swap volume.

The more swap space you have, the more virtual memory your processes can allocate. The best virtual memory performance is achieved when the swap area is split among several drives. Of course, the best option of all is to not swap; consider adding RAM if you find yourself needing to optimize swap performance.

The proper amount of swap space to allocate depends on how a machine is used. There is no penalty to overprovisioning except that you lose the extra disk space. We suggest half the amount of RAM as a rule of thumb, but never less than 2GB on a physical server.

If a system will hibernate (personal machines, usually), it needs to be able to save the entire contents of memory to swap in addition to saving all the pages that would be swapped in normal operation. On these machines, increase the swap space recommended above by the amount of RAM.

Cloud and virtualized instances have their own peculiarities with respect to swap space. Paging is always a performance killer, so some sources recommend running without swap space entirely; if you need more memory, you need a larger instance. On the other hand, small instances usually have such meager RAM allotments that they can barely boot without a swap area. The general rule is that it's fine for instances to *have* swap space as long as you don't *use* it at steady state (or pay extra for it). Whatever approach you decide to take, check your base images to see how they're set up. Some come with swap preconfigured and some don't.

Some Amazon EC2 instances come with a local "instance store." This is essentially a slice of a local hard disk on the machine that runs the hypervisor. The contents of the instance store don't persist across starts and stops. The store is included in the price of the instance, so you may as well use it for swap space if nothing else.



On Linux systems, you initialize swap areas with **mkswap**, which takes the device name of the swap volume as an argument. **mkswap** writes some header information to the swap area. That data includes a UUID, which is why swap partitions count as "filesystems" from the perspective of **/etc/fstab** and can be identified there by UUID.

You can manually enable swapping to a particular device with **swapon** *device*. However, you will generally want to have this function automatically performed at boot time. Just list swap

areas in the regular **fstab** file and give them a filesystem type of swap.

To review the system's active swapping configuration, run **swapon -s** on Linux systems or **swapctl -l** on FreeBSD.

20.11 Next-generation filesystems: ZFS and Btrfs

Although ZFS and Btrfs are usually referred to as filesystems, they represent vertically integrated approaches to storage management that include the functions of a logical volume manager and a RAID controller. Although the current versions of both systems have a few limitations, most fall into the “not yet implemented” category rather than the “can’t do for architectural reasons” category.

Copy-on-write

Both ZFS and Btrfs avoid overwriting data in place and instead use a scheme known as “copy on write.” To update a block of metadata, for example, the filesystem modifies the in-memory copy and then writes it to a previously vacant disk block. Of course, that data block probably has a parent block that points to it, so the parent is rewritten as well, as is the parent’s parent, and so on back to the topmost level of the filesystem. (In practice, caching and careful design of data structures optimize-out most of these writes, at least in the short term.)

The advantage of this architecture is that the on-disk copy of the filesystem remains perpetually consistent. Before the root block is updated, the filesystem looks exactly as it did the last time the root was updated. A few “empty” blocks have been modified, but nothing points to them, so it makes no difference. The filesystem as a whole moves directly from one consistent state to another.

Error detection

ZFS and Btrfs also take data integrity far more seriously than do traditional filesystems. These systems store checksums for every disk block, and they verify all blocks read to ensure that misreads are detected. On storage pools that include mirroring or parity, bad data is automatically reconstructed from a known-good copy.

Disk drives implement their own layers of error detection and error correction, and although they fail frequently, they're not supposed to do so without reporting an error back to the host computer. Nevertheless, they sometimes do return bad data without an error indication.

One commonly cited rule of thumb is to expect an instance of silent data corruption for every 75TB of data read. A 2008 study by Bairavasundaram et al. examined service records of more than 1.5 million disk drives in NetApp servers and found that 0.5% of drives showed evidence of silent read errors in each year of service. (Interestingly, one key finding of this study was that enterprise-grade hard disks were an order of magnitude less likely to experience these types of errors.)

These error rates are small, but by all indications they're staying about the same even as disk capacities and the volumes of data stored on disks expand exponentially. Soon we'll have hard disks so large that you can't read the entire contents without a better-than-even chance of encountering a silent error. The extra validation done by ZFS and Btrfs is starting to look really important. (A related issue is the risk of random bit errors in RAM. They are infrequent but they do happen. All production servers should use—and monitor!—ECC memory.)

Parity RAID does not address this issue, at least in normal use. Parity can't be checked without a reading of the contents of an entire stripe, and it's inefficient to expand every disk access into a full-stripe read. Scrubbing can help find latent errors, but only if they're reproducible.

Performance

All the traditional filesystems that remain in common use have similar performance. It's possible to contrive workloads for which one filesystem or another has an edge, but general-purpose benchmarks rarely show much difference.

Copy-on-write filesystems access storage media somewhat differently from traditional filesystems, and they lack the decades of iterative refinement that have brought the old-guard filesystems to their current state of polish. Usually, the traditional filesystems set the upper bound on filesystem performance.

In many benchmarks, ZFS and Btrfs show performance comparable to traditional filesystems. But at their worst, these filesystems can be about half as fast as the traditional options.

Judging from Linux benchmarks (the only platform on which direct comparison is possible, since Btrfs is Linux-only), Btrfs currently has a slight performance edge over ZFS. However, the results vary widely by access pattern. It is not uncommon for one of these filesystems to perform well on a particular benchmark while the other lags far behind.

The performance picture is complicated by the fact that each of these filesystems has some potential tricks up its sleeve to increase performance. Benchmarks usually don't take account of these end-arounds. ZFS lets you add caching SSDs to a storage pool; it automatically copies frequently read data to the cache and avoids hitting the hard disks entirely. On Btrfs, you can use **chattr +C** to disable copy-on-write semantics for the data in specific files (usually large or frequently modified ones), thereby skirting some common low-performance scenarios.

For general use as root filesystems and home directory storage, ZFS and Btrfs perform well and offer many useful advantages. They can also work well as data storage for specific server workloads. However, in these latter scenarios, it's worth taking some time to double-check their behavior in your particular environment.

20.12 ZFS: ALL YOUR STORAGE PROBLEMS SOLVED

ZFS was introduced in 2005 as a component of OpenSolaris, and it quickly made its way to Solaris 10 and to various BSD-based distributions. In 2008, it became usable as a root filesystem, and it has been the front-line filesystem of choice for Solaris ever since. UFS remains the default root filesystem on FreeBSD, but ZFS has been an officially supported option since FreeBSD 10.

ZFS is more than just a filesystem, RAID controller, and volume manager wrapped into one. As originally conceived for OpenSolaris, it was a comprehensive rethinking of storage-related administration that addressed everything from the way filesystems were mounted to the way they were exported to other systems over NFS and SMB.

Modern BSD and Linux systems need to accommodate a variety of filesystems, so they've been forced to back off a bit from ZFS's original comprehensive approach. Nevertheless, ZFS remains a thoughtfully designed system that solves quite a few administrative problems through its architecture rather than through the addition of features.

ZFS on Linux

Although ZFS is free software, its use on Linux has been hampered by the fact that the source code is covered by Sun Microsystems' Common Development and Distribution License (CDDL). The Free Software Foundation maintains that the CDDL is incompatible with the GNU Public License, which covers the Linux kernel. Although add-on versions of ZFS for Linux have long been available through the OpenZFS project (openzfs.org), the FSF's position has discouraged Linux distributions from bundling ZFS into their base systems.

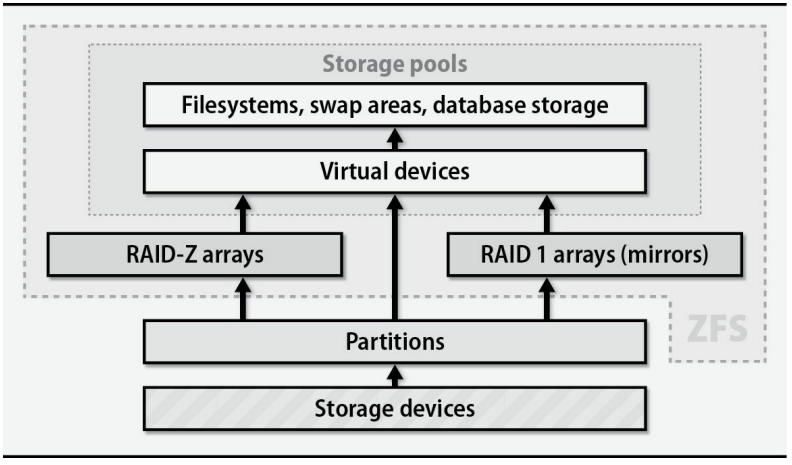
After nearly a decade of impasse over this issue, the FSF's position is at last being challenged by Canonical Ltd., developers of Ubuntu. After a legal review, Canonical formally disputed the FSF's interpretation of the GPL and included ZFS in Ubuntu 16.04 in the form of a loadable kernel module. So far (mid 2017), no lawsuit has resulted. If Canonical remains unpunished, it's possible that ZFS might become a fully supported root filesystem on Ubuntu and that other distributions might follow suit in supporting it.

If nothing else, the story of ZFS is an interesting case in which the GPL has actively impeded the development of an open source software package and blocked its adoption by users and distributors. If you're interested in the legal details, Richard Fontana's wrap-up of open source legal news for 2016 at goo.gl/PC9i3t includes a helpful summary.

ZFS architecture

Exhibit C shows a schematic of the major objects in the ZFS system and their relationship to each other.

Exhibit C: ZFS architecture



A ZFS "storage pool" is analogous to a "volume group" in other logical volume management systems. Each pool is composed of "virtual devices," which can be raw storage devices (disks, partitions, SAN devices, etc.), mirror groups, or RAID arrays. ZFS RAID is similar in spirit to RAID 5 in that it uses one or more parity devices to implement redundancy for the array. However, ZFS calls the scheme RAID-Z and uses variable-sized stripes to eliminate the RAID 5 write hole. All writes to the storage pool are striped across the pool's virtual devices, so a pool that contains only individual storage devices is effectively an implementation of RAID 0, although the devices in this configuration are not required to be of the same size.

Unfortunately, the current ZFS RAID is a bit brittle: you cannot add new devices to an array once it has been defined; nor can you permanently remove a device. As in most RAID implementations, devices in a RAID set must be the same size. You can force ZFS to accept mixed sizes, but the size of the smallest volume then dictates the overall size of the array. To use disks of different sizes efficiently in combination with ZFS RAID, you must partition the disks ahead of time and define the leftover regions as separate devices.

Most configuration and management of ZFS is done through two commands: **zpool** and **zfs**. Use **zpool** to build and manage storage pools. Use **zfs** to create and manage the entities created from pools, chiefly filesystems and raw volumes used as swap space, database storage, or backing for SAN volumes.

Example: disk addition

Before we descend into the details of ZFS, here's a high-level example. Suppose you've added a new disk to your FreeBSD system and the disk has shown up as **/dev/ada1**. (An easy way to determine the correct device is to run **geom disk list**.)

The first step is to add the disk to a new storage pool:

```
$ sudo zpool create demo ada1
```

Step two is… well, there is no step two. ZFS creates the pool "demo," creates a filesystem root inside that pool, and mounts that filesystem as **/demo**. The filesystem is automatically remounted when the system boots.

```
$ ls -a /demo
.   ..
```

It would be even more impressive if we could simply add our new disk to the existing storage pool of the root disk, which on FreeBSD is called "zroot" by default. (The command would be **sudo zpool add rpool ada1**.) Unfortunately, the root pool can contain only a single virtual device. Other pools can be painlessly extended in this manner, however.

Filesystems and properties

It's fine for ZFS to automatically create a filesystem on a new storage pool—by default, ZFS filesystems consume no particular amount of space. All filesystems that live in a pool can draw from the pool's available space.

Unlike traditional filesystems, which are independent of one another, ZFS filesystems are hierarchical and interact with their parent and child filesystems in several ways. You create new filesystems with **zfs create**:

```
$ sudo zfs create demo/new_fs
$ zfs list -r demo
NAME           USED  AVAIL  REFER  MOUNTPOINT
demo           432K   945G    96K  /demo
demo/new_fs     96K   945G    96K  /demo/new_fs
```

The **-r** flag to **zfs list** makes it recurse through child filesystems. Most other **zfs** subcommands understand **-r**, too. Ever helpful, ZFS automounts the new filesystem as soon as you create it.

To simulate traditional filesystems of fixed size, you can adjust the filesystem's properties to add a "reservation" (an amount of space reserved in the storage pool for the filesystem's use) and a quota. This adjustment of filesystem properties is one of the keys to ZFS management, and it's something of a paradigm shift for administrators who are accustomed to other systems. Here, we set both values to 1GB:

```
$ sudo zfs set reservation=1g demo/new_fs
$ sudo zfs set quota=1g demo/new_fs
$ zfs list -r demo
NAME           USED  AVAIL  REFER  MOUNTPOINT
demo          1.00G   944G    96K  /demo
demo/new_fs     96K  1024M    96K  /demo/new_fs
```

The new quota is reflected in the AVAIL column for **/demo/new_fs**. Similarly, the reservation shows up immediately in the USED column for **/demo**. That's because the reservations of **/demo**'s descendant filesystems are included in its size tally. (The REFER column shows the amount of data referenced by the active copy of each filesystem. **/demo** and **/demo/new_fs** have similar REFER values because they're both empty filesystems, not because there's any inherent relationship between the numbers.)

Both property changes are purely bookkeeping entries. The only change to the actual storage pool is the update of a block or two to record the new settings. No process goes out to format the 1GB of space reserved for **/demo/new_fs**. Most ZFS operations, including the creation of new storage pools and new filesystems, are similarly lightweight.

Using this hierarchical system of space management, you can easily group several filesystems to guarantee that their collective size does not exceed a certain threshold; you need not specify

limits on individual filesystems.

You must set both the quota and reservation properties to properly emulate a traditional fixed-size filesystem. The reservation alone simply ensures that the filesystem has enough room available to grow *at least* that large. The quota limits the filesystem's maximum size *without* guaranteeing that space is available for this growth; another object could snatch up all the pool's free space, leaving no room for **/demo/new_fs** to expand.

On the other hand, there are few reasons to set up a filesystem this way in real life. We show the use of these properties simply to demonstrate ZFS's space accounting system and to emphasize that ZFS is compatible with the traditional model, should you wish to enforce it.

Property inheritance

Many properties are naturally inherited by child filesystems. For example, if we wanted to mount the root of the demo pool in **/mnt/demo** instead of **/demo**, we could simply set the root's mountpoint parameter:

```
$ sudo zfs set mountpoint=/mnt/demo demo

$ zfs list -r demo
NAME           USED  AVAIL  REFER  MOUNTPOINT
demo          1.00G   944G    96K  /mnt/demo
demo/new_fs     96K  1024M    96K  /mnt/demo/new_fs

$ ls /mnt/demo
new_fs
```

Setting the mountpoint parameter automatically remounts the filesystems, and the mount point change affects child filesystems in a predictable and straightforward way. The usual rules regarding filesystem activity still apply, however; see this page.

Use **zfs get** to see the effective value of a particular property; **zfs get all** dumps them all. The SOURCE column tells you why each property has its particular value: local means that the property was set explicitly, and a dash means that the property is read-only. If the property value is inherited from an ancestor filesystem, SOURCE shows the details of that inheritance as well.

```
$ zfs get all demo/new_fs
NAME          PROPERTY         VALUE                  SOURCE
demo/new_fs   type             filesystem             -
demo/new_fs   creation         Mon Apr 03  0:12 2017  -
demo/new_fs   used             96K                    -
demo/new_fs   available        1024M                  -
demo/new_fs   referenced       96K                    -
demo/new_fs   compressratio    1.00x                  -
demo/new_fs   mounted          yes                    -
demo/new_fs   quota            1G                     local
demo/new_fs   reservation      1G                     local
demo/new_fs   mountpoint       /mnt/new_fs            inherited from demo
demo/new_fs   checksum         on                     default
demo/new_fs   compression      off                    default
... <many more, about 55 in all>
```

Vigilant readers might notice that the available and referenced properties look suspiciously similar to the AVAIL and REFER columns shown by **zfs list**. In fact, **zfs list** is just a different way of displaying filesystem properties. If we had included the full output of our **zfs get** command above, there would be a used property in there, too. Use the **-o** option to specify the properties you want **zfs list** to show.

It wouldn’t make sense to assign values to `used` and to the other size properties, so these properties are read-only. If the specific rules for calculating `used` don’t meet your needs, other properties such as `usedbychildren` and `usedbysnapshots` may give you better insight into how your disk space is being consumed.

You can set additional, nonstandard properties on filesystems for your own use and for the use of your local scripts. The process is the same as that for standard properties. For example, many backup and snapshot utilities for ZFS read their configuration information from filesystem properties.

The names of custom properties must include a colon to distinguish them from standard properties.

One filesystem per user

Since filesystems consume no space and take no time to create, the optimal number of them is closer to “a lot” than “a few.” If you keep users’ home directories on a ZFS storage pool, you may find it helpful to make each home directory a separate filesystem.

There are several benefits:

- If you need to set disk usage quotas, home directories are a natural granularity at which to do this. You can set quotas on both individual users’ filesystems and on the filesystem that contains all users.
- Snapshots are per filesystem. If each user’s home directory is a separate filesystem, the user can access old snapshots through **~/.zfs**. This feature alone is a huge time saver for administrators because it means that users can service most of their own file restore needs. (The **.zfs** directory is hidden by default. You can make it visible with **zfs set snapdir=visible** *filesystem*.)
- ZFS lets you delegate permission to perform various operations such as taking snapshots or rolling back the filesystem to an earlier state. If you prefer, you can give users control over these operations for their own home directories. We do not describe the details of ZFS permission management in this book, however; see the man page entry for **zfs allow**.

Snapshots and clones

Just like a logical volume manager, ZFS brings copy-on-write to the user level by allowing you to create instantaneous snapshots. However, there's an important difference: ZFS snapshots are implemented per filesystem rather than per volume, so they have arbitrary granularity.

On the command line, you create snapshots with **zfs snapshot**. For example, the following command sequence illustrates creation of a snapshot, use of the snapshot through the filesystem's **.zfs/snapshot** directory, and reversion of the filesystem to its previous state.

```
$ sudo touch /mnt/demo/new_fs/now_you_see_me
$ ls /mnt/demo/new_fs
now_you_see_me
$ sudo zfs snapshot demo/new_fs@snap1
$ sudo rm /mnt/demo/new_fs/now_you_see_me
$ ls /mnt/demo/new_fs
$ ls /mnt/demo/new_fs/.zfs/snapshot/snap1
now_you_see_me
$ sudo zfs rollback demo/new_fs@snap1
$ ls /opt/demo/new_fs
now_you_see_me
```

You assign a name to each snapshot at the time it's created. The complete specifier for a snapshot is usually written in the form *filesystem@snapshot.*

Use **zfs snapshot -r** to create snapshots recursively. The effect is the same as executing **zfs snapshot** on each contained object individually: each subcomponent receives its own snapshot. All the snapshots have the same name, but they're logically distinct because the *filesystem* portion is different.

ZFS snapshots are read-only, and although they can bear properties, they are not true filesystems. However, you can instantiate a snapshot as a full-fledged, writable filesystem by "cloning" it:

```
$ sudo zfs clone demo/new_fs@snap1 demo/subclone
$ ls /mnt/demo/subclone
now_you_see_me
$ sudo touch /mnt/demo/subclone/and_me_too
$ ls /mnt/demo/subclone
and_me_too  now_you_see_me
```

The snapshot that is the basis of the clone remains undisturbed and read-only. However, the new filesystem (**demo/subclone** in this example) retains a link to both the snapshot and the filesystem on which it's based, and neither of those entities can be deleted as long as the clone exists.

Cloning isn't a common operation, but it's the only way to create a branch in a filesystem's evolution. The **zfs rollback** operation demonstrated above can only return a filesystem to its most recent snapshot, so to use it you must permanently delete (**zfs destroy**) any snapshots made

since the snapshot that is your reversion target. Cloning lets you go back in time without losing access to recent changes.

For example, suppose that you've discovered a security breach that occurred some time within the last week. For safety, you want to return a filesystem to its state of a week ago to be sure today that it contains no hacker-installed back doors. At the same time, you don't want to lose recent work or the data for forensic analysis. The solution is to clone the week-ago snapshot to a new filesystem, **zfs rename** the old filesystem, and then **zfs rename** the clone in place of the original filesystem.

For good measure, also **zfs promote** the clone; this operation inverts the relationship between the clone and the filesystem of origin. After promotion, the main-line filesystem has access to all the old filesystem's snapshots, and the old, moved-aside filesystem becomes the "cloned" branch.

Raw volumes

You create swap areas and raw storage areas with **zfs create**, just as you create filesystems. The **-V** *size* argument makes **zfs** treat the new object as a raw volume instead of a filesystem. The *size* can use any common unit, for example, **128m**.

Since the volume does not contain a filesystem, it is not mounted; instead, it shows up in the **/dev/zvol** directory and can be referenced as if it were a hard disk or partition. ZFS mirrors the hierarchical structure of the storage pool in these directories, so **sudo zfs create -V 128m demo/swap** creates a 128MB swap volume located at **/dev/zvol/demo/swap**.

You can create snapshots of raw volumes just as you can with filesystems, but because there's no filesystem hierarchy in which to put a **.zfs/snapshot** directory, the snapshots show up in the same directory as their source volumes. Clones work too, just as you'd expect.

By default, raw volumes receive a space reservation equal to their specified size. You're free to reduce the reservation or to do away with it entirely, but note that this configuration can make writes to the volume return an "out of space" error. Clients of raw volumes might not be designed to deal with such an error.

Storage pool management

Now that we've waded into some of the features that ZFS offers at the filesystem and block-client level, we can go for a longer swim in ZFS's storage pools.

Up to now, we've used a pool called "demo" that we created from a single disk back on this page. Here it is in the output of **zpool list**:

```
$ zpool list
NAME    SIZE  ALLOC   FREE  EXPANDSZ   FRAG    CAP  DEDUP  HEALTH  ALTROOT
demo    976M   516K   976G         -     0%     0%  1.00x  ONLINE  -
zroot  19.9G  16.3G  3.61G         -    24%    81%  1.00x  ONLINE  -
```

The pool named "zroot" contains the bootable root filesystem. Bootable pools are currently restricted in several ways: they can contain only a single virtual device, and that device must be either a mirror array or a single disk drive; it cannot be a striping set or a RAID-Z array. (This is either an implementation limit or a strong push in the direction of robustness for the root filesystem; we're not sure which.)

zpool status adds more detail about the virtual devices that make up a storage pool and reports their current status.

```
$ zpool status demo
  pool: demo
 state: ONLINE
  scan: none requested
config:

    NAME       STATE    READ WRITE CKSUM
    demo      ONLINE      0    0    0
      ada1    ONLINE      0    0    0

errors: No known data errors
```

Time to get rid of this demo pool and set up something a bit more sophisticated. We've attached five 1TB drives to our example system. We first create a pool called "monster" that includes three of those drives in a RAID-Z single-parity configuration.

```
$ sudo zpool destroy demo
$ sudo zpool create monster raidz1 ada1 ada2 ada3
$ zfs list monster
NAME      USED  AVAIL  REFER  MOUNTPOINT
monster  87.2K  1.84T  29.3K  /monster
```

ZFS also understands **raidz2** and **raidz3** for double and triple parity configurations. The minimum number of disks is always one more than the number of parity devices. Here, one drive out of three is used for parity, so roughly 2TB is available for use by filesystems.

For illustration, we then add the remaining two drives configured as a mirror:

```
$ sudo zpool add monster mirror ada4 ada5
invalid vdev specification
use '-f' to override the following errors:
mismatched replication level: pool uses raidz and new vdev is mirror
$ sudo zpool add -f monster mirror ada4 ada5
```

zpool initially balks at this configuration because the two virtual devices have different redundancy schemes. This particular configuration is OK since both vdevs have some redundancy. In actual use, do not mix redundant and nonredundant vdevs since there's no way to predict which blocks might be stored on which devices; partial redundancy is useless.

```
$ zpool status monster
  pool: monster
 state: ONLINE
  scan: none requested
config:

    NAME        STATE     READ WRITE CKSUM
    monster     ONLINE       0     0     0
      raidz1-0  ONLINE       0     0     0
        ada1    ONLINE       0     0     0
        ada2    ONLINE       0     0     0
        ada3    ONLINE       0     0     0
      mirror-1  ONLINE       0     0     0
        ada4    ONLINE       0     0     0
        ada5    ONLINE       0     0     0

errors: No known data errors
```

ZFS distributes writes among all a storage pool's virtual devices. As demonstrated in the preceding example, it is not necessary for all virtual devices to be the same size. (In this example the *disks* are all the same size, but the virtual devices are not.)However, the components within a redundancy group should be of similar size. If they are not, only the smallest size is used on each component. Multiple simple disks used together in a storage pool is essentially a RAID 0 configuration.

You can add additional vdevs to a pool at any time. However, existing data won't be redistributed to take advantage of parallelism. Unfortunately, you cannot currently add additional devices to an existing RAID array or mirror. This is an area in which Btrfs has a distinct advantage, since it accommodates all sorts of reorganizations in a relatively clean and automatic manner.

ZFS has an especially nice implementation of read caching that makes good use of SSDs. To set up this configuration, just add the SSDs to the storage pool as vdevs of type **cache**. The caching system uses an adaptive replacement algorithm developed at IBM that is smarter than a normal

LRU (least recently used) cache. It knows about the frequency at which blocks are referenced as well as their recentness of use, so reads of large files are not supposed to wipe out the cache.

Hot spares are handled as vdevs of type **spare**. You can add the same disk to multiple storage pools; whichever pool experiences a disk failure first gets to claim the spare disk.

20.13 Btrfs: “ZFS lite” for Linux

Oracle’s Btrfs filesystem project (“B-tree file system,” officially pronounced “butter FS” or “better FS,” though it’s hard not to think “butter face”) aimed to repeat many of ZFS’s advances on the Linux platform during the long interregnum when ZFS seemed like it might be lost to Linux because of licensing issues.

Although Btrfs remains under active development, it’s been a standard part of the Linux kernel trunk since 2009. It’s available and ready to use on nearly all Linux systems, and SUSE Enterprise Linux has even made it a supported option for the root filesystem. Because the code base evolves quickly, it’s probably best to avoid Btrfs on stability-oriented distributions such as Red Hat for now; old versions have known issues.

Btrfs vs. ZFS

Because they share some technical underpinnings, comparisons between Btrfs and ZFS are probably inevitable. However, Btrfs is not a ZFS clone, and it doesn't seek to reproduce ZFS's architecture. For example, you mount Btrfs volumes just like those of other filesystems, by running the **mount** command or by listing them in the **/etc/fstab** file.

Although Btrfs volumes and their subvolumes exist in a unified namespace, there's no hierarchical relationship among them. To make a change to a group of Btrfs subvolumes, you must modify each of them individually. Btrfs commands do not operate recursively, and volume properties are not inheritable. This isn't an omission so much as a design choice: why load up the filesystem (the developers ask) with features that you can emulate in a shell script?

Btrfs reflects this preference for simplicity in a variety of ways. For example, Btrfs storage pools can include only one group of disks in one particular configuration (e.g., RAID 5), whereas ZFS pools can include multiple disk groups as well as caching disks, intent logs, and hot spares.

As is common in the software arena, debates over the relative merits of ZFS and Btrfs tend to become heated and to focus on stylistic distinctions. However, several important differences between the two systems rise above the level of nitpicking and personal preference.

- Btrfs is the clear winner when it comes to changing your hardware configuration; ZFS didn't even show up for this fight. You can add or remove disks at any time, or even change RAID type, and Btrfs redistributes existing data accordingly while remaining on-line. In ZFS, such changes are usually impossible without your dumping your data to external media and starting over.
- Even without memory-intensive features (such as deduplication) enabled, ZFS functions best with a generous amount of RAM. 2GB is the recommended minimum. That's a lot of memory for a virtual server.
- ZFS's ability to cache frequently read data on separate cache SSDs is a killer feature for many use cases, and one for which Btrfs currently has no answer.
- As of 2017, the Btrfs implementations of parity raid (RAID 5 and 6) are not yet ready for production use. That's not our opinion; it's the official word from the developers. This is a significant missing feature.

Setup and storage conversion

In this section we demonstrate a few common Btrfs procedures analogous to those shown for ZFS in previous sections. We first set up Btrfs for use on a set of two 1TB hard disks configured for RAID 1 (mirroring):

```
$ sudo mkfs.btrfs -L demo -d raid1 /dev/sdb /dev/sdc
Label:              demo
UUID:
Node size:          16384
Sector size:        4096
Filesystem size:    1.91TiB
Block group profiles:
  Data:             RAID1             1.00GiB
  Metadata:         RAID1             1.00GiB
  System:           RAID1             8.00MiB
SSD detected:       no
Incompat features:  extref, skinny-metadata
Number of devices:  2
Devices:
   ID        SIZE  PATH
    1   978.00GiB  /dev/sdb
    2   978.00GiB  /dev/sdc

$ sudo mkdir /mnt/demo
$ sudo mount LABEL=demo /mnt/demo
```

We could name any of the component devices in the **mount** command line, but it's simplest to just use the label we assigned to the group, "demo."

The **btrfs filesystem usage** command shows how the space on these disks is currently being used:

```
$ sudo btrfs filesystem usage /mnt/demo
Overall:
    Device size:                   1.91TiB
    Device allocated:              4.02GiB
    Device unallocated:            1.91TiB
    Device missing:                  0.00B
    Used:                          1.25MiB
    Free (estimated):            976.99GiB      (min: 976.99GiB)
    Data ratio:                       2.00
    Metadata ratio:                   2.00
    Global reserve:                16.00MiB      (used: 0.00B)

Data,RAID1: Size:1.00GiB, Used:512.00KiB
   /dev/sdb        1.00GiB
   /dev/sdc        1.00GiB

Metadata,RAID1: Size:1.00GiB, Used:112.00KiB
   /dev/sdb        1.00GiB
   /dev/sdc        1.00GiB

System,RAID1: Size:8.00MiB, Used:16.00KiB
   /dev/sdb        8.00MiB
   /dev/sdc        8.00MiB

Unallocated:
   /dev/sdb      975.99GiB
   /dev/sdc      975.99GiB
```

btrfs subcommands can be abbreviated to any unique prefix. For example, **btrfs filesystem usage** is also accessible as **btrfs f u**. We spell out commands for clarity and propriety.

The interesting thing to note in the output above is the small initial allocations into the RAID 1 groups for data, metadata, and system blocks. Most disk space remains in an unallocated pool that has no intrinsic structure. The mirroring we requested isn't imposed on the disks as a whole, just on the blocks that are actually in use. It's not a rigid structure so much as a policy to be implemented at the level of block groups.

This distinction is key to understanding how Btrfs can adapt to changing requirements and hardware provisioning. Here's what happens when we store some files into the new filesystem and then add a third disk:

```
$ mkdir /mnt/demo/usr
$ cd /usr; tar cf - . | (cd /mnt/demo/usr; sudo tar xfp -)
$ sudo btrfs device add /dev/sdd /mnt/demo
$ sudo btrfs filesystem usage /mnt/demo
Overall:
   <omitted from this output>

Data,RAID1: Size:3.00GiB, Used:2.90GiB
  /dev/sdb       3.00GiB
  /dev/sdc       3.00GiB

Metadata,RAID1: Size:1.00GiB, Used:148.94MiB
  /dev/sdb       1.00GiB
  /dev/sdc       1.00GiB

System,RAID1: Size:8.00MiB, Used:16.00KiB
  /dev/sdb       8.00MiB
  /dev/sdd     978.00GiB

  /dev/sdc     973.99GiB
  /dev/sdb     973.99GiB
Unallocated:

  /dev/sdc       8.00MiB
```

The new disk, **/dev/sdd**, has become available to the pool, but the existing block groups are fine as they were, so none of them reference the new disk. Future allocations would automatically take advantage of the new disk. If we like, we can force Btrfs to level the data among all disks:

```
$ sudo btrfs balance start --full-balance /mnt/demo
Starting balance without any filters.
Done, had to relocate 5 out of 5 chunks
```

Conversion among RAID levels is also a form of balancing. Now that we have three disks available, we can convert to RAID 5:

```
$ sudo btrfs balance start -dconvert=raid5 -mconvert=raid5 /mnt/demo
Done, had to relocate 5 out of 5 chunks
```

If we had glanced at the usage data during the conversion, we'd have seen block groups for both RAID 1 and RAID 5 active simultaneously. Disk removals work similarly: Btrfs incrementally copies all blocks to groups that don't include the leaving disk, and eventually no data remains there.

Volumes and subvolumes

Snapshots and quotas are filesystem-level entities in Btrfs, so it's helpful to be able to define portions of the file tree as distinct entities. Btrfs calls these "subvolumes." A subvolume looks a lot like a regular filesystem directory, and in fact, it remains accessible as a subdirectory of its parent volume, as shown below.

```
$ sudo btrfs subvolume create /mnt/demo/sub
Create subvolume '/mnt/demo/sub'
$ sudo touch /mnt/demo/sub/file_in_a_subvolume
$ ls /mnt/demo/sub
file_in_a_subvolume
```

The subvolume is not automatically mounted; it's visible here as part of the parent volume. However, you *can* mount a subvolume independently of its parent with the **subvol** mount option. For example,

```
$ mkdir /sub
$ sudo mount LABEL=demo -o subvol=/sub /sub
$ ls /sub
file_in_a_subvolume
```

There is no way to prevent a subvolume from showing up within its parent volume when the parent is mounted. To create the illusion of multiple, independent, noninteracting volumes, just make them subvolumes of the root and mount each of them separately with the **subvol** option. The root itself is not required to be mounted anywhere. In fact, Btrfs lets you specify a volume other than the root to be the default mount target when no **subvol** is requested; see **btrfs subvolume set-default**.

To see or manipulate the full Btrfs hierarchy under this configuration, just mount the root on a scratch directory with **subvol=/**. It's fine for volumes to be mounted several times and accessible through multiple paths.

Volume snapshots

Btrfs's version of volume snapshots works a lot like **cp**, except that copies are shallow and initially share all their storage with the parent volume:

```
$ sudo btrfs subvolume snapshot /mnt/demo/sub /mnt/demo/sub_snap
Create a snapshot of '/mnt/demo/sub' in '/mnt/demo/sub_snap'
```

Unlike ZFS snapshots, Btrfs snapshots are writable by default. In fact, there is no such thing as a "snapshot" per se in Btrfs; a snapshot is just a volume that happens to share some storage with another volume:

```
$ sudo touch /mnt/demo/sub/another_file
$ ls /mnt/demo/sub
another_file  file_in_a_subvolume
$ ls /mnt/demo/sub_snap
file_in_a_subvolume
```

For an immutable snapshot, just pass the **-r** option to **btrfs subvolume snapshot**. Btrfs does not make a fundamental distinction between read-only snapshots and writable copies in the way that ZFS does. (In ZFS, writable copies are "clones." To create one, first make a read-only snapshot, then create a clone based on that snapshot.)

Btrfs does not enforce any particular naming or location conventions when it comes to defining subvolumes and snapshots, so it's up to you to decide how these entities should be organized and named. The Btrfs documentation at btrfs.wiki.kernel.org suggests a couple of conventions for your consideration.

Btrfs also has no "rollback" operation that resets a volume to its state as of a particular snapshot. Instead, you can just move the original volume aside and **mv** or copy a snapshot in its place:

```
$ ls /mnt/demo/sub
another_file  file_in_a_subvolume
$ sudo mv /mnt/demo/sub /mnt/demo/sub.old
$ sudo btrfs subvolume snapshot /mnt/demo/sub_snap /mnt/demo/sub
Create a snapshot of '/mnt/demo/sub_snap' in '/mnt/demo/sub'
$ ls /mnt/demo/sub
file_in_a_subvolume
```

Note that this change confuses direct mounts of the subvolume. They'll need to be remounted afterward.

Shallow copies

The analogy between Btrfs snapshots and **cp** is more than just coincidence. You cannot create snapshots—as such—of files or of directories that are not subvolume roots. But interestingly, you can create shallow copies of arbitrary files and directories with **cp --reflink**, even across subvolume boundaries.

This option activates Btrfs-specific magic inside **cp** that negotiates directly with the filesystem to arrange for copy-on-write duplication. The semantics are identical to those of a normal **cp** and also perilously close to those of a snapshot.

Btrfs doesn't track shallow copies for you as it would with snapshots, and it also doesn't necessarily guarantee perfect point-in-time consistency for actively modified directory hierarchies. But in other respects, the two operations are markedly similar. One nice feature of shallow copies is that they require no special permissions; any user can take advantage of them.

If you specify the **cp** option in the form **--reflink=auto**, **cp** shallow-copies when it can and behaves normally otherwise. That makes it a tempting target for a **~/.bashrc** alias:

```
alias cp="cp --reflink=auto"
```

20.14 Data backup strategy

On a good day, your main focus within the storage environment is to ensure that performance remains good and that sufficient free space is available. Unfortunately, not every day is a good day. With the Google Labs study finding that a disk drive has less than a 75% chance of surviving for five years, the deck is stacked against us. Always have systems in place to protect valuable data against catastrophic loss, and be prepared to activate your recovery procedure at short notice.

RAID and other data replication schemes protect against the failure of a single facility or piece of hardware. However, there are many other ways to lose data that these technologies do not address. For example, if you experience a security breach or an infection by ransomware, your data can be altered or corrupted even though the physical layer remains perfectly intact. Automated replication of compromised data to multiple disks or sites only increases the misery. You need immutable, point-in-time backups of critical data that you can revert to as a fallback option.

In past decades, media such as magnetic tapes were a popular storage method for off-line backups. However, the capacity of these media proved unable to keep up with the exponentially growing sizes of hard drives and SSDs. Along with the physical challenges of transporting and storing tapes and of maintaining finicky mechanical tape drives, the capacity issues ultimately relegated tape media to the status of 35mm camera film: it's still technically on the market, but you have to wonder who's actually buying the stuff.

Today, most cloud platforms let you capture point-in-time backups in the form of snapshots, usually on an automated schedule. You pay a monthly fee for the storage consumed by each snapshot and can set your own retention policies.

Regardless of the exact technology you use to implement backups, you need a written plan that answers at least the following questions.

Overall strategy:

- What data is to be backed up?
- What system or technology will perform the backups?
- Where will backup data be stored?
- Will backups be encrypted? If so, where will encryption keys be stored?
- How much will it cost to store backups over time?

Timelines:

- How often will backups be performed?
- How often will backups be validated and restore-tested?
- How long will backups be retained?

People:

- Who will have access to backup data?
- Who will have access to the encryption keys that protect backup data?
- Who will be in charge of verifying the execution of backups?
- Who will be in charge of validating and restore-testing backups?

Use and protection:

- How will backup data be accessed or restored in an emergency?
- How will you ensure that neither a hacker nor a bogus process can corrupt, modify, or delete backups? (That is, how will you achieve immutability?)
- How will backup data be protected against being taken hostage by an adversarial cloud provider, vendor, or government?

The best answers to these questions vary by organization, type of data, regulatory environment, technology platform, and budget, just to name a few potential factors.

Take time today to map out a backup plan for your environment or to review your existing backup plan.

20.15 Recommended reading

Lucas, Michael W., and Allan Jude. *FreeBSD Mastery: ZFS*. Tilted Windmill Press, 2015.

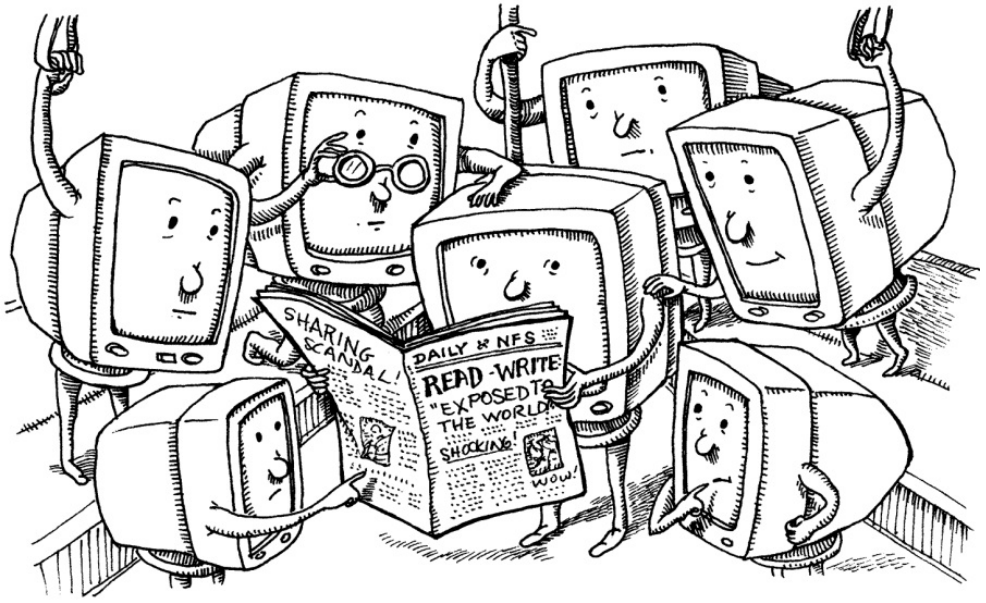
Jude, Allan, and Michael W. Lucas. *FreeBSD Mastery: Advanced ZFS*. Tilted Windmill Press, 2016.

The two titles above are the go-to references for modern ZFS. Although they purport to be FreeBSD-specific, most of the material applies to ZFS on Linux as well. The *Advanced ZFS* book is particularly useful in its coverage of topics as varied as jails, permission delegation, caching strategies, and performance analysis.

Lucas, Michael W., and Allan Jude. *FreeBSD Mastery: Storage Essentials*. Tilted Windmill Press, 2015.

McKusick, Marshall Kirk, George V. Neville-Neil, and Robert N. M. Watson. *The Design and Implementation of the FreeBSD Operating System (2nd Edition)*. Upper Saddle River, NJ: Addison-Wesley Professional, 2014. This book addresses a variety of kernel-related subjects, but it includes complete chapters on UFS, ZFS, and the VFS layer.

21 The Network File System



The Network File System protocol, commonly known as NFS, lets you share filesystems among computers. NFS is nearly transparent to users, and no information is lost when an NFS server crashes. Clients can simply wait until the server returns and then continue as if nothing had happened.

NFS was introduced by Sun Microsystems in 1984. It was originally implemented as a surrogate filesystem for diskless clients, but the protocol proved to be well designed and useful as a general file sharing solution. These days, all UNIX vendors and Linux distributions offer some version of NFS. The NFS protocol is an open standard that is documented in RFCs (see RFCs 1094, 1813, and 7530 in particular).

21.1 Meet network file services

The goal of a network file service is to grant shared access to files and directories that are stored on the disks of remote systems. User applications must be able to read and write to these files with the same system calls they use for local files; that files are stored elsewhere on the network should be transparent to applications. If more than one network client or application attempts to modify a file simultaneously, the file sharing service must resolve any conflicts that arise.

The competition

See Chapter 22 for more details on SMB and Samba.

NFS is not the only file sharing system around. The Server Message Block (SMB) protocol underlies the file sharing capabilities built into Windows and macOS. However, UNIX and Linux can also speak SMB by running the Samba add-on package. If you run a hybrid network that includes a variety of different operating systems, you may find that SMB is the path that presents the fewest compatibility hurdles.

NFS is most commonly used in shops where UNIX and Linux are predominant. In those contexts it offers a somewhat more natural fit and a higher degree of integration. But even in these environments, SMB remains a plausible option. It's uncommon—but not unheard of—for sites consisting exclusively of UNIX and Linux systems to rely on SMB as their primary file sharing protocol.

Sharing files over a network seems like a simple task, but in fact it's a confoundingly complex problem that abounds with edge cases and subtleties. Many protocol issues have come to light only through bugs encountered in unusual situations. Both NFS and SMB show the scars of battles fought to maintain security, performance, and reliability over decades of development and widespread use. Today's administrators can be confident that these protocols will not regularly corrupt data or otherwise incur the wrath of irate users, but it has taken a lot of work and experience to get to this point.

Storage area network (SAN) systems are another option for high-performance storage management on a network. SAN servers need not understand filesystems because they serve only disk blocks, unlike NFS and SMB, which operate at the level of filesystems and files rather than raw storage devices. A SAN affords fast read/write access, but it's unable to manage concurrent access by multiple clients without the help of a clustered filesystem.

For big data projects, several open source distributed filesystems have come into common use. GlusterFS and Ceph implement both POSIX-compliant filesystems and RESTful object storage distributed among a cluster of nodes for fault tolerance. Commercial versions of both systems are sold by Red Hat, which acquired both developers. Both systems are production-ready, highly capable filesystems worthy of consideration for use cases like big data processing and high performance computing.

Cloud-based systems have additional options. Refer to this page.

Issues of state

One decision made when designing a network filesystem is to determine what part of the system will track the files that each client has open, information referred to generically as “state.” A server that records the status of files and clients is said to be stateful; one that does is stateless. Both approaches have been used over the years, and both have benefits and drawbacks.

Stateful servers keep track of all open files across the network. This mode of operation introduces many layers of complexity (more than you might expect) and makes recovery in the event of a crash far more difficult. When the server returns from a hiatus, a negotiation between the client and server must occur to reconcile the last known state of the connection. Statefulness lets clients maintain more control over files and facilitates the robust management of files opened in read/write mode.

On a stateless server, each request is independent of the requests that have preceded it. If either the server or the client crashes, nothing is lost in the process. Under this design, it is painless for servers to crash or reboot, since they do not maintain any context. However, it’s impossible for the server to know which clients have opened a file for writing, so it cannot manage concurrency.

Performance concerns

Network filesystems should present a seamless experience to users. Accessing a file over the network should be no different from accessing a file on a local filesystem. Unfortunately, wide area networks have high latencies, which make operations behave erratically, and low bandwidth, which yields slow performance on large files. Most file service protocols, including NFS, incorporate techniques to minimize performance problems on both local and wide area networks.

Most protocols try to minimize the number of network requests. For example, read-ahead caching preloads portions of a file into a local memory buffer to avoid a delay when a new section of the file is read. A little extra network bandwidth is consumed in an effort to avoid the latency of a full round-trip exchange with the server.

Similarly, some systems cache writes in memory and send updates in batches, reducing the delay incurred when communicating write operations to the server. These types of batch operations are referred to generically as request coalescing.

Security

Any service that grants convenient access to files on a network has great potential to cause security problems. Local filesystems implement complex access control algorithms and safeguard files with granular permissions. On a network, these tasks are greatly complicated by differences in configurations among machines and by vagaries such as race conditions, bugs in file service software, and unresolved edge cases in file sharing protocols.

The rise of directory and centralized authentication services has improved the security of network filesystems. The bottom line is that no client can be trusted to authenticate itself sanely, so a trusted, central system must verify identities and approve access to files. Most file sharing services can be integrated with a variety of different authentication providers.

File sharing protocols do not typically address the issues of privacy and integrity—or at least, they do not do so directly. As with authentication, this responsibility is generally outsourced to another layer such as a Kerberos, SSH, or a VPN tunnel. However, recent versions of SMB have added strong encryption and integrity checking. Many sites that run NFS on a trusted LAN choose to forgo cryptography because an easy and high-performance solution is unavailable.

21.2 The NFS approach

The newest version of the NFS protocol has been refined to increase platform independence, to improve performance over wide area networks such as the Internet, and to add strong, modular security features. Most implementations also include diagnostic utilities to help debug configuration and performance problems.

NFS is a network protocol, so in theory it could be implemented in user space just like most other network services. However, the traditional approach has been for parts of the NFS implementation (on both server and client sides) to live inside the kernel, mostly to improve performance. This general pattern continues even on Linux, where locking functions and certain system calls have proved difficult to export to user space. Fortunately, the kernel-resident parts of NFS need no configuration and are largely transparent to administrators.

NFS is not an off-the-shelf solution for all file sharing problems. High availability can only be achieved with warm standbys, but NFS has no built-in provisions for synchronizing with backup servers. The sudden disappearance of an NFS server from the network can result in clients holding stale file handles that can be cleaned up only with a reboot. Strong security is possible but is overly complex. Despite these drawbacks, NFS remains the most common choice for UNIX and Linux file sharing on a LAN.

Protocol versions and history

The first public release of the NFS protocol was version 2 in 1989. The original protocol made some expensive tradeoffs to favor consistency over performance and was quickly replaced. It's highly unlikely that you'll encounter this version in use today.

NFS version 3, which dates from the early 1990s, eliminates this bottleneck with a coherency scheme that permits asynchronous writes. It also updates several other aspects of the protocol that were found to have caused performance problems, and it improves the handling of large files. The net result is that NFS version 3 is quite a bit faster than version 2.

NFS version 4, dating from 2003 but not used widely until later in that decade, is a major overhaul that includes many new fixes and features. Highlighted enhancements include

- Compatibility and cooperation with firewalls and NAT devices
- Integration of the lock and mount protocols into the core NFS protocol
- Stateful operation
- Strong, modular security
- Support for replication and migration
- Support for both UNIX and Windows clients
- Access control lists (ACLs)
- Support for Unicode filenames
- Good performance even on low-bandwidth connections

The various NFS protocol versions cannot talk to one another, but NFS servers (including those on all our example systems) typically implement all three versions. In practice, all combinations of NFS clients and servers can interoperate with some version of the protocol. Always use the V4 protocol if both sides support it.

NFS remains actively developed and in widespread use. Version 4.2, written by some of the original stakeholders from Sun's heyday, reached RFC draft status in early 2015. The Elastic File System service from AWS, which became generally available in mid-2016, adds NFSv4.1 filesystems for use by EC2 instances.

Although V4 is a significant step forward in many ways, it hasn't much altered the process of configuring and administering NFS. In some ways this is a feature; for example, you still use the same configuration files and commands to administer all versions of NFS. In other ways it's a problem; some aspects of the configuration process feel jury-rigged (particularly on FreeBSD),

and some options have become ambiguous or overloaded, with different meanings or configuration formats depending on which version of NFS you are using.

Remote procedure calls

When Sun developed the first versions of NFS in the 1980s, they realized that many of the network-related problems that needed solving for NFS would apply to other network-based services, too. They developed a more general framework for remote procedure calls known as RPC or SunRPC, and built NFS on top of that. This work opened the door for applications of all kinds to run procedures on remote systems as if they were being run locally.

Sun's RPC system was primitive and somewhat hackish; far better systems exist today to fill this need (as do infinitely more horrifying monstrosities than SunRPC; check out SOAP). Nevertheless, NFS still relies on Sun-style RPCs for much of its functionality. Operations that read and write files, mount filesystems, access file metadata, and check file permissions are all implemented as RPCs. The NFS protocol specification is written generically, so a distinct RPC layer is not technically required. However, we are aware of no NFS implementations that stray from the original architecture in this regard.

Transport protocols

NFS version 2 originally used UDP because that was what performed best on the LANs and computers of the 1980s. Although NFS does its own packet sequence reassembly and error checking, UDP and NFS both lack the congestion control algorithms that are essential for good performance on a large IP network.

To remedy these problems (and others), NFS migrated to a choice of UDP or TCP in version 3, and to TCP only in version 4. The TCP option was first explored as a way to help NFS work through routers and over the Internet. Over time, most of the original reasons for preferring UDP over TCP have evaporated in the warm light of fast CPUs, cheap memory, and high-speed networks.

State

A client must explicitly mount an NFS filesystem before using it, just as a client must mount a filesystem stored on a local disk. However, NFS versions 2 and 3 are stateless, and the server does not keep track of which clients have mounted each filesystem. Instead, the server simply discloses a secret “cookie” at the conclusion of a successful mount negotiation. The cookie identifies the mounted directory to the NFS server and so opens a way for the client to access its contents. Cookies persist between reboots of the server, so a crash does not leave the client in an unrecoverable muddle. The client can simply wait until the server is available again and resubmit the request.

NFSv4, on the other hand, is a stateful protocol: both client and server maintain information about open files and locks. When the server fails, the clients assist in the recovery process by sending the server their pre-crash state information. A returning server waits for a predefined grace period for former clients to report their state information before it permits new operations and locks. The cookie management of V2 and V3 no longer exists in NFSv4.

Filesystem exports

---

*See this page for more information about the **fstab** file.*

NFS servers maintain a list of directories (called “exports” or “shares”) that they make available to clients over the network. By definition, all servers export at least one directory. Clients can then mount these exports and add them to their **fstab** files.

In V2 and V3, each export is treated as an independent entity that is exported separately. In the V4 specification, a server exports a single hierarchical pseudo-filesystem that incorporates all its exported directories. Essentially, the pseudo-filesystem is the server’s own filesystem namespace skeletonized to remove anything that is not exported.

For example, consider the following list of directories, with the directories to be exported in boldface:

/www/domain1
/www/domain2
/www/domain3
/var/logs/httpd
/var/spool

In NFS version 3, each exported directory must be separately configured. Client systems must execute three different mount requests to obtain access to all the server’s exports.

In NFS version 4, however, the pseudo-filesystem bridges the disconnected portions of the directory structure to create a single view for NFS clients. Rather than requesting a separate mount for each of **/www/domain1**, **/www/domain2**, and **/var/logs/httpd**, the client can simply mount the server’s entire pseudo-root directory and browse the hierarchy.

The directories that are not exported, **/www/domain3** and **/var/spool**, do not appear during browsing. In addition, individual files contained in **/**, **/var**, **/www**, and **/var/logs** are not visible to the client because the pseudo-filesystem portion of the hierarchy includes only directories. Thus, the client view of the NFSv4-exported filesystem is

```
/
├── var
│   └── logs
│       └── httpd
└── www
    ├── domain1
    └── domain2
```

The server specifies the root of the exported filesystems in a configuration file known as the **exports** file, usually kept in **/etc**. Pure NFSv4 clients cannot peruse the list of mounts on a

remote server. Instead, they simply mount the pseudo-root and then all available exports become accessible through that mount point.

That’s the story according to the RFC specification. In practice, the situation is somewhat fuzzy. The Solaris implementation conformed to this specification. Linux made a halfhearted attempt at support for the pseudo-filesystem in the early NFSv4 code, but later revised it to support the scheme more fully; today’s version appears to respect the intent of the RFC. FreeBSD does not implement the pseudo-filesystem as described by the RFC. The FreeBSD export semantics are essentially the same as in version 3; all subdirectories within an export are available to clients.

File locking

File locking (as implemented by the **flock**, **lockf**, or **fcntl** systems calls) has been a sore point on UNIX systems for a long time. On local filesystems, it has been known to work less than perfectly. In the context of NFS, the ground is shakier still. By design, early versions of NFS servers are stateless: they have no idea which machines are using any given file. However, to implement locking, state information is needed. What to do?

The traditional answer was to implement file locking separately from NFS. In most systems, two distinct daemons, **lockd** and **statd,** attempted to make a go of it. Unfortunately, the task was difficult for a variety of subtle reasons, and NFS file locking under **lockd** and **statd** is generally unreliable.

NFSv4 removed the need for **lockd** and **statd** by folding locking (and hence, statefulness and all that it implies) into the core protocol. This change introduces significant complexity but obviates many of the related problems of earlier NFS versions. Unfortunately, separate **lockd**s and **statd**s are still needed to support V2 and V3 clients if your site has them. Our example systems all ship with the earlier versions of NFS enabled, so the separate daemons still run by default.

Security concerns

In many ways, NFS V2 and V3 are poster children for everything that is or ever has been wrong with UNIX and Linux security. The protocol was originally designed with essentially no concern for security, and convenience has its price. NFSv4 has addressed the security concerns of earlier versions by mandating support for strong security services and establishing better means of user identification.

All versions of the NFS protocol are intended to be security-mechanism independent, and most servers support multiple "flavors" of security. A few of the common flavors include

- AUTH_NONE – no authentication
- AUTH_SYS – UNIX-style user and group access control
- RPCSEC_GSS – a stronger flavor that enables flexible security schemes

Historically, most sites used AUTH_SYS authentication, which depends on UNIX user and group identifiers. In this scheme, the client simply sends the local UID and GID of the user requesting access to the server. The server compares the values to those from its own **/etc/passwd** file (or a network database equivalent such as NIS or LDAP) and determines whether the user should have access. Thus, if users mary and bob share the same UID on two different clients, they will have access to each other's files. Furthermore, users that have root access on a system can **su** to whatever UID they wish; the server will then give them access to the corresponding user's files.

Enforcing **passwd** file consistency among systems is essential in environments that use AUTH_SYS. But even this is only a security fig leaf; any rogue host (or heaven forbid, Windows machine) can "authenticate" its users however it likes and thereby subvert NFS security.

See this page for more information about Kerberos.

To prevent such problems, most sites can use a more robust security mechanism such as Kerberos in combination with the NFS RPCSEC_GSS layer. This configuration requires both the client and server to participate in a Kerberos realm. The Kerberos realm authenticates clients centrally, avoiding the problems of self-identification described above. Kerberos can also provide strong encryption and guaranteed integrity for files transferred over the network. All protocol-conformant NFS version 4 systems must implement RPCSEC_GSS, but it's optional in version 3.

See this page for more information about firewalls.

NFS version 4 requires TCP as a transport protocol and communicates over port 2049. Since V4 does not rely on any other ports, opening access through a firewall is as simple as opening TCP port 2049. As with all access list configurations, be sure to specify source and destination addresses in addition to the port. If your site doesn't need to provide NFS services to hosts on the Internet, block access through the firewall or use a local packet filter.

File service over wide area networks with NFSv2 and V3 is not recommended because of the long history of bugs in the RPC protocols and the lack of strong security mechanisms. Administrators of NFS version 3 servers should block access to TCP and UDP ports 2049 and also the **portmap** port, 111.

Given the myriad and obvious shortcomings of AUTH_SYS security, we strongly recommend discontinuing all use of NFSv3. If you have ancient operating systems that can't be updated to NFSv4 compatibility, at least implement packet filters to restrict network connectivity.

Identity mapping in version 4

Before launching into the following discussion, we should warn you that we consider all implementations of AUTH_SYS security to be more or less broken for security purposes. We strongly suggest Kerberos and RPCSEC_GSS authentication; it's the only reasonable choice.

As discussed in Chapter 8, *User Management*, UNIX operating systems identify users through a collection of UIDs and GIDs in the local **passwd** file or an LDAP directory. NFS version 4, on the other hand, represents users and groups as string identifiers of the form *user@nfs-domain* and *group@nfs-domain*. NFSv4 clients and servers run an identity mapping daemon that translates UNIX identifier values to strings that match this format.

When a V4 client performs operations that return identities, such as listing the owners of a set of files with **ls -l** (the underlying operation is a series of **stat** calls), the server's identity mapping daemon uses its local **passwd** file to convert the UID and GID of each file object to a string, such as ben@admin.com. The client's identity mapper then reverses the process, converting ben@admin.com into local UID and GID values, which may or may not be the same as the server's. If a string value does not match any local identity, the nobody user account is assigned as a placeholder.

At this point, the remote filesystem call (**stat**) has completed and returned UID and GID values to its caller (here, the **ls** command). Since **ls** was called with the **-l** option, it needs to display text names instead of numbers. So, **ls** in turn retranslates the IDs back to textual names using the **getpwuid** and **getgrgid** library routines. These routines once again consult the **passwd** file or its network database equivalent. What a long, strange trip it's been.

Confusingly, the identity mapper is used only when retrieving and setting file attributes, typically ownerships. *Identity mapping plays no role in authentication or access control*, all of which is handled in the traditional form by RPC. The identity mapper may do a better job of mapping than the underlying NFS protocol, causing the apparent file permissions to conflict with the permissions the NFS server will actually enforce.

Consider, for example, the following commands on an NFSv4 client:

```
[ben@nfs-client]$ id ben
uid=1000(ben) gid=1000(ben) groups=1000(ben)

[ben@nfs-client]$ id john
uid=1010(john) gid=1010(john) groups=1010(john)

[ben@nfs-client]$ ls -ld ben
drwxr-xr-x   2   john   root     4096  May 27 16:42        ben

[ben@nfs-client]$ touch ben/file
[ben@nfs-client]$ ls -l ben/file
-rw-rw-r-- 1   john   nfsnobody  0   May 27 17:07        ben/file
```

First, ben is shown to have UID 1000 and john to have UID 1010. An NFS-exported home directory called **ben** appears to have permissions 755 and is owned by john. However, ben is able to create a file in the directory even though the **ls -l** output indicates that he lacks write permission.

On the server, john has UID 1000. Since john has UID 1010 on the client, the identity mapper performs UID conversion as described above, with the result that "john" appears to be the owner of the directory. However, the identity mapping daemon plays no role in access control. For the file creation operation, ben's UID of 1000 is sent directly to the server, where it is interpreted as john's UID and permission is granted.

How do you know which operations are identity mapped and which are not? It's simple: whenever a UID or GID appears *in the filesystem API* (as with **stat** or **chown**), it is mapped. Whenever the user's own UIDs or GIDs are used *implicitly* for access control, they are routed through the designated authentication system.

For this reason, enforcing consistent **passwd** files or relying on LDAP is essential for users of AUTH_SYS "security."

Unfortunately for administrators, identity mapping daemons are not standardized across systems, so their configuration processes may be different. Specifics for our example systems are covered here.

Root access and the nobody account

Although users should generally be given identical privileges wherever they go, it's traditional to prevent root from running rampant on NFS-mounted filesystems. By default, the NFS server intercepts incoming requests made on behalf of UID 0 and changes them to look as if they came from some other user. This modification is called "squashing root." The root account is not entirely shut out, but it is limited to the abilities of a normal user.

A placeholder account named "nobody" is defined specifically to be the pseudo-user as whom a remote root masquerades on an NFS server. The traditional UID for nobody is 65,534 (the 16-bit twos-complement equivalent of UID -2).

Although the Red Hat NFS server defaults to UID -2, the nobody account in the **passwd** file uses UID 99. You can leave things as they are, add a **passwd** entry for UID -2, or change anonuid and anongid to 99 if you wish. It really doesn't matter. Some systems also have an nfsnobody account.

You can change the default UID and GID mappings for root in the **exports** file. Some systems have an all_squash option to map all client UIDs to the same pseudo-user UID on the server. This configuration eliminates all distinctions among users and creates a sort of public-access filesystem.

The intent behind these precautions is nice, but their ultimate value is not as great as it might seem. Remember that root on an NFS client can **su** to whatever UID it wants, so user files are never really protected. The only real effect of root squashing is to prevent access to files that are owned by root and not readable or writable by the world.

Performance considerations in version 4

NFSv4 was designed to achieve good performance over wide area networks. Most WANs have higher latency and lower bandwidth than those of LANs. NFS takes aim at these problems with the following refinements:

- An RPC called COMPOUND clumps multiple file operations into one request, reducing the overhead and latency incurred from multiple remote procedure calls.
- A delegation mechanism allows client-side caching of files. Clients can maintain local control over files, including those open for writing.

These features are part of the core NFS protocol and do not require much attention from system administrators.

21.3 SERVER-SIDE NFS

An NFS server is said to “export” a directory when it makes the directory available for use by other machines. Exports are presented to NFSv4 clients as a single filesystem hierarchy through the pseudo-filesystem.

In NFS version 3, the process used by clients to mount a filesystem is separate from the process used to access files. The operations use separate protocols, and the requests are served by different daemons: **mountd** for mount discovery and requests, and **nfsd** for actual file service. On some systems, these daemons are called **rpc.nfsd** and **rpc.mountd** as a reminder that they rely on RPC as an underlying mechanism (and hence require the **portmap** daemon to be running). In this chapter, we omit the **rpc** prefix for readability.

NFSv4 does not use **mountd** at all. If you absolutely must run old clients that only support NFSv3, **mountd** must remain active.

Both **mountd** and **nfsd** should start when the system boots, and both should remain running as long as the system is up. Both Linux and FreeBSD automatically run the daemons when you enable NFS service.

NFS uses a single access-control database that tells which filesystems should be exported and which clients can mount them. The operative copy of this database is usually kept in a file called **xtab** and also in tables internal to the kernel. **xtab** is a binary file maintained for use by the server daemon.

Maintaining a binary file by hand is not much fun, so most systems assume that you would rather maintain a text file, usually **/etc/exports**, that enumerates the system’s exported directories and their access settings. The system can then consult this text file at boot time to automatically construct the **xtab** file.

/etc/exports is the canonical, human-readable list of exported directories. Its contents are read by **exportfs -a** on Linux, and at a simple restart of the NFS server on FreeBSD. Hence, when you edit **/etc/exports**, run **exportfs -a** to activate your changes on Linux, or run **service nfsd restart** on FreeBSD. If you serve V3 clients from FreeBSD, restart **mountd** as well (**service mountd reload**).

NFS deals with the logical layer of the filesystem. Any directory can be exported; it doesn’t have to be a mount point or the root of a physical filesystem. However, for security, NFS does pay attention to the boundaries between filesystems and does require each device to be exported separately. For example, on a machine that has set up **/chimchim/users** as a separate partition, you could export **/chimchim** without implicitly exporting **/chimchim/users**.

Clients are usually allowed to mount subdirectories of an exported directory if they wish, although the protocol does not require this feature. For example, if a server exports

/chimchim/users, a client could mount only **/chimchim/users/joe** and ignore the rest of the **users** directory.

Linux exports



On Linux, the **exports** file consists of a list of exported directories in the leftmost column followed by the hosts that are allowed to access them and associated options on the right. Whitespace separates the filesystem from the list of clients, and each client is followed immediately by a parenthesized list of comma-separated options. Lines can be continued with a backslash. For example, the line

```
/home   *.users.admin.com(rw) 172.17.0.0/24(ro)
```

lets **/home** be mounted read/write by all machines in the users.admin.com domain, and read-only by all machines on the 172.17.0.0/24 class C network. If a system in the users.admin.com domain resides on the 172.17.0.0/24 network, that client will be granted read-only access. The least privileged rule wins.

Filesystems listed in the **exports** file without a specific set of hosts are usually mountable by *all* machines. That's a sizable security hole.

The same sizable security hole can be created accidentally by a misplaced space. For example, the line

```
/home   *.users.admin.com (rw)
```

permits any host read/write access *except* for *.users.admin.com, which has read-only permission, the default. Oops.

There is unfortunately no way to list multiple client specifications for a single set of options. You must repeat the options for all desired clients. Table 21.1 lists the types of client specifications that can appear in the **exports** file.

Table 21.1: Client specifications in the Linux /etc/exports file

Type	Syntax	Meaning
Hostname	*hostname*	Individual hosts
Netgroup	*@groupname*	NIS netgroups (infrequently used)
Wild cards	* and ?	FQDNs[a] with wild cards; * will not match a dot
IPv4 networks	*ipaddr/mask*	CIDR-style specifications (e.g., 128.138.92.128/25)
IPv6 networks	*ipaddr/mask*	IPv6 addresses with CIDR notation (2001:db8::/32)

a. Fully qualified domain names

Table 21.2 shows the most commonly used export options understood by Linux.

Table 21.2: Common export options in Linux

Option	Description
ro	Exports read-only
rw	Exports for reading and writing (the default)
rw=*list*	Exports read-mostly. The *list* enumerates the hosts allowed to mount for writing; all others must mount read-only.
root_squash	Maps ("squashes") UID 0 and GID 0 to the values specified by anonuid and anongid. This is the default.
no_root_squash	Allows normal access by root. Dangerous.
all_squash	Maps all UIDs and GIDs to their anonymous versions [a]
anonuid=*xxx*	Specifies the UID to which remote roots should be squashed
anongid=*xxx*	Specifies the GID to which remote roots should be squashed
noaccess	Blocks access to this dir and subdirs (use with nested exports)
wdelay	Delays writes in hopes of coalescing multiple updates
no_wdelay	Writes data to disk as soon as possible
async	Makes server reply to write requests before actual disk write
nohide	Reveals filesystems mounted within exported file trees
hide	Is the opposite of nohide
subtree_check	Verifies that each requested file is within an exported subtree
no_subtree_check	Verifies only that file requests refer to an exported filesystem
secure_locks	Requires authorization for all lock requests
insecure_locks	Specifies less stringent locking criteria (supports older clients)
sec=*flavor*	Lists security methods for the exported directory [b]
pnfs	Enables V4.1 parallel NFS extensions for direct client access
replicas=*path@host*	Sends clients a list of alternative locations for this export

a. This option is useful for supporting PCs and other untrusted single-user hosts.
b. Values include sys (UNIX authentication, the default), dh (DES, not recommended), krb5 (Kerberos authentication), krb5i (Kerberos authentication and integrity), krb5p (Kerberos authentication, integrity, and privacy), and none (anonymous access, not recommended).

The subtree_check option (the default) verifies that every file accessed by a client lies within an exported subdirectory. If you turn off this option, only the fact that the file is within an exported filesystem is verified. Subtree checking can cause occasional problems when a requested file is renamed while the client has the file open. If you anticipate many such situations, consider setting no_subtree_check.

async tells the NFS server to ignore the protocol spec and reply to requests before they are written to disk. This might result in a slight performance boost, but might also result in corrupted data if the server restarts unexpectedly. The default behavior is sync.

The replicas option is merely a convenience that helps clients discover mirrors if the server goes off-line. The actual replication of the filesystem must be handled out of band through some other mechanism such as **rsync** or DRBD (replication software for Linux). The replica referral feature was added in NFSv4.1.

Early versions of the Linux NFSv4 implementation required administrators to designate a pseudo-filesystem root with the fsid=0 flag in **/etc/exports**. This is no longer required. To create a pseudo-fileystem as described by the RFC, just list exports as normal and, from an NFSv4 client, mount / on the server. The subdirectories under the mount point will be the exported filesystems. If you do designate an export as fsid=0, that filesystem and all its subdirectories are exported for V4 clients.

FreeBSD exports



In keeping with longstanding UNIX tradition, the **exports** format used on FreeBSD is entirely different from that of Linux. Each line in the file (except for lines that start with #, which are comments) is composed of three components: a list of directories to export, the options to apply to those exports, and the set of hosts to which the export applies. As on Linux, a backslash denotes a line continuation.

```
/var/www -ro,alldirs www*.admin.com
```

The line above exports **/var/www** and all of its subdirectories read-only to all hosts matching the pattern www*.admin.com. To implement different mount options for different clients, simply repeat the line and specify different values. For example,

```
/var/www -alldirs,sec=krb5p -network 2001:db8::/32
```

allows read/write access for all hosts in the named IPv6 network. Kerberos is used for authentication, integrity, and privacy.

On FreeBSD, exports are per server-filesystem. Multiple exports to the same set of client hosts from the same filesystem must be named on the same line. For example,

```
/var/www1 /var/www2 -ro,alldirs www*.admin.com
```

It would be an error for www1 and www2 to be on separate lines with the same host designations, assuming that www1 and www2 reside within the same filesystem.

To enable NFSv4 you must designate a root by prefixing a line with `V4:`, for example,

```
V4: /exports -sec=krb5p,krb5i,krb5,sys -network *.admin.com
```

Only one effective V4 root path is allowed. However, it can be specified more than once with different options for different clients. The root can appear anywhere in the **exports** file.

The `V4:` line does not actually export any filesystems. It simply chooses a base directory for NFSv4 clients to mount. To activate it, list an export within the root:

```
/exports/www -network *.admin.com
```

Despite the V4 root designation, the FreeBSD NFS server does not implement the pseudo-filesystem as described by the RFC. When a V4 root is designated and at least one export is present under that root, a V4 client can mount the root and access all of the files and directories within it regardless of their export status. This information is not clear in the **exports**(5) documentation, and the ambiguity can be quite dangerous. Do not designate the server's own filesystem root (/) as the V4 root; otherwise, the server's entire root filesystem will be available to clients.

Because of the V4 root, V2 and V3 clients have a different path to mount than V4 clients have. For example, given the following exports

```
/exports/www -network 10.0.0.0 -mask 255.255.255.0
V4: /exports -network 10.0.0.0 -mask 255.255.255.0
```

a V2 or V3 client in the 10.0.0.0/24 network mounts **/exports/www**, but because of the pseudo-filesystem designation on **/exports**, a V4 client must mount the export as **/www**. Alternatively, a V4 client can mount / and access the **www** directory under that mount point.

Use network ranges for best performance when exporting to a large number of clients. For IPv4 you can use CIDR notation or a subnet mask. For IPv6 you *must* use CIDR; the `-mask` option is not permitted. For example:

```
/var/www -network 10.0.0.0 -mask 255.255.255.0
/var/www -network 10.0.0.0/24
/var/www -network 2001:db8::/32
```

FreeBSD has fewer export options than Linux affords. Table 21.3 summarizes them.

Table 21.3: Common export options in FreeBSD

Option	Description
`alldirs`	Allows mounts at any point in the filesystem
`ro`	Exports read-only (read/write is the default)
`o`	Synonym for `ro`; exports read-only
`maproot=`*xxx*	The username or UID to map for access from a remote root user
`mapall=`*xxx*	Maps all client users to the specified user (like `maproot`)
`sec=`*flavor*	Specifies allowable security methods[a]

a. Specify multiple flavors in a comma-separated list, in order of preference. Possible values are `sys` (UNIX authentication, the default), `krb5` (Kerberos authentication), `krb5i` (Kerberos authentication and integrity), `krb5p` (Kerberos authentication, integrity, and privacy), and `none` (anonymous access, not recommended).

nfsd: serve files

Once a client's mount request has been validated, the client can request various filesystem operations. These requests are handled on the server side by **nfsd**, the NFS operations daemon. (In reality, **nfsd** simply makes a nonreturning system call to NFS server code embedded in the kernel.) **nfsd** does not need to run on an NFS client machine unless the client exports filesystems of its own.

nfsd has no configuration file; options are passed as command-line arguments. You start and stop **nfsd** with your system's standard service mechanisms, i.e., **systemctl** on Linux systems running **systemd**, and the **service** command on FreeBSD. Table 21.4 shows which file and option to adjust in order to change the arguments passed to **nfsd**.

Table 21.4: Where to set startup options for nfsd

System	Config file	Option to set
Ubuntu	**/etc/default/nfs-kernel-server**	RPCNFSDOPTS[a]
Red Hat	**/etc/sysconfig/nfs**	RPCNFSDARGS
FreeBSD	**/etc/rc.conf**	nfs_server_flags

a. Some versions of the **nfs-kernel-server** package incorrectly suggest that you edit RPCMOUNTDOPTS to set some **nfsd** options. Do not be fooled.

On Linux systems, run **systemctl restart nfs-config.service nfs-server.service** to enable **nfsd** configuration changes. In FreeBSD, use **service nfsd restart** and **service mountd restart.**

The -N option to **nfsd** disables the specified version of NFS. For example, to disable versions 2 and 3, add -N 2 -N 3 to the appropriate file and option specified in Table 21.4 and restart the service. This is a good idea if you are sure you don't need to support older clients.

nfsd takes a numeric argument that specifies how many server threads to fork. Selecting the appropriate number of **nfsd**s is important and is unfortunately something of a black art. If the number is too low or too high, NFS performance can suffer.

The optimal number of **nfsd** threads depends on the operating system and the hardware in use. If you notice that **ps** usually shows the **nfsd**s in state D (uninterruptible sleep) and that some idle CPU is available, consider increasing the number of threads. If you find the load average (as reported by **uptime**) rising as you add **nfsd**s, you've gone too far; back off a bit from that threshold.

Run **nfsstat** regularly to check for performance problems that might be associated with the number of **nfsd** threads. See this page for more details on **nfsstat**.



On FreeBSD, the **--minthreads** and **--maxthreads** options to **nfsd** enable automatic management of the number of threads within the specified bounds. On FreeBSD, see **man**

rc.conf and refer to the options prefixed with nfs_ for more NFS server settings.

21.4 Client-side NFS

NFS filesystems are mounted in much the same way as local disk filesystems. The **mount** command understands the notation *hostname*:*directory* to mean the path *directory* on the host *hostname*. As with local filesystems, **mount** maps the remote *directory* on the remote *host* into a directory within the local file tree. After the mount completes, you access an NFS-mounted filesystem just like a local filesystem. The **mount** command and its associated NFS extensions represent the most significant concerns to the system administrator of an NFS client.

Before an NFS filesystem can be mounted, it must be properly exported (see this page). On an NFSv3 client, you can verify that a server has properly exported its filesystems by running the **showmount** command:

```
$ showmount -e monk
Export list for monk:
/home/ben harp.atrust.com
```

This example reports that the directory **/home/ben** on the server monk has been exported to the client system harp.atrust.com.

If an NFS mount is not working, first verify that the filesystems have been properly exported on the server. Make sure that after updating the server's **exports** file, you ran **exportfs -a** (Linux) or **service nfsd restart** and **service mountd reload** (FreeBSD). Next, recheck the **showmount** output.

If the directory is properly exported on the server but **showmount** returns an error or an empty list, double-check that all the necessary processes are running on the server (**portmap** and **nfsd**; add **mountd**, **statd**, and **lockd** for V3). Make sure the **hosts.allow** and **hosts.deny** files allow access to those daemons and that you are on the right client system.

The path information displayed by **showmount** (e.g., **/home/ben** above) is valid only for NFS version 2 and 3 servers. NFS version 4 servers export a single unified pseudo-filesystem and do not use the mount protocol. The traditional NFS concept of separate mount points doesn't jibe with version 4's model, so **showmount** simply doesn't apply to the V4 world.

Unfortunately, NFSv4 has no good replacement for **showmount**. On the server, the command **exportfs -v** shows the existing exports, but of course this works only locally. If you don't have direct access to the server, you can try to mount the server's V4 root and traverse the directory structure manually. You can also mount any subdirectory of the exported root filesystem.

To actually mount the filesystem in versions 2 and 3, you'd use a command such as

```
$ sudo mount -o rw,hard,intr,bg server:/home/ben /nfs/ben
```

To accomplish the same under version 4 on a Linux system, you'd type

```
$ sudo mount -o rw,hard,intr,bg server:/ /nfs/ben
```

In this case, the options after **-o** specify that the filesystem be mounted read/write (**rw**), that operations be interruptible (**intr)**, and that retries be done in the background (**bg**). Table 21.5 introduces the most common Linux mount options.

Table 21.5: NFS mount flags and options for Linux

Flag	Description
`rw`	Mounts the filesystem read/write (must be exported that way)
`ro`	Mounts the filesystem read-only
`bg`	If the mount fails (server doesn't respond), keeps trying it in the background and continues with other mount requests
`hard`	If a server goes down, makes operations that access it block until the server comes back up
`soft`	If a server goes down, makes operations that access it fail and return an error, to avoid processes hanging on inessential mounts
`intr`	Allows users to interrupt blocked operations (they return an error)
`nointr`	Does not allow user interrupts
`retrans=`*n*	Specifies the number of times to repeat a request before returning an error on a soft-mounted filesystem
`timeo=`*n*	Sets the timeout period (in tenths of a second) for requests
`rsize=`*n*	Sets the read buffer size to *n* bytes
`wsize=`*n*	Sets the write buffer size to *n* bytes
`sec=`*flavor*	Specifies the security flavor
`nfsvers=`*n*	Sets the NFS protocol version
`proto=`*proto*	Selects a transport protocol; must be `tcp` for NFS version 4

The client side of NFS usually tries to autonegotiate a suitable version of the protocol. You can specify a specific version by passing **-o nfsvers**=*n*.

On FreeBSD, **mount** is a wrapper that calls **/sbin/mount_nfs** for NFS mounts. This wrapper sets NFS options and invokes the **nmount** system call. To mount a version 4 server on FreeBSD, type:

```
$ sudo mount -t nfs -o nfsv4 server:/ /mnt
```

If you don't specify a version explicitly, **mount** negotiates one automatically in descending order. In fact, a simple **mount server:/ /mnt** does the trick in this case because **mount** can infer from the format that the filesystem you're referring to is NFS.

Filesystems mounted `hard` (the default) cause processes to hang when their servers go down. This behavior is particularly bothersome when the processes in question are standard daemons, so we

do not recommend serving critical system binaries over NFS. In general, the intr option reduces the number of NFS-related headaches.

Jeff Forys, one of our technical reviewers, advises, “Most mounts should use hard, intr, and bg, because these options best preserve NFS’s original design goals. soft is an abomination, an ugly Satanic hack! If the user wants to interrupt, cool. Otherwise, wait for the server and all will eventually be well again with no data lost.”

Automount solutions such as autofs, discussed starting here, also prescribe some remedies for mounting ailments.

The read and write buffer sizes are negotiated to the highest value supported by both client and server. You can set them to any value between 1KiB and 1MiB.

You can see the available space on an NFS mount with **df**, just as you would on a local filesystem:

```
$ df /nfs/ben
Filesystem         1k-blocks    Used  Available  Use%  Mounted on
leopard:/home/ben   17212156 1694128   14643692   11%  /nfs/ben
```

umount works on NFS filesystems just like it does on local filesystems. If the NFS filesystem is in use when you try to unmount it, you get an error such as

```
umount: /nfs/ben: device is busy
```

Use **fuser** or **lsof** to find processes with open files on the filesystem. Kill them, or in the case of shells, change directories. If all else fails or your server is down, try running **umount -f** to force the filesystem to be unmounted.

Mounting remote filesystems at boot time

---

*See this page for more information about the **fstab** file.*

You can use the **mount** command to establish temporary network mounts, but you should list mounts that are part of a system's permanent configuration in **/etc/fstab** so that they are mounted automatically at boot time. Alternatively, mounts can be handled by an automatic mounting service such as autofs.

The following **fstab** entries mount the **/home** filesystem from the server monk:

```
# filesystem mountpoint  fstype flags                        dump  fsck
monk:/home   /nfs/home   nfs    rw,bg,intr,hard,nodev,nosuid  0     0
```

You can make your changes take effect immediately (without rebooting) by running **mount -a -t nfs**.

The flags field of **/etc/fstab** specifies options for NFS mounts; these options are the same ones you would specify on the **mount** command line.

Restricting exports to privileged ports

NFS clients are free to use any TCP or UDP source port they like when connecting to an NFS server. However, some servers may insist that requests come from a privileged port (a port numbered lower than 1,024). Others allow this behavior to be set as an option. The use of privileged ports delivers little actual security.

Nevertheless, most NFS clients adopt the traditional (and still recommended) approach of defaulting to a privileged port to avert the potential for conflict. Under Linux, you can accept mounts from unprivileged ports with the `insecure` export option.

21.5 Identity mapping for NFS version 4

We introduced the general ideas behind NFSv4's identity mapping system starting here. In this section we discuss the administrative aspects of the identity mapping daemon.

All systems that participate in an NFSv4 network should have the same NFS domain. In most cases, it's reasonable to use your DNS domain as the NFS domain. For example, admin.com is a straightforward choice of NFS domain for the server ulsah.admin.com. Clients in subdomains (e.g., books.admin.com) may or may not want to use the same domain name (e.g., admin.com) to facilitate NFS communication.

Unfortunately for administrators, NFSv4 UID mapping has no standard implementation, so the details of administration differ slightly among systems. Table 21.6 names the mapping daemons on Linux and FreeBSD and notes the location of their configuration files.

Table 21.6: NFSv4 identity mapping daemons and their configuration files

System	Daemon	Configuration file	man page
Linux	**/usr/sbin/rpc.idmapd**	**/etc/idmapd.conf**	**nfsidmap**(5)
FreeBSD	**/usr/sbin/nfsuserd**	`nfsuserd_flags` in **/etc/rc.conf**	**idmap**(8)

Other than having their NFS domains set, identity mapping services require little assistance from administrators. The daemons are started at boot time by the same scripts that manage other NFS daemons. After making configuration changes, you'll need to restart the identity mapper daemon. Options such as verbose logging and alternative management of the nobody account are usually available.

21.6 NFSSTAT: DUMP NFS STATISTICS

nfsstat displays various statistics maintained by the NFS system. **nfsstat -s** shows server-side statistics, and **nfsstat -c** shows information for client-side operations. By default, **nfsstat** shows statistics for all protocol versions. For example:

```
$ nfsstat -c

Client rpc:
  calls  badcalls  retrans  badxid  timeout  wait  newcred   timers
  64235      1595        0       3     1592     0        0      886

Client nfs:
  calls  badcalls   nclget  nclsleep
  62613         3    62643         0
  null    getattr  setattr  readlink  lookup  root     read
    0%        34%       0%       21%     30%    0%       2%
  write   wrcache   create    remove  rename  link  symlink
     3%        0%       0%        0%      0%    0%       0%
  mkdir   readdir    rmdir    fsstat
     0%        6%       0%        0%
```

This example is from a relatively healthy NFS client. If more than 3% of RPC calls time out, it's likely that your NFS server or network has a problem. You can usually discover the cause by checking the badxid field. If badxid is near 0 with timeouts greater than 3%, packets to and from the server are getting lost on the network. You might be able to solve this problem by lowering the rsize and wsize mount parameters (read and write block sizes).

If badxid is nearly as high as timeout, then the server is responding, but too slowly. Either replace the server or increase the timeo mount parameter.

Running **nfsstat** and **netstat** occasionally and becoming familiar with their output helps you discover NFS problems before your users do. We suggest including this data as part of your site's monitoring and alerting system.

21.7 Dedicated NFS file servers

Fast, reliable file service is an essential element of a production computing environment. Although you can certainly roll your own file servers from workstations and off-the-shelf hard disks, doing so is often not the best-performing or easiest-to-administer solution (though it is usually the cheapest).

Dedicated NFS file server products have been on the market for many years. They offer a host of potential advantages over the homebrew approach:

- They are optimized for file service and typically deliver the best possible NFS performance.
- As storage requirements grow, they can scale smoothly to support tera-bytes of storage and hundreds of users.
- They are more reliable than stand-alone boxes thanks to their simplified software, redundant hardware, and use of disk mirroring.
- They usually handle file service for both UNIX and Windows clients. Most even contain integrated HTTPS, FTP, and SFTP servers.
- They often include backup and checkpoint facilities that are superior to those found on vanilla UNIX systems.

Some of our favorite dedicated NFS servers are made by NetApp. Their products run the gamut from very small to very large, and their pricing is OK. EMC is another player in the high-end server market. They make good products, but be prepared for sticker shock and build up your tolerance for marketing buzzwords.

In an AWS environment, the Elastic File System service is a scalable NFSv4.1 server-as-a-service that exports filesystems to EC2 instances. Each filesystem can support multiple GiB/s throughput, depending upon the size of the filesystem. See aws.amazon.com/efs for more information.

21.8 AUTOMATIC MOUNTING

Mounting filesystems at boot time by listing them in **/etc/fstab** can cause administrative headaches on large networks. First, it's tedious to maintain the **fstab** file on hundreds of machines, even with help from scripts and configuration management systems. Each host may have slightly different needs and so require individual attention. Second, if shared filesystems are mounted from many different hosts, clients become dependent on many different downstream servers. Chaos ensues when one of those servers crashes. Every command that accesses that server's mount points will hang.

You can moderate these problems with an automounter, a type of daemon that mounts filesystems when they are referenced and unmounts them when they are no longer being used. In addition to deferring mounts until they are actually needed, most automounters can also accept a list of "replicas" (identical backup copies) for a filesystem. These backups let the network continue to function even when a primary server becomes unavailable.

As described by Edward Tomasz Napierała, author of the FreeBSD automounter, this magic requires the cooperation of several related pieces of software:

- autofs, a kernel-resident filesystem driver that watches a filesystem for mount requests, pauses the calling program, and invokes the automounter to mount the target filesystem before returning control to the caller
- **automountd** and **autounmountd**, which read the administrative configuration and actually mount or unmount filesystems
- **automount**, an administrative utility

For the most part, automounters are transparent to users. Instead of mirroring an actual filesystem, the automounter "makes up" a virtual filesystem hierarchy according to the specifications given in its configuration files. When a user references a directory within the automounter's virtual filesystem, the **automountd** intercepts the reference and mounts the actual filesystem the user is trying to reach. The NFS filesystem is mounted beneath the autofs filesystem in normal UNIX fashion.

The idea of an automounter originally comes from Sun. The Linux version functionally mimics that of Sun, although it is in fact an independent implementation. FreeBSD maintains yet another implementation, having sacrificed a once widely used automounter, **amd**, in the FreeBSD 10.1 release.

The various **automount** implementations understand three different kinds of configuration files, referred to as "maps": direct maps, indirect maps, and master maps. Direct and indirect maps contain information about the filesystems to be automounted. A master map lists the direct and indirect maps that **automount** should pay attention to. Only one master map can be active at

once; the default master map is kept in **/etc/auto_master** on FreeBSD and in **/etc/auto.master** on Linux.

A direct map can also be managed as an NIS database or in an LDAP directory, but doing so is tricky.

On most systems, **automount** is a stand-alone command that reads its configuration files, sets up any necessary autofs mounts, and exits. Actual references to automounted filesystems are handled (through autofs) by a separate daemon process, **automountd**. This daemon does its work silently and does not need additional configuration.


On Linux systems, the daemon is called **automount** instead of **automountd**, and the setup function is performed by a system startup script (**systemd** for modern distributions). Linux details are given here. In the following discussion, we refer to the setup command as **automount** and the daemon as **automountd**.

If you change the master map or one of the direct maps that it references, you must rerun **automount** to pick up the changes. With the **-v** option, **automount** shows you the adjustments it's making to its configuration. You can add **-L** to achieve a dry run effect that lets you examine your configuration and debug problems.

automount (**autounmountd** on FreeBSD) accepts a **-t** argument that tells how long (in seconds) an automounted filesystem can remain unused before being unmounted. The default is 300 seconds (10 minutes). Since an NFS mount whose server has crashed can cause programs that touch it to hang, it's good hygiene to clean up automounts that are no longer in use; don't raise the timeout too much. (The other side of this issue is the time required to mount a filesystem. System response is faster and smoother if filesystems aren't being continually remounted.)

Indirect maps

Indirect maps automount several filesystems under a common directory. However, the path of the directory is specified in the master map, not in the indirect map itself. For example, an indirect map might look like this:

```
users    harp:/harp/users
devel    -soft harp:/harp/devel
info     -ro harp:/harp/info
```

The first column names the subdirectory in which each automount should be installed, and subsequent items list the mount options and the NFS path of the filesystem. This example (perhaps stored in **/etc/auto.harp**) tells **automount** that it can mount the directories **/harp/users**, **/harp/devel**, and **/harp/info** from the server harp, with **info** being mounted read-only and **devel** being mounted soft.

In this configuration, the paths on harp and the local host are the same. However, this correspondence is not required.

Direct maps

Direct maps list filesystems that do not share a common prefix, such as **/usr/src** and **/cs/tools**. A direct map (e.g., **/etc/auto.direct**) that described both of these filesystems to **automount** might look something like this:

```
/usr/src     harp:/usr/src
/cs/tools    -ro monk:/cs/tools
```

Because they do not share a common parent directory, these automounts must each be implemented with a separate autofs mount. This configuration requires more overhead, but it has the added advantage that the mount point and directory structure are always accessible to commands such as **ls**. Running **ls** on a directory full of indirect mounts can be confusing to users because **automount** doesn't show the subdirectories until their contents have been accessed. (**ls** doesn't look inside the automounted directories, so it does not cause them to be mounted.)

Master maps

A master map lists the direct and indirect maps that **automount** should pay attention to. For each indirect map, it also specifies the root directory to be used by the mounts defined in the map.

A master map that referenced the direct and indirect maps shown in the previous examples would look something like this:

```
# Directory  Map
/harp        /etc/auto.harp -proto=tcp
/-           /etc/auto.direct
```

The first column is a local directory name for an indirect map or the special token `/-` for a direct map. The second column identifies the file in which the map is stored. You can have several maps of each type. When you specify mount options at the end of a line, they set the defaults for all mounts within the map. Linux administrators should always specify the `-fstype=nfs4` mount flag for NFS version 4 servers.



On most systems, the default options set on a master map entry do not blend with the options specified in the direct or indirect map to which it points. If a map entry has its own list of options, the defaults are ignored. Linux merges the two sets, however. If the same option is specified in both places, the map entry’s value overrides the default.

Executable maps

If a map file is executable, it's assumed to be a script or program that dynamically generates automounting information. Instead of reading the map as a text file, the automounter executes it with an argument (the "key") that indicates which subdirectory a user has attempted to access. The script prints an appropriate map entry; if the specified key is not valid, the script can simply exit without printing anything.

This powerful feature makes up for many of the deficiencies in **automounter**'s rather strange configuration system. In effect, you can easily define a site-wide automount configuration file in a format of your own choice. You can write a simple script to decode the global configuration on each machine. Some systems come with a handy **/etc/auto.net** executable map that takes a hostname as a key and mounts all exported filesystems on that host.

Since automount scripts run dynamically as needed, it's unnecessary to distribute the master configuration file after every change or to convert it preemptively to the **automounter** format; in fact, the global configuration file can have a permanent home on an NFS server.

Automount visibility

When you list the contents of an automounted filesystem's parent directory, the directory appears empty no matter how many filesystems have been automounted there. You cannot browse the automounts in a GUI filesystem browser.

An example:

```
$ ls /portal
$ ls /portal/photos
art_class_2010  florissant_1003     rmnp03
blizzard2008    frozen_dead_guy_Oct2009 rmnp_030806
boston021130    greenville.021129       steamboat2006
```

The **photos** filesystem is alive and well and is automounted under **/portal**. It's accessible through its full pathname. However, a review of the **/portal** directory does not reveal its existence. If you had mounted this filesystem through the **fstab** file or a manual **mount** command, it would behave like any other directory and would be visible as a member of the parent directory.

One way around the browsing problem is to create a shadow directory that contains symbolic links to automount points. For example, if **/automounts/photos** is a link to **/portal/photos**, you can **ls** the contents of **/automounts** to discover that **photos** is an automounted directory. References to **/automounts/photos** are still routed through the automounter and work correctly.

Unfortunately, these symbolic links require maintenance and can go out of sync with the actual automounts unless they are periodically reconstructed by a script.

Replicated filesystems and automount

In some cases, a read-only filesystem such as **/usr/share** might be identical on several different servers. In this case, you can tell **automount** about several potential sources for the filesystem. It then chooses a server according to its own idea of which servers are closest, given network routes, NFS protocol versions, and response times to an initial query.

Although **automount** itself does not see or care how the filesystems it mounts are used, replicated mounts should represent read-only filesystems such as **/usr/share** or **/usr/local/X11**. There's no way for **automount** to synchronize writes across a set of servers, so replicated read/write filesystems are of little practical use.

You can assign explicit priorities to determine which replica to select first. The priorities are small integers, with larger numbers indicating lower priority. The default priority is 0, most eligible.

An **auto.direct** file that defines **/usr/man** and **/cs/tools** as replicated filesystems might look like this:

```
/usr/man    -ro harp:/usr/share/man monk(1):/usr/man
/cs/tools   -ro leopard,monk:/cs/tools
```

Note that server names can be listed together if the source path on each is the same. The (1) after monk in the first line sets that server's priority with respect to **/usr/man**. The lack of a priority after harp indicates an implicit priority of 0.

Automatic automounts (V3; all but Linux)

Instead of listing every possible mount in a direct or indirect map, you can tell **automount** a little about your filesystem naming conventions and let it figure things out for itself. The key piece of glue that makes this work is that the **mountd** running on a remote server can be queried to find out what filesystems the server exports. In NFS version 4, the export is always /, which eliminates the need for this automation.

"Automatic automounts" can be configured in several ways, the simplest of which is the -hosts mount type on FreeBSD. If you list -hosts as a map name in your master map file, **automount** then maps remote hosts' exports into the specified automount directory:

```
/net    -hosts -nosuid,soft
```

For example, if harp exported **/usr/share/man**, that directory could then be reached through the automounter at the path **/net/harp/usr/share/man**.

The implementation of -hosts does not enumerate all possible hosts from which filesystems can be mounted; that would be impossible. Instead, it waits for individual subdirectory names to be referenced, then runs off and mounts the exported filesystems from the requested host.

A similar but finer-grained effect can be achieved with the * and & wild cards in an indirect map file. Also, a number of macros available for use in maps expand to the current hostname, architecture type, and so on. See the **automount**(1M) man page for details.

Specifics for Linux


The Linux implementation of **automount** has diverged a bit from the original Sun standards. The changes mostly relate to the naming of commands and files.

First, **automount** is the daemon that actually mounts and unmounts remote filesystems. It fills the same niche as the **automountd** daemon on other systems and generally does not need to be run by hand.

The default master map file is **/etc/auto.master**. Its format and the format of indirect maps are as described previously. The documentation can be hard to find, however. The master map format is described in **auto.master**(5) and the indirect map format in **autofs**(5); be careful, or you'll get **autofs**(8), which documents the syntax of the **autofs** command. (As one of the man pages says, "The documentation leaves a lot to be desired.") To cause changes to the master map to take effect, run the command **/etc/init.d/autofs reload**, which is equivalent to **automount** in Sunland.

The Linux implementation does not support the Solaris-style -hosts clause for automatic automounts.

21.9 Recommended reading

Table 21.7 lists the various RFCs for the NFS protocol.

Table 21.7: NFS-related RFCs

RFC	Title	Author	Date
1094	Network File System Protocol Specification	Sun Microsystems	Mar 1989
1813	NFS Version 3 Protocol Specification	B. Callaghan et al.	Jun 1995
2623	NFS Version 2 and Version 3 Security Issues	M. Eisler	Jun 1999
2624	NFS Version 4 Design Considerations	S. Shepler	Jun 1999
3530	NFS Version 4 Protocol	S. Shepler et al.	April 2003
5661	NFS Version 4 Minor Version 1 Protocol	S. Shepler et al.	Jan 2010
7862	NFS Version 4 Minor Version 2 Protocol	T. Haynes	Nov 2016

22 SMB



Chapter 21, *The Network File System*, covers the most popular system for sharing files among UNIX and Linux systems. However, UNIX systems also need to share files with systems, such as Windows, that don't natively support NFS. Enter SMB.

In the early 1980s, Barry Feigenbaum created the BAF protocol to afford shared network access to files and resources. Before release, the name was changed from the author's initials to Server Message Block (SMB). The protocol was rapidly embraced by Microsoft and the PC community because it gave "just like local" access to files on remote systems.

In 1996, a version called the Common Internet File System (CIFS) was released by Microsoft, mostly as a marketing exercise. Sun Microsystems had also entered the fray in 1996 with its WebNFS offering, and Microsoft saw an opportunity to market SMB with a more user-friendly implementation and name. CIFS introduced (often-buggy) changes to the original SMB protocol. As a result, Microsoft released SMB 2.0 in 2006 and then SMB 3.0 in 2012. Although it's common within the industry to refer to SMB fileshares as CIFS, the truth is that CIFS was deprecated long ago; only SMB lives on.

If you're working in a homogeneous UNIX and Linux environment, then this chapter probably isn't for you. But if you need a way to share files between UNIX and Windows systems, read on.

22.1 SAMBA: SMB SERVER FOR UNIX

Samba is a popular software package, available under the GNU Public License, that implements the server side of the SMB protocol on UNIX and Linux hosts. It was originally created by Andrew Tridgell, who first reverse-engineered the SMB protocol and published the resulting code in 1992. Here, we focus on Samba version 4.

Samba is well supported and under active development to expand its functionality. It offers a stable, industrial strength way to share files between UNIX and Windows systems. The real beauty of Samba is that you install only one package on the server side; no special software is needed on the Windows side.

In the Windows world, a filesystem or directory made available over the network is known as a "share." It sounds a bit strange to UNIX ears, but we follow this convention when referring to SMB filesystems.

Although we explore only file sharing in this chapter, Samba can also implement a variety of other cross-platform services, including

- Authentication and authorization
- Network printing
- Name resolution
- Service announcement (file server and printer "browsing")

Samba can also perform the basic functions of a Windows Active Directory controller. This configuration involves a certain amount of hubris, though; we suspect that being an AD controller is probably a job best left to Windows servers.

There is certainly value in getting your UNIX and Linux systems added to an AD domain as clients, however. This arrangement lets you share identity and authentication information site-wide. See Chapter 17, *Single Sign-On*, for more information.

Likewise, we don't recommend Samba as a print server. CUPS is probably your best bet there. See Chapter 12 for more information about printing in UNIX and Linux with CUPS.

Most of Samba's functionality is implemented by two daemons, **smbd** and **nmbd**. **smbd** implements file and print services as well as authentication and authorization. **nmbd** is responsible for the other major SMB components: name resolution and service announcement.

Unlike NFS, which requires kernel-level support, Samba requires no drivers or kernel modifications and runs entirely as a user process. It binds to the sockets used for SMB requests and waits for a client to request access to a resource. Once a request has been authenticated, **smbd** forks an instance of itself that runs as the user who is making the requests. As a result, all normal file access permissions (including group permissions) are obeyed. The only special

functionality that **smbd** adds on top of this is a file locking service that gives Windows systems the locking semantics to which they are accustomed.

If you're left wondering why you'd use SMB over, say, a more UNIX-integrated remote filesystem such as NFS, the answer is ubiquity. Almost all OSs support SMB at some level. Table 22.1 summarizes some of the main differences between SMB and NFS.

Table 22.1: SMB vs. NFS

SMB	NFS
User-space servers and processes	Kernel server with threads
Per-user server processes	Same server (one process) for all clients
Uses underlying OS for access control	Has its own access control system
Mounters: usually individual users	Mounters: usually systems
Pretty good performance	Best performance

Chapter 21 explores NFS in more detail.

22.2 Installing and configuring Samba

Samba is available for all our example systems. Most Linux distributions include it by default. Patches, documentation, and other goodies are available from samba.org. Make sure you are using the most current Samba packages available for your system, since many updates fix security vulnerabilities.

If Samba is not already installed on your system, you can install it on FreeBSD with **pkg install samba44**. On Linux systems, grab the **samba-common** package through your package manager of choice.

You configure Samba in the **/etc/samba/smb.conf** file (**/usr/local/etc/smb4.conf** on FreeBSD). The file specifies the directories to share, their access rights, and Samba's general operational parameters. Linux packages are kind enough to supply a heavily commented sample configuration that's a good starting point for new setups.

Samba comes with sensible defaults for its configuration options, and most sites need only a small configuration file. Run the command **testparm -v** for a listing of all the Samba configuration options and the values to which they are currently set. This listing includes your settings from the **smb.conf** or **smb4.conf** file as well as any default values you have not overridden. Note that once Samba is running, it checks its configuration file every few seconds and loads any changes—no restart required!

The most common use of Samba is to share files with Windows clients. Access to these shares must be authenticated through a user account by one of two options. The first option uses local accounts, for which users specify a password that is managed separately from their other accounts (such as their domain login). The second option integrates Active Directory authentication and so piggybacks on the user's domain login credentials.

File sharing with local authentication

The simplest way to authenticate users who want to access Samba shares is by creating a local account for them on the UNIX or Linux server.

Because Windows passwords work quite differently from UNIX passwords, Samba cannot control access to SMB shares by means of users' existing account passwords. Hence, to use local accounts, you must store (and maintain) a separate SMB password hash for every user.

Sometimes, however, simplicity outweighs user convenience, and this authentication system is really simple. Here's the start of an example **smb.conf** file that uses it:

```
[global]
workgroup = ulsah
security = user
netbios name = freebsd-book
```

The security = user parameter tells Samba to use local UNIX accounts. Be sure the workgroup name is set to fit your environment. This is typically the Active Directory domain if you're in a Windows environment. If you're not, you can omit this setting.

Samba has its own command, **smbpasswd**, for setting up Windows-style password hashes. For example, here we add the user tobi and set a password for him:

```
$ sudo smbpasswd -a tobi
New SMB password: <password>
Retype new SMB password: <password>
```

The UNIX account should already exist before you attempt to set its Samba password. Users can change their Samba password by running **smbpasswd** without any options:

```
$ smbpasswd
New SMB password: <password>
Retype new SMB password: <password>
```

This example changes the Samba password of the current user on the Samba server. Unfortunately, Windows-only users must log in to a shell prompt on the server to change their share password. The ability to log in remotely must be set up separately, most likely through SSH.

File sharing with accounts authenticated by Active Directory

As simple as the basic process is, maintaining a separate authentication database for shares with **smbpasswd** does seem archaic in today's hyper-integrated world. In most cases, you'll want users to authenticate through some form of centralized authority such as Active Directory or LDAP.

Recent years have brought great advances to UNIX and Linux in this area. Chapter 17, *Single Sign-On*, covers the necessary components, including directory services, **sssd**, the **nsswitch.conf** file, and PAM. Once you have deployed those components, configuring Samba to take advantage of them is easy. (Historically, **winbind** was used integrate Active Directory with Samba. These days, **sssd** is the preferred method.)

Here's an example of the start of an **smb.conf** file for an environment in which Active Directory performs user authentication (via **sssd**):

```
[global]
    workgroup = ulsah
    realm = ulsah.example.com
    security = ads
    dedicated keytab file = FILE:/etc/samba/samba.keytab
    kerberos method = dedicated keytab
```

In this case, the `realm` parameter should be the same as the local Active Directory domain name. The `dedicated keytab file` and `kerberos method` parameters enable Samba to work properly with Active Directory's Kerberos implementation.

The keytab file is created by **sssd** if you've set it up according to the instructions in Chapter 17. For more information about keytabs in Samba, see goo.gl/ZxCUKA (deep link within wiki.samba.org).

Configuring shares

After you've configured Samba's general settings and authentication, you can specify in the **smb.conf** file which directories should be shared through SMB. Each share that you expose needs its own stanza in the configuration file. The name of the stanza becomes the share name that is advertised to SMB clients.

Here's an example:

```
[bookshare]
   path = /storage/bookshare
   read only = no
```

Here, SMB clients see a mountable share named *sambaserver*\bookshare. It yields access to the file tree located at **/storage/bookshare** on the server.

Sharing home directories

You can automatically convert users' home directories into distinct SMB shares with the magic stanza name [homes] in the **smb.conf** file:

```
[homes]
comment = Home Directories
browseable = no
valid users = %S
read only = no
```

For example, this configuration would allow user janderson to access her home directory through the path *sambaserver*\janderson from any Windows system on the network.

At some sites, the default permissions on home directories let users browse one another's files. Because Samba relies on UNIX file permissions to implement access control, Windows users coming in through Samba can then read one another's home directories, too. However, experience shows that this behavior tends to confuse Windows users and make them feel exposed.

The variable %S listed as the value of valid users in the example above expands to the username associated with each share; it thus restricts access to the owner of the home directory. Omit this line if that is not the behavior you want.

Samba uses its magic [homes] section as a last resort. If a particular user's home directory has an explicitly defined share in the configuration file, the parameters set there override the values set through [homes].

Sharing project directories

Samba can map Windows access control lists (ACLs) to either traditional UNIX file permissions or ACLs, if the underlying filesystem supports them. But in practice, we find that ACLs are too complex for most users to deal with.

See this page for more information about ACLs.

Instead of using ACLs, we normally set up a special share for each user group that needs a collective work area. When a user attempts to mount this share, Samba checks that the applicant is in the appropriate UNIX group before allowing access. In the example below, a user must be a part of the eng group to mount the share and access files:

```
[eng]
comment = Group Share for engineering
; Everybody who is in the eng group may access this share.
; People will have to log in with their Samba account.
valid users = @eng
path = /home/eng

; Disable NT ACLs since we do not use them here.
nt acl support = no

; Make sure that all files have sensible permissions and that dirs
; have the setgid (inherit group) bit set.
create mask = 0660
directory mask = 2770
force directory mode = 2000
force group = eng

; Normal share parameters.
browseable = no
read only = no
guest ok = no
```

This configuration does not require you to create a pseudo-user to act as the owner of the shared directory. You just need a UNIX group (here, eng) that includes the intended users of the share.

Users mount the share under their own accounts, but to facilitate collaboration, we would prefer that any files created within the share be owned by group eng. That way, other team members can access newly created files by default.

The first step toward ensuring this behavior is to use the `force group` option to coerce mounters' effective group IDs to eng, the UNIX group that controls access to the share. However, this step alone is not enough to ensure that new files and directories are assigned a group owner of eng.

As explained here, the setgid option on a directory makes new files created within that directory inherit the directory's group owner—or at least, it does so on Linux. (FreeBSD does not honor

the setgid bit on a directory; however, its default behavior is to inherit the group, just as Linux does with the setgid bit turned on. Setting the setgid bit on FreeBSD does no harm, however.)

We can ensure that new files are owned by eng by setting the group of the share's root to eng and then turning on the setgid bit on that directory:

```
$ sudo chown root:eng /home/eng
$ sudo chmod u=rwx,g=rwxs,o= /home/eng
```

These measures are sufficient to manage files created in the root of the share. However, to make the system work for complex hierarchies of files, we also need to ensure that newly created directories' setgid bits are also turned on. The example configuration above implements this requirement with the force directory mode and directory mask options.

22.3 Mounting SMB file shares

Mounting for SMB file shares works quite differently from how it's done for other network filesystems. In particular, SMB volumes are mounted by a specific user rather than being mounted by the system itself.

You need local permission to perform an SMB mount. You also need the password for an identity that the remote SMB server will allow to mount the share. A typical command line on Linux is

```
$ sudo mount -t cifs -o username=joe //redmond/joes /home/joe/mnt
```

And the equivalent on FreeBSD:

```
$ sudo mount -t smbfs //joe@redmond/joes /home/joe/mnt
```

Windows conceptualizes network mounts as being established by a particular user (hence the **username=joe** option above), whereas UNIX regards them as more typically belonging to the system as a whole. Windows servers generally cannot deal with the concept that several different people might be accessing a mounted Windows share.

From the perspective of the UNIX client, all files in the mounted directory appear to belong to the user who mounted it. If you mount the share as root, then all files belong to root, and garden-variety users might not be able to write files on the Windows server.

The mount options **uid**, **gid**, **fmask**, and **dmask** let you tweak these settings so that ownership and permission bits are more in tune with the intended access policy for that share. Check the **mount.cifs** (Linux) or **mount_smbfs** (FreeBSD) man page for more information about these options.

22.4 Browsing SMB file shares

Samba includes a command-line utility called **smbclient** that lets you list file shares without actually mounting them. It also defines an FTP-like interface for interactive access. This feature can be useful when you are debugging or when a script needs access to a share.

For example, here's how to list shares available to user dan on the server hoarder:

```
$ smbclient -L //hoarder -U dan
Enter dan's password: <password>
Domain=[WORKGROUP] OS=[Unix] Server=[Samba 3.6.21]

        Sharename    Type    Comment
        ---------    ----    -------
        Temp         Disk    Temp Storage
        Programs     Disk    Various Programs and Applications
        Docs         Disk    Shared Documents
        Backups      Disk    Backups of all sorts
```

To connect to a share and transfer files, omit the **-L** flag and include the share name:

```
$ smbclient //hoarder/Docs -U dan
```

Once you're connected, type help for a list of available commands.

22.5 Ensuring Samba security

It’s important to be aware of the security implications of sharing files and other resources over a network. For a typical site, you need to do two things to ensure a basic level of security:

- Explicitly specify which clients can access the resources shared by Samba. This part of the configuration is controlled by the `hosts allow` clause in the **smb.conf** file. Make sure that it contains only the IP addresses, address ranges, or hostnames that it should.

  You can include a `hosts deny` clause in the **smb.conf** file as well, but note that denials have priority. If you include a hostname or address in both the `hosts deny` clause and the `hosts allow` clause, that host will not be able to access the resource.

- Block access to the server from outside your organization. Samba uses encryption only for password authentication. It does not use encryption for its data transport. In almost all cases, you should block access from outside your organization to prevent users from accidentally downloading files in plain text across the Internet.

  Blocking is typically implemented at the network firewall level. Samba uses UDP ports 137–139 and TCP ports 137, 139, and 445.

Since the release of Samba version 3, excellent security documentation has been available on Samba’s wiki, wiki.samba.org.

22.6 Debugging Samba

Samba usually runs without requiring much attention. If you do experience a problem, you can consult two primary sources of debugging information: the **smbstatus** command and Samba's logging facilities.

Querying Samba's state with smbstatus

smbstatus shows currently active connections and locked files; it's the first place to look when issues arise. This information is especially useful for tracking down locking problems (e.g., "Which user has file **xyz** open read/write exclusive?").

```
$ sudo smbstatus                # Some output condensed for clarity
Samba version 4.3.11-Ubuntu
PID     Username      Group         Machine
-------------------------------------------------------------

6130    clay          atrust        192.168.20.48
23006   dan           atrust        192.168.20.25

Service      pid     machine       Connected at
-------------------------------------------------------

admin        6130   192.168.20.48  Wed Apr 12 07:25:15 2017
swdepot2     6130   192.168.20.48  Wed Apr 12 07:25:15 2017
clients      6130   192.168.20.48  Wed Apr 12 07:25:15 2017
clients      23006  192.168.20.25  Fri Apr 28 14:32:25 2017

Locked files:
Pid     Uid  DenyMode   R/W    Oplock SharePath        Name
--------------------------------------------------------------------------

23006  1009  DENY_NONE  RDONLY  NONE  /atrust/clients  Acme_Supply/Con...
6130   1035  DENY_ALL   RDONLY  NONE  /home/clay       .
6130   1035  DENY_NONE  RDONLY  NONE  /atrust/admin    New Hire Proces...
```

The first section of output lists the users that have connected. The Service column in the next section shows the actual shares they've mounted. The last section, from which we've removed a couple of columns to save space, lists any active file locks.

If you kill the **smbd** associated with a certain user, all that user's locks disappear. Some applications handle this situation gracefully and reacquire any locks they need. Others freeze and die a horrible death, with much clicking required on the Windows side just to close the unhappy application. As dramatic as this may sound, we have yet to see any file corruption resulting from such a procedure.

Be careful when Windows claims that files have been locked by another application; it is often right. Fix the problem on the client side by closing the offending application instead of brute-forcing the locks on the server.

Configuring Samba logging

Configure logging parameters in your **smb.conf** file:

```
[global]
# The %m causes a separate file to be written for each client.
log file = /var/log/samba.log.%m
max log size = 10000

# If you want Samba to log only through syslog then set the following
# parameter to 'yes'.
syslog only = no

# We want Samba to log a minimal amount of information to syslog.
# Everything should go to /var/log/samba/log.{smbd,nmbd} instead.
# If you want to log through syslog, increase the following parameter
syslog = 7
```

Higher log levels produce more information. Logging uses system resources, so don't ask for too much detail unless you are actively debugging.

The following example shows log entries generated by an unsuccessful connection attempt:

```
[2017/04/30 08:44:47.510724,  2, pid=87498, effective(0,
    0), real(0, 0), class=auth] ../source3/auth/
    auth.c:315(auth_check_ntlm_password)
  check_ntlm_password:  Authentication for user [dan] -> [dan] FAILED
    with error NT_STATUS_WRONG_PASSWORD
[2017/04/30 08:44:47.510821,  3] ../source3/smbd/
    error.c:82(error_packet_set)
  NT error packet at ../source3/smbd/sesssetup.c(937) cmd=115
    (SMBsesssetupX) NT_STATUS_LOGON_FAILURE
```

A successful attempt looks like this:

```
[2017/04/30 08:45:30.425699,  5, pid=87502, effective(0,
    0), real(0, 0), class=auth] ../source3/auth/
    auth.c:292(auth_check_ntlm_password)
  check_ntlm_password:  PAM Account for user [dan] succeeded
[2017/04/30 08:45:30.425864,  2, pid=87502, effective(0,
    0), real(0, 0), class=auth] ../source3/auth/
    auth.c:305(auth_check_ntlm_password)
  check_ntlm_password:  authentication for user [dan] -> [dan]
    -> [dan] succeeded
```

The **smbcontrol** command is handy for altering the debug level of a running Samba server without altering the **smb.conf** file. For example,

```
$ sudo smbcontrol smbd debug "4 auth:10"
```

This command sets the global debug level to 4 and sets the debug level for authentication-related matters to 10. The **smbd** argument specifies that all **smbd** daemons on the system should have their debug levels set. To debug a specific established connection, use the **smbstatus** command to determine which **smbd** daemon handles the connection, then pass its PID to **smbcontrol** to debug just that one connection. At log levels over 100, you'll start to see encrypted passwords in the logs.

Managing character sets

Starting with version 3.0, Samba encodes all filenames in UTF-8. If your server runs with a UTF-8 locale, which we recommend, this a great match. Type **echo $LANG** to see if your system is running in UTF-8 mode.

If you are in Europe and are still using one of the ISO 8859 locales on the server, you may find that Samba-created filenames that include accented characters (e.g., ä, ö, ü, é, or è) do not display correctly when you run **ls**. The solution is to tell Samba to use the same encoding as your server:

```
unix charset = ISO8859-15
display charset = ISO8859-15
```

Make sure the filename encoding is correct right from the start. Otherwise, files with improperly encoded names accumulate. Fixing them later is a surprisingly complex task.

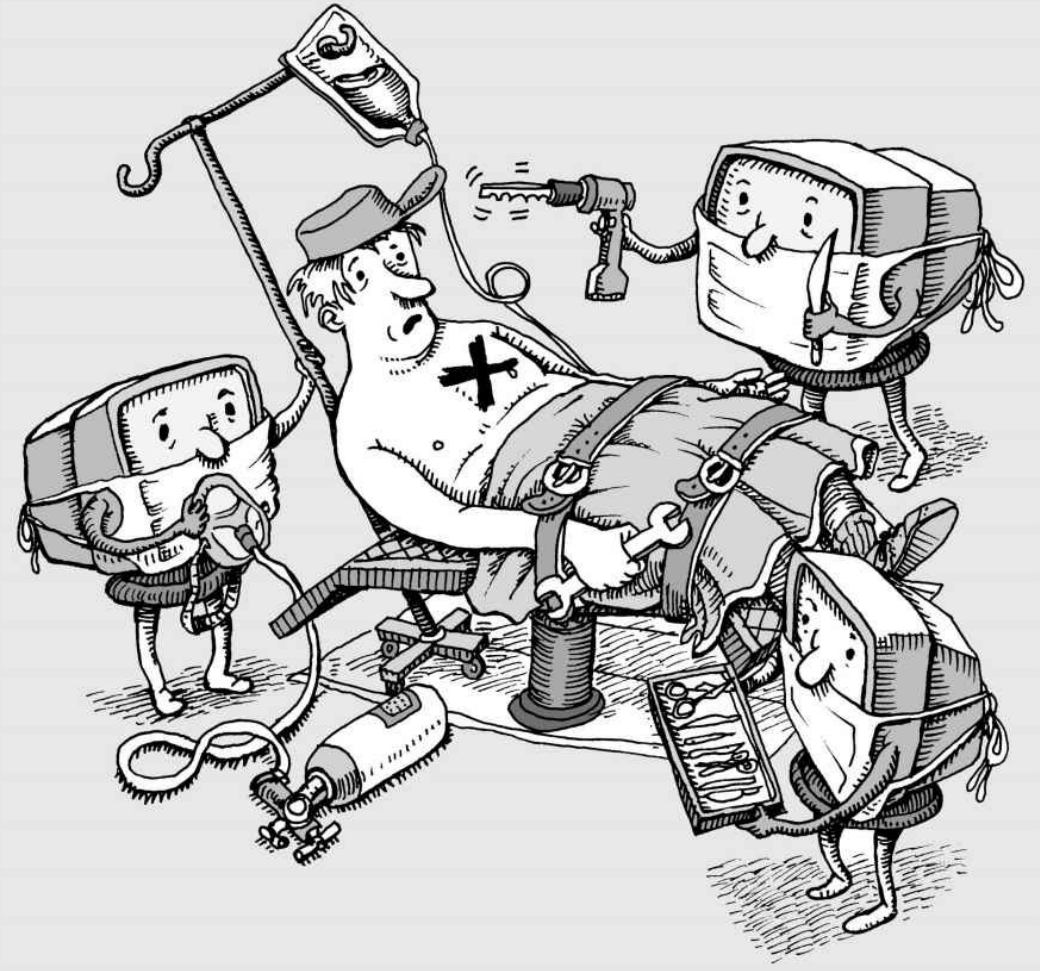
22.7 Recommended reading

Red Hat. *Red Hat Enterprise Linux System Administrator's Guide: File and Print Servers*. goo.gl/LPjNXa (deep link into access.redhat.com/documentation).

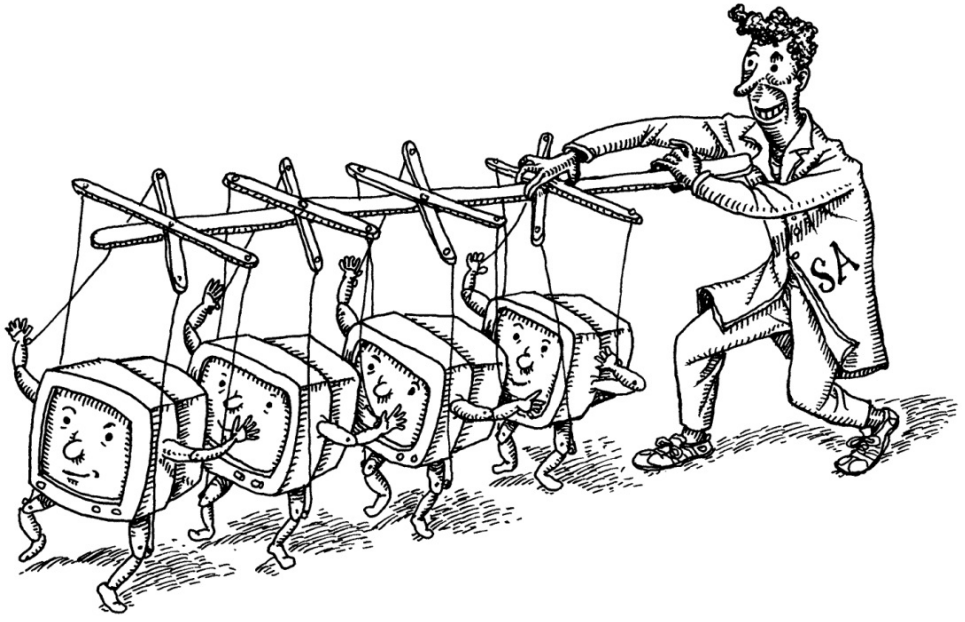
Samba Project. *Samba Wiki Page*. wiki.samba.org. This wiki is updated relatively frequently and is an authoritative source of information, though portions are somewhat disorganized.

SECTION FOUR

OPERATIONS



23 Configuration Management



A longstanding tenet of system administration is that changes should be structured, automated, and applied consistently among machines. But that's easier said than done when you're confronted with a heterogeneous fleet of systems and networks in various states of health.

Configuration management software automates the management of operating systems on a network. Administrators write specifications that describe how servers should be configured, and the configuration management software then brings reality into conformance with the specifications. Several open source implementations of configuration management are in widespread use. In this chapter, we introduce the basics of configuration management and describe the major players.

As an automation tool, configuration management is closely affiliated with the DevOps philosophy of IT operations, which we describe in more detail starting here. People sometimes conflate DevOps and configuration management, so you may occasionally hear these terms used interchangeably. Nevertheless, they are distinct. In this chapter, we show several examples that demonstrate how configuration management enables—but not is not identical to—several key elements of DevOps.

"Configuration management system" is a bit of a handful to read and write, so we often shorten that term to "CM system" (or even simply CM). (Unfortunately, the abbreviation CMS is already in widespread use for "content management system.")

23.1 CONFIGURATION MANAGEMENT IN A NUTSHELL

The traditional approach to sysadmin automation is an intricate complex of home-grown shell scripts supplemented by ad hoc fire fighting when scripts fail. This scheme works about as well as you might expect. Over time, systems managed in this way usually degenerate into a chaotic wreckage of package versions and configurations that can't be reliably reproduced. It's sometimes called the snowflake model of system administration because no two systems are ever alike.

See Chapter 7 for more information about shell scripting.

Configuration management is a better approach. It captures desired state in the form of code. Changes and updates can then be tracked over time in a version control system, which creates an audit trail and a point of reference. The code also acts as informal documentation of a network. Any administrator or developer can read the code to determine how the system is configured.

When all a site's servers are under configuration management, the CM system effectively acts as both an inventory database and a command-and-control center for the network. CM systems also offer "orchestration" features, which let you apply changes and run ad hoc commands remotely. You can target groups of hosts whose hostnames match particular patterns or whose configuration variables match a given set of values. Managed clients report information about themselves to the central database for analysis and monitoring.

Most configuration management "code" uses a declarative—as opposed to procedural—idiom. Rather than writing scripts that tell the system what changes to make, you describe the state you want to achieve. The configuration management system then uses its own logic to adjust target systems as necessary.

Ultimately, the job of a CM system is to apply a series of configuration specifications, aka operations, to an individual machine. Operations vary in granularity, but they are typically coarse enough to correspond to items that might plausibly appear on a sysadmin's to-do list: create a user account, install a software package, and so on. A subsystem such as a database might require anywhere between 5 and 20 operations to fully configure. Full configuration of a freshly booted system might entail dozens or hundreds of operations.

23.2 Dangers of configuration management

Configuration management is a major improvement over the ad hoc approach, but it is not a magic wand. A few sharp edges are particularly important for administrators to be aware of in advance.

Although all major CM systems use similar conceptual models, they describe these models with different lexicons. Unfortunately, the terminology used by a particular CM system often has more to do with conforming to a marketing theme than with maximizing clarity.

The result is a general lack of conformity and standardization among systems. Most administrators will encounter several CM systems throughout the course of their careers and will develop preferences derived from that experience. Unfortunately, knowledge of one system is not directly portable to another.

As a site grows, so too must the infrastructure needed to support its configuration management system. A site with a few thousand servers needs a handful of systems dedicated to running the CM workloads. This overhead imposes both direct and indirect costs in the form of hardware resources and ongoing maintenance. CM system upgrades can be major projects in their own right.

A certain level of operational maturity and rigor is necessary for a site to fully embrace configuration management. Once a host is under the control of a CM system, it must not be modified manually, or it immediately reverts to the status of a snowflake system.

On more than one occasion, we have seen cases in which hurried (or lazy) administrators have manually updated a configuration-managed host and set their changes to be immutable, thus overriding the expected state and preventing the CM system from applying future changes. This kind of hack results in great confusion when an admin’s colleagues cannot quickly determine why the expected configuration is not being applied. In one case, it caused a major service outage.

Although some CM systems are easier to pick up than others, they are all notorious for having a steep learning curve, especially for administrators who lack prior experience with automation. If you match this description, consider practicing on a lab of virtual machines to hone your skills before you tackle your production network.

23.3 Elements of configuration management

In this section, we review the components of a CM system and the concepts used to configure it at a greater level of detail. Then, starting here, we survey four of the most commonly used CM systems: Ansible, Salt, Puppet, and Chef.

Rather than adopt any particular CM system's terminology, we use the clearest and most directly descriptive term we can find for each concept. Table 23.2 maps the correspondences between our vocabulary and those of the four CM systems listed above. If you're already familiar with one of those CM systems, you might find it helpful to refer to this table as you read the material below.

Operations and parameters

We've already introduced the concept of operations, which are the small-scale actions and checks used by a CM system to achieve a particular state. Every CM system includes a large set of supported operations, and more arrive with each new release.

Here are some sample operations that all CM systems can handle right out of the box:

- Create or remove a user account or set its attributes
- Copy files to or from the system being configured
- Synchronize directory contents
- Render a configuration file template
- Add a new line in a configuration file
- Restart a service
- Add a **cron** job or **systemd** timer
- Run an arbitrary shell command
- Create a new cloud server instance
- Create or remove a database account
- Set database operating parameters
- Perform Git operations

This is just a sampling; most CM systems define hundreds of operations, including many that perform potentially complex niche operations, such as setting up specific databases, run-time environments, or even pieces of hardware.

If operations seem suspiciously similar to shell commands, your intuition is accurate. They are scripts, usually written in the implementation language of the CM system itself and exploiting the system's standard tools and libraries. In many cases, they run standard shell commands under the hood as part of their implementation.

Just as UNIX commands accept arguments, most operations accept parameters. For example, a package management operation would accept parameters that specify the package name, version, and whether the package is to be installed or removed.

Parameters vary according to the operation. As a convenience, they usually have default values that are suitable for the most common use cases.

CM systems let you use variable values (see the next section) to define parameters. They can also infer parameter values according to the environment of the system, such as the network it lives on, whether a particular configuration property is present, or whether the system's hostname matches a given regular expression.

A well-behaved operation knows nothing about the host or hosts to which it might eventually be applied. The implementation is written to be relatively generic and OS-agnostic. Binding operations to specific systems occurs at a higher level of the configuration management hierarchy.

Despite CM systems' focus on declarative configuration, operations must ultimately run like any other command. Execution has a start and an end. It can succeed or fail. It reports its status back to the calling environment.

However, operations differ from typical UNIX commands in a few important ways:

- Most operations are designed to be applied repeatedly without causing problems. Borrowing a term from linear algebra, you'll sometimes see this latter property referred to as "idempotence."
- Operations know when they change the system's actual state.
- Operations know when the system state *needs* to be changed. If the current configuration already conforms to the specification, the operation exits without doing anything.
- Operations report their results to the CM system. Their report data is richer than a UNIX-style exit code and can aid in debugging.
- Operations strive to be cross-platform. They usually define a constrained set of functions that are common to all supported platforms, and they interpret requests in accordance with the local system.

Some operations can't be made idempotent without a little help from a sysadmin who knows more about the context. For example, if an operation runs a garden-variety UNIX command, the CM system has no direct way of knowing what effect that command had on the system.

You also have the option of writing your own custom operations. They're just scripts, and the CM system typically provides a well-greased path for integrating your custom operations with the standard ones.

Variables

Variables are named values that influence how configurations are applied to individual machines. They commonly set parameter values and fill in the blanks in configuration templates.

Variable management in CM systems is often quite rich. A few points of note:

- Variables can typically be defined in many different places and contexts within the configuration base.
- Each definition has a scope in which it's visible. Scope types vary by CM system and might encompass a single machine, a group of machines, or a particular set of operations.
- Multiple scopes can be active in any given context. Scopes can be nested, but more commonly they are simply coactive.
- Because multiple scopes can define values for the same variable, some form of conflict resolution is necessary. Some systems merge values, but most use precedence rules or definition order to pick a winning value.

Variables are not limited to having scalar values; arrays and hashes are also acceptable variable values in all CM systems. Some operations accept nonscalar parameter values directly, but such values are more typically used above the level of individual operations. For example, an array might be enumerated in a loop to apply the same operation more than once with different parameters.

Facts

CM systems investigate each configuration client to determine descriptive facts such as the IP address of the primary network interface and the OS type. This information is then accessible from within the configuration base through variable values. As with any other variable, these values can be used to define parameter values or to expand templates.

It can take awhile to determine all the facts associated with a particular system. Therefore, CM systems generally cache facts, and they do not necessarily rebuild the cache on every run. If you find that a particular configuration flow is encountering stale configuration data, you might need to explicitly invalidate the cache.

All CM systems let target machines add their own values to the fact database, either by including a static file of declarations or by running custom code on the target machine. This feature is useful both for extending the types of information that can be accessed through the facts database and for moving static configuration information onto client machines.

Client-side hints can be particularly useful for managing cloud and virtual servers. You simply apply cloud-level markers (such as EC2 tags) as an instance is created, and the configuration management system can then flesh out the appropriate configuration by checking the markers. Keep in mind the security implications of this approach, however: the client controls the facts that it reports, so make sure that a compromised client can’t exploit the configuration management system to gain additional privileges.

Depending on the CM system, you may be able to transcend your local context when sniffing around in the variable or fact space. In addition to accessing the configuration information for the current host, you may also be able to access data for other hosts, or even to introspect the state of the configuration base itself. This is a useful feature for coordinating a distributed system such as a cluster of servers.

Change handlers

If you change a web server's configuration file, you had better restart the web server. That's the basic concept behind handlers, which are operations that run in response to some sort of event or situation rather than as part of a baseline configuration.

In most systems, a handler runs whenever one or more of a designated set of operations reports that it has modified the target system. The handler isn't told anything about the exact nature of the change, but because the association between operations and their handlers is fairly specific, additional information isn't needed.

Bindings

Bindings complete the basic configuration model by associating specific sets of operations to specific hosts or groups of hosts. You can also bind operations to a dynamic set of clients that's defined by the value of a fact or variable. CM systems can also define host groups by looking up information in a local inventory system or by calling a remote API.

In addition to their basic linking role, bindings in most CM systems also act as variable scopes. This feature lets you customize the behavior of the operations you're assigning by defining or customizing variable values for the clients that are targeted.

A given host can match criteria for many different bindings. For example, a node might live on a certain subnet, be managed by a particular department, or fill an explicitly designated role (e.g., Apache web server). The CM system takes account of all of these factors and activates the operations associated with each binding.

Once you set up the bindings for a host, you can invoke your CM system's top-level "configure everything" mechanism to make the CM system identify all the operations that should run on the target and execute them in order.

Bundles and bundle repositories

A bundle is a collection of operations that perform a specific function, such as installing, configuring, and running a web server. CM systems let you package bundles into a format that's suitable for distribution or reuse. In most cases, a bundle is defined by a directory, and the name of the directory defines the name of the bundle.

CM vendors maintain public repositories that include both officially blessed and user-contributed bundles. You can use these "as is" or modify them to suit your needs. Most CM systems provide native commands for interacting with repositories.

Environments

It's often useful to segregate configuration-managed clients into multiple "worlds," such as the traditional categories of development, test, and production. Large installations can create even finer distinctions to support processes such as the gradual ("staged") rollout of new code into production.

These different worlds are known generically as "environments," both inside and outside the configuration management context. This seems to be the single term on which all configuration management systems agree.

When properly implemented, environments are not just groups of clients. They're an additional axis of variation that can affect multiple aspects of the configuration. The development and production environments might both include web servers and database servers, for example, but the details of how those roles are defined might vary among environments.

For example, it's common for the database and web server to run on the same machine in the development environment. However, the production environment usually has multiple servers of each type. A production environment might also define server types that don't exist in the development environment, such as those that do load balancing or act as DMZ proxies.

The environment system is usually thought of as a sort of pipeline for configuration code. As a thought experiment, you can imagine that fixed groups of clients run the development, test, and production environments. As a given configuration base is validated, it propagates from one environment to the next, ensuring that changes are properly vetted before they reach the all-important production systems.

On most CM systems, different environments are just different versions of the same configuration base. If you're a Git user, think of them as tags in a Git repository: the development tag points to the most recent version of the configuration base, and the production tag might point to a commit that's several weeks old. The tags move forward as releases make their way through testing and deployment.

Different environments can provide different variable values to clients. For example, the database credentials used in development likely differ from those used on production systems, as do the details of the network configuration and perhaps the users and groups who are permitted access.

See Chapter 26, *Continuous Integration and Delivery*, for more information about environments.

Client inventory and registration

Because CM systems define lots of ways to segregate clients into categories, the overall universe of machines under configuration management must be well defined. The inventory of managed hosts can live in a flat file or in a proper relational database. In some cases, it may even be entirely dynamic.

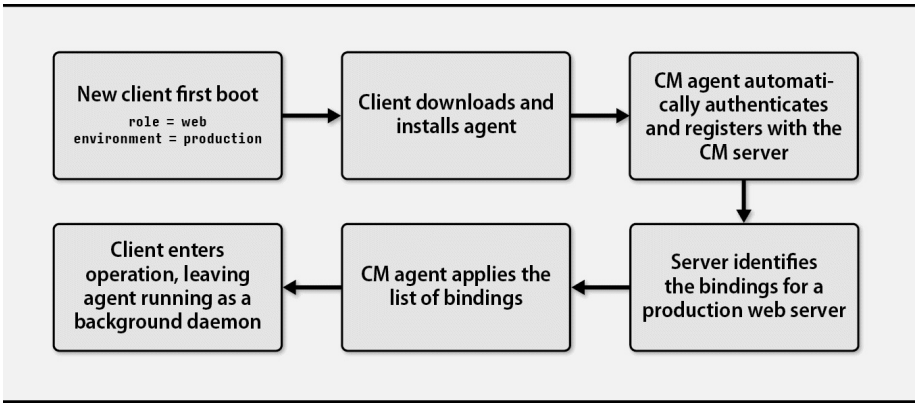
The exact mechanism through which configuration code is distributed, parsed, and executed varies among CM systems. Most systems actually give you several options in this regard. Here are a few common approaches:

- A daemon runs continuously on each client. The daemon pulls its configuration code from a designated CM server (or server group).
- A central CM server pushes configuration data to each client. This process can run on a regular schedule, or it can be initiated by administrators.
- Each managed node runs a client that wakes up periodically, reads configuration data from a local clone of the configuration base, and applies the relevant configuration to itself. There is no central configuration server.

Configuration information is sensitive and often includes secrets such as passwords. To protect this data, all CM systems define some way for clients and servers to authenticate each other and to encrypt private information.

The process of putting a new client under configuration management can be made as simple as installing the appropriate client-side software. If the environment has been configured to support automatic bootstrapping, a new client can automatically contact its configuration server, authenticate itself, and initiate the configuration process. OS-specific initialization mechanisms typically kick off this chain of events the first time the client is bootstrapped. The flow is depicted in Exhibit A.

Exhibit A: Initialization process for a new CM-managed client


New client first boot
role = web
environment = production
Client downloads and installs agent
CM agent automatically authenticates and registers with the CM server
Server identifies the bindings for a production web server
CM agent applies the list of bindings
Client enters operation, leaving agent running as a background daemon

23.4 Popular CM systems compared

Currently, four major players own the market for general configuration management on UNIX and Linux systems: Ansible, Salt, Puppet, and Chef. Table 23.1 shows some general information about these packages.

Table 23.1: Major configuration management systems

		Languages and formats			Daemons	
System	**Web site**	**Impl**	**Config**	**Template**	**Server**	**Client**
Ansible	ansible.com	Python	YAML	Jinja	No	No
Salt	saltstack.com	Python	YAML	Jinja	Optional	Optional
Puppet	puppet.com	Ruby	custom	ERB[a]	Optional	Optional
Chef	chef.io	Ruby	Ruby	ERB	Optional	Yes

a. ERB (embedded Ruby) is a basic syntax for embedding Ruby code in templates.

All of these packages are relatively young. The oldest, Puppet, debuted in 2005. It still claims the largest market share, in large part because of its early head start. Chef was released in 2009, followed by Salt in 2011 and Ansible in 2012.

The general category of configuration management software was pioneered by Mark Burgess's CFEngine in 1993. CFEngine is still around and is still actively developed, but the majority of its user base has been siphoned away by newer systems. See cfengine.com for current information.

Microsoft has its own CM solution in the form of PowerShell Desired State Configuration. Although it originates in the Windows world and is primarily designed to configure Windows clients, Microsoft has also published extensions for configuring Linux systems. It's worth noting that all four of the systems in Table 23.1 can configure Windows clients as well.

A number of projects focus on specific subdomains of configuration management, notably new-system provisioning (e.g., Cobbler) and software deployment (e.g., Fabric and Capistrano). The general proposition behind these systems is that by more closely modeling a specific problem domain, they can provide a simpler and more targeted set of features.

Depending on your needs, you may or may not find that these specialized systems provide a reasonable rate of return on your learning investment. Generic configuration management systems like those in Table 23.1 are not perfectly suited to all possible activities.

The systems in Table 23.1 work with pretty much any type of contemporary UNIX-compatible client machine, although there's always a support frontier. Chef has a modest edge in compatibility and supports even AIX.

See Chapter 25 for more information about containers.

OS support on the configuration server side (for those systems that use a configuration server) is more limited. Chef, for example, requires RHEL or Ubuntu for its server. Containerized versions of the server can run anywhere, though, so this isn’t as much an obstacle as it might seem.

Terminology

Table 23.2 shows the terms used by each of our example CM systems for the entities outlined in *Elements of configuration management*.

Table 23.2: Configuration management Rosetta Stone

Our term	Ansible	Salt	Puppet	Chef
operation	task	state	resource	resource
op type	module	function	resource type, provider	provider
op list	tasks	states	class, manifest	recipe
parameter	parameter	parameter	property, attribute	attribute
binding	play(book)	top file	classification, declaration	run list
master host	control	master	master	server
client host	host	minion	agent, node	node
client group	group	nodegroup	node group	role
variable	variable	variable	parameter, variable	attribute
fact	fact	grain	fact	automatic attribute
notification	notification	requisite	notify	notifies
handler	handler	state	subscribe	subscribes
bundle	role	formula	module	cookbook
bundle repo	galaxy	GitHub	forge	supermarket

Business models

All the products we discuss are freemium-model packages, which means that the basic systems are open source and free, but that each system has a corporate backer that sells support, consulting services, and add-on packages.

In theory, vendors have a potential motivation to withhold useful functionality from the open source releases to motivate add-on sales. In the configuration management space, however, this effect has not been evident. The open source versions of the software are full-featured and more than adequate for most sites.

Add-on services are mostly of interest to large organizations. If your site falls into this category, you may want to evaluate configuration management systems with respect to the functionality and pricing of the full-stack offerings. The main upsells are usually support, custom development, training, GUIs, and reporting and monitoring solutions. In this book, we discuss only the basic, free versions.

Architectural options

In theory, CM systems don't require servers. Software could run only on the machines being configured. You'd copy the configuration base to each target host and simply run a command to say, "Here, configure yourself according to these specifications."

In practice, it's nice not to have to fuss with the details of getting configuration information pushed out to clients and executed. CM systems always make some provision for centralized control, even if the master machine is defined as "wherever you happen to be logged in and have a clone of the configuration base."

Ansible uses no daemons at all (other than **sshd**), which is an appealing simplification. Configuration runs happen when an administrator (or **cron** job) on the server runs the **ansible-playbook** command. **ansible-playbook** executes the appropriate remote commands over SSH, leaving no trace of its presence on the client machine after configuration has completed. The only requirements for client machines are that they be accessible through SSH and have Python 2 installed. (Depending on the system, you might need a Python add-on or two as well. For example, Fedora requires the **python-dnf** package.)

Salt, Puppet, and Chef include both master- and client-side daemons. Typical deployment scenarios run daemons on both sides of the relationship, and this is the environment you'll see described in most documentation. It's possible to run each of these without a server also, but this configuration is less common.

It's tempting to assume that CM systems with daemons must be more heavyweight and more complex than those without (i.e., Ansible). However, that isn't necessarily true. In Salt and Puppet, the daemons are facilitators and accelerators. They're useful but optional, and they don't change the fundamental architecture of the system, although they do enable some advanced features. If you prefer, you are free to run these systems without daemons and to replicate the configuration base by hand. Salt even has an SSH-based mode that works similarly to Ansible.

Given that, why would you want to mess around with a bunch of optional daemons? Several reasons:

- It's faster. Ansible works hard to overcome the performance limits imposed by SSH and by its lack of client-side caching, but it is still noticeably more sluggish than Salt. When you're reading a system administration book, ten seconds sounds insignificant. In the midst of resolving an outage, it feels endless, especially when repeated across dozens or hundreds of clients.
- Some features can't exist without central coordination. For example, Salt lets clients notify the configuration master of events such as full disks. You can then respond to these events through the normal configuration management facilities. Having a central connection point facilitates a variety of interclient data-sharing features.

- Only the master-side daemon is really a potential source of administrative complexity. CM systems work hard to make client bootstrapping a one-line operation, regardless of whether a daemon is involved.
- The presence of active agents on both client and server opens a variety of architectural options not available in one-sided configurations.

In terms of architecture, Chef is the outlier among configuration management systems in that its server daemon is a top-tier entity within the conceptual model. Salt and Puppet serve configuration data directly from plaintext files on disk; to make changes, you simply edit the files. By contrast, the Chef server is an opaque and authoritative source of configuration information. Changes must be uploaded to the server with the **knife** command or they will not be available to clients. (However, even Chef has a serverless mode of operation in the form of **chef-solo**.)

We mention all this not to promote serverful systems per se, but simply to point out that the main fault line among CM systems runs between Chef and everything else. Ansible, Salt, and Puppet all have about the same, modest level of overall complexity. Chef requires significantly more investment to maintain and master, especially when its extensive line of add-on modules is added to the mix.

Because of its serverless model, Ansible is often tagged as a sort of “easy option” for configuration management. But in fact, the basic architectures of Salt and Puppet are similarly approachable. A strong case could be made that Salt is the simplest system of all, if you disregard its advanced facilities and somewhat peculiar documentation. Don’t write off Salt and Puppet as advanced options.

The converse is also true: Ansible is more than just a gimped-out starter system for short-attention-span sysadmins. It’s a legitimate option for complex sites, although its sluggish performance becomes increasingly more apparent in this context.

Language options

Ansible and Salt are written in Python. Puppet and Chef are written in Ruby. But except in the case of Chef, this information is probably less relevant than it might initially appear.

No Python code appears in the average Ansible or Salt configuration. Both of these systems use YAML (an alternate syntax for expressing JavaScript object notation, aka JSON), as their primary configuration language. YAML is just structured data, not code, so it has no inherent behavior other than the interpretation assigned by the configuration management system.

Here's a simple example from Salt that keeps the SSH service enabled and running:

```
ssh.server.run_ssh:
  service:
    - name: sshd
    - running
    - enable: true
```

To make YAML files more dynamically expressive, both Ansible and Salt augment them with a templating system, Jinja2, as a preprocessor. Jinja has its roots in Python, but it's not just a simple Python wrapper. In use, it feelsmore like a template system than a real programming language. Even Salt, which relies more heavily on Jinja than does Ansible, cautions against putting too much logic into Jinja code.

In fairness, Salt is actually format- and preprocessor-agnostic, and it supports several input pipelines (including raw Python) right out of the box. However, departing from the greased path of Jinja and YAML means leaving the documentation and the rest of the world behind. It's probably best deferred until you're quite familiar with Salt.

The bottom line is that unless you write your own custom operation types or use explicit escapes into Python, you won't be encountering much Python in the Ansible and Salt worlds. Extending the CM system with your own code can in fact be quite easy and quite helpful, however.

Jon Corbet, one of our technical reviewers, agrees that these systems don't expose much Python…until things go horribly wrong. "At that point," he adds, "familiarity with Python tracebacks and data structure representations helps a lot."

Both Puppet and Chef use Ruby-based, domain-specific languages as their primary configuration systems. Chef's version is a lot like a configuration management analog of Rails from the web development world. That is, it has been extended with a few concepts that are designed to facilitate configuration management, but it's still recognizably Ruby. For example:

```
service 'sshd' do
  supports :restart => true, :status => true
  action [:enable, :start]
end
```

Most configuration management tasks can be achieved without delving below the surface of Ruby, but Ruby's full power is available if you need it. You'll appreciate this hidden depth more and more as your comfort with Ruby and Chef increases.

By contrast, Puppet has put in quite a bit of work to be conceptually independent of Ruby and to use it only as an implementation layer. Although the language remains Ruby under the hood and is amenable to the insertion of Ruby code, the Puppet language has its own idiosyncratic structure that is more akin to a declarative system such as YAML than a programming language:

```
service  {
  "ssh":
    ensure => "running",
    enable => "true"
}
```

In our opinion, Puppet hasn't done administrators any favors with this architecture. Instead of letting you leverage your existing knowledge of Ruby (or parlay your Puppet experience into a more general familiarity with Ruby), Puppet just defines its own insulated world.

Dependency management options

No matter how your configuration management system structures its data, the work list for a given client ultimately boils down to a set of operations for the client to execute. Some of those operations have execution-order dependencies, and some don't.

For example, consider the following Ansible tasks for installing a www user account, such as might be used to own the files for a web application:

```
- name: Ensure that www group exists
  group: name=www state=present

- name: Ensure that www user exists
  user: name=www group=www state=present createhome=no
```

We want the www user to have its own dedicated group, also named www. Ansible's `user` module does not create groups automatically, so we must do that in a separate step. And the group creation must precede the creation of the www account; it's an error to specify a nonexistent `group` in a `user` operation.

Ansible executes operations in the order in which they are presented by the configuration, so this configuration snippet works just fine. Chef works this way, too, in part because it's much harder to rearrange code than data. Even if it wanted to, Chef couldn't reliably break your code into pieces and reassemble the pieces as it sees fit.

By contrast, Puppet and Salt allow dependencies to be explicitly declared. For example, in Salt, the equivalent set of states would be

```
www-user:
  user.present:
    - name: www
    - gid: www
    - createhome: false
    - require:
      - www-group

www-group:
  group.present:
    - name: www
```

We inverted the order of the operations here for dramatic effect. But because of the `require` declaration, the operations run in the correct order regardless of how they appear in the source file. The following command applies the configuration:

```
$ sudo salt test-system state.apply order-test
test-system:
----------
          ID: www-group
    Function: group.present
        Name: www
      Result: True
     Comment: Group www is present and up to date
     Started: 23:30:39.825839
    Duration: 3.183 ms
     Changes:
----------
          ID: www-user
    Function: user.present
        Name: www
      Result: True
     Comment: User www is present and up to date
     Started: 23:30:39.829218
    Duration: 27.435 ms
     Changes:

Summary for test-system
------------
Succeeded: 2
Failed:    0
------------
Total states run:     2
```

The require parameter can be added to any operation (“state,” in Salt) to ensure that the named prerequisites run before the current operation. Salt defines several types of dependency relationships, and declarations can appear on either side of a relationship.

Puppet works similarly. It also helps to ease the pain of declaring dependencies by inferring them automatically in some common circumstances. For example, a user configuration that names a particular group automatically becomes dependent on the resource that configures that group. Nice!

So… Why would you want to declare your dependencies explicitly when configuration order seems natural and effortless? Apparently, lots of administrators have been asking this question, as both Salt and Puppet have moved to a hybrid dependency model in which presentation order is significant. However, it’s only a factor within a given configuration file; inter-file dependencies must still be explicitly declared.

The main benefit of declaring dependencies is that it makes configurations more resilient and explicit. The CM system is not obliged to abort the configuration process at the first sign of trouble, because it knows which subsequent operations might be affected by a failure. It can abort one dependency chain while allowing others to continue. Nice, but in our view not a significant payback for the extra work of declaring dependencies.

In theory, a CM system that knows dependency information can parallelize the execution of independent operation chains on a particular host. However, neither Salt nor Puppet attempts this feat.

General comments on Chef

We've seen deployments of the mainline CM packages at organizations of various sizes, and they all display something of a tendency toward entropy. The *Ansible access options* section includes some hints for keeping things organized. However, an even more fundamental rule is to avoid taking on more complexity than is helpful for your environment.

In practice, this means you need to be clear about whether you're living in Chef territory or not. Chef thinks big. To get the most out of Chef, you should have

- Hundreds or thousands of machines under configuration management
- An administrative staff of nonuniform privileges and experience (Chef's internal permissions system and multiple interfaces are quite helpful here)
- Specific reporting, compliance, or regulatory requirements to enforce
- The patience to train new team members without prior Chef experience

Sure, you can run Chef stand-alone on a single machine for free. Nobody's stopping you! But you'll still have to pay the cognitive overhead for many of the enterprise-level features you aren't using. They're baked into the architecture and the documentation.

We like Chef. It's complete, robust, and scalable—more so than the alternatives. But at heart, it's just another configuration management system that does the same basic stuff as Ansible, Salt, and Puppet. Keep Chef in perspective, and resist the temptation to adopt it just "because it's the most powerful" (or "because it uses Ruby," for that matter).

We have found that bringing beginners up to speed with Chef can be a significant challenge, especially for those without prior configuration management experience. Chef requires a developer mindset more than the other systems do. Prior programming experience is helpful.

Chef's attribute precedence system is powerful but can also be a source of frustration. Its peculiar combination of foodie and Internet-meme nomenclature is annoying and uninformative. Resolving dependencies among cookbooks can be challenging; sometimes an upstream dependency breaks, and all your systems develop problems unless you remembered to pin all your dependencies to a particular version.

General comments on Puppet

Puppet is the oldest of the four main CM systems and the one with the largest installed base. It has lots of users, lots of contributed modules, and a free web GUI. Still, it's losing market share fairly steadily to more recent competitors.

As a determinedly middle-of-the-road option, Puppet is under pressure from both ends of the market. It's famous for server-side bottlenecks that cause problems when managing thousands of hosts, and several major Puppet deployments have abandoned it over the last couple of years (most publicly, Lyft, which adopted Salt). These days, such large-scale scenarios seem to be better handled with a tiered Chef or Salt network.

In the arena of small deployments, Ansible and Salt are mounting a serious challenge with their relatively low barriers to entry. As discussed on this page, Puppet is not complex at heart. However, it does drag along some historical baggage that tends to impede newcomers. For example, relatively few operations are built into the Puppet core. Most sites will need to go prospecting for user-contributed modules to complete their basic configurations.

Our subjective impression is that Puppet went through some false starts early in its design and development. Although Puppet has worked hard to correct these issues, history and backward compatibility take an inevitable toll on the current product.

It doesn't help that Puppet transmutes the golden treasure of Ruby into the lump of coal that is the Puppet configuration language. That was probably a sensible decision back in 2005, when Ruby was obscure and Rails had not yet appeared on the scene to propel it to stardom. These days, the Puppet configuration language just seems gratuitous.

None of these issues is a deal breaker, but Puppet seems to have no clear and compelling advantage that might counterbalance such concerns. We are not aware of any bake-off or comparative review written within the last few years in which Puppet emerged as a recommended option.

Of course, if you've inherited an existing Puppet installation, there's no need to start looking for an immediate replacement. Puppet works fine; the distinctions among these systems are mostly a matter of style and marginal advantage.

General comments on Ansible and Salt

Ansible and Salt are both nice systems, and we recommend one of these options for the majority of sites.

We've taken a deeper look at both of these systems in *Introduction to Ansible* and *Introduction to Salt*, which begin here and here, respectively. Each of those sections reviews the system's configuration syntax and the general flavor of day-to-day use.

Ansible and Salt look deceptively similar on the surface, mostly because they both use YAML and Jinja as their default formats. Under the hood, however, they almost couldn't be more different. Accordingly, we defer our head-to-head comparison of Ansible and Salt until this page, once we've discussed them both in a bit more detail.

Before we look into the systems themselves, however, we cast a jaundiced eye on YAML itself.

YAML: a rant

In theory, a YAML document should start with three dashes on a line by themselves, and the Ansible documentation often follows this convention. However, this “start YAML document” line is essentially vestigial. As far as we are aware, it can safely be omitted in all cases.

As mentioned earlier, YAML is just an alternate syntax for JSON. For example, this YAML for Ansible:

```
- name: Install cpdf on cloud servers
  hosts: cloud
  become: yes
  tasks:
    - name: Install OCAML packages
      package: name={{ item }} state=present
      with_items:
        - gmake
        - ocaml
        - ocaml-opam
```

maps to the following JSON:

```
[{
  "name": "Install cpdf on cloud servers",
  "hosts": "cloud",
  "become": "yes",
  "tasks": [{
    "name": "Install OCAML packages",
    "package": {
      "name": "{{ item }}",
      "state": "present",
    },
    "with_items": [ "gmake", "ocaml", "ocaml-opam" ]
  }]
}]
```

In the JSON world, brackets enclose lists and curly braces enclose hashes. A colon separates a hash key from its value. These delimiters can appear directly in YAML, but YAML also understands indentation to indicate structure, much like Python. YAML marks items in a list with a preceding dash.

Take a moment to verify that you understand how the YAML example above maps into JSON, because Ansible and Salt are actually JSON-based worlds. The YAML is just a shorthand. We pick on Ansible below, as its version of YAML is a bit more idiosyncratic, but most of the general points apply to Salt as well. (Once again, Salt partisans will protest that Salt can’t be blamed for YAML and Jinja because it has no actual dependencies on these systems. You’re free

to use any one of a number of alternatives. That's all true. At the same time, it's a lot like saying that you're not responsible for the country's government because you didn't vote.)

Clearly, the YAML version is more readable than the JSON version. The problem isn't YAML per se, but rather the compromises inherent in trying to force data of the complexity found in configuration management systems into the mold of JSON.

YAML is good for representing simple data structures, but it's not a tool that scales well to arbitrary complexity. When cracks appear in the model, they have to be puttied over with a variety of ad hoc fixes.

The example above already contains such a patch. Did you spot it?

```
package: name={{ item }} state=present
```

Ignore the `{{ item }}` part; that's just a Jinja expansion. The crime here is the `name=value` syntax, which is really just a nonstandard shorthand for defining a sub-hash:

```
package:
  name: {{ item }}
  state: present
```

Or is it? Actually not, because Ansible doesn't allow hash values that start with a Jinja expansion. That Jinja term now must sport quotes:

```
package:
  name: "{{ item }}"
  state: present
```

And what if the operation accepts a "free form" argument?

```
- name: Cry for help
  shell: echo "Please, sir, I just want the syntax to be consistent"
  args:
    warn: no
```

Visually, this doesn't look so bad. But think about what's actually going on: `shell` is the operation type, and `warn` is a parameter for `shell` just as `state` is a parameter for `package` in the previous example. So what is that extra `args` dictionary doing there?

Well, `shell` typically has a complex string as its main argument (the shell command to run), so it's been made a special type of operation that accepts a string instead of a parameter hash as its value. The `args` dictionary is actually a property of the task-item wrapper, not the `shell` operation. Its contents are covertly stuffed down into the `shell` operation on your behalf to make the whole construction work.

No problem; keep calm and carry on. But it's a confusing subtlety that muddles a relatively basic example.

The problem isn't this specific scenario. It's the constant drip of edge cases, ambiguities, and compromises that are needed to coerce configuration data into JSON format. Does this particular argument go in the operation? In the state? In the binding? It's all just a big JSON hierarchy, so the answer is rarely obvious.

Look again at the "Install **cpdf** on cloud servers" playbook on this page. Is it obvious that `with_items` should be at the same level as `package` and not at the same level as `name` and `state` (which are in fact logically beneath `package`)? Probably not.

The underlying intent behind the YAML approach is praiseworthy: use an existing format that people already know, and represent configuration information as data instead of code. Still, these systems have syntactic warts that would probably not be permitted in a real programming language.

Despite the looseness of YAML as used in configuration management systems, its specification is actually quite lengthy. In fact, it's longer than the specification for the entire Go programming language.

23.5 Introduction to Ansible

Ansible has no server daemon and installs no software of its own on clients, so it's really just a set of commands (most notably, **ansible-playbook**, **ansible-vault**, and **ansible**) that you install on any system from which you wish to manage clients.

See this page for more information about EPEL.

Standard OS-level packages are widely available for Ansible, although the package names vary from system to system. On RHEL and CentOS, make sure you have the EPEL repository enabled on the master systems. As with most things, OS-specific packages are often somewhat out of date with respect to the trunk. If you don't mind forgoing package management, Ansible is easy to install from the GitHub repository (ansible/ansible) or through **pip**. (**pip** is a package manager for Python. Try **pip install ansible** to pull the latest version from PyPI, the Python Package Index. You might need to install **pip** from your distribution's packaging system first.)

The default location of Ansible's master configuration file is **/etc/ansible/ansible.cfg**. (As with most add-ons, FreeBSD moves the **ansible** directory to **/usr/local/etc**.) The default **ansible.cfg** file is short and sweet. The only change we'd recommend is to add the following lines to the end:

```
[ssh_connection]
pipelining = true
```

---

*See this page to allow **sudo** without a control terminal.*

These lines turn on pipelining, an SSH feature that significantly improves performance. Pipelining requires that **sudo** on clients not be configured to require interactive terminals; however, that is the default.

If you keep your configuration data under **/etc/ansible**, you'll need to use **sudo** to make changes, and you tie yourself to one particular server machine. Alternatively, you can easily set up Ansible for use under your own account. The server just runs **ssh** to reach other systems, so root privileges are unnecessary unless you need to run privileged commands on the server side.

Fortunately, Ansible makes it a snap to combine system-wide and personal configurations. Don't remove the system-wide configuration; just shadow it by creating **~/.ansible.cfg** and setting the location of your inventory file and roles directory:

```
[defaults]
inventory = ./hosts
roles_path = ./roles
```

The inventory is the list of client systems, and roles are bundles that abstract various aspects of client configuration. We return to both of these topics shortly.

Here, we define both locations as relative paths, which assumes that you'll **cd** to your clone of the configuration base and that you'll follow the stated naming conventions. Ansible also understands the shell's ~ notation for home directories if you prefer to use fixed paths. (Ansible allows ~ pretty much everywhere else, too.)

Ansible example

Before we delve into too much more detail, we first look at a small example that demonstrates a few basic Ansible operations.

The following set of steps would set up **sudo** on a new system (as might be required, e.g., on FreeBSD, which does not include **sudo** by default).

1. Install the **sudo** package.
2. Copy a standard **sudoers** file from a server and install it locally.
3. Make sure the **sudoers** file has appropriate permissions and ownerships.
4. Create a UNIX group called “sudo”.
5. Add every system administrator with an account on the local machine to the sudo group.

The Ansible code below implements these steps. Because this code is designed to illustrate several points about Ansible, it isn’t necessarily an example of idiomatic Ansible code.

```
- name: Install sudo package
  package: name=sudo state=present

- name: Install sudoers file
  template:
    dest: "{{ sudoers_path }}"
    src: sudoers.j2
    owner: root
    group: wheel
    mode: 0640

- name: Create sudo group
  group: name=sudo state=present

- name: Get current list of usernames
  shell: "cut -d: -f1 /etc/passwd"
  register: userlist

- name: Add administrators to the sudo group
  user: name={{ item }} groups=sudo append=true
  with_items: "{{ admins }}"
  when: "{{ item in userlist.stdout_lines }}"
```

The statements are applied in order, much as they would be in a shell script.

The expressions enclosed in double curly braces (e.g., `"{{ admins }}"`) are variable expansions. Ansible interpolates facts in a similar manner. This kind of parameter management flexibility is a common characteristic of configuration management systems, and it’s one of their main

advantages over raw scripts. You define the general procedure in one place and the configuration specifics elsewhere. The CM system then collapses the global specification and ensures that the proper parameters are applied to each target host.

The file **sudoers.j2** is a Jinja2 template that expands to become the **sudoers** file on the target machine. The template can consist of static text or it can have internal logic and variable expansions of its own.

Templates are usually kept along with configurations in the same Git repository, allowing for one-stop shopping when configurations are applied. There's no need to maintain a separate file server from which templates can be copied. The configuration management system uses its existing access to the target host to install templates, so credential management need be set up only once.

We had to work around a couple of rough edges. Ansible's user module, used here to add system administrators to the sudo UNIX group, normally ensures that the specified account exists, and it creates the account if it does not. In this scenario, we want to affect only accounts that already exist, so we're forced to manually check for the existence of each account before we permit user to modify it. In a more typical scenario, the configuration management system would be responsible for setting up administrators' accounts as well as **sudo** access. The configuration specifications for both functions would likely refer to the same admins variable, and so there would be no possibility of conflict and no need to validate each account name.

To check for the existence of accounts, the configuration runs the shell command **cut –d: -f1 /etc/passwd** to obtain a list of existing accounts and captures ("registers") the output under the name userlist. It's similar in principle to the **sh** line

```
userlist=$(cut -d: -f1 /etc/passwd)
```

Each account listed in the admins variable (with_items: "{{ admins }}") is considered separately. During its turn, the account name is assigned to the variable item. (The name item is an Ansible convention; the configuration does not specify it.) For each account found within the output of the **cut** command (the when clause), the user clause is invoked.

There's a bit of extra glue we haven't shown that binds this configuration to a particular set of target hosts and that tells Ansible to make the changes as root. When we activate that binding (by running **ansible-playbook example.yml**), Ansible starts working to configure several target hosts in parallel. If any operation fails, Ansible reports the error and stops working on the host that generated it. Other hosts can continue until they're done.

Client setup

Ansible needs three things from each configuration management client:

- SSH access
- **sudo** permission
- A Python 2 interpreter

Ansible does not actually require **sudo** access per se. It's only needed if you want to run privileged operations. But you typically will.

If the client is a Linux cloud server, it may be Ansible-accessible right out of the box. Systems like FreeBSD that don't install **sudo** or Python by default might need a bit more tweaking, but you can do some of the initial bootstrapping through Ansible with `raw` operations, which execute commands remotely without the usual Python wrapper. Or you can just write your own bootstrapping script.

Several choices must be made when Ansible clients are set up. We suggest a reasonable game plan in *Ansible access options*, but for now, let's assume you've created a dedicated "ansible" user on the client, that the appropriate SSH key is in your default set, and that you're willing to enter the **sudo** password by hand.

Clients don't introduce themselves to Ansible, so you need to add them to Ansible's host inventory. By default, the inventory is a single file called **/etc/ansible/hosts**.

One nice feature of Ansible is that you can replace any flat configuration file with a directory of the same name. Ansible then merges the contents of the files the directory contains. This feature is useful for structuring your configuration base, but it's also Ansible's way of incorporating dynamic information: if a particular file is executable, Ansible runs it and captures the output instead of reading the file directly.Actually, Ansible is even smarter than this. It ignores certain file types entirely, e.g., **.ini** files. So not only can you put in scripts, but also configuration files for scripts.

This aggregation feature is so useful and so commonly used that we recommend bypassing the larval flat-file stage of most configuration files and skipping directly to directories. For example, we can define an Ansible client by adding the following line to **/etc/ansible/hosts/static** (or to ~/**hosts/static** within a personal configuration base):

```
new-client.example.com ansible_user=ansible
```

 FreeBSD clients put Python in an unusual location, so you'll need to tell Ansible about that:

```
freebsd.example.com ansible_python_interpreter=/usr/local/bin/python
    ansible_user=ansible
```

This should all be on a single line in the **hosts** file. (On this page we present a much better way to set these variables, but that method is just a generalization of this same idea.)

To check connectivity with a new host, run the setup operation, which returns the client's fact catalog:

```
$ ansible new-client.example.com -m setup
new-client.example.com | SUCCESS => {
  "ansible_facts": {
    "ansible_all_ipv4_addresses": [
      "172.31.25.123"
    ],
    ...
<200+ more lines omitted>
```

The name "setup" is unfortunate, as no explicit setup is actually required. You can go directly to actual configuration operations if you wish. In addition, you can run the setup operation as often as you like to review the client's fact catalog.

Check to be sure that privilege escalation through **sudo** is also working correctly:

```
$ ansible new-client.example.com -a whoami --become --ask-become-pass
SUDO password: <password>
new-client.example.com | SUCCESS | rc=0 >>
root
```

Here, the command operation, which runs shell commands, is the default. We could have said **-m command** explicitly with equivalent results. The **-a** flag introduces operation parameters; in this case, the actual command to execute.

"Becoming" is Ansible's odd locution for privilege escalation; you "become" another user. The "other user" is root by default, but you can specify a different one with the **-u** option. Unfortunately, you have to force Ansible to ask you for the **sudo** password (with **--ask-become-pass**), and it does so regardless of whether the remote system actually prompts for the password.

Client groups

Groups are defined within the **hosts** directory as well, although the syntax can become a bit awkward:

```
client-four.example.com

[webservers]
client-one.example.com
client-two.example.com

[dbservers]
client-one.example.com
client-three.example.com
```

If this doesn't look so bad, that's because we've skirted the main problem areas. The **.ini** format is flat(ish), so some tricks are needed if you want to define hierarchical groups or add extras directly to the **hosts** file (e.g., variable assignments for a group). These features aren't actually that important in practice, however.

Note that we had to list client-four at the top of the file because that host is not included in any groups. We can't just append to the **hosts** file, because that would make client-four a member of the dbservers group, even if we added a blank line as a separator.

This is another reason why configuration directories are helpful. In practice, we'd probably want to put each group definition in a separate file.

Ansible lets you freely intermix client names and group names on command lines and within the configuration base. Neither is specially marked, and both can be subject to globbing. Regular-expression-style matching is also available for both; just start the pattern with a ~. There's also a set algebra notation for combining cohorts of clients in various ways.

For example, the following command uses a globbing expression to select the webservers group. It executes the ping operation on each member of that group.

```
$ ansible 'web*' -m ping
client-one.example.com | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
client-two.example.com | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

Variable assignments

As we saw here, variable values can be assigned within inventory files. But that's gauche; don't do it that way.

Every host and group can have its own collection of variable definitions in YAML format. By default, these definitions are stored under **/etc/ansible/host_vars** and **/etc/ansible/group_vars** in files named for the host or group. You can use a **.yml** suffix if you want; Ansible finds the appropriate files either way.

As with other Ansible configurations, these files can be converted to directories if you'd like to add some additional structure or scripting. Ansible does its usual routine of ignoring configuration files, running scripts, and combining all the results into a final package.

Ansible automatically defines a group called "all" for you. Just like other groups, "all" can have its own group variables. For example, if you standardize on using client accounts named "ansible" for configuration management, that's a good fact to put in the global configuration (here, in, say, **group_vars/all/basics**):

```
ansible_user: ansible
```

If multiple value declarations exist for a variable, Ansible selects a final value according to precedence rules rather than declaration order. Ansible currently has 14 different precedence categories, but the relevant point in this case is that host variables trump group variables.

Conflicts among overlapping groups are resolved at random, which can make for inconsistent behavior and tricky debugging. Try to structure your variable declarations so that there's no possibility for overlaps.

Dynamic and computed client groups

Ansible's grouping system really comes into its own when dynamic scripting is added to the mix. The dynamic inventory scripts used with cloud providers, for example, don't simply list all the available servers. They also slice and dice those servers into ad hoc groupings according to metadata from the cloud.

For example, Amazon's EC2 lets you assign arbitrary tags to each instance. You might assign the tag webserver to every instance that needs an NGINX stack and dbserver to every instance that needs PostgreSQL. The **ec2.py** dynamic inventory script would then create groups named tag_webserver and tag_dbserver. These groups can have their own group variables and can be named in bindings ("playbooks"), just like any other group.

The situation gets murkier when it comes to grouping clients on criteria internal to Ansible, such as the values of facts. You can't do this directly. What you can do instead is target playbooks to broader groups (such as "all") and apply conditional expressions to individual operations which, when the proper conditions do not apply, cause the operation to be skipped.

For example, the following playbook ensures that **/etc/rc.conf** contains a line to configure the hostname on each FreeBSD client:

```
- name: Set hostname at startup on FreeBSD systems
  hosts: all
  tasks:
    - lineinfile:
        dest: /etc/rc.conf
        line: hostname="{{ hostname }}"
        regexp: ^hostname
      when: ansible_os_family == "FreeBSD"
```

(If the last line looks like it needs some {{ }}, your instinct is good. This is actually just a bit of Ansible syntactic sugar to help keep configurations tidy. when clauses are always going to be Jinja expressions, so Ansible surrounds their contents with double braces for you automatically. This feature is helpful, but it's just one of a fairly extensive list of irregularities in Ansible's YAML parsing.)

In this example, every host in inventory is considered for the lineinfile operation. But thanks to the when clause, only FreeBSD hosts actually run it. This approach works fine, but it doesn't make the FreeBSD hosts into a true group. They can't, for example, have a normal **group_vars** entry, although you can simulate the effect with some jury-rigging.

A structurally preferable but slightly more verbose alternative is to use a group_by operation, which runs locally and classifies hosts according to an arbitrary key value for which you designate a template:

```
- name: Group hosts by OS type
  hosts: all
  tasks:
    - group_by: key={{ ansible_os_family }}

- name: Set hostname at startup on FreeBSD systems
  hosts: FreeBSD
  tasks:
    - lineinfile:
        dest: /etc/rc.conf
        line: hostname="{{ hostname }}"
        regexp: ^hostname
```

The basic game plan is similar, but the classification occurs in a separate “play” (Ansible’s term for what we call a binding; see this page). We then start a new play so that we can specify a different set of target hosts, this time using the FreeBSD group that the first play defined for us.

The advantage of using group_by is that we perform the classification only once. We can then hang any number of tasks off the second play with confidence that we’re targeting only the intended clients.

Task lists

Ansible calls operations "tasks," and a collection of tasks in a separate file is called a task list. Like all but a few parts of an Ansible configuration, task lists are just YAML, so the files have a **.yml** suffix.

The binding of task lists to specific hosts is done in higher-level objects called playbooks, which are described here. For now, let's focus on the operations themselves and not worry about how they come to be applied to a particular host.

As an example, we revisit the "install **sudo**" example from this page with a slightly different focus and implementation. This time, we create the administrator accounts from scratch and give each one its own UNIX group of the same name. We then set up a **sudoers** file that lists the administrators explicitly (instead of just assigning privileges to a "sudo" UNIX group).

Some input data is needed to drive these operations: in particular, the location of the **sudoers** file and the names and usernames of administrators. We should put this information in a separate variable file, say, **group_vars/all/admins.yml**:

```
sudoers_path: /etc/sudoers
admins:
  - { username: manny, fullname: Manny Calavera }
  - { username: moe, fullname: Moe Money }
```

The value of `admins` is an array of hashes; we iterate through this array to create all the accounts. Here's what the complete task list would look like:

```
- name: Install sudo package
  package: name=sudo state=present

- name: Create personal groups for admins
  group: name={{ item.username }}
  with_items: "{{ admins }}"

- name: Create admin accounts
  user:
    name: "{{ item.username }}"
    comment: "{{ item.fullname }}"
    group: "{{ item.username }}"
    groups: wheel
  with_items: "{{ admins }}"

- name: Install sudoers file
  template:
    dest: "{{ sudoers_path }}"
    src: templates/sudoers.j2
    owner: root
    group: wheel
    mode: 0600
```

From the perspective of YAML and JSON, the tasks form a list. Each dash at the left margin starts a new task, which is represented by a hash.

In this example, every task has a `name` field that describes its function in English. The names are technically optional, but if you don't include them, Ansible tells you very little about what it's doing when you run the configuration (other than listing the module names: `package`, `group`, etc.).

Each task must have among its keys the name of exactly one operation module. The value of that key is itself a hash that enumerates the operation parameters. Parameters that you do not explicitly set assume default values.

The notation

```
- name: Install sudo package
  package: name=sudo state=present
```

is an Ansible extension to YAML that's essentially equivalent to

```
- name: Install sudo package
  package:
    name: sudo
    state: present
```

There's some potential weirdness here in the case of operations like `shell` that have "freeform" arguments, but we won't rehash that here. See the YAML rant here.

The one-line format is not only more compact, but it also lets you set parameters whose values are Jinja expressions without quotes, as seen in the task that creates personal groups for admins. In the normal syntax, a Jinja expression cannot appear at the start of a value unless the entire value is in quotes. The quoting is benign, but it does add visual noise. Despite appearances, the quotes do not force the value to be a string.

Now we're ready to break out a few of the more notable aspects of this example task list in the sections below.

state parameters

In Ansible, operation modules can often perform several different tasks depending on the `state` you request. For the `package` module, for example, `state=present` installs the package, `state=absent` removes it, and `state=latest` ensures that the package is both present and up to date. Operations often look for different sets of parameters depending on the `state` being invoked.

In a few cases (e.g., the `service` module with `state=restarted`, which restarts a daemon), this model wanders a bit from what might normally be conceived of as a "state," but overall it works well. The `state` can be omitted (as shown here when creating the sudo group), in which case it assumes a default value, usually something positive and empowering such as `present`, `configured`, or `running`.

Iteration

with_items is an iteration construct that repeats a task once for each element it's supplied with. For quick reference, here's another copy of the two tasks in our example that use with_items:

```
- name: Create personal groups for admins
  group: name={{ item.username }}
  with_items: "{{ admins }}"

- name: Create admin accounts
  user:
    name: "{{ item.username }}"
    comment: "{{ item.fullname }}"
    group: "{{ item.username }}"
    groups: wheel
  with_items: "{{ admins }}"
```

Note that with_items is an attribute of the task, not the operation that the task runs.

On each pass through a loop, Ansible sets the value of item to one of the items supplied to with_items. In this case, we assigned the variable admins a list of hashes, so item is always a hash. The notation item.username is shorthand for item['username']. Use whichever you prefer.

Each of these tasks loops through the admins array separately. One pass creates UNIX groups and the other creates user accounts. Although Ansible does define a grouping mechanism for tasks (called a block), that construct unfortunately does not support with_items.

If you really need the effect of a single loop that executes multiple tasks in sequence, you can achieve it by moving the loop body into a separate file and including it into the main task list:

```
- include: sudo-subtasks.yml
  with_items: "{{ admins }}"
```

with_items is not the only loop available in Ansible. There are also loop forms dedicated to iterating over hashes (termed "dictionaries" in Python), over lists of files, and over globbing patterns.

Interaction with Jinja

The Ansible documentation is not very specific about how YAML and Jinja interact, but it's important to understand the details. As constructs like `with_items` demonstrate, Jinja is not simply a preprocessor that's run over a file before it is handed off to YAML (as is the case in Salt). In fact, Ansible parses YAML with Jinja expressions intact. It then Jinja-expands each string value immediately before use. Parameters of iterated operations are reevaluated during each iteration.

Jinja has control structures of its own, including loops and conditionals. However, they are inherently incompatible with Ansible's delayed-evaluation architecture, and so they are not allowed in Ansible's YAML files (although they can be used in templates). Ansible constructs such as `when` and `with_items` are not just window dressing for the equivalent Jinja. They represent a rather different approach to structuring the configuration.

Template rendering

Ansible uses the Jinja2 template language both to add dynamic features to YAML files and to flesh out configuration file templates installed by the `template` module. We use a template in this example to set up the **sudoers** file. Here are the variable definitions again for reference:

```
sudoers_path: /etc/sudoers
admins:
  - { username: manny, fullname: Manny Calavera }
  - { username: moe, fullname: Moe Money }
```

And the task code:

```
- name: Install sudoers file
  template:
    dest: "{{ sudoers_path }}"
    src: templates/sudoers.j2
    owner: root
    group: wheel
    mode: 0600
```

The file **sudoers.j2** is a mix of plain text and Jinja2 code for the dynamic bits. For example, here's a skeletal example that gives "**sudo** ALL" privileges to each admin:

```
Defaults env_keep += "HOME"

{% for admin in admins %}
{{ admin }} ALL=(ALL) ALL
{% endfor %}
```

The `for` loop wrapped by `{% %}` is Jinja2 syntax. Unfortunately, you can't indent loop bodies sensibly as you might in a real programming language, because doing so would cause the output of the template to be indented as well.

The expanded version looks like this:

```
Defaults env_keep += "HOME"

manny ALL=(ALL) ALL
moe ALL=(ALL) ALL
```

Note that variable values automatically flow through to templates. The values are available to configuration files under exactly the same names used to define them; no prefix or additional hierarchy is imposed. Autodiscovered fact variables are in the top-level namespace, too, but to forestall potential name conflicts they all begin with the prefix `ansible_`.

Ansible's module for installing static files is called `copy`. However, you may as well treat all configuration files as templates, even if their contents initially consist of static text. You can then

add customizations in the future without having to touch the configuration code; just edit the template. Reserve `copy` for binary files and for static files that will never need expansion, such as public keys.

Bindings: plays and playbooks

Bindings are the mechanism through which tasks become associated with sets of client machines. Ansible's binding object is called a play. Here's a simple example:

```
- name: Make sure NGINX is installed on web servers
  hosts: webservers
  tasks:
    - package: name=nginx state=present
```

Just as multiple tasks can be concatenated to form a task list, multiple plays in sequence form a "playbook."

As in other systems, the basic elements of a binding are a set of hosts and a set of tasks. However, Ansible's system allows several additional options to be specified at the play level. They're listed in Table 23.3.

Table 23.3: Ansible play elements

Key	Format	What it specifies
`name`	string	Name to print out when executing the play, optional
`hosts`	list, string	Client systems on which to run associated tasks and roles
`vars`	hash	Variable values to set for the scope of this play
`vars_files`	list	Files from which to read variable values
`become*`	strings	Privilege escalation (e.g., **sudo**) options
`tags`	list	Categories for selective execution
`tasks`	list	Operations to run; may include separate files
`handlers`	list	Operations to run in response to `notify`
`roles`	list	Bundles (roles) to invoke for these hosts

The biggies here are the variable-related options, not so much because they appear in plays per se, but because they're available pretty much anywhere—even when executing `include`s. Ansible can activate the same task list or playbook again and again with different sets of variable values. It's a lot like defining a function (e.g., "make a user account") and then calling it with different sets of arguments.

Ansible formalizes this system in its implementation of bundles (called "roles"), which we discuss here. Roles are powerful, but under the hood, they're just a set of standardized conventions for doing includes, so they're also easy to understand.

Here's a simple play that demonstrates the use of handlers:

```
- name: Update cow-clicker web app
  hosts: clickera,clickerb
  tasks:
    - name: rsync app files to /srv
      synchronize:
        mode: pull
        src: web-repo:~sites/cow-clicker
        dest: /srv/cow-clicker
        notify: restart nginx
  handlers:
    - name: restart nginx
      service: name=nginx state=restarted
```

This playbook runs on hosts clickera and clickerb. It mirrors files from a central (local) repository by running **rsync** (using the synchronize module), then restarts the NGINX web server if any updates were made.

When a task with a notify clause makes changes to the system, Ansible runs the handler of the requested name. Handlers themselves are just tasks, but they're declared in a separate section of the play.

Playbooks are the primary unit of execution in Ansible. You run them with **ansible-playbook**:

```
$ ansible-playbook global.yml --ask-sudo-pass
```

Ansible approaches multihost execution task by task. As it reads a playbook, each task is run in parallel on the targeted hosts. When every host has completed the task, Ansible continues to the next task. By default, Ansible runs tasks simultaneously on up to five hosts, but you can set a different limit with the **-f** flag.

When debugging problems, it's often helpful to include the **-vvvv** argument to increase the amount of debugging output. You'll see the exact commands that are executed on the remote system and their detailed responses.

Roles

As we described generically starting here, bundles (our term) are the packaging mechanism defined by a CM system to facilitate reuse and sharing of configuration fragments.

Ansible calls its bundles "roles," and they are in fact nothing but a structured system of include operations and variable precedence rules. They make it easy to put the variable definitions, task lists, and templates associated with a configuration into a single directory, making them readily available for reuse and sharing.

Each role is a subdirectory of a directory called **roles** that's normally found at the top level of your configuration base. You can also add site-wide role directories by setting the roles_path variable in **ansible.cfg**, as shown here. All known role directories are searched whenever you include a role in a playbook.

Role directories can have the subdirectories shown in Table 23.4.

Table 23.4: Subdirectories of an Ansible role

Subdir	Contents
defaults	Default values for variables (overridable)
vars	Variable definitions (not overridable, but can reference overrides)
tasks	Tasks lists (sets of operations)
handlers	Operations that respond to notifications
files	Data files (typically used as a source for `copy` operations)
templates	Templates to be processed by Jinja before installation
meta	List of bundles to run in preparation for this bundle

Roles are invoked through playbooks and nowhere else. Ansible looks for a file called **main.yml** within each of the role's subdirectories. If it exists, the contents are automatically incorporated into any playbook that invokes the role. For example, the playbook

```
- name: Set up cow-clicker app throughout East region
  hosts: web-servers-east
  roles:
    - cow-clicker
```

is roughly equivalent to

```
- name: Set up cow-clicker app throughout East region
  hosts: web-servers-east
  vars_files:
    - roles/cow-clicker/defaults/main.yml
    - roles/cow-clicker/vars/main.yml
  tasks:
    - include: roles/cow-clicker/tasks/main.yml
  handlers:
    - include: roles/cow-clicker/handlers/main.yml
```

However, variable values from the **default** folder do not override values that have already been set. In addition, Ansible makes it easy to refer to files from the **files** and **templates** directories, and it sub-includes any roles mentioned as dependencies in the **meta/main.yml** file.

Files other than **main.yml** are ignored by the roles system, so you can break the configuration into whatever pieces are appropriate and just include those parts into **main.yml**.

Ansible lets you pass a set of variable values to a particular instance of a role. In effect, this makes the role act as a sort of parameterized function. For example, you might define a bundle that's used to deploy a Rails app. You could invoke that bundle several times within a playbook, supplying the parameters of a different app for each invocation:

```
- name: Install ULSAH Rails apps
  hosts: ulsah-server
  roles:
    - { role: rails_app, app_name: ulsah-reviews }
    - { role: rails_app, app_name: admin-com }
```

In this example, the rails_app role would probably depend on a role for nginx or some other web server, so it would not be necessary to mention the web server role explicitly. If you wanted to customize the web server installation, you could simply include the appropriate variable values in the rails_app invocation, and those values would be propagated downward.

Ansible's public role repository is located at galaxy.ansible.com. You can search for roles with the **ansible-galaxy** command, but you're better off using the web site. It lets you sort by rating or download count, and you can easily click through to the GitHub repo that hosts the actual code for each role. Several roles are usually available to address most common scenarios, so it's worth examining the code to determine which version will serve your needs best.

Once you've settled on a role implementation, copy the files to your **roles** directory by running **ansible-galaxy install**. For example:

```
$ ansible-galaxy install ANXS.postgresql
- downloading role 'postgresql', owned by ANXS
- downloading role from https://github.com/ANXS/postgresql/v1.4.0.tar.gz
- extracting ANXS.postgresql to /etc/ansible/roles/ANXS.postgresql
- ANXS.postgresql was installed successfully
```

Recommendations for structuring the configuration base

Most configuration bases are organized hierarchically. That is, various pieces of the configuration feed into a master playbook that controls the global state. However, you can also define task-specific playbooks that are unrelated to the global scheme.

Try to keep task lists and handlers out of playbook files. Instead, put them in separate files and interpolate them with `include`. This structure makes a clean separation between bindings and behavior, and it puts all tasks on an equal footing. For extra style points, avoid freestanding task lists entirely and standardize on roles.

It's sometimes recommended that a single playbook should cover all the tasks that relate to each logically distinct group of hosts. For example, all the roles and tasks that relate to web servers should be included in a single **webserver.yml** playbook. This approach avoids replication of host groups and provides a clear locus of control for each host group.

On the other hand, following this rule means that there's no direct way to run a portion of the global configuration, even for debugging. Ansible can run only playbooks; there is no simple command that runs a specific task list on a given machine.

The official solution for this issue is tagging, which works fine but requires some setup. You can include a `tags` field in or above any task to classify it. At the command line, use **ansible-playbook**'s **-t** option to specify the subset of tags you want to run. In most debugging scenarios, you'll also want to use the **-l** option to limit execution to a specified test host.

Assign tags at as high a level as you can within the configuration hierarchy. Under normal circumstances, you should have no temptation to assign tags to individual tasks. (If you do, it may be a sign that the particular task list should be split up.) Instead, attach `tags` to the `include` or `roles` clause that incorporates a specific task list or role into the configuration. The tags then cover all the included tasks.

Alternatively, you can just construct scratch playbooks that run parts of the configuration base on a test host. Setting up these scratch playbooks is a minor annoyance, but so is tagging.

Ansible access options

Ansible needs SSH and **sudo** access on every client system, which sounds straightforward and familiar until you consider that the configuration management system holds the master keys to the entire organization. It's hard for daemon-based systems to be more secure than the root account on the configuration server, but Ansible can potentially do better than this with some thoughtful planning.

For simplicity, it's best if SSH access is funneled through a dedicated account such as "ansible" that has the same name on each client. That account should use a simple shell and should have a minimal dot-file configuration.

On cloud servers, you can use a standard bootstrapping account (such as ec2-user on EC2) for Ansible control. Just make sure that after the initial setup, the account has been properly locked down and does not allow, e.g., **su** to root without a password.

You have some flexibility regarding the actual security design. But keep the following points in mind:

- Ansible needs one credential (password or private key) to gain access to a remote system, and another to escalate privileges with **sudo**. Proper security hygiene suggests that these be separate credentials. A single compromised credential should not grant an intruder root access to a target machine.
- If both credentials are stored in the same place with the same form of protection (encryption, file permissions), they are effectively a single credential.
- Credentials can be reused on machines that are peers (e.g., web servers in a farm), but it should not be possible to use credentials from one server to access a more sensitive —or even substantially different—server.
- Ansible has transparent support for encrypted data through the **ansible-vault** command, but only if the data is contained in a YAML or **.ini** file.
- Administrators can remember only a few passwords.
- It's unreasonable to demand more than one password for a given operation.

Some sites set up client-side "ansible" accounts with the NOPASSWD option in the **sudoers** file, such that no password is required for the ansible account to run **sudo**. This is a terribly insecure configuration. If you can't bring yourself to type a password, at least install the PAM SSH agent module and require a forwarded SSH key for **sudo** access. See this page for more information about PAM.

Sites will arrive at their own tradeoffs, but we suggest the following system as a robust but usable baseline that conforms to these guidelines:

- SSH access is controlled by key pairs that are used by Ansible only.
- Password-based SSH access is prohibited on client systems (by setting `PasswordAuthentication no` in **/etc/ssh/sshd_config**).
- SSH private keys are protected by a passphrase (set with **ssh-keygen -p**). All private keys have the same passphrase.
- Private SSH keys are kept in a known location on the Ansible master machine. They do not live within the configuration base, and administrators agree not to copy them elsewhere.
- Remote accounts ("ansible" accounts) have random UNIX passwords that are listed in the configuration base in encrypted form. All of them are encrypted with the same passphrase, but it's different from the passphrase used for SSH private keys. You will need to add some Ansible glue to make sure the right passwords are used with the right client hosts.

In this scheme, both sets of credentials are encrypted, which makes them resistant to simple violations of file permissions. This layer of indirection also lets you change the master passphrases easily without changing the underlying keys.

Administrators need remember only two passphrases: the passphrase that gives access to SSH private keys, and the Ansible vault password, which allows Ansible to decrypt the host-specific **sudo**-passwords (as well as any other confidential information included in your configuration base).

If you require more granularity for administrator permissions (which is likely), you can encrypt multiple sets of credentials with different passphrases. If the sets are cumulative (as opposed to disjoint), no individual administrator needs to remember more than two passphrases.

---

*See this page for more details on **ssh-agent**.*

It's assumed in this system that administrators will use **ssh-agent** to manage access to private keys. All keys can be activated with a single **ssh-add** command, and the SSH password need be entered only once per session. To work on a system other than the usual Ansible master, admins can use SSH's `ForwardAgent` option to tunnel keys through to the machine on which work is being done. All other security information is included in the configuration base itself.

It's true that **ssh-agent** and key forwarding are only as secure as the machines on which they run. (Less so, really: like **sudo** with a grace period, they are only as secure as your personal account.) However, the risk is mitigated by limits on time and context. Use the **-t** argument to **ssh-agent** or **ssh-add** to cap the lifetime of activated keys, and terminate connections that have access to forwarded keys once you are no longer using them.

If possible, private keys should never be deployed onto client systems. If clients need privileged access to controlled resources (e.g., to clone a controlled Git repository), use the proxying features built into SSH and Ansible, or use **ssh-agent** to make private keys temporarily available to the client without copying them.

For some reason, Ansible cannot currently recognize encrypted files in the configuration base and prompt you to enter the passphrase for decryption. You have to force its hand with the **--ask-vault-pass** argument to the **ansible-playbook** and **ansible** commands. There's a **--vault-password-file** option available for noninteractive use, but of course, that reduces security. If you decide to use a password file, it should be accessible only to the dedicated ansible account.

23.6 Introduction to Salt

Out in the world, you might see Salt referred to as Salt, SaltStack, or Salt Open. These terms are essentially interchangeable. The vendor's name is SaltStack, and they use SaltStack as a generic term to refer to the complete product line, which includes some enterprise add-ons that we don't discuss in this book. However, many people call the open source system SaltStack, too.

Salt Open is a more recently introduced name that designates only the open source components of Salt. But currently, that name doesn't seem to be used anywhere outside of saltstack.com.

SaltStack maintains its own package repository at repo.saltstack.com which hosts up-to-date packages for every Linux packaging system. See the web site for instructions on how to add the repo to your configuration. Some distributions include free-range Salt packages of their own, but it's generally best to go directly to the source.

You'll need the **salt-master** package on the configuration server (the "master"). If you have any dealings with cloud providers, also install the **salt-cloud** package. It wraps a variety of cloud providers into a standard interface and simplifies the process of creating new cloud servers to be managed through Salt. It's essentially similar to cloud providers' native CLI tools, but it handles machines at both the Salt and cloud layers. New machines are automatically bootstrapped, enrolled, and approved. Deleted machines are removed from Salt as well as the provider's cloud.



SaltStack doesn't host a package repo for FreeBSD, but it is a supported platform. The web installer is FreeBSD aware:

```
$ curl -L https://bootstrap.saltstack.com -o /tmp/saltboot
$ sudo sh /tmp/saltboot -P -M
```

By default, the web installer installs client-side software as well as the master server. If you don't want that, pass the **-N** option to **saltboot**.

See Chapter 25 for more information about containers.

Salt's configuration files go in **/etc/salt**, both on the master server and on clients ("minions"). It's theoretically possible to run the server daemon as an unprivileged user, but that requires manually **chown**ing a bunch of system directories that Salt expects to interact with. If you're tempted to head down this road, you're probably better off using a containerized version of the server or saving the configuration into a pre-baked machine image.

Salt has a simple access control system that you can configure to allow unprivileged users to initiate Salt operations on minions. However, you must do manual permission hacking similar to that required for nonroot operation. Considering that the master has direct root access to all

minions, we find this feature rather suspect from a security perspective. If you do use it, keep a tight lid on the permissions that are granted.

Salt maintains a separation between configuration files that set variable values (the “pillar”) and configuration files that define operations (“states”). The distinction goes all the way to the top: you must set separate locations for these configuration hierarchies. They both default to living under **/srv**, which is equivalent to the following **/etc/salt/master** file:

```
file_roots:
  base:
    - /srv/salt

pillar_roots:
  base:
    - /srv/pillar
```

Here, base is a required common environment on top of which additional environments (e.g., development) can be layered. Variable definitions go in the **/srv/pillar** root, and everything else lives in **/srv/salt**.

Note that the paths themselves are list elements, since they’re prefixed with dashes. You can include multiple directories, which makes the **salt-master** daemon serve a merged view of the listed directories to minions. This is a useful feature when you are organizing a large configuration base, since it permits you to add structure that Salt wouldn’t natively understand.

Typically, you’ll want to manage the configuration base as a single Git repository that includes both the **salt** and **pillar** subdirectories. This isn’t a good fit for the default layout because it means that **/srv** would be the repo root; consider moving everything down a level to **/srv/salt/salt** and **/srv/salt/pillar**.

The Salt documentation doesn’t do a very good job of explaining why the pillar and the states have to be completely separate, but in fact this distinction is central to Salt’s architecture. The **salt-master** daemon doesn’t pay the slightest attention to state files; it simply makes them available to minions, who are responsible for parsing and executing them.

The pillar is entirely different. It’s evaluated on the master and propagated to minions as a single, unified JSON hierarchy. Each minion sees a different view of the pillar, but none of them can see the implementation machinery behind these views.

In part, this is a security measure: Salt makes a strong guarantee that minions cannot access each other’s pillars. It’s also a data-sourcing distinction, as dynamic pillar content always originates from the master. This makes for a nice complementarity with grains (Salt’s version of facts), which originate on minions.

See this page for more information about network firewalls.

Salt's communication bus uses TCP ports 4505 and 4506 on the server. Make sure these ports are allowed through any firewalls or packet filters that lie between the server and the prospective clients. The clients themselves do not accept network connections, so this step needs to be done only once, for the server.

When first investigating Salt, you might find it informative to run **salt-master -l debug** in a terminal window (instead of as a system service). This makes **salt-master** run in the foreground and print out activity on Salt's communication bus as it occurs.

Minion setup

As on the master side, you have a choice of native packages from SaltStack's repo or a universal bootstrap script. The repo is hardly worth fussing with on minions, so we recommend the latter:

```
$ curl -o /tmp/saltboot -sL https://bootstrap.saltstack.com
$ sudo sh /tmp/saltboot -P
```

The bootstrap script works on any supported system. On systems without **curl**, **wget** and **fetch** also work fine. See the saltstack/salt-bootstrap repository on GitHub for specific installation scenarios and source code.

For production systems that are started automatically, minimize your exposure to external events by downloading a locally cached version of the boot script. Install a specific version of the Salt client, also from a local cache, or preload it on the machine image. Run the boot script with a **-h** option to see all the options it supports.

See Chapter 16 for more information about DNS.

By default, the **salt-minion** daemon tries to register itself with a master machine named "salt". (This "magic name" system was first popularized by Puppet.) You can use DNS wizardry to make the name resolve appropriately, or you can set an explicit master in **/etc/salt/minion** (**/usr/local/etc/salt/minion** on FreeBSD):

```
master: salt.example.com
```

Restart **salt-minion** after modifying this file (usually, **service salt_minion restart**, note the underscore rather than a dash).

salt-master accepts client registrations from any random machine that can reach it, but you must approve each client with the **salt-key** command on the master configuration server before it becomes active:

```
$ sudo salt-key -l unaccepted
Unaccepted Keys:
new-client.example.com

# If everything looks good, accept all pending keys

$ sudo salt-key -yA
The following keys are going to be accepted:
Unaccepted Keys:
new-client.example.com
Key for minion new-client.example.com accepted.
```

You can now check connectivity from the server with the test module:

```
$ sudo salt new-client.example.com test.ping
new-client.example.com:
    True
```

In this example, new-client.example.com looks suspiciously like a hostname, but it really isn't. It's just the machine's Salt ID, a string that defaults to the hostname but can be set to anything you like in the client's **/etc/salt/minion** file:

```
master: salt.example.com
id: new-client.example.com
```

IDs and IP addresses have nothing to do with each other. For example, even if 52.24.149.191 were the client's actual IP address, you could not directly target it that way with Salt commands:

```
$ sudo salt 52.24.149.191 test.ping
No minions matched the target. No command was sent, no jid was assigned.
ERROR: No return received
```

(Of course, you *can* do IP-based matching. It just has to be explicit. See this page.)

Variable value binding for minions

As we saw in the server setup section, Salt has separate filesystem hierarchies for state bindings and variable-value bindings (the "pillar"). Each of these directory trees has a **top.sls** file at the root that binds groups of minions to files within the tree. The two **top.sls** files both use the same layout. (**.sls** is just Salt's standard extension for YAML files.)

As an example, here's the layout of a simple Salt configuration base that shows both the **salt** and **pillar** roots:

```
$ tree /srv/salt
/srv/salt
├── salt
│   ├── top.sls
│   ├── hostname.sls
│   ├── bootstrap.sls
│   ├── sshd.sls
│   └── baseline.sls
└── pillar
    ├── top.sls
    ├── baseline.sls
    ├── webserver.sls
    └── freebsd.sls

2 directories, 9 files
```

To bind the variables defined in **pillar/baseline.sls** and **pillar/freebsd.sls** to our example client, we could include the following lines in **pillar/top.sls**:

```
base:
  new-client.example.com:
    - baseline
    - freebsd
```

As in the **master** file, `base` is a required, common environment that can be overlaid in more sophisticated setups. See this page for more about this.

It's possible for **baseline.sls** and **freebsd.sls** to define some of the same variable values. For scalar and array values, the last source listed in **top.sls** is the one that takes effect. Hashes, however, are merged.

For example, if a minion binds to one variable file that looks like this:

```
admin-users:
  manny:
    uid: 724
  moe:
    uid: 740
```

and one that looks like this:

```
admin-users:
  jack:
    uid: 1004
```

then Salt merges the two versions.

The pillar data presented to minions is

```
admin-users:
  manny:
    uid: 724
  moe:
    uid: 740
  jack:
    uid: 1004
```

Minion matching

In the scenario above, what we probably want is to apply **baseline.sls** to all clients, and to apply **freebsd.sls** to all clients that are running FreeBSD. Here's how we can do that with selection patterns in the **pillar/top.sls** file:

```
base:
  '*.example.com':
    - baseline
  'G@os:FreeBSD':
    - freebsd
```

The star matches all client IDs in example.com. We could have just used '*' here, but we wanted to emphasize that it's a globbing pattern. The G@ prefix requests a match on grain values. The grain being inspected is named os, and the value sought is FreeBSD. Globbing is allowed here, too.

A less magical way to write the matching expression for FreeBSD would be

```
  'os:FreeBSD':
    - match: grain
    - freebsd
```

The choice is up to you, but the @ notation expands cleanly to complex expressions that involve parentheses and Boolean operations. Table 23.5 lists most of the common matching types, although a few have been omitted.

Table 23.5: Salt minion match types

Code	Target	Match type	match:	Example
–[a]	ID	glob	glob	*.cloud.example.com
E	ID	regex	pcre	E@(nw|wc)-link-\d+
L	ID	list	list	L@hosta,hostb,hostc[b]
G	grain	glob	grain	G@domain:*.example.com
E	grain	regex	grain_pcre	E@virtual:(xen|VMWare)
I	pillar	glob	pillar	I@scaling_type:autoscale
J	pillar	regex	pillar_pcre	J@server-class:(web|database)
S	IP address	CIDR block	ipcidr	S@52.24.9/20
–[c]	compound	compound	compound	not G@os_family:RedHat

a. This is the default. No match-type code is needed (or defined).
b. Note the lack of spaces; individual expressions can't include them.
c. Codes are used to label individual terms.

If Table 23.5 looks disturbingly complex, take heart; these are just options. Real-world selectors look a lot more like our simple examples.

If you're wondering what all those grains or pillar values are that you can match against, it's easy to find out. Just use

```
$ sudo salt minion grains.items
```

or

```
$ sudo salt minion pillar.items
```

to obtain a complete list.

You can define named groups in the **/etc/salt/master** file. They're called nodegroups, and they are useful for moving complex group selectors out of **top.sls** files. However, they're not really a true grouping mechanism so much as a way to name patterns for reuse. As a result, their behavior is a bit squirrelly. They can only be defined in terms of compound-type selectors (not, for example, by a simple list of clients, unless you use an L@ clause), and you must use an explicit match: type of nodegroup to match against them. There's no global shorthand notation.

Salt states

Salt operations are called “states.” As in Ansible, they’re defined in YAML format, and in fact they look vaguely similar to Ansible tasks. However, the fine-grained details are quite different. You can include a series of state definitions in a **.sls** file.

States are bound to specific minions in the **top.sls** file at the root of the **salt** arm of the configuration base. This file looks and functions exactly like the **top.sls** file for variable bindings; see the examples here.

Take a look at the following Salt version of the same example we worked through with Ansible starting on this page: we install **sudo** and create a corresponding sudo group to which we assign administrators who should have **sudo** privileges. We then create a group of administrator accounts, each of which has its own UNIX group of the same name. Finally, we then copy in a **sudoers** file from the configuration base.

As it happens, we can use exactly the same variable file for Salt that we used for Ansible:

```
sudoers_path: /etc/sudoers
admins:
  - { username: manny, fullname: Manny Calavera }
  - { username: moe, fullname: Moe Money }
```

To make these definitions available to all minions, we put them in the configuration base at **pillar/example.sls** and add a binding to **top.sls**:

```
base:
  '*':
    - example
```

Here’s a Salt version of the operations:

```
install-sudo-package:
  pkg.installed:
    - name: sudo
    - refresh: true

create-sudo-group:
  group.present:
    - name: sudo

{% for admin in pillar.admins %}

create-group-{{ admin.username }}:
  group.present:
    - name: {{ admin.username }}

create-user-{{ admin.username }}:
  user.present:
    - name: {{ admin.username }}
    - gid: {{ admin.username }}
    - groups: [ wheel, sudo ]
    - fullname: {{ admin.fullname }}

{% endfor %}

    - mode: '0600'

    - user: root
    - name: {{ pillar.sudoers_path }}
    - group: wheel
install-sudoers-file:
    - source: salt://files/sudoers
  file.managed:
```

This version shows the operations in their most canonical form for easier comparison with the equivalent Ansible task list that starts here. We can make a few additional changes to clean things up a bit, but first, a look at this longer version.

Salt and Jinja

The first thing to notice is that the file includes a Jinja loop delimited by {% and %}. These delimiters are similar to {{ and }} except that {% and %} do not return values. The contents of the loop are interpolated into the YAML file as many times as the loop runs.

See this page for general information about Python.

Although Jinja uses Python-like syntax, YAML already "owns" the indentation in a **.sls** file, so Jinja is forced to define block-ending tokens such as endfor. In straight Python, blocks would normally be defined through indentation.

Salt defines only a rudimentary iteration construct in its basic YAML scheme (see the comments regarding names here). Conditionals and robust iteration have to be provided by Jinja, or by whatever template language the **.sls** file is run through. (In fact, Salt does not care about YAML, either. It just expands configuration files through a designated pipeline and consumes the final JSON output, which must be fully literal.)

On one hand, this approach is clean. There's no conceptual ambiguity about what's going on, and it's easy to examine an expanded **.sls** file to make sure it means what you intended. On the other hand, it means you'll be using Jinja to provide any logic required by your configuration. The mix of templating code and YAML can easily become somewhat dazzling. It's a bit like writing the logic of a web app using only HTML templates.

Several rules of thumb can help keep Salt configurations tidy. First, Salt has usable and well-defined mechanisms for implementing variable-value overlays. Use these to keep as much configuration as possible in the domain of data rather than code.

Many examples in the Salt documentation use Jinja conditionals when they aren't the best solution, for example. (In fairness, the examples are usually designed to illustrate some point other than general tidiness.) The following **.sls** file installs the Apache web server, which has different package names on different distributions:

```
# apache-pkg.sls
apache:
  pkg.installed:
    {% if grains['os'] == 'RedHat' %}
    - name: httpd
    {% elif grains['os'] == 'Ubuntu' %}
    - name: apache2
    {% endif %}
```

This variation could be dealt with more elegantly through the pillar:

```
# apache-pkg.sls
{{ pillar['apache-pkg'] }}:
  pkg.installed

# pillar/top.sls
base:
  '*':
    - defaults
  'G@os:Ubuntu':
    - ubuntu

# pillar/defaults.sls
apache-pkg: httpd

# pillar/ubuntu.sls
apache-pkg: apache2
```

Although replacing one file with four might not initially seem like a simplification, it's now an extensible and code-free system. Multi-OS environments will encounter many such variations, and they can all be dealt with in one place.

If a value has to be dynamically calculated, consider whether you can put the code at the top of the **.sls** file and simply memorialize it for later use in a variable. For example, another way to write the Apache package installation above would be

```
{% set pkg_name = 'httpd' %}
{% if grains['os'] == 'Ubuntu' %}
  {% set pkg_name = 'apache2' %}
{% endif %}

{{ pkg_name }}:
  pkg.installed:
```

This at least has the advantage of separating the Jinja logic from the actual configuration.

If you must intermix Jinja logic with YAML, consider whether you can break out some of the YAML segments into separate files. You can then interpolate these segments as appropriate. Once again, the idea is simply to separate the code and YAML rather than alternating back and forth between them.

For nontrivial calculations, you can abandon YAML altogether and replace it with pure Python, or with one of the Python-based DSLs that Salt includes by default. See the Salt documentation for "renderers" for more information.

State IDs and dependencies

To return to our **sudo** example from this page, here are its first two states again for reference:

```
install-sudo-package:
  pkg.installed:
    - name: sudo
    - refresh: true

create-sudo-group:
  group.present:
    - name: sudo
```

You can see that the individual states are not items in a list (as they are in Ansible) but rather the elements of a hash. The hash key for each state is an arbitrary string called the ID. As usual with hashes, IDs must be unique or they'll collide.

But wait! The potential domain for collisions is not just this particular file, but the entire client configuration. State IDs must be globally unique, because Salt is eventually going to stuff them all together into one big hash.

It's a bit of a funny hash, though, because it preserves the order of keys. In a standard hash, keys emerge in random order when the hash is enumerated. That's the way that Salt used to work, too, and as a result, all dependencies among states had to be explicitly declared. These days, the hash preserves the order of presentation by default, although that can still be overridden if explicit dependencies are declared (or if this behavior is turned off in the **master** file).

There's still some trickiness, though. In the absence of other constraints, order of execution conforms to the original **.sls** files. However, Salt still presumes that states are not logically dependent on one another unless you say so. If a state fails to execute, Salt notes the error but then continues and runs the next state.

If you want a dependent state not to run if its ancestors fail, you can declare that explicitly. For example:

```
create-sudo-group:
  group.present:
    - name: sudo
    - require:
      - install-sudo-package
```

In this configuration, Salt won't try to create a sudo group unless the **sudo** package was successfully installed.

Requisites also come into play when ordering states from multiple files. Unlike Ansible, Salt does not interpolate the contents of an include file at the point the include was encountered. It simply adds the file to its to-read list. If multiple files attempt to include the same source, there is still be

only one copy of the source in the final assembly, and the order of states might not be what you expected. In-order execution is guaranteed only within a file; if any states depend on externally defined operations, they must declare explicit requisites.

The requisite mechanism is also used to achieve an effect analogous to Ansible's notifications. Actually, a handful of alternatives to `require` are syntactically interchangeable with it but imply subtle shadings of behavior. One of those, `watch`, is particularly useful for doing things when another state makes changes to the system.

For example, the following configuration sets the system's time zone and the arguments to be passed to **ntpd** when it starts up. This configuration always makes sure that **ntpd** is running and configured to start at boot time. In addition, it restarts **ntpd** if either the system time zone or the **ntpd** flags are updated.

```
set-timezone:
  timezone.system:
    - name: America/Los_Angeles

set-ntpd-opts:
  augeas.change:
    - context: /files/etc/rc.conf
    - lens: shellvars.lns
    - changes:
      - set ntpd_flags '"-g"'

ntpd:
  service.running:
    - enable: true
    - watch:
      - set-ntpd-opts
      - set-timezone
```

Augeas is a tool that understands many different file formats and facilitates automated changes.

State and execution functions

In a **.sls** file, the names that appear directly under state IDs are the operations those states should run. Some specific cases from our example scenario are pkg.installed and group.present.

These names include both a “module” part and a “function” part. Together, they are roughly analogous to an Ansible module name together with a state value. For example, Ansible uses a package module with state=present for installing packages, whereas Salt uses a dedicated pkg.installed function within the pkg module.

Salt makes a big whoop-de-do of distinguishing operations that do things to target systems (“execution functions”) from those that idempotently enforce a particular configuration (“state functions”). State functions usually call their associated execution functions when they need to make changes.

The general idea is that only state functions should be mentioned in **.sls** files, and only execution functions should appear on command lines. Salt primly enforces these rules, sometimes to confusing effect.

State and execution functions live in separate Python modules, but related modules usually share the same name. For example, there’s both a timezone state module and a timezone execution module. There can’t be any overlap in function names between the two modules, though, because that would create ambiguity. The end result is that to set the time zone from a **.sls** file, you must use timezone.system:

```
set-timezone:
  timezone.system:
    - name: America/Los_Angeles
```

But to set a minion’s time zone from the command line, you use timezone.set_zone:

```
$ sudo salt minion timezone.set_zone America/Los_Angeles
```

If you get it wrong and need to consult the documentation, you’ll find the two halves of timezone in different sections of the manual. It’s also not always clear from behavior exactly which type of function is which. For example, git.config_set, which sets Git repository options, is a state function, but state.apply, which idempotently enforces configurations, is an execution function.

Ultimately, you just have to know which functions are which and the contexts to which they belong. If you need to call a function from the “wrong” context—which is sometimes necessary—you can use the adapter functions module.run (runs an execution function from a state context) and state.single (runs a state function from an execution context). For example, the adapted timezone calls above would be

```
set-timezone:
  module.run:
    - name: timezone.set_zone
    - timezone: America/Los_Angeles
```

and

```
# salt minion state.single timezone.system name=America/Los_Angeles
```

Parameters and names

Once again, here are the first two states from this page for reference:

```
install-sudo-package:
  pkg.installed:
    - name: sudo
    - refresh: true

create-sudo-group:
  group.present:
    - name: sudo
```

Indented under the name of each operation (that is, the *module.function* construction) is its list of parameters. In Ansible, the parameters for an operation form one big hash. Salt wants them as a list, with each entry prefaced by a dash. More specifically, Salt wants a list of hashes, though there's typically only one key in each hash.

Most parameter lists include a parameter called `name`, which is the standard label for "the thing this operation is configuring." Alternatively, you can supply a list of targets in a parameter called `names`. For example:

```
create-groups:
  group.present:
    - names:
      - sudo
      - rvm
```

If you provide a `names` parameter, Salt reruns the operation multiple times, substituting one item from the `names` list into the `name` parameter on each pass. This is a mechanical process, and the operation itself is not aware of the iteration. It's a run-time (as opposed to parse-time) operation, much like Ansible's `with_items` construction. But because Jinja expansion has already completed, there's no opportunity to base the values of other parameters on the `name`. If you need to adjust multiple parameters, ignore `names` and just iterate with a Jinja loop.

Some operations can handle multiple arguments at once. For example, `pkg.installed` can hand off multiple package names at once to the underlying OS package manager, which may be useful for efficiency or dependency resolution. Because Salt hides `names` iteration, such operations are forced to use a separate parameter name to enable bulk operations. For example, the states

```
install-packages:
  pkg.installed:
    - names: [ sudo, curl ]
```

and

```
install-packages:
  pkg.installed:
    - pkgs: [ sudo, curl ]
```

both install **sudo** and **curl**. The first version does it in two distinct operations, and the second does it in one.

We stress this seemingly minor point because it's easy to make mistakes with `names`. Because it's mechanical, `names` iterates even operations that pay no attention to the `name` parameter. On reviewing the Salt log, you'll see that multiple executions have run successfully, but somehow the target system still doesn't seem to be properly configured. So it's helpful to understand exactly what's going on.

If you don't specify an explicit `name` for a state, Salt copies the state ID to this field. You can use this behavior to simplify state definitions a bit. For example,

```
create-sudo-group:
  group.present:
    - name: sudo
```

becomes

```
sudo:
  group.present
```

or even just

```
sudo: group.present
```

YAML doesn't allow hash keys without values, so now that `group.present` no longer has any listed parameters, it has to become a simple string instead of a hash key with a parameter list as a value. That's fine; Salt checks for this explicitly.

The shorthand style is usually clearer than the long form. A separate ID field can theoretically serve as a comment or an explanation, but most IDs seen in the wild simply restate behavior that is already obvious. If you want comments, add comments.

The shorthand form has a potential problem, though: since state IDs must be globally unique, short IDs named for common system entities are more vulnerable to ID collisions. Salt detects and reports conflicts, so this is really more an annoyance than a serious issue. But if you're writing a Salt formula with the intention of reusing it in several configuration bases or you are planning to share it with the Salt community, stick with IDs that are less likely to clash.

Salt allows several operations to be included in a single state. Since the two operations above share a `name` field, we can combine them into a single state without having to state any explicit `name`s. However, there's yet another YAML snare awaiting us:

```
sudo:
  pkg.installed:
    - refresh: true
  group.present: []
```

The value of the `sudo` key now has to be a hash; it can't be a hash with the string `group.present` somehow tacked on. Accordingly, we now have to treat `group.present` as a hash key and provide an explicit parameter list as a value, even though that list is empty. That's true even if we drop the `refresh` parameter from `pkg.installed`:

```
sudo:
  pkg.installed: []
  group.present: []
```

Just as we collapsed these two states, we can collapse our two states that do user account management. A more idiomatic version of the state list from [this page] is thus

```
sudo:
  pkg.installed: []
  group.present: []

{% for admin in pillar.admins %}
{{ admin.username }}:
  group.present: []
  user.present:
    - gid: {{ admin.username }}
    - groups: [ wheel, sudo ]
    - fullname: {{ admin.fullname }}
{% endfor %}

{{ pillar.sudoers_path }}:
  file.managed:
    - source: salt://files/sudoers
    - user: root
    - group: wheel
    - mode: '0600'
```

State binding to minions

As it happens, there's not much more to say about Salt state bindings. They work exactly like pillar bindings. There's a **top.sls** file at the root of the state hierarchy, and it maps minion groups to state files. Here's a skeletal example:

```
base:
  '*':
    - bootstrap
    - sitebase
  'G@os:Ubuntu':
    - ubuntu
  'G@webserver':
    - nginx
    - webapps
```

In this configuration, all hosts apply states from **bootstrap.sls** and **sitebase.sls** from the root of the state hierarchy. Ubuntu systems also run **ubuntu.sls**, and web servers (that is, minions that have a top-level webserver entry in their grains databases) run states to configure NGINX and local web apps.

Order in **top.sls** corresponds to the general order of execution on each minion. But as usual, explicit dependency information within states overrides the default order.

Highstates

Salt refers to the bindings in **top.sls** as a minion's "highstate." There's a bit of potential terminological confusion in that Salt also uses "highstate" to mean "a parsed and assembled JSON tree of states," which it then processes to form a "lowstate"—also a JSON tree—which is the low-level input to the execution engine.

You activate the highstate by telling the minion to run the state.apply function with no arguments:

```
$ sudo salt minion state.apply
```

The state.highstate function is equivalent to state.apply with no arguments. You'll see both forms used.

Especially when debugging new state definitions, you might want a minion to run only a single state file. That's easily accomplished with state.apply:

```
$ sudo salt minion state.apply statefile
```

Leave out the **.sls** suffix on the state file name; Salt will add it. Also keep in mind that the path to the state file has nothing to do with your current directory. It's always interpreted relative to the state root as defined in the minion's configuration file. This command does not redefine the minion's highstate in any way; it simply runs the specified state file.

The **salt** command accepts a variety of flags for targeting different sorts of minion groups, but it's easiest to just remember **-C** for "compound" and use one of the shorthands from Table 23.5.

For example, to highstate all Red Hat minions:

```
$ sudo salt -C G@os:RedHat state.highstate
```

The default match type is ID globbing, so the command

```
$ sudo salt '*' state.highstate
```

is the command for "validate the entire site's configuration."

In keeping with Salt's minion-centric execution model, all parallel executions begin simultaneously, and minions do not report back until they have completed execution. The **salt** command prints each minion's results as soon as it receives them. There is no way to display incremental results while a state file is executing.

If you have lots of minions or a complex configuration base, the **salt** command's default output can be quite a lot to look through because it reports on every operation considered by every minion. Add the option **--state-output=mixed** to reduce this output to one line for operations that succeed and cause no changes. The option **--state-verbose=false** suppresses output for no-change operations entirely, but **salt** still prints a header and summary for each minion.

Salt formulas

Salt calls its bundles “formulas” (well, “formula,” really). Like Ansible roles, they’re just a directory of files, although Salt formulas have an outer wrapper that includes some metadata and versioning information as well. In actual use, you just need the inner formula directory.

Formula directories go in one of the **salt** roots defined in the **master** file. If you want, you can create a root just for formulas. Formulas sometimes include example pillar data, but you’re responsible for installing that yourself.

Salt does nothing special to support formulas, except that if you name a directory in a **top.sls** file or include statement, Salt looks for an **init.yml** file within that directory and reads that. This convention provides a clear default path into the formula. Many formulas also include stand-alone states that you can reference by specifying both the directory and filename.

Nothing in Salt can be included in a configuration more than once, and that includes formulas. You can make multiple inclusion requests, but they’ll be coalesced. As a result, formulas cannot be instantiated multiple times in the way that Ansible roles can.

It doesn’t matter anyway, because Salt defines no way to pass parameters to a formula other than by putting variable values in the pillar. (Jinja expressions can set the values of variables, but those settings exist only within the context of the current file.)

To simulate the effect of invoking a formula repeatedly, you can supply pillar data in the form of a list or hash that the formula can iterate through on its own. However, the formula must be explicitly written with this structure in mind. You can’t impose it after the fact.

The central Salt repository for community-contributed formulas is currently just GitHub. Look for the username salt-formulas. Each formula is a separate project.

Environments

See this page for more information about environments.

Salt makes several gestures toward explicit support for environments (e.g., the separation of development, test, and production universes). Unfortunately, its environment facilities are somewhat peculiar, and they don't map straightforwardly to the most common real-world use cases. It's possible to get environments up and running with a little bit of determination and a tube of Jinja glue, but we find that in practice, many sites simply punt and run separate master servers for each environment instead. This jibes well with security and compliance standards that require separation of environments at the network layer.

As we saw back on this page, the **/etc/salt/master** file enumerates the various places where configuration information can be stored. It also associates an environment with each set of paths:

```
file_roots:
  base:
    - /srv/salt

pillar_roots:
  base:
    - /srv/pillar
```

Here, **/srv/salt** and **/srv/pillar** are the state and pillar root directories for the default environment, called `base`. For simplicity, we have omitted mention of pillar data in the discussion below; environment management works the same way for both arms of the configuration base.

Sites with more than one environment will typically add an additional layer to the configuration directory hierarchy to represent that fact:

```
file_roots:
  base:
    - /srv/base/salt
  development:
    - /srv/development/salt
  production:
    - /srv/production/salt
```

(Evidently, these example cowboys have no test environment. Don't try this at home!)

An environment can list multiple root directories. If there's more than one, the server transparently merges their contents. However, each environment performs a separate merge, and the final results remain segregated.

Inside **top.sls** files (the bindings that associate minions to particular states and pillar files), top-level keys are always environment names. So far, we've only seen examples that used the `base` environment, but of course any valid environment can go in this spot. For example:

```
base:
  '*':
    - global
development:
  '*-dev':
    - webserver
    - database
production:
  '*web*-prod':
    - webserver
  '*db*-prod':
    - database
```

The exact import of an environment's appearance in a **top.sls** file depends on how you've configured Salt. In all cases, environments must already be defined in the **master** file; top files cannot create new environments. In addition, state files are required to originate from the environment context in which they are mentioned.

By default, Salt does not associate minions with any particular environment, and minions can receive state assignments from any or all of the environments in **top.sls**. In the snippet above, for example, all minions run the **global.sls** state from the base environment. Depending on their IDs, individual minions may also receive states from the production or development environments. (When you set a minion's ID to match the development or production pattern, you are functionally associating it to the corresponding environment. However, Salt itself does not make an explicit association—at least, not in this configuration.)

The Salt documentation encourages this way of configuring environments, but we have some reservations. One potential issue is that minions end up as frankenservers that draw configuration elements from multiple environments. You can't trace any given minion's configuration back to one particular environment at one particular point in time, because every minion has multiple parents.

This distinction is important because a single base environment must be shared among all other environments. Which one should it be? The development version of the base environment? The production version? A completely separate and staged configuration base? Exactly when should you migrate the base environment to a new release?

There's also some additional complexity lurking under the covers. Each environment is a full-fledged Salt configuration hierarchy, so it can, in theory, have its own **top.sls** file. Each of those **top.sls** files can, in theory, refer to multiple environments. When confronted with this situation, Salt tries to merge all the top files into one composite frankenconfiguration. (Merging occurs at the YAML level, though, so you'd better hope that multiple top files don't try to assign states to

the same matching pattern within the same environment. If they do, some states will be silently discarded.) Environments can demand the execution of one another's states—states that they don't own, control, or know anything about. It would be horrifying if it weren't so silly.

It's not clear exactly what use cases this architecture attempts to enable. Although top file merging is the default behavior, the docs repeatedly warn you away from setting things up this way. Instead, you're encouraged to designate a single **top.sls** file, most likely in `base`, to control all environments.

If you do that, though, it soon becomes apparent that there's some organizational friction between this "external" top file and the rest of the environments. The top file is an integral part of an environment's configuration, so states and top files are normally co-developed; a change to one often requires changes to the other. With a separate top file, you must effectively separate each environment into two pieces that must be manually kept synchronized with each other. In addition, the master top file must be shared with, synchronized with, and compatible with all other environments. When you promote the test environment to production, for example, you must make sure the master **top.sls** is adjusted to reflect the proper settings for that specific version of the new production release.

Alternatively, you can hard-wire minions to a given environment, either by setting the value of `environment` in the minion's **/etc/salt/minion** file or by including the flag `saltenv=`*environment* on **salt** command lines. Under this regime, a minion sees only the **top.sls** file of its assigned environment. Within that top file, its view is also further limited to entries that appear under that environment.

For example, a machine pinned to the development environment might see the **top.sls** file from this page in the following abbreviated form (assuming that the **top.sls** file was found at the root of the development state tree):

```
development:
  '*-dev':
    - webserver
    - database
```

This mode of operation is quite a bit closer than the default to the traditional concept of environments. There can be no unintended cross-talk among environments, which limits the potential for unintended behavior. It also has the advantage that as a particular version of the configuration base is promoted through the environment chain, different portions of the **top.sls** file automatically apply themselves to clients.

The main disadvantage is that you lose the ability to factor out parts of the configuration that are common to more than one environment. There's no built-in way to "see" outside the context of the current environment, so elements of the baseline configuration must be replicated into every environment.

Rewritten to work in the context of this approach, the **top.sls** file from this page would look something like this:

```
development:
  '*':
    - global
  '*-dev':
    - webserver
    - database
production:
  '*':
    - global
  '*web*-prod':
    - webserver
  '*db*-prod':
    - database
```

The base environment itself is now vestigial, so we've dropped it from the **top.sls** file and copied that key's former contents directly into the development and production environments.

Keep in mind that we're now operating in a world where every environment tree has its own **top.sls** file. For this example, we assume that the **top.sls** file hasn't diverged between the two environments, so the same contents would appear in both copies of **top.sls**.

Of course, manually reproducing the elements of the common configuration inside each environment is prone to error. A better option is to define the common configuration as a Jinja macro so that it can automatically be repeated:

```
{% macro baseline() %}
  '*':
    - global
{% endmacro %}

development:
  {{ baseline() }}
  '*-dev':
    - webserver
    - database
production:
  {{ baseline() }}
  '*web*-prod':
    - webserver
  '*db*-prod':
    - database
```

We're assuming in this scenario that all minions are pinned to specific environments, so we can now potentially remove the environment indicators from minion IDs. However, it's a good idea to retain them for security.

The issue is that minions control their own environment settings. If a minion in the development environment were compromised, for example, it could declare itself to be a production server and potentially gain access to the keys and configurations used in the production environment. (This is perhaps one reason why the Salt documentation seems a bit skittish about recommending environment pinning.)

Making environment-specific configurations contingent on both the environment settings and the minion IDs protects against this line of attack. If a minion changes its ID, the master no longer recognizes it as an approved client and ignores it until an administrator approves the change with the **salt-key** command.

If you prefer not to use IDs in this way, an alternative is to use pillar data as a cross-check. Whatever you do, you can't just drop the suffix and turn '*-dev' into '*', because the shared portion of the configuration already uses '*' as a key. Duplicate patterns within an environment are a YAML violation.

When debugging environments, you'll find a couple of execution functions especially helpful. config.get shows the value that a particular minion (or set of minions) is using for a configuration option:

```
$ sudo salt new-client-dev config.get environment
new-client-dev:
  development
```

Here, we can see that the minion with ID new-client-dev has been pinned to the development environment, just as its ID would suggest. To see what the **top.sls** configuration looks like from that minion's perspective, use state.show_top:

```
$ sudo salt new-client-dev state.show_top
new-client-dev:
  ----------
  development:
    - global
    - webserver
    - database
```

The output shows only the states that are active and selected for the target minion. In other words, they are the states that would run if you invoked state.highstate on that minion.

Note that all the displayed states come from the development environment. Because the minion is pinned, that will always be the case.

Documentation roadmap

Salt's documentation (docs.saltstack.com) will likely earn your admiration, but perhaps only after a period of frustration. The main sticking point is that topics are nested several layers deep, but the headings at the top two layers do not necessarily hint at what you'll find at layer three. The *Architecture* section, for example, contains no information about Salt's architecture (it's really about multiserver deployments).

Some of the most useful reference material lies within sections that are organized as scenarios or tutorials. Front-to-back reading can sometimes evoke a dying sysadmin's fever dream: themes cyclically loom and recede without fully resolving. Once in a while, you'll experience a moment of lucidity in which to appreciate the severity of your condition.

Some pointers:

- The top-level *Using Salt* section is an overview by concept, and most of *Configuration Management* is labeled as a tutorial. Because of their formats, these sections look like supplemental materials. But that's not true; they are pretty much the primary documentation for the material they cover. Don't skip.
- The best reference information is beneath *State System Reference*, under *Configuration Management*. A lot of the stuff in here is not important for a first reading, but *Highstate data structure definitions*, *Requisites and other global state arguments*, and *The top file* are particularly worth reading. (*The top file* is also the authoritative documentation for environments.)
- The docs you'll use most frequently—the ones covering state and execution functions—are concealed under *Salt Module Reference* and camouflaged among 19 other module types that are of interest mostly to module developers. Bookmark the sections for *Full list of builtin state modules* and *Full list of builtin execution modules*.

23.7 Ansible and Salt compared

We like both Ansible and Salt. Each of them has some friction points, however, and we recommend them for different environments. The sections below comment on a few of the factors you might consider when choosing between them.

Deployment flexibility and scalability

Salt covers a broader range of deployment environments than does Ansible. It's simple enough that you can reasonably use it to manage a single server, but it also scales effortlessly and essentially without limit. If you want to learn one system that covers the broadest possible range of use cases, Salt is a good choice.

In part, that's because Salt's architecture makes relatively few demands of the master server. Minions receive their instructions and don't report back until they're done, with all status information being reported at once. Minions call the server to obtain configuration data, but aside from serving pillar data, the server itself performs relatively little computation.

Once your site outgrows a single Salt master, you can convert your infrastructure to a tiered or replicated server scheme. We don't cover those options in this book, but they're easy to set up and work well.

Large deployments are a comparative weak spot for Ansible. It does include some features to help you implement multitier server systems, but the transition to this model is not as transparent as it is in Salt.

Ansible is an order of magnitude slower than Salt, and because of its architecture, it must handle clients in batches. However, most servers can handle far more than the default 5 simultaneous clients. You can also change Ansible's execution strategy so that clients aren't kept in strict lockstep with each other. Even a tuned Ansible system won't approach the speed of Salt, but it's better than one might naïvely anticipate.

Built-in modules and extensibility

Bake-offs of configuration management software sometimes attempt to compare the number of operation types that various systems support out of the box. However, these comparisons are hard to get right because of underlying structural differences. Functions that are spread across several modules in Ansible might be addressed by one in Salt, for example. An atomic operation in one system may correspond to several operations in another.

At present, Salt and Ansible are roughly comparable in this respect. In addition to extensive standard libraries, both systems have a structure in place for absorbing community-written modules into the core or an easily accessible add-on pack.

In any event, total module count doesn't matter nearly as much as coverage of the systems and software your site is actually using. All CM systems cover basic operations pretty well, but as you move into the long tail, offerings vary dramatically.

It's likely that you'll eventually want to tackle some tasks for which your CM system doesn't have an off-the-shelf solution. Fortunately, Salt and Ansible are both easy to extend with your own Python code. Embrace this extensibility early on and make it part of your repertoire.

Security

As outlined in *Ansible access options*, Ansible can be made almost arbitrarily secure. The only limit to security is your own willingness to retype passwords and deal with security red tape.

Ansible's vault system lets you keep configuration data in an encrypted format. That's actually a pretty big deal, because it means that neither the Ansible server nor the configuration base needs to be particularly secure. (Salt's modular architecture probably makes this an easy feature to add, but it doesn't come in the box.)

By contrast, Salt can only be as secure as the root account on the master server. Although the master daemon itself is simple to set up, the server on which it runs should receive your site's most aggressive securement. Ideally, the master should be a machine or virtual server dedicated to this task.

In practice, administrators hate intrusive security protocols as much as anyone else does. Most real-world Ansible installations have relatively lax security. Just as Ansible can be made arbitrarily secure, it can also be made arbitrarily insecure.

Even if you strive to keep Ansible fully secured, you may have trouble maintaining this approach once your site grows beyond the point at which configuration management can be handled by an administrator typing commands in a terminal window. Nothing that runs out of **cron**, for example, can depend on the presence of an administrator to enter passwords. Working around that constraint inevitably ends up lowering security to the level of the root account.

The bottom line on security is that Ansible gives you both more options and more opportunities to shoot yourself in the foot. It's more securable, but that doesn't necessarily mean that it's more secure. Either system is fine for the average site. Keep your own needs and constraints firmly in mind when evaluating these systems.

Miscellaneous

Table 23.6 and Table 23.7 summarize some of the additional strengths and weakness of both Ansible and Salt.

Table 23.6: Ansible pros and cons

Advantages	Disadvantages
Requires only SSH and Python; no daemons	Very slow
Clear and concise documentation	Server-heavy; harder to scale
Built-in loops and conditionals, minimal Jinja	Lots of files with identical names
Works fine as a nonroot user	Idiosyncratic YAML syntax
Operations can use each other's output	Hand-managed client inventory
Clean and flexible use of config directories	Minimal grouping facilities
Secure, general encryption facility	Many different variable scopes
Roles can be instantiated repeatedly	No daemons means fewer options
Larger user base than Salt	No real support for environments

Table 23.7: Salt pros and cons

Advantages	Disadvantages
Fast	Relies heavily on Jinja
At heart, simpler than Ansible	Documentation oddly organized
Flexible, consistent bindings	Poor support for nonroot use
Integrated support for cloud servers	Formulas can't be instantiated
Concise configuration syntax	No built-in encryption solution
Multitier server deployments	No access to results of operations
Structured event monitoring	Minimal support for variable value defaults
Execution logs easily exported	Requires explicit dependency declarations
Fanatically modularized	Fanatically modularized

23.8 BEST PRACTICES

If you've worked on a software project, you might find many of the issues addressed by configuration management systems to be familiar from the development world. Development environments encompass many of the same vagaries: multiple platforms, multiple products derived from the same code base, multiple types of builds and configurations, and deployment through successive steps of development, testing, and production.

These are complex issues, and development environments are only tools. Developers use a variety of additional controls—development guidelines, design reviews, coding standards, internal documentation, and clear architectural boundaries, among others—to limit the slide toward entropy.

Unfortunately, administrators often wander into configuration management territory without a proper suit of developer's armor. At first glance, configuration management seems deceptively straightforward, like a slightly more general and sophisticated way to approach routine scripting tasks. Configuration management vendors work hard to reinforce this impression. Their web sites are siren songs of ease and grace; each one features a tutorial in which you deploy a web server by running ten lines of configuration code.

In reality, the edge of the abyss may be closer than it seems, particularly when multiple administrators contribute to the same configuration base over time. Real-world specifications for even a single-purpose server run to hundreds of lines of code segmented into multiple different functional roles. Without coordination, it's easy to turn the CM system into a muddle of conflicting or parallel code.

Best practices vary by configuration management system and by environment, but a few rules apply to most situations:

- Keep the configuration base under version control. This isn't a best practice so much as a basic requirement for CM sanity. Not only does Git provide change tracking and history, but it has already solved many of the mechanical problems involved in coordinating projects across administrative boundaries.
- Configuration bases are inherently hierarchical, at least in a logical sense. Some standards apply site-wide, some apply to every server in a particular department or region, and some are specific to particular hosts. In addition, you'll most likely need the ability to make exceptions in certain cases. Depending on your site's operations, you might also need to maintain multiple independent hierarchies.

  Plan for all of this structure in advance, and consider how you might manage scenarios in which different groups control different parts of the configuration base. At the very least, conventions for classifying hosts (e.g., EC2 instances, Internet-

facing hosts, database servers) should be coordinated site-wide and adhered to consistently.

- CM systems allow different parts of the configuration base to be kept in different directories or repositories. However, this structure provides little actual benefit, and it complicates day-to-day configuration work. We recommend one big, integrated configuration base. Manage hierarchy and coordination at the Git layer.
- Sensitive data (keys, passwords) should not be put under version control unless encrypted, even in private code repositories. Git in particular is not designed to maintain security. Your CM system may have some encryption features built in, but if not, roll your own.
- Because it effectively has root access to many other hosts, a configuration server is one of the most concentrated sources of security risk in your organization. It's reasonable to dedicate a server to this role, and it should receive your most stringent security hardening.
- Configurations should run without reporting spurious changes. Scripts and shell commands are usually the biggest sticking points. Check your CM system's documentation for advice on this topic, as it's one of the most frequent issues that users encounter.
- Don't test on production servers. But do test! It's easy to spin up a test system in the cloud or in Vagrant. Chef even provides an elaborate testing and development system in the form of Kitchen. Make sure your test system matches your real systems by using the same machine image and network configuration.
- Read the code for add-on bundles that you obtain from public repositories. It's not that these sources are particularly suspicious; it's just that systems and conventions differ widely. In many cases, you'll find that a few local tweaks are needed. If you can bypass the CM system's package manager and clone bundles directly from a Git repo, then you can easily upgrade to later releases without losing your customizations.
- Subdivide configurations ruthlessly. Every file should have a clear and single purpose. (Ansible users might want to select a text editor that deals well with 50 different files all named **main.yml**.)
- Configuration-managed servers should be 100% managed. That is, there should be no penumbra of administrative work that was performed by hand and that no one knows how to replicate. This issue appears primarily when moving existing servers onto configuration management. When converting an existing "snowflake" server to configuration management, you may find it useful to clone the original system for use as a basis for comparison. It can take multiple cycles of configuration management and testing to home in on all the system's particulars.

- Do not allow yourself or your team to “temporarily disable” the CM system on a node or to use a heavy-handed method of overriding the CM system (for example, by setting the immutable attribute on a configuration file that is typically under CM control). These changes are inevitably forgotten, and confusion or outages ensue.

- It’s not hard to open a gateway from existing administrative databases into a configuration management system, and there’s a lot of value in doing so. CM systems are designed for this kind of interfacing. For example, you might identify system administrators and their zones of activity in your site-wide LDAP database, and make this information available within the configuration management environment through gateway scripts. Ideally, every piece of information should have a single authoritative source.

- CM systems are excellent for managing the state of a machine. They are not intended for stateful, coordinated activities such as software deployment operations, although the documentation and even some examples might lead you to believe that they are. In our experience, a dedicated continuous deployment system is more suitable.

- In elastic cloud environments where computational capacity is added in response to real-time demand, the time that it takes for a new node to bootstrap through configuration management can be agonizingly slow. Optimize by including packages and long-running configuration items within the baseline machine image rather than downloading and installing them at boot time.

  If you use configuration management to set configuration parameters for an application, make sure that that step comes early in the bootstrapping process so that the application comes on-line more quickly. We try to limit the CM run time to less than 60 seconds for dynamically scaled nodes.

- As an administrator working with a CM system, you will allot much of your time to writing CM code, testing changes against a representative set of systems, committing the updates to a repository, and applying changes to your site in a staged fashion. To be most effective, you should perfect this process by investing time up front to learn best practices and tricks for your system of choice.

23.9 Recommended reading

Cowie, Jon. *Customizing Chef: Getting the Most Out of Your Infrastructure Automation.* Sebastopol, CA: O'Reilly Media, 2014.

Frank, Felix, and Martin Alfke. *Puppet 4 Essentials (2nd Edition).* Birmingham, UK: Packt Publishing, 2015.

Geerling, Jeff. *Ansible for DevOps: Server and configuration management for humans*. St. Louis, MO: Midwestern Mac, LLC, 2015. This book is focused mostly on basic Ansible wrangling, but it does include some helpful material about combining Ansible with specific systems such as Vagrant, Docker, and Jenkins.

Hochstein, Lorin. *Ansible: Up and Running (2nd Edition)*. Sebastopol, CA: O'Reilly Media, 2017. Like *Ansible for DevOps*, this book covers both the Ansible basics and interactions with common environments such as Vagrant and EC2. Some highlights are the inclusion of a larger-scale example configuration, a chapter on writing your own Ansible modules, and tips on debugging.

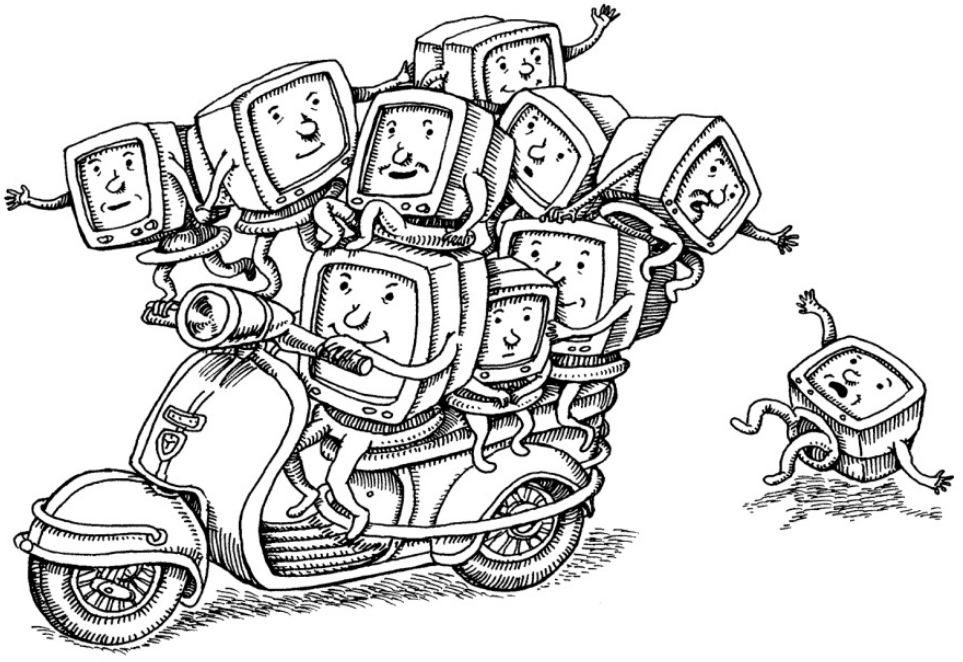
Morris, Kief. *Infrastructure as Code: Managing Servers in the Cloud*. Sebastopol, CA: O'Reilly Media, 2016. This book includes few specifics about configuration management per se, but it's helpful for understanding how configuration management integrates into the larger scheme of DevOps and structured administration.

Sebenik, Craig, and Thomas Hatch. *Salt Essentials*. Sebastopol, CA: O'Reilly Media, 2015. This is a short and rather skeletal book that sticks pretty closely to the basics of Salt. It's not the style of book we'd ordinarily recommend, but given the unevenness of the official documentation, it's a potentially useful reference for those who seek a "second opinion."

Taylor, Mischa, and Seth Vargo. *Learning Chef: A Guide to Configuration Management and Automation*. Sebastopol, CA: O'Reilly Media, 2013.

Uphill, Thomas, and John Arundel. *Puppet Cookbook (3rd Edition)*. Birmingham, UK: Packt Publishing, 2015.

24 Virtualization



Server virtualization makes it possible to run multiple operating system instances concurrently on the same physical hardware. Virtualization software parcels out CPU, memory, and I/O resources, dynamically allocating their use among several “guest” operating systems and resolving resource conflicts. From the user’s point of view, a virtual server walks and talks like a full-fledged physical server.

See Chapter 25 for more information about containers.

This decoupling of hardware from the operating system affords numerous luxuries. Virtualized servers are more flexible than their “bare metal” kin. They’re portable and can be programmatically managed. The underlying hardware is used more efficiently because it can service multiple guests simultaneously. And if that isn’t enough, virtualization technology underpins both cloud computing and containers.

Implementations of virtualization have changed over the years, but the core concepts are not new to the industry. Big Blue used virtual machines on early mainframes while researching time-sharing concepts in the 1960s. The same techniques were used throughout the mainframe heyday

of the 1970s until the client/server boom of the 1980s, when the difficulty of virtualizing the Intel x86 architecture led to a short period of relative dormancy.

The ever-growing size of server farms rekindled interest in virtualization for modern systems. VMware and other providers conquered the challenges of x86 and made it easy to automatically provision operating systems. These facilities eventually led to the rise of on-demand, Internet-connected virtual servers: the infrastructure we now know as cloud computing. More recently, advances in OS-level virtualization have ushered in a new era of OS abstraction in the form of containers.

In this chapter, we begin by clarifying the terms and concepts you need in order to understand virtualization for UNIX and Linux. We then introduce the leading virtualization solutions used on our example operating systems.

24.1 VIRTUAL VERNACULAR

The terminology used to describe virtualization is somewhat opaque, largely because of the way the technology evolved. Competing vendors worked independently without the benefit of standards, yielding a bewildering array of ambiguous phrases and acronyms. Conway's Law comes to mind: "Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations."

To further confuse the issue, "virtualization" itself is an overloaded term that describes more than the scenario described above, in which guest operating systems run within the context of virtualized hardware. OS-level virtualization—more commonly referred to as containerization—is a related but distinct set of facilities that has become just as ubiquitous as server virtualization. For those without hands-on exposure to these technologies, it can often be difficult to grasp the differences. We contrast the two approaches later in this section.

Hypervisors

A hypervisor (also known as a virtual machine monitor) is a software layer that mediates between virtual machines (VMs) and the underlying hardware on which they run. Hypervisors are responsible for sharing system resources among the guest operating systems, which are isolated from one another and which access the hardware exclusively through the hypervisor.

Guest operating systems are independent, so they needn't be the same. CentOS can run alongside FreeBSD and Windows, for example. VMware ESX, XenServer, and FreeBSD bhyve are examples of hypervisors. The Linux kernel-based virtual machine (KVM) converts the Linux kernel into a hypervisor.

Full virtualization

The first hypervisors fully emulated the underlying hardware, defining virtual replacements for all the basic computing resources: hard disks, network devices, interrupts, motherboard hardware, BIOSs, and so on. This mode, called full virtualization, runs guests without modification but incurs a performance penalty because the hypervisor must constantly translate between the system's actual hardware and the virtual hardware exposed to guests.

Simulating an entire PC is a complex task. Most hypervisors that offer full virtualization separate the task of maintaining multiple environments (virtualization) from the task of simulating the hardware within each environment (emulation).

The most common emulation package used in these systems is an open source project called QEMU. You can find more information at qemu.org, but in most cases the emulator doesn't require much attention from administrators.

Paravirtualization

The Xen hypervisor introduced "paravirtualization," in which modified guest operating systems detect their virtualized state and actively cooperate with the hypervisor to access hardware. This approach improves performance by an order of magnitude or more. However, guest operating systems need substantial updates to run this way, and the exact modifications depend on the specific hypervisor in use.

Hardware-assisted virtualization

In 2004 and 2005, Intel and AMD introduced CPU features (Intel VT and AMD-V) that facilitate virtualization on the x86 platform. These extensions gave rise to "hardware-assisted virtualization," also known as "accelerated virtualization." In this scheme, the CPU and memory controller are virtualized by the hardware, albeit under the control of the hypervisor. Performance is very good, and guest operating systems need not know that they're running on a virtualized CPU. These days, hardware-assisted virtualization is the assumed baseline.

Although the CPU is a primary point of contact between the hardware and guest operating systems, it is only one component of the system. The hypervisor still needs some way to present or emulate the rest of the system's hardware. Either full virtualization or paravirtualization can be used for this task. In some cases, a mix of approaches is used; it depends on the virtualization-awareness of the guest.

Paravirtualized drivers

One great advantage of hardware-assisted virtualization is that it largely restricts the need for paravirtualization support to the level of device drivers. All operating systems allow add-on drivers, so setting up a guest with paravirtualized disk drives, display cards, and network interfaces is as simple as installing the appropriate drivers. The drivers know the secret handshake that lets them connect with the hypervisor's paravirtualization support, and the guest OS remains none the wiser.

A few pesky aspects of the PC architecture, such as the interrupt controller and the BIOS resources, fall into the domain of neither the CPU nor the device drivers. In the past, the predominant approach has been to implement these remaining components through full virtualization. For example, Xen's HVM (Hardware Virtual Machine) mode combines support for CPU-level virtualization extensions with a copy of the QEMU PC emulator. And PVHVM (ParaVirtualized HVM) mode adds to this scheme paravirtualized drivers on guest operating systems, greatly reducing the amount of full virtualization needed to keep the system running. However, the hypervisor still needs an active copy of QEMU for each virtual machine so that it can cover the odds and ends not addressed by the paravirtualized drivers.

Modern virtualization

The most recent versions of Xen and other hypervisors have more or less eliminated the need to emulate legacy hardware. Instead, they rely on CPU-level virtualization features, paravirtualized guest-OS drivers, and a few additional sections of paravirtualized code within guest kernels. Xen calls this mode PVH (ParaVirtualized Hardware), and it's considered to be a close-to-ideal blend that yields optimal performance but imposes the lowest possible requirements on guest kernels.

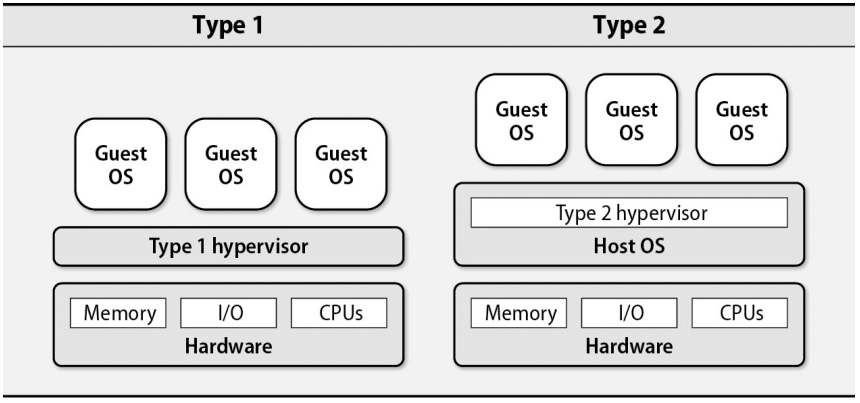
You might encounter any of the varieties of virtualization described above in the wild or when reading documentation. However, it's not worth memorizing any particular taxonomy or worrying too much about virtualization modes. The boundaries among these modes are porous, and the hypervisor generally works out the best options for a given guest. If you keep your software updated, you automatically benefit from the latest enhancements. The only reason to choose anything other than the default operating mode is to support older hardware or ancient hypervisors.

Type 1 vs. type 2 hypervisors

Many reference materials draw a somewhat dubious distinction between "type 1" and "type 2" hypervisors. A type 1 hypervisor runs directly on the hardware without a supporting OS, and for

that reason is sometimes called a bare-metal or native hypervisor. Type 2 hypervisors are user-space applications that run on top of another general-purpose OS. Exhibit A depicts these two models.

Exhibit A: Comparison of type 1 and type 2 hypervisors



VMware ESXi and XenServer are considered type 1, and FreeBSD's bhyve is type 2. Likewise, workstation-oriented virtualization packages such as Oracle's VirtualBox and VMware Workstation are also type 2.

It's true that type 1 and type 2 systems are different, but the delineation is not always so binary. KVM, for example, is a Linux kernel module that gives virtual machines direct access to CPU virtualization features. Differentiating among types of hypervisor is more an academic exercise than a point of practice.

Live migration

Virtual machines can move between hypervisors running on different physical hardware in real time, in some cases without interruptions in service or loss of connectivity. This feature is called live migration. The magic lies in a memory dance between the source and target hosts. The hypervisor copies changes from the source to the destination, and as soon as the memory is identical between the two, the migration completes. Live migration is helpful for high-availability load balancing, disaster recovery, server maintenance, and general system flexibility.

Virtual machine images

Virtual servers are created from images, which are templates of configured operating systems that a hypervisor can load and execute. The image file format varies by hypervisor. Most hypervisor projects maintain a collection of images that you can download and use as a basis for your own customizations. You can also take a snapshot of a virtual machine to create an image, either as a backup of important data or to use as the basis for creating more virtual machines.

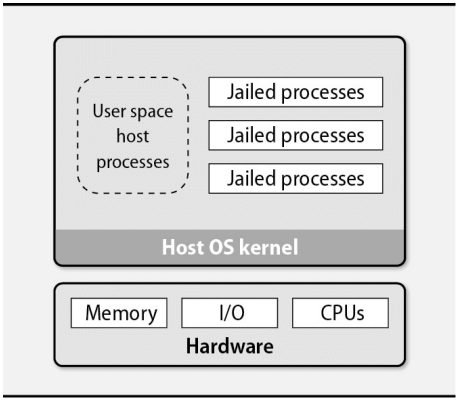
Because the virtual machine hardware presented by the hypervisor is standardized, images are portable among systems even if their actual hardware differs. Images are specific to a particular hypervisor, but conversion tools that port images among hypervisors are available.

Containerization

OS-level virtualization—or containerization—is a different approach to isolation that does not use a hypervisor. Instead, it relies on kernel features that isolate processes from the rest of the system. Each process "container" or "jail" has a private root filesystem and process namespace. The contained processes share the kernel and other services of the host OS, but they cannot access files or resources outside of their containers. Exhibit B illustrates this architecture.

See Chapter 25 for more information about containers.

Exhibit B: Containerization



Because it does not require virtualization of the hardware, the resource overhead of OS-level virtualization is low. Most implementations offer near-native performance. Linux's LXC, Docker containers, and FreeBSD jails are implementations of containers.

This type of virtualization largely precludes the use of multiple operating systems because the host kernel is shared by all containers. However, FreeBSD's Linux emulation layer permits Linux containers to run on FreeBSD hosts.

It's easy to confuse containers with virtual machines. Both define portable, isolated execution environments, and both look and act like full operating systems with root filesystems and running processes. Yet their implementations are entirely different.

A true virtual machine has an OS kernel, an **init** process, drivers to interact with hardware, and the full trappings of a UNIX operating system. A container, on the other hand, is merely the facade of an operating system. It uses the strategies described above to give individual processes a suitable execution environment. Table 24.1 illustrates some of the practical differences.

Table 24.1: Comparing virtual machines with containers

Virtual machine	Container
A full-fledged OS that shares underlying hardware through a hypervisor	An isolated group of processes managed by a shared kernel
Requires a complete boot procedure to initialize; starts in 1-2 minutes	Processes run directly by the kernel; no boot required; starts in < 1 second
Long-lived	Frequently replaced
Has one or more dedicated virtual disks attached through the hypervisor	Filesystem view is a layered construct defined by the container engine
Images measured in gigabytes	Images measured in megabytes
A few dozen or fewer per physical host	Many per virtual or physical host
Complete isolation among guests	OS kernel and services shared with host
Multiple independent operating systems running side by side	Must run the same kernel as the host (OS distribution may differ)

It's common to use containers in combination with virtual machines. Virtual machines are the best way to subdivide physical servers into manageable chunks. You can then run applications in containers atop the VMs to achieve optimal system density (this procedure is sometimes called "bin packing"). The containers-on-VMs architecture is standard for containerized applications that need to run on public cloud instances.

We focus on true virtualization for the rest of this chapter. See Chapter 25, *Containers*, for more details on containerization.

24.2 Virtualization with Linux

Xen and KVM are the leading open source virtualization projects for Linux. Xen, now a project of the Linux Foundation, powers some of the largest public clouds, including Amazon Web Services and IBM's SoftLayer. KVM is the kernel-based virtual machine integrated into the mainline Linux kernel. Both Xen and KVM have demonstrated their stability through many production installations at large sites.

Xen

Initially developed by Ian Pratt as a research project at the University of Cambridge, the Linux-friendly Xen has grown to become a formidable virtualization platform that challenges even the commercial giants in terms of performance, security, and especially, cost.

As a paravirtual hypervisor, Xen claims a mere 0.1%–3.5% overhead, far less than fully virtualized solutions. Because Xen is open source, a variety of management tools are available with varying levels of feature support. The Xen source code is available from xenproject.org, but many Linux distributions include native support.

Xen is a bare-metal hypervisor that runs directly on the physical hardware. A running virtual machine is called a domain. There is always at least one domain, referred to as domain zero or dom0. Dom0 has full hardware access, manages the other domains, and runs all the hypervisor's own device drivers. Unprivileged domains are referred to as domU.

Dom0 typically runs a Linux distribution. It looks just like any other Linux system but includes the daemons, tools, and libraries that complete the Xen architecture and enable communication among domU, dom0, and the hypervisor.

The hypervisor is responsible for CPU scheduling and memory management for the system as a whole. It controls all domains, including dom0. However, the hypervisor itself is in turn controlled and managed from dom0. What a tangled web we weave.

Table 24.2 lists the most interesting puzzle pieces of a Linux dom0.

Table 24.2: Xen components in dom0

Path	Contents
/etc/xen	Primary configuration directory
auto	Guest OS config files to autostart at boot time
scripts	Utility scripts that create network interfaces, etc.
/var/log/xen	Xen log files
/usr/sbin/xl	Xen guest domain management tool

Each Xen guest-domain configuration file in **/etc/xen** specifies the virtual resources available to a domU, including disk devices, CPU, memory, and network interfaces. Each domU has a separate configuration file. The format is flexible and gives administrators granular control over the constraints applied to each guest. If a symbolic link to a domU configuration file is added to the **auto** subdirectory, that guest OS is automatically started at boot time.

Xen guest installation

It takes several steps to get a guest server up and running under Xen. We recommend the use of a tool such as **virt-manager** (virt-manager.org) to simplify the process. **virt-manager** was originally a Red Hat project, but it has now been deproprietarized and is available for most Linux distributions. **virt-install**, its command-line OS provisioning tool, accepts installation media from a variety of sources, including SMB or NFS mounts, physical CDs or DVDs, and HTTP URLs.

Guest domains' disks are normally stored in virtual block devices (VBDs) in dom0. The VBD can be connected to a dedicated resource such as a physical disk drive or logical volume. Or, it can be a loopback file, also known as a file-backed VBD, created with **dd**. Performance is better with a dedicated disk or volume, but files are more flexible and can be managed with normal Linux commands (such as **mv** and **cp**) in dom0. Backing files are sparse files that grow as needed.

Unless the system is experiencing performance bottlenecks, a file-backed VBD is usually the best choice. It's a simple process to transfer a VBD onto a dedicated disk if you change your mind.

The installation of a guest domain might look like this:

```
$ sudo virt-install -n chef -f /vm/chef.img -l http://example.com/myos
    -r 512 --nographics
```

This is a typical Xen guest domain with the name "chef," a disk VBD location of **/vm/chef.img**, and installation media obtained through HTTP. The instance has 512MiB of RAM and uses no X Windows graphics support during installation. **virt-install** downloads the files needed to start the installation and then kicks off the installer process.

When the screen clears, install Linux through the standard text-based process, which includes network configuration and package selection. After the installation completes, the guest domain reboots and is ready for use. To disconnect from the guest console and return to dom0, just press <Control-]>.

It's worth noting that although this example uses text-based installation, graphics-based installation through Virtual Network Computing (VNC) is also available.

virt-install saves the domain's configuration in **/etc/xen/chef**. Here's what it looks like:

```
name = "chef"
uuid = "a85e20f4-d11b-d4f7-1429-7339b1d0d051"
maxmem = 512
memory = 512
vcpus = 1
bootloader = "/usr/bin/pygrub"
on_poweroff = "destroy"
on_reboot = "restart"
on_crash = "restart"
vfb = [  ]
disk = [ "/vm/chef.dsk,xvda,w" ]
vif = [ "mac=00:16:3e:1e:57:79,bridge=xenbr0" ]
```

You can see that the NIC defaults to bridged mode. In this case, the VBD is a “block tap” file that affords better performance than does a standard loopback file. The writable disk image file is presented to the guest as **/dev/xvda**.

The **xl** tool is convenient for day-to-day management of virtual machines. It lets you start and stop VMs, connect to their consoles, and investigate their current state. Below, we show the running guest domains, then connect to the console for the chef domU. IDs are assigned in increasing order as guest domains are created, and they are reset when the host reboots.

```
$ sudo xl list
Name        ID   Mem(MiB)  VCPUs   State    Time(s)
Domain-0    0    2502      2       r-----   397.2
chef        19   512       1       -b----   12.8
$ sudo xl console 19
```

To change the configuration of a guest domain (e.g., to attach another disk or to change the network to NAT mode instead of bridged), edit the guest’s configuration file in **/etc/xen** and reboot the guest.

KVM

KVM, the Kernel-based Virtual Machine, is a full virtualization platform that is the default for most Linux distributions. Like Xen's HVM mode, KVM takes advantage of the Intel VT and AMD-V CPU extensions and relies (in a typical setup) on QEMU to implement a fully virtualized hardware system. Although the system is native to Linux, it has also been ported to FreeBSD as a loadable kernel module.

Since KVM defaults to full virtualization, many guest operating systems are supported, including Windows. Paravirtualized Ethernet, disk, and graphics card drivers are available for Linux, FreeBSD, and Windows. Their use is optional but recommended for performance.

Under KVM, the Linux kernel itself serves as the hypervisor. Memory management and scheduling are handled through the host's kernel, and guest machines are normal Linux processes. Enormous benefits accompany this unique approach to virtualization. For example, the complexity introduced by multicore processors is handled by the kernel, and no hypervisor changes are required to support them. Linux commands such as **top**, **ps**, and **kill** show and control virtual machines, just as they would for other processes. The integration with Linux is seamless.

KVM guest installation

Although the technologies behind Xen and KVM are fundamentally different, the tools that install and manage guest operating systems are eerily similar. As under Xen, you can use **virt-install** to create new KVM guests. Use the **virsh** command to manage them.

The flags passed to **virt-install** vary slightly from those used for a Xen installation. To begin with, the **--hvm** flag says that the guest should be hardware virtualized, as opposed to paravirtualized. In addition, the **--connect** argument guarantees that the correct default hypervisor is chosen, since **virt-install** supports more than one hypervisor. Finally, the use of **--accelerate** is recommended, to take advantage of the acceleration capabilities in KVM. Ergo, a full command for installing an Ubuntu server guest from DVD-ROM looks something like this:

```
$ sudo virt-install --connect qemu:///system -n UbuntuYakkety
    -r 512 -f ~/ubuntu-Yakkety.img -s 12 -c /dev/dvd --os-type linux
    --accelerate --hvm --vnc

Would you like to enable graphics support? (yes or no)
```

Assuming that the Ubuntu installation DVD has been inserted, this command launches the installation and stores the guest in the file **~/ubuntu-Yakkety.img**, allowing it to grow to 12GB. Since we specified neither **--nographics** nor **--vnc**, **virt-install** asks whether to enable graphics.

The **virsh** utility spawns its own shell from which you can run commands. To open the shell, type **virsh --connect qemu:///system**. The following series of commands demonstrates some of the core functionality of **virsh**. Type **help** in the shell to see a complete list, or see the man page for the nitty-gritty details.

```
$ sudo virsh --connect qemu:///system
virsh # list --all
     Id      Name                State
----------------------------------------
      3       UbuntuYakkety       running
      7       CentOS              running
      -       Windows2016Server   shut off

virsh # start Windows2016Server
Domain WindowsServer started

virsh # shutdown CentOS
Domain CentOS is being shutdown

virsh # quit
```

24.3 FreeBSD bhyve

FreeBSD's virtualization software is bhyve, a relatively new system first added in FreeBSD 10.0. It can run BSD, Linux, and even Windows guests. However, it runs on a limited set of hardware and is missing some of the core features found in other implementations.

With so many virtualization platforms that support FreeBSD on the market already, it's unclear why the bhyve effort started when it did. Unless you are developing a custom platform that requires embedded FreeBSD virtualization, we recommend choosing another solution until this project matures.

24.4 VMware

VMware is the biggest player in the virtualization industry and was the first vendor to develop techniques to virtualize the fractious x86 platform. VMware is a commercial entity, but some of its products are free. They're all worthy of consideration when you are choosing a site-wide virtualization technology.

The primary product of interest to UNIX and Linux administrators is ESXi, which is a bare-metal hypervisor for the Intel x86 architecture. The name stands for "Elastic Sky X, integrated"—you really can't make this stuff up. ESXi is free (free like a box of puppies), but some useful functionality is limited to paid licensees.

In addition to ESXi, VMware offers some powerful, advanced products that facilitate centralized deployment and management of virtual machines. They also have the most mature live migration technology we've seen. In-depth coverage of the full VWware product suite is beyond the scope of this chapter, however.

24.5 VirtualBox

VirtualBox is a consumer-grade, cross-platform, type 2 hypervisor. It performs “probably good enough” virtualization of systems, typically for individuals. It’s popular among developers and end users because it is free, easy to install, easy to use, and often simplifies the creation and management of test environments.

Performance and hardware support are both weak points. VirtualBox is generally not suitable for “production” virtualization use, although its web site does describe it as a “professional” solution which is licensed for “enterprise” use. (This may in fact be the case with regard to Oracle operating systems, which are the only prebuilt VMs available.)

The history of VirtualBox is long and sordid. It originally began as a commercial product of Innotek GmbH but was released as open source before Innotek was acquired by Sun Microsystems in 2008. After Oracle swallowed Sun in 2010, the product was rebranded as Oracle VM VirtualBox. VirtualBox lives on today (available under the GPLv2 open source license) and remains under active development at Oracle.

VirtualBox runs on Linux, FreeBSD, Windows, macOS, and Solaris. Oracle does not publish or support the FreeBSD version of the host, but it’s available as a community port. Supported guest OSs include Windows, Linux, and FreeBSD.

By default, you wrangle virtual machines through VirtualBox’s GUI. If you’re interested in running VMs on a system that doesn’t run a GUI, explore VBoxHeadless, the morbid name for VirtualBox’s CLI tool. You can download VirtualBox and read more about it at virtualbox.org.

24.6 PACKER

Packer (packer.io), from the esteemed open source company HashiCorp, is a tool for building virtual machine images from a specification file. It can build images for a variety of virtualization and cloud platforms. Integrating Packer into your workflow lets you be more or less virtualization-platform-agnostic. You can easily build your customized image for whatever platform you're using on a given day.

To create an image, Packer launches an instance from a source image of your choosing. It then customizes the instance by running scripts or invoking other provisioning steps that you specify. Finally, it saves a copy of the virtual machine's state as a new image.

This process is particularly helpful for supporting an "infrastructure as code" way of managing servers. Instead of manually applying changes to images, you modify a template that describes the image in abstract terms. You then check the specification into a repository as you would with traditional source code. This technique supplies you with outstanding transparency, repeatability, and reversibility. It also creates a clear audit trail.

Packer configurations are JSON files. Most administrators agree that JSON is a poor choice of format since it's notoriously picky about quotes and commas and doesn't allow comments. With luck, HashiCorp will soon convert Packer to their much improved custom configuration format, but until then you're stuck editing JSON.

In a template, "builders" define how to create an image and "provisioners" configure and install software for the image. Builders exist for AWS, GCP, DigitalOcean, VMware, VirtualBox, and Vagrant, among others. Provisioners can be shell scripts, Chef cookbooks, Ansible roles, or other configuration management tools.

The following template, **custom_ami.json**, demonstrates AWS's amazon-ebs builder and the shell provisioner.

```
{
    "builders": [{
        "type": "amazon-ebs",
        "access_key": "AKIAIOSFODNN7EXAMPLE",
        "secret_key": "wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY",
        "region": "us-west-2",
        "source_ami": "ami-d440a6e7",
        "instance_type": "t2.medium",
        "ssh_username": "ubuntu",
        "ssh_timeout": "5m",
        "subnet_id": "subnet-ef67938a",
        "vpc_id": "vpc-516b8934",
        "associate_public_ip_address": true,
        "ami_virtualization_type": "hvm",
        "ami_description": "ULSAH AMI",
        "ami_name": "ULSAH5E",
        "tags": {
            "Name": "ULSAH5E Demo AMI"
        }
    }],
    "provisioners": [
        {
            "type": "shell",
            "source": "customize_ami.sh"
        }
    ]
}
```

Just as the CLI tool needs certain parameters to launch an instance, the amazon-ebs builder needs data such as API credentials, instance type, the source AMI on which to base the new image, and a VPC subnet where the instance should be located. Packer uses SSH to execute the provisioning step, so we make sure the instance has a public IP address.

In this case, the provisioner is a shell script called **customize_ami.sh**. Packer copies the script to the remote system with **scp** and runs it. There's nothing special about such a script; it can do anything you'd normally do from a script. For example, it can add new users, download and configure software, or execute security hardening steps.

To create the AMI, invoke **packer build**:

```
$ packer build custom_ami.json
```

packer build notes each step of the creation process on the console. The amazon-ebs builder takes the following steps:

1. It automatically creates a key pair and a security group.
2. It starts the instance and waits for it to become accessible on the network.

3. It uses **scp** and **ssh** to perform the requested provisioning steps.
4. It creates an AMI by calling the EC2 CreateImage API.
5. It cleans up by terminating the instance.

If everything works correctly, Packer prints the AMI ID as soon as it is available for use. If a problem occurs during the build, Packer prints a magenta-colored error message and exits after cleaning up after itself.

The **-debug** argument to **packer build** pauses at each step to let you troubleshoot problems. You can also use the `null` builder to fix any errors without launching an instance each time you try to run a build.

24.7 VAGRANT

Also developed by HashiCorp, Vagrant is a wrapper that sits on top of virtualization platforms such as VMware, VirtualBox, and Docker. However, it is not itself a virtualization platform.

Vagrant simplifies virtual environment provisioning and configuration. Its mission is to quickly and easily create disposable, preconfigured development environments that closely mirror production environments. This glue function lets developers write and test code with minimal involvement from sysadmins or an operations team.

It's possible (but not required) to use Vagrant in combination with Packer. For example, you might standardize the base image that you use for your production platforms through Packer, then distribute a Vagrant build of that image to developers. The developers can then spin up an instance of the image on their laptop or cloud provider of choice, with any necessary customizations. This method balances the need for centralized management of production images with developers' need for access to a similar environment that they can directly control.

24.8 Recommended reading

The web site virtualization.info is an excellent source of current news, trends, and gossip in the virtualization and cloud computing sectors.

Hashimoto, Mitchell. *Vagrant: Up and Running: Create and Manage Virtualized Development Environments*. Sebastopol, CA: O'Reilly Media, 2013.

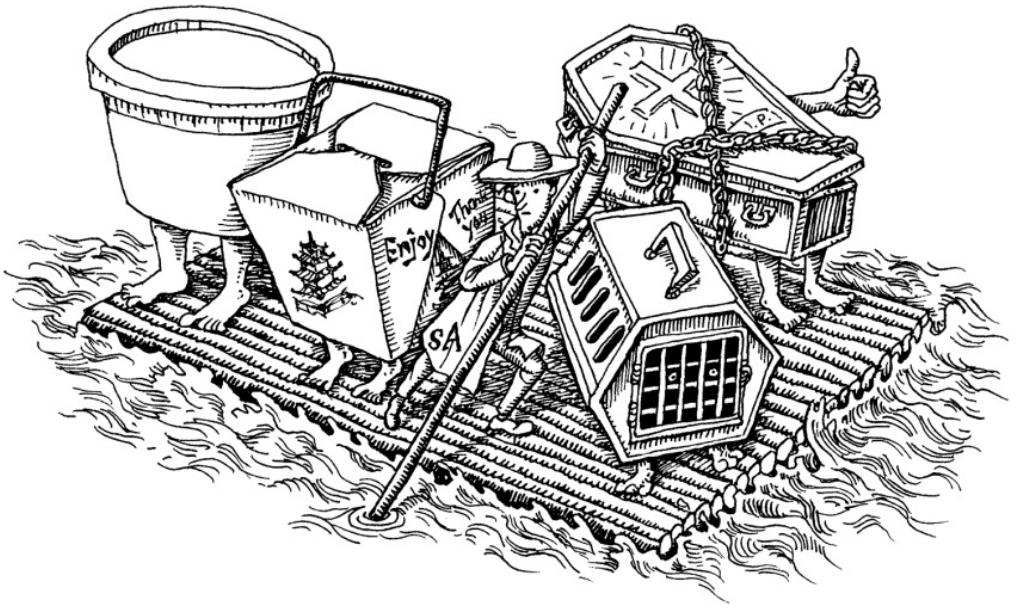
Kusnetsky, Dan. *Virtualization: A Manager's Guide: Big Picture of the Who, What, and Where of Virtualization*. Sebastopol, CA: O'Reilly Media, 2011.

Mackey, Tim, and J. K. Benedict. *XenServer Administration Handbook: Practical Recipes for Successful Deployments*. Sebastopol, CA: O'Reilly Media, 2016.

Senthil, Nathan. *VirtualBox at Warp Speed: Virtualization with VirtualBox*. Seattle, WA: Amazon Digital Services, 2015.

Troy, Ryan, and Matthew Helmke. *VMware Cookbook: A Real-World Guide to Effective VMware Use, 2nd Edition*. Sebastopol, CA: O'Reilly Media, 2012.

25 Containers



Few technologies have generated as much excitement and hype in recent years as the humble container, whose explosion in popularity coincided with the release of the open source Docker project in 2013. Containers are of particular interest to system administrators because they standardize software packaging, an ambition that has long been tantalizingly out of reach.

To illustrate the utility of containers, consider a typical web application developed in any modern language or framework. At a minimum, the following ingredients are needed to install and run the app:

- The code for the application and its correct configuration
- Libraries and other dependencies, potentially numbered in the dozens, each pinned to a specific version that is known to be compatible
- An interpreter (e.g., Python or Ruby) or run time (JRE) to execute the code, also version pinned
- Localizations such as user accounts, environment settings, and services provided by the operating system

A typical site runs dozens or hundreds of such applications. Maintaining uniformity in each of these areas across multiple application deployments is a constant challenge, even with the assistance of the tools discussed in Chapter 23, *Configuration Management*, and Chapter 26,

Continuous Integration and Delivery. Incompatible dependencies required by separate applications lead to systems that are underutilized because they cannot be shared. In addition, at sites where software developers and system administrators are functionally separated, careful coordination is needed because it's not always straightforward to identify who's responsible for what parts of the operating environment.

A container image simplifies matters by packaging an application and its prerequisites into a standard, portable file. Any host with a compatible container run-time engine can create a container by using the image as a template. Tens or hundreds of containers can run simultaneously without conflicts. With images typically being a few hundred megabytes in size or less, it's practical to copy them among systems. This easy application portability is perhaps the primary reason for the popularity of containers.

This chapter focuses on Docker. The eponymous business behind Docker has played a central role in bringing containers into mainstream use, and the Docker ecosystem is the one you're most likely to encounter as a system administrator. Docker, Inc., offers several products related to containers, but we limit our discussion to the main container engine and the Swarm cluster manager.

Several viable alternative container engines are available. rkt, from CoreOS, is the most complete. It has a cleaner process model than Docker and a more secure default configuration. rkt integrates well with the Kubernetes orchestration system. **systemd-nspawn**, from the **systemd** project, is another option for lightweight containers. It has fewer features than Docker or rkt, but in some cases that can be a good thing. rkt cooperates with **systemd-nspawn** to configure container namespaces.

25.1 Background and core concepts

The container's rapid rise to grace can be attributed more to timing than to the emergence of any single technology. Containers are a fusion of numerous existing kernel features, filesystem tricks, and networking hacks. A container engine is the management software that pulls it all together.

In essence, a container is an isolated group of processes that are restricted to a private root filesystem and process namespace. The contained processes share the kernel and other services of the host OS, but by default they cannot access files or system resources outside their container. Applications that run within a container are not aware of their containerized state and do not require modification.

After you read the following sections, it should be clear that containers contain no magic. In fact, they rely on some features of UNIX and Linux that have been around for many years. See Chapter 24, *Virtualization*, for a description of how containers differ from virtual machines.

Kernel support

The container engine uses several kernel features that are essential for isolating processes. In particular:

- *Namespaces* isolate containerized processes from the perspective of several operating system facilities, including filesystem mounts, process management, and networking. The mount namespace, for example, shows processes a customized view of the filesystem hierarchy. Containers can run with varying levels of integration with the host operating system, depending on how these namespaces have been configured.
- *Control groups* (contextually abbreviated to cgroups) limit the use of system resources and prioritize certain processes over others. Cgroups prevent runaway containers from consuming all available CPU and memory.
- *Capabilities* allow processes to execute certain sensitive kernel operations and system calls. For example, a process might have a capability that permits it to change the ownership of a file or to set the system time.
- *Secure computing mode* (usually shortened to seccomp) restricts access to system calls. It allows more fine-grained control than do capabilities.

Development of these features was driven in part by the Linux Containers project, LXC, which began at Google in 2006. LXC was the basis of Borg, Google's internal virtualization platform. LXC supplies the raw functions and tools needed to create and run Linux containers, but with more than 30 command-line tools and configuration files, it's quite complicated. The first few releases of Docker were essentially user-friendly wrappers that made LXC easier to use.

Docker now relies on an improved, standards-based container run time dubbed **containerd**. It too relies on Linux namespaces, cgroups, and capabilities to isolate containers. Learn more at containerd.io.

Images

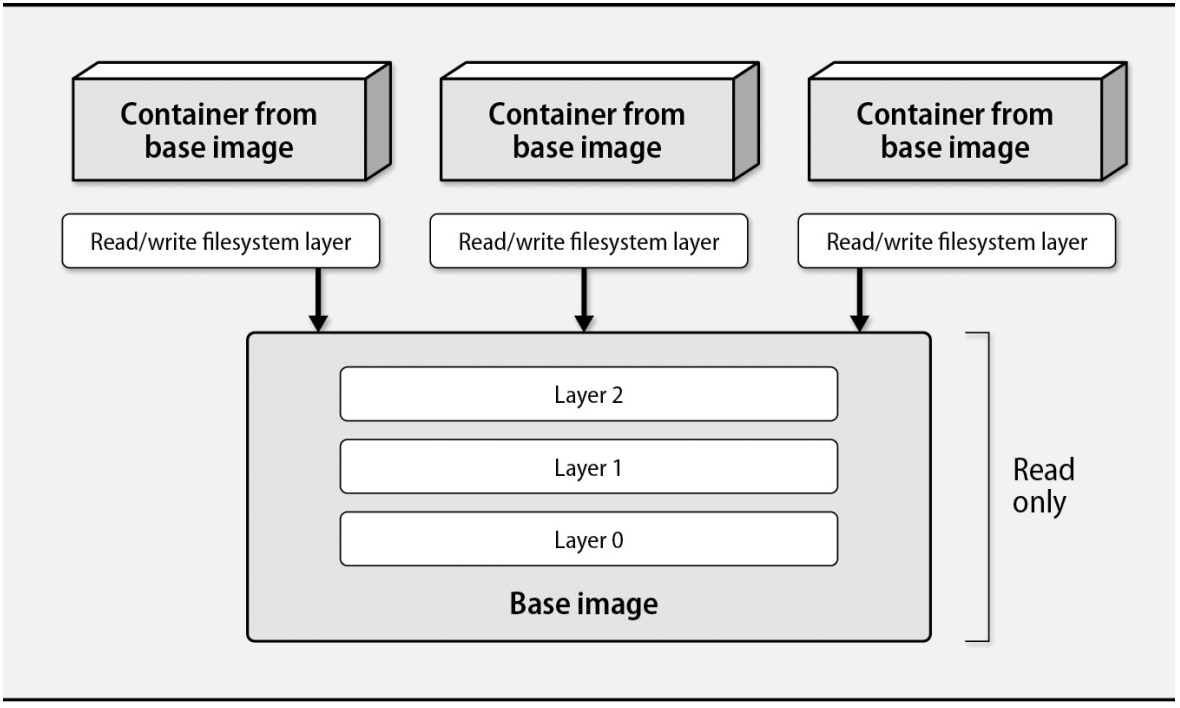
A container image is akin to a template for a container. Images rely on union filesystem mounts for performance and portability. Unions overlay multiple filesystems to create a single, consistent hierarchy. The LWN.net article “A brief history of union mounts” describes the relevant background. The related articles are interesting reading, too. See lwn.net/Articles/396020.

Container images are union filesystems that are organized to resemble the root filesystem of a typical Linux distribution. The directory layout and the locations of binaries, libraries, and supporting files all conform to standard Linux filesystem hierarchy specifications. Specialized Linux distributions have been developed for use as the basis of container images.

To create a container, Docker points to the read-only union filesystem of an image and adds a read/write layer that the container can update. When containerized processes modify the filesystem, their changes are transparently saved within the read/write layer. The base remains unmodified. This is known as a copy-on-write strategy.

Many containers can share the same immutable base layers, thus improving storage efficiency and reducing startup times. Exhibit A depicts the scheme.

Exhibit A: Docker images and the union filesystem


Container from base image
Container from base image
Container from base image
Read/write filesystem layer
Read/write filesystem layer
Read/write filesystem layer
Layer 2
Layer 1
Layer 0
Base image
Read only

Networking

The default way to connect containers to the network is to use a network namespace and a bridge within the host. In this configuration, containers have private IP addresses that aren't reachable from outside the host. The host acts as a poor man's IP router and proxies traffic between the outside world and the containers. This architecture gives administrators control over which container ports are exposed to the outside world.

It's also possible to forgo the private container addressing scheme and expose entire containers directly to the network. This is called host mode networking, and it means that the container has unfettered access to the host's network stack. This might be desirable in some situations, but it also presents a security risk because the container is not fully isolated.

See *Docker networks* for more details.

25.2 Docker: the open source container engine

Docker, Inc.'s primary product is a client/server application that builds and manages containers. The Docker container engine, written in Go, is highly modular. Separate, individual projects manage pluggable storage, networking, and other features.

Docker, Inc., is not without controversy. Its tools tend to evolve rapidly, and new versions have sometimes been incompatible with existing deployments. Some sites worry that relying on Docker's ecosystem will result in vendor lock-in. And as with any new technology, containers introduce complexity and require some study to understand.

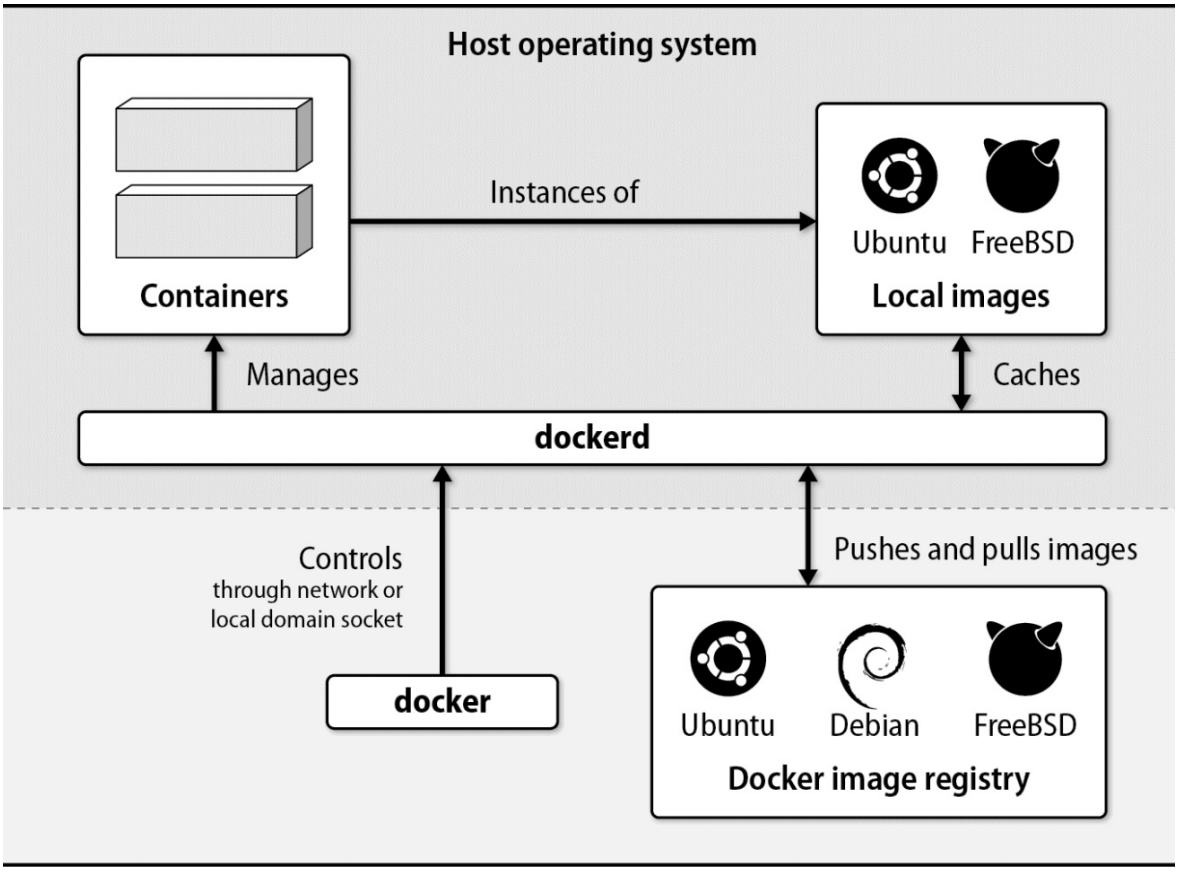
To counter these sources of resistance, Docker, Inc., became one of the founding members of the Open Container Initiative, a consortium whose mission is to guide the growth of container technology in a healthily competitive direction that fosters standards and collaboration. You can learn more at opencontainers.org. In 2017, Docker founded the Moby project and contributed the primary Docker Git repository to it to facilitate easier community development of the Docker execution engine. Refer to mobyproject.org for details.

Our discussion of Docker is based on version 1.13.1. Docker maintains an exceptionally rapid pace of development, and the current features are a moving target. We focus on the nuts and bolts here, but be sure to supplement our tutorial with the reference material at docs.docker.com. You might also dip your toes into the Moby sandbox at play-with-moby.com and the Docker lab environment at labs.play-with-docker.com.

Basic architecture

docker is an executable command that handles all management tasks for the Docker system. **dockerd** is the persistent daemon process that implements container and image operations. **docker** can run on the same system as **dockerd** and can communicate with it through UNIX domain sockets, or it can contact **dockerd** from a remote host over TCP. The architecture is depicted in Exhibit B.

Exhibit B: Docker architecture



dockerd owns all the scaffolding needed to run containers. It creates the virtual network plumbing and maintains the data directory in which containers and images are stored (**/var/lib/docker** by default). It's responsible for creating containers by invoking the appropriate system calls, setting up union filesystems, and executing processes. In short, it is the container management software.

Administrators interface with **dockerd** from the command line by running subcommands of the **docker** client. You can create a container with **docker run**, for example, or view information about the server with **docker info**. Table 25.1 summarizes some frequently used subcommands.

Table 25.1: Frequently used docker subcommands

Subcommand	What it does
docker info	Displays summary information about the daemon
docker ps	Displays running containers
docker version	Displays extensive version info about the server and client
docker rm	Removes a container
docker rmi	Removes an image
docker images	Displays local images
docker inspect	Displays the configuration of a container (JSON output)
docker logs	Displays the standard output from a container
docker exec	Executes a command in an existing container
docker run	Runs a new container
docker pull/push	Downloads images from or uploads images to a remote registry
docker start/stop	Starts or stops an existing container
docker top	Displays containerized process status

An image is the *template* for a container. It includes the files that processes running within the container instance depend on, such as libraries, operating system binaries, and applications. Linux distributions can function as convenient base images because they define a complete operating environment. However, an image is not necessarily based on a Linux distribution. The "scratch" image is an explicitly empty image intended as a basis for creation of other, more practical images.

A container relies on the image template as a basis for execution. When **dockerd** runs a container, it creates a writable filesystem layer that is separate from the source image. The container can read any of the files and other metadata stored within the image, but any writes are confined to the container's own read/write layer.

An image registry is a centralized collection of images. **dockerd** communicates with registries when you **docker pull** an image that isn't already present or when you **docker push** one of your own images. The default registry is Docker Hub, which stockpiles images for many popular applications. Most standard Linux distributions also publish Docker images.

You can run your own registry, or you can add your custom images to private registries that are hosted on Docker Hub. Any system with Docker can pull images from a registry as long as the registry server is accessible over the network.

Installation

Docker runs on Linux, macOS, Windows, and FreeBSD, but Linux is the flagship platform. FreeBSD support is considered experimental. Visit docker.com to choose the installation method that best suits your environment.

Users in the docker group can control the Docker daemon through its socket, which effectively gives those users root privileges. This is a significant security risk, so we suggest that you use **sudo** to control access to **docker** rather than adding users to the docker group. In the examples below, we run **docker** commands as the root user.

The installation process may not immediately start the daemon. If it isn't running, start it through the system's normal **init** system. On CentOS, for example, run **sudo systemctl start docker**.

Client setup

If you're connecting to a local **dockerd** and you're in the docker group or have **sudo** privileges, no client configuration is necessary. The **docker** client connects to **dockerd** through a local socket by default. You can modify the default client behavior by setting environment variables.

To connect to a remote **dockerd**, set the DOCKER_HOST environment variable. The usual HTTP port for the daemon is 2375, and the TLS version is 2376.

For example:

```
$ export DOCKER_HOST=tcp://10.0.0.10:2376
```

Always use TLS to communicate with remote daemons. If you use plain HTTP, you may as well hand out root privileges freely to anyone on your network. You can find additional details on Docker TLS configuration in *Use TLS*.

We also suggest enabling the content trust:

```
$ export DOCKER_CONTENT_TRUST=1
```

This feature validates the integrity and publisher of Docker images. Enabling the content trust prevents the client from pulling images that are not trusted.

If you run **docker** through **sudo**, you can prevent **sudo** from purging your environment variables with the **-E** flag. You can also whitelist specific environment variables by setting the value of the env_keep variable in **/etc/sudoers**. For example,

```
Defaults env_keep += "DOCKER_CONTENT_TRUST"
```

The container experience

To create a container, you need an image to use as a template. The image has all the filesystem bits needed to run programs. A new installation of Docker has no images. You can review the available images by browsing hub.docker.com. To download images from the Docker Hub, use **docker pull**.

```
# docker pull debian
Using default tag: latest
latest: Pulling from library/debian
f50f9524513f: Download complete
d8bd0657b25f: Download complete
Digest: sha256:e7d38b3517548a1c71e41bffe9c8ae6d6...
Status: Downloaded newer image for debian:latest
```

The hex strings are the layers of the union filesystem. If the same layer is used by more than one image, Docker needs only a single copy. We didn't request a specific tag, or version, of the Debian image, so Docker downloaded the "latest" tag by default.

Examine the locally available images with **docker images**:

```
# docker images
REPOSITORY     TAG          IMAGE ID          CREATED          SIZE
ubuntu         latest       07c86167cdc4      2 weeks ago      187.9 MB
ubuntu         wily         b5e09e0cd052      5 days ago       136.1 MB
ubuntu         trusty       97434d46f197      5 days ago       187.9 MB
ubuntu         15.04        d1b55fd07600      8 weeks ago      131.3 MB
centos         7            d0e7f81ca65c      2 weeks ago      196.6 MB
centos         latest       d0e7f81ca65c      2 weeks ago      196.6 MB
debian         jessie       f50f9524513f      3 weeks ago      125.1 MB
debian         latest       f50f9524513f      3 weeks ago      125.1 MB
```

This machine has the images for several Linux distributions, including the just-downloaded Debian image. The same image can be tagged more than once. Notice that debian:jessie and debian:latest share an image ID; they are two different names for the same image.

Armed with an image, it's remarkably simple to run a basic container:

```
# docker run debian /bin/echo "Hello World"
Hello World
```

What just happened? Docker created a container from the Debian base image and ran the command **/bin/echo "Hello World"** inside it. (This is GNU **echo**, not to be confused with the **echo** command built into most shells**.** They do exactly the same thing.)

The container stops running when the command exits: in this case, immediately after **echo** completes. If the "debian" image didn't already exist locally, the daemon would attempt to

automatically download it before running the command. We didn't specify a tag, so the "latest" image was used by default.

We start an interactive shell with the **-i** and **-t** flags to **docker run.** The command below starts a **bash** shell within the container and connects the "outer" shell's I/O channels to it. We also assign the container a hostname, which is helpful for identifying it in logs. (Otherwise, we'd see the container's random ID in log messages.)

```
ben@host$ sudo docker run --hostname debian -it debian /bin/bash
root@debian:/# ls
bin   dev  home  lib64  mnt  proc  run   srv  tmp  var
boot  etc  lib   media  opt  root  sbin  sys  usr
root@debian:/# ps aux
USER      PID  %CPU %MEM    VSZ   RSS TTY   STAT  START  TIME  COMMAND
root        1   0.5  0.4  20236  1884 ?       Ss  19:02  0:00  /bin/bash
root        7   0.0  0.2  17492  1144 ?       R+  19:02  0:00  ps aux
root@debian:/# uname -r
3.10.0-327.10.1.el7.x86_64
root@debian:/# exit
exit
ben@host$ uname -r
3.10.0-327.10.1.el7.x86_64
```

The experience is oddly similar to accessing a virtual machine. There is a complete root filesystem, but the process tree appears nearly empty. **/bin/bash** is PID 1 because it's the command that Docker started in the container.

The result of **uname -r** is the same both inside and outside the container. That will always be the case; we show it as a reminder that the kernel is shared.

Processes in containers cannot see other processes running on the system because of PID namespacing. However, processes on the host can see the containerized processes. The PID of a process as seen from within a container differs from the PID that is visible from the host.

For real work, you need long-lived containers that run in the background and accept connections over the network. The following command runs in the background (**-d**) a container named "nginx" that's generated from the official NGINX image. We tunnel port 80 from the host into the same port within the container:

```
# docker run -p 80:80 --hostname nginx --name nginx -d nginx
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
fdd5d7827f33: Already exists
a3ed95caeb02: Pull complete
e04488adab39: Pull complete
2af76486f8b8: Pull complete
Digest: sha256:a234ab64f6893b9a13811f2c81b46cfac885cb141dcf4e275e
    d3ca18492ab4e4
Status: Downloaded newer image for nginx:latest
0cc36b0e61b5a8211432acf198c39f7b1df864a8132a2e696df55ed927d42c1d
```

We didn't have the "nginx" image locally, so Docker had to pull it from the registry. Once the image was downloaded, Docker started the container and printed its ID, a unique 65-character hexadecimal string.

docker ps shows a brief summary of running containers:

```
# docker ps
IMAGE  COMMAND                  STATUS         PORTS
nginx  "nginx -g 'daemon off"   Up 2 minutes   0.0.0.0:80->80/tcp
```

We didn't tell **docker** what to run in the container, so it used the default command that was specified when the image was created. The output shows this command to be **nginx -g 'daemon off'** which runs **nginx** as a foreground process rather than as a background daemon. The container has no **init** to manage processes, and if the **nginx** server were started as a daemon, the container would run but immediately exit when the **nginx** process forked and exited to enter the background.

Most server daemons offer a command-line flag that forces them to run in the foreground. If your software doesn't run in the foreground or if you need to run several processes in a container, you can assign a process control system such as **supervisord** to act as a lightweight **init** for the container.

With NGINX running in the container and port 80 mapped from the host, we can make HTTP requests to the container with **curl**. NGINX serves a generic HTML landing page by default.

```
host$ curl localhost
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
```

We can use **docker logs** to view the STDOUT from the container, which in this case is the NGINX access log. The only traffic is our **curl** request:

```
# docker logs nginx
172.17.0.1 - - [24/Feb/2017:19:12:24 +0000] "GET / HTTP/1.1" 200
    612 "-" "curl/7.29.0" "-"
```

We can also use **docker logs -f** to get a real-time stream of a container's output, just like running **tail -f** on a growing log file.

docker exec creates a new process in an existing container. For example, to debug or troubleshoot, we could start an interactive shell in a container:

```
# docker exec -ti nginx bash
root@nginx:/# apt-get update && apt-get -y install procps
root@nginx:/# ps ax
  PID TTY   STAT   TIME  COMMAND
    1 ?     Ss     0:00  nginx: master process nginx -g daemon off;
    7 ?     S      0:00  nginx: worker process
    8 ?     Ss     0:00  bash
   21 ?     R+     0:00  ps ax
```

Container images are as slim as possible and are often missing common administrative utilities. In this sequence, we first updated the package index and then installed **ps**, which is part of the procps package.

The process list reveals the **nginx** master daemon, an **nginx** worker, and our **bash** shell. When we exit the shell created with **docker exec**, the container continues to run. If PID 1 exited while our shell was active, the container would terminate and our shell would also exit.

We can stop and start the container:

```
# docker stop nginx
nginx
# docker ps
IMAGE  COMMAND               STATUS       PORTS
# docker start nginx
# docker ps
IMAGE  COMMAND               STATUS       PORTS
nginx  "nginx -g 'daemon off"  Up 2 minutes  0.0.0.0:80->80/tcp
```

docker start starts the container with the same arguments that were passed when the container was created with **docker run**.

When containers exit, they remain on the system in a dormant state. You can list all containers, including those that are stopped, with **docker ps -a**. It's not particularly harmful to keep unneeded old containers lying around, but it's considered poor hygiene and might cause name collisions if you reuse container names.

When we finish with the container, we can stop and remove it:

```
# docker stop nginx && docker rm nginx
```

docker run --rm runs a container and removes it automatically when it exits, but this works only for containers that are not daemonized with **-d**.

Volumes

The filesystem layers for most containers consist of static application code, libraries, and other supporting or OS files. The read/write filesystem layer allows containers to make local modifications to these layers. However, heavy reliance on the overlay filesystem isn't the best storage solution for data-intensive applications such as databases. For those kinds of apps, Docker has the notion of volumes.

A volume is an independent, writable directory within a container that's maintained separately from the union filesystem. If the container is removed, the data in the volume persists and can be accessed from the host. Volumes can also be shared among multiple containers.

We add a volume to a container with **docker**'s **-v** argument:

```
# docker run -v /data --rm --hostname web --name web -d nginx
```

If **/data** already exists within the container, any files found there are copied to the volume. We can find the volume on the host by running **docker inspect**:

```
# docker inspect -f '{{ json .Mounts }}' web
...
 "Mounts": [
    {
      "Name": "8f026ebb9c0cda27441fb7fd275c8e767685f260...f5fd1939823558",
      "Source": "/var/lib/docker/volumes/8f026ebb9c0cda...93823558/_data",
      "Destination": "/data",
      "Driver": "local",
      "Mode": "",
      "RW": true,
      "Propagation": ""
    }
  ]
```

The **inspect** subcommand returns verbose output; we applied a filter so that only the mounted volumes would be printed. If the container terminates or needs to be removed, we can find the data volume at the Source directory on the host. The Name looks more like an ID, but it's useful if we need to identify the volume later.

For a higher-level overview of volumes on the system, we run **docker volume ls**.

Docker also supports "bind mounts," which mount volumes on the host and in containers simultaneously. For example, we can bind-mount **/mnt/data** from the host to **/data** in the container with the following command:

```
# docker run -v /mnt/data:/data --rm --name web -d nginx
```

When the container writes to **/data**, the changes are also visible in **/mnt/data** on the host.

For bind-mounted volumes, Docker does not copy existing files from the container's mount directory to the volume. As with a traditional filesystem mount, the volume's contents supersede the original contents of the container's mount directory.

When running containers in the cloud, we suggest combining bind mounts with the block storage options offered by cloud providers. For example, AWS's Elastic Block Storage volumes make great backing stores for Docker bind mounts. They have built-in snapshot facilities and can move among EC2 instances. They can also be copied between nodes, which makes it straightforward for other systems to retrieve a container's data. You can leverage EBS's native snapshotting facilities to create a simple backup system.

Data volume containers

One helpful pattern that has emerged from real-world experience is the data-only container. Its purpose is to hold a volume configuration on behalf of other containers so that those containers can be easily restarted and replaced.

Create a data container by using either a normal volume or a bind-mounted volume from the host. The data container never actually runs.

```
# docker create -v /mnt/data:/data --name nginx-data nginx
```

Now you can use the data container's volume in the nginx container:

```
# docker run --volumes-from nginx-data -p 80:80 --name web -d nginx
```

The "web" container has read and write access to the **/data** volume of the data-only "nginx-data" container. "web" can be restarted, removed, or replaced, but so long as it is started with **--volumes-from**, the files in **/data** will remain persistent.

In truth, combining data persistence with containers is a bit of an impedance mismatch. Containers are meant to be created and removed at a moment's notice in response to external events. The ideal is to have a fleet of identical servers that run **dockerd**, with containers being deployable to any of the servers. Once you add persistent data volumes, however, the container becomes coupled to a particular server. As much as we'd like to be living in the ideal world, many applications do need persistent data.

Docker networks

As discussed in *Networking*, there is more than one way to connect containers to the network. During installation, Docker creates three default networking options. List them with **docker network ls**:

```
# docker network ls
NETWORK ID          NAME                DRIVER
6514e7108508        bridge              bridge
1a72c1e4b230        none                null
e0f4e608c92c        host                host
```

---

*See this page for more information about **iptables**.*

In the default bridge mode, containers reside on a private namespaced network within the host. The bridge connects the host's network to the container namespace. When you create a container and map a port from the host with **docker run -p**, Docker creates **iptables** rules that route traffic from the host's public interface to the container's interface on the bridge network.

With "host" networking, no separate network namespace is used. Instead, the container shares the network stack with the host, including all its interfaces. Ports exposed by the container are also exposed on the interfaces of the host. Some software behaves better when running with host networking, but this configuration can also lead to port conflicts and other problems.

"None" networking indicates that Docker shouldn't take any steps whatsoever to configure networking. It is intended for advanced use cases that have custom networking requirements.

Pass the **--net** argument to **docker run** to select a container's network.

Namespaces and the bridge network

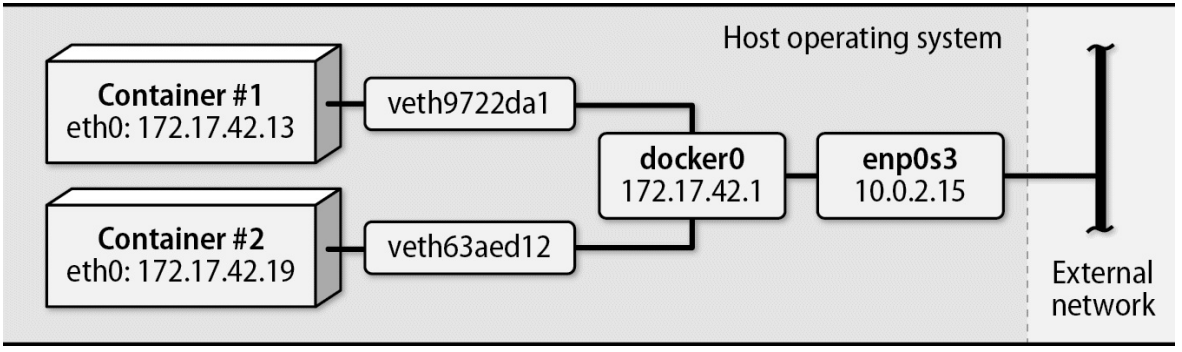
A bridge is a Linux kernel feature that connects two network segments. During installation, Docker quietly creates a bridge called docker0 on the host. Docker chooses an IP address space for the far side of the bridge that it calculates as unlikely to collide with any networks reachable by the host. Each container is given a namespaced virtual network interface that has an IP address within the bridged network range.

The address selection algorithm is practical but not perfect. Your network may have routes that aren't visible from the host. If a collision occurs, the host will no longer be able to access the remote network that has the overlapping address space, but it will be able to reach local containers. If you find yourself in this situation or if you need to customize the bridge's address space for some other reason, use the **--fixed-cidr** argument to **dockerd**.

Network namespaces rely on virtual interfaces, strange constructs that are created in pairs, where one side is in the host's namespace and the other is in the container's. Data flows in one end of

the pair and out the other end, thus connecting the container to the host. In most cases a container has only one such pair. Exhibit C illustrates the concept.

Exhibit C: A docker bridge network



One half of each pair is visible from the host's networking stack. For example, here are the visible interfaces on a CentOS host with just one container running:

```
centos$ ip addr show
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast
    state UP qlen 1000
    link/ether 08:00:27:c3:36:f0 brd ff:ff:ff:ff:ff:ff
    inet 10.0.2.15/24 brd 10.0.2.255 scope global dynamic enp0s3
       valid_lft 71368sec preferred_lft 71368sec
    inet6 fe80::a00:27ff:fec3:36f0/64 scope link
       valid_lft forever preferred_lft forever
3: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
    state UP
    link/ether 02:42:d4:30:59:24 brd ff:ff:ff:ff:ff:ff
    inet 172.17.42.1/16 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:d4ff:fe30:5924/64 scope link
       valid_lft forever preferred_lft forever
53: veth584a021@if52: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
    noqueue master docker0 state UP
    link/ether d6:39:a7:bd:bf:eb brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet6 fe80::d439:a7ff:febd:bfeb/64 scope link
       valid_lft forever preferred_lft forever
```

The output shows enp0s3, the primary interface on the host, and docker0, the virtual Ethernet bridge, which uses the 172.17.42.0/16 range. The veth interface is the host side of the virtual interface pair that connects the container to the bridged network.

The container's side of the bridged pair is not visible from the host without low-level inspection of the networking stack. This invisibility is just a side effect of the way network namespaces

work. However, we can find the interface by inspecting the container itself:

```
# docker inspect -f '{{ json .NetworkSettings.Networks.bridge }}' nginx
    "bridge": {
      "Gateway": "172.17.42.1",
      "IPAddress": "172.17.42.13",
      "IPPrefixLen": 16,
      "MacAddress": "02:42:ac:11:00:03"
    }
```

The container's IP address is 172.17.42.13, and the default gateway is the docker0 bridge interface. (This is the bridge network depicted in Exhibit C.)

In the default bridge configuration, all containers can communicate with one another because they are all on the same virtual network. However, you can create additional network namespaces to isolate containers from one another. That way, you can serve multiple, isolated environments from the same set of container instances.

Network overlays

Docker has lots of additional networking flexibility available to help with advanced use cases. For example, you can create user-defined private networks that automatically have container linking. With network overlay software, containers running on separate hosts can route traffic to each other through a private network address space. Virtual eXtensible LAN (VXLAN) technology, described in RFC7348, is one system that can be combined with containers to implement advanced networking capabilities. See the Docker networking documentation for more details.

Storage drivers

UNIX and Linux systems offer multiple ways to implement a union filesystem. Docker is technology-agnostic in this regard and filters all filesystem operations through a storage driver that you select.

The storage driver is configured as part of the **docker daemon** launch options. Your choice of storage engine has important consequences for performance and stability, especially in production environments that support many containers. Table 25.2 shows the current menu of drivers.

Table 25.2: Docker storage drivers

Driver	Description and comments
`aufs`	A reimplementation of the original UnionFS The original Docker storage engine Default for Debian and Ubuntu Now deprecated because it's not part of the mainline Linux kernel
`btrfs`	Uses the Btrfs copy-on-write filesystem Btrfs is stable and is included in the mainline Linux kernel Docker's use is distribution-limited and somewhat experimental
`devicemapper`	Default for RHEL/CentOS 6 Direct LVM mode strongly recommended but needs configuration Has a history of bugs Study Docker's devicemapper documentation
`overlay`	Based on OverlayFS Considered the replacement for AuFS The default in CentOS 7 if the overlay kernel module is loaded
`vfs`	Not a real union filesystem Slow but stable, suitable for some production environments Good as a proof of concept or as a testbed
`zfs`	Uses the ZFS copy-on-write filesystem Default for FreeBSD Considered experimental on Linux

The VFS driver effectively disables the use of a union filesystem. Docker creates a complete copy of an image for each container, resulting in higher disk usage and longer container start times. However, this implementation is simple and robust. If your use case involves long-lived containers, VFS is a reliable choice. We've never encountered a site that uses VFS in production, however.

Btrfs and ZFS are also not true union filesystems. However, they implement overlays efficiently and reliably because they natively support copy-on-write filesystem clones. Docker support for Btrfs and ZFS is currently limited to a few specific Linux distributions (and FreeBSD, for ZFS), but these are good options to keep an eye on for the future. The less filesystem magic specific to the container system, the better.

Storage driver selection is a nuanced topic. Unless you or somebody on your team has comprehensive knowledge of one of these filesystems, we recommend that you stick with the default for your distribution. The Docker storage driver documentation has further information.

dockerd option editing

You'll inevitably need to modify some of **dockerd**'s settings. Tuning options include the storage engine, DNS options, and the base directory in which images and metadata are stored. Run **dockerd -h** to see a complete list of arguments.

You can examine a running daemon's configuration with **docker info**:

```
centos# docker info
Containers: 6
  Running: 0
  Paused: 0
  Stopped: 6
Images: 9
Server Version: 1.10.3
Storage Driver: overlay
  Backing Filesystem: xfs
Logging Driver: json-file
Plugins:
  Volume: local
  Network: bridge null host
Kernel Version: 3.10.0-327.10.1.el7.x86_64
Operating System: CentOS Linux 7 (Core)
OSType: linux
Architecture: x86_64
```

This is a good place to check that any customizations you made have taken effect.

Docker conforms to the operating system's native **init** system for managing daemon processes, including settings for startup options. For example, on a distribution that uses **systemd**, the following command edits the Docker service unit to set a nondefault storage driver, a set of DNS servers, and a custom address space for the bridge network:

```
$ systemctl edit docker
[Service]
ExecStart=
ExecStart=/usr/bin/docker daemon -D --storage-driver overlay \
  --dns 8.8.8.8 --dns 8.8.4.4 --bip 172.18.0.0/19
```

The redundant `ExecStart=` is not a mistake. It's a **systemd**-ism that clears the default setting, ensuring that the new definition is used exactly as shown. Once the edits are complete, we restart the daemon with **systemctl** and review the changes:

```
centos$ sudo systemctl restart docker
centos$ sudo systemctl status docker
  docker.service
     Loaded: loaded (/etc/systemd/system/docker.service; static;
       vendor preset: disabled)
  Drop-In: /etc/systemd/system/docker.service.d
           └─override.conf
   Active: active (running) since Wed 2016-03-09 23:14:56 UTC; 12s ago
 Main PID: 4328 (docker)
   CGroup: /system.slice/docker.service
           └─4328 /usr/bin/docker daemon -D --storage-driver overlay
     --dns 8.8.8.8 --dns 8.8.4.4 --bip 172.18.0.0/19
...
```

On systems running **upstart**, configure daemon options in **/etc/default/docker**. For older systems with SysV-style **init**, use **/etc/sysconfig/docker**.

By default, **dockerd** listens for connections from **docker** on the UNIX domain socket at **/var/run/docker.sock**. To set the daemon to listen on a TLS socket instead, use the daemon option **-H tcp://0.0.0.0:2376**. See this page for more details about how to set up TLS.

Image building

You can containerize your own applications by building images that include your application code. The build process begins with a base image. You add your application by committing any changes as new layers and saving the image to the local image database. You can then create containers from the image. You can also push your image to a registry to make it accessible to other systems running Docker.

Each layer of an image is identified by a cryptographic hash of its contents. The hash serves as a validation system that lets Docker confirm that no corruption or malicious intervention has modified the contents of the image.

Choosing a base image

Before creating a custom image, choose a suitable base. The rule of thumb for base images is that the smaller footprint, the better. The base should have what you need to run your software and nothing more.

Many of the official images are based on a distribution called Alpine Linux, which weighs in at a lean 5MB but may have library incompatibilities with some applications. The Ubuntu image is larger at 188MB, but still small in comparison to a typical server installation. You might be able to find a base image that has your application run-time components already configured. Default base images exist for the most common languages, run-times, and application platforms.

Thoroughly vet your base image before you make a final decision. Examine the base's **Dockerfile** (see the next section) and any nonobvious dependencies to avoid surprises. Base images may have unexpected requirements or include vulnerable versions of software. In some circumstances, you may need to copy the **Dockerfile** of a base image and rebuild it to suit your needs.

When **dockerd** downloads an image, it downloads only the layers that it doesn't already have. If all your applications use the same base, there is less data for the daemon to download and containers start faster when first run.

*Building from a **Dockerfile***

A **Dockerfile** is a recipe for building an image. It contains a series of instructions and shell commands. The **docker build** command reads the **Dockerfile**, runs its instructions in sequence, and commits the result as an image. Software projects that have a **Dockerfile** usually keep it in the root directory of the Git repository to facilitate building new images that contain that software.

The first instruction in a **Dockerfile** specifies an image to use as the base. Each subsequent instruction commits a change to a new layer, which is used in turn as the base for the next

instruction. Each layer includes only the changes from the previous layer. The union filesystem merges the layers to create a container's root filesystem.

Here is a **Dockerfile** that builds the official NGINX image for Debian. This is slightly simplified from github.com/nginxinc/docker-nginx.

```
FROM debian:jessie
MAINTAINER NGINX Docker Maintainers "docker-maint@nginx.com"
ENV NGINX_VERSION 1.10.3-1~jessie
RUN apt-get update \
    && apt-get install -y ca-certificates nginx=${NGINX_VERSION} \
    && rm -rf /var/lib/apt/lists/*
# forward request and error logs to docker log collector
RUN ln -sf /dev/stdout /var/log/nginx/access.log \
    && ln -sf /dev/stderr /var/log/nginx/error.log
EXPOSE 80 443
CMD ["nginx", "-g", "daemon off;"]
```

NGINX uses the debian:jessie image as a base. After declaring a maintainer, the file sets an environment variable (NGINX_VERSION) that's then available to every subsequent instruction in the **Dockerfile** and also to any process that runs inside the container once the image has been built and instantiated. The first RUN instruction does the heavy lifting by installing NGINX from a package repository.

By default, NGINX sends log data to **/var/log/nginx/access.log**, but the convention for containers is to log messages to STDOUT. In the final RUN command, the maintainers use a symbolic link to redirect the access log to the STDOUT device file. Similarly, errors are redirected to the container's STDERR.

The EXPOSE command tells **dockerd** which ports the container listens on. The exposed ports can be overridden at container run time with the **-p** option to **docker run**.

The final instruction in the NGINX **Dockerfile** is the command that **dockerd** should execute when it starts the container. In this case, the container runs the **nginx** binary as a foreground process.

See Table 25.3 for a rundown of common **Dockerfile** instructions. The reference manual at docs.docker.com is the authoritative documentation.

Table 25.3: Abbreviated list of Dockerfile instructions

Instruction	What it does
ADD	Copies files from the build host to the image[a]
ARG	Sets variables that can be referenced during the build but not from the final image; not intended for secrets
CMD	Sets the default commands to execute in a container
COPY	Like ADD, but only for files and directories
ENV	Sets environment variables available to all subsequent build instructions and containers spawned from this image
EXPOSE	Informs **dockerd** of the network ports exposed by the container
FROM	Sets the base image; must be the first instruction
LABEL	Sets image tags (visible with **docker inspect**)
RUN	Runs commands and saves the result in the image
STOPSIGNAL	Specifies a signal to send to the process when told to quit with **docker stop**; defaults to SIGKILL
USER	Sets the account name to use when running the container and any subsequent build instructions
VOLUME	Designates a volume for storing persistent data
WORKDIR	Sets the default working directory for subsequent instructions

a. The source can be a file, directory, tarball, or remote URL.

*Composing a derived **Dockerfile***

We can use a very simple **Dockerfile** to build a derived NGINX image that adds a custom **index.html**, replacing the default from the official image:

```
$ cat index.html
<!DOCTYPE html>
<title>ULSAH index.html file</title>
<p>A simple Docker image, brought to you by ULSAH.</p>
$ cat Dockerfile
FROM nginx
# Add a new index.html to the document root
ADD index.html /usr/share/nginx/html/
```

Other than having a custom **index.html**, our new image will be identical to the base image. Here's how we build the customized image:

```
# docker build -t nginx:ulsah .
Step 1 : FROM nginx
  ---> fd19524415dc
Step 2 : ADD index.html /usr/share/nginx/html/
  ---> c0c25eaf7415
Removing intermediate container 04cc3278fdb4
Successfully built c0c25eaf7415
```

We use **docker build** with **-t nginx:ulsah** to create an image with the name nginx and the tag ulsah to distinguish it from the official NGINX image. The trailing dot tells **docker build** where to search for the **Dockerfile** (in this case, the current directory).

Now we can run the image and see our customized **index.html**:

```
# docker run -p 80:80 --name nginx-ulsah -d nginx:ulsah
$ curl localhost
<!DOCTYPE html>
<title>ULSAH index.html file</title>
<p>A simple Docker image, brought to you by ULSAH.</p>
```

We can check that our image is listed among the local images by running the command **docker images**:

```
# docker images | grep ulsah
REPOSITORY     TAG        IMAGE ID       CREATED            SIZE
nginx          ulsah      c0c25eaf7415   3 minutes ago      134.6 MB
```

To remove images, run **docker rmi**. You can't remove an image until you've stopped and removed any containers that are using it:

```
# docker ps | grep nginx:ulsah
IMAGE        COMMAND                  STATUS         PORTS
nginx:ulsah  "nginx -g 'daemon off"   Up 37 seconds  0.0.0.0:80->80/tcp
# docker stop nginx-ulsah && docker rm nginx-ulsah
nginx-ulsah
nginx-ulsah
# docker rmi nginx:ulsah
```

Both **docker stop** and **docker rm** echo the name of the container they affect, resulting in "nginx-ulsah" being printed twice.

Registries

A registry is an index of Docker images that **dockerd** can access through HTTP. When an image is requested that doesn't exist on the local disk, **dockerd** pulls it from the registry. Images are uploaded to a registry with **docker push**. Although image operations are initated by the **docker** command, only **dockerd** actually interacts with registries.

Docker Hub is a hosted registry service run by Docker, Inc. It hosts images for many distributions and open source projects, including all our example Linux systems. The integrity of these official images is verified through a content trust system, thus ensuring that the image you download is provided by the vendor whose name is on the label. You can also publish your own images to Docker Hub for others to use.

Anyone can download public images from Docker Hub, but with a subscription you can also create private repositories. Once you have a paid account at hub.docker.com, log in from the command line with **docker login** to access the private registry so that you can push and pull your own custom images. You can also trigger an image build whenever a commit is detected on a GitHub repository.

Docker Hub is not the only subscription-based registry. Others include quay.io, Artifactory, Google Container Registry, and the Amazon EC2 Container Registry.

Docker Hub is the generous benefactor of the greater image ecosystem, and it also benefits from being the default registry when nothing more specific is requested. For example, the command

```
# docker pull debian:jessie
```

first looks for a local copy of the image. If the image isn't available locally, the next stop is Docker Hub. You can tell **docker** to use a different registry by including a hostname or URL in the image specification:

```
# docker pull registry.admin.com/debian:jessie
```

Similarly, when building an image to push to a custom registry, you must tag it with the registry's URL, and you must authenticate before you push:

```
# docker tag debian:jessie registry.admin.com/debian:jessie .
...
# docker login https://registry.admin.com
Username: ben
Password: <password>
# docker push registry.admin.com/debian:jessie
```

Docker saves the login details to a file in your home directory called **.dockercfg** so that you need not log in again the next time you interact with the private registry.

For performance or security reasons, you might prefer to run your own image registry. The registry project is open source (github.com/docker/distribution), and a simple registry is easy to run as a container:

```
# docker run -d -p 5000:5000 --name registry registry:2
```

The **registry:2** tag differentiates the latest-generation registry from the previous version, which implements an API that is incompatible with current versions of Docker.

After running this command, the registry service is now running on port 5000. You can pull an image from it by qualifying the name of the image you're seeking:

```
# docker pull localhost:5000/debian:jessie
```

The registry implements two authentication methods: `token` and `htpasswd`. `token` delegates authentication to an external provider, which is likely to require custom development effort. `htpasswd` is simpler and allows HTTP basic authentication for registry access. Alternatively, you can set up a proxy (e.g., NGINX) to handle authentication. Always run the registry with TLS.

The default private registry configuration is not appropriate for a large-scale deployment. Considerations for production use include storage space, authentication and authorization requirements, image cleanup, and other maintenance tasks.

As your containerized environment expands, your registry will be inundated with new images. For users working in the cloud, an object store such as Amazon S3 or Google Cloud Storage is one possible way to store all this data. The registry natively supports both services.

Better yet, you can outsource your registry functions to the registries built into your cloud platform of choice and have one less thing to worry about. Both Google and Amazon run managed container registry services. You pay for storage and for the network traffic to upload and download images.

25.3 CONTAINERS IN PRACTICE

Once you're comfortable with the general way that containers work, you'll find that certain administrative chores need to be approached differently in a containerized world. For example, how do you manage log files for containerized applications? What are some security considerations? How do you troubleshoot errors?

The list below offers a few rules of thumb to help you adjust to life inside a container:

- When your application needs to run a scheduled job, don't run **cron** in a container. Use the **cron** daemon from the host (or a **systemd** timer) to schedule a short-lived container that runs the job and exits. Containers are meant to be disposable.
- Need to log in and check out what a process is doing? Don't run **sshd** in your container. Log in to the host with **ssh**, then use **docker exec** to open an interactive shell.
- If possible, set up your software to accept its configuration information from environment variables. You can pass environment variables to containers with the **-e** *KEY=value* argument to **docker run**. Or set up many variables at once from a separate file with **--env-file** *filename*.
- Ignore the commonly dispensed advice "one process per container." That's nonsense. Split processes into separate containers only when it makes sense to do so. For example, it's usually a good idea to run an application and its database server in separate containers because they are separated by clear architectural boundaries. But it's perfectly OK to have more than one process in a container when that's appropriate. Use common sense.
- Focus on the automatic creation of containers for your environment. Write scripts to build images and upload them to registries. Make sure that software deployment procedures involve replacing containers, not updating them in place.
- On that note, avoid maintaining containers. If you're accessing a container manually to fix something, figure out what the problem is, resolve it in the image, then replace the container. Immediately update your automation tooling if necessary.
- Stuck? Ask questions on the Docker User mailing list, on the Docker Community Slack, or in the #docker IRC channel on freenode.

Everything an application needs to function should be available within its container: the filesystem, network access, and kernel facilities. The only processes that run in a container are the ones that you start. It is atypical of containers to run normal OS services such as **cron**, **rsyslogd**, and **sshd**, although it is certainly possible to do so. Those duties are best left to the host

OS. If you find yourself needing these services within a container, reconsider your problem and see if you can solve it in a more container-friendly way.

Logging

UNIX and Linux applications traditionally use syslog (now the **rsyslogd** daemon) to process log messages. Syslog handles log filtering, sorting, and routing to remote systems. Some applications don't use syslog and instead write directly to log files.

Containers do not run syslog. Instead, Docker collects the logs for you through logging drivers. Container processes need only write logs to STDOUT and errors to STDERR. Docker collects those messages and sends them to a configurable destination.

If your software supports logging only to files, apply the same technique as the NGINX example on this page: create symbolic links from log files to **/dev/stdout** and **/dev/stderr** when you build the image.

Docker forwards the log entries it receives to a selectable logging driver. Table 25.4 lists some of the more common and useful logging drivers.

Table 25.4: Docker logging drivers

Driver	What it does
`json-file`	Writes JSON logs in the daemon's data directory (default) [a]
`syslog`	Writes logs to a configurable syslog destination [b]
`journald`	Writes logs to the **systemd** journal [a]
`gelf`	Writes logs in the Graylog Extended Log Format
`awslogs`	Writes logs to the AWS CloudWatch service
`gcplogs`	Writes logs to Google Cloud Logging.
`none`	Does not collect logs

a. Log entries stored this way are accessible through the **docker logs** command.
b. Supports UDP, TCP, and TCP+TLS.

When using `json-file` or `journald`, you can access log data from the command line through **docker logs** *container-id*.

You set the default logging driver for **dockerd** with the **--log-driver** option. You can also assign a logging driver at container run time with **docker run --logging-driver**. Some drivers accept additional options. For example, the **--log-opt max-size** option configures log file rotation for the `json-file` driver. Use this option to avoid filling up the disk with log files. Refer to the Docker logging documentation for complete details.

Security advice

Container security relies on processes within containers being unable to access files, processes, and other resources outside their sandbox. Vulnerabilities that allow attackers to escape containers—known as breakout attacks—are serious but rare. The code that underlies container isolation has been in the Linux kernel since at least 2008; it's mature and stable. As with bare-metal or virtualized systems, insecure configurations are a far more likely source of compromises than are vulnerabilities in the isolation layer.

Docker maintains an interesting list of known software vulnerabilities that are and are not mitigated by containerization. See docs.docker.com/engine/security/non-events.

Restrict access to the daemon

Above all, protect the docker daemon. Because **dockerd** necessarily runs with elevated privileges, it's trivial for any user with access to the daemon to gain full root access to the host.

The following sequence of commands demonstrates the risk:

```
$ id
uid=1001(ben) gid=1001(ben) groups=1001(ben),992(docker)
# docker run --rm -v /:/host -t -i debian bash
root@e51ae86c5f7b:/# cd /host
root@e51ae86c5f7b:/host# ls
bin   dev  home  lib64  mnt  proc  run   srv  test  usr
boot  etc  lib   media  opt  root  sbin  sys  tmp   var
```

This transcript shows that any user in the docker group can mount the host's root filesystem to a container and gain full control of its contents. This is just one of many possible ways to elevate privileges through Docker.

If you use the default UNIX domain socket to communicate with the daemon, add only trusted users to the docker group, which has access to the socket. Better yet, control access through **sudo**.

Use TLS

See this page for information about TLS.

We said it before, and we'll say it again: if the docker daemon must be remotely accessible (**dockerd -H**), require the use of TLS to encrypt network communications and to mutually authenticate the client and server.

Setting up TLS involves having a certificate authority issue certificates to the docker daemon and clients. Once the key pairs and certificate authority are in place, actually enabling TLS for

docker and **dockerd** is a simple matter of supplying the right command-line arguments. Table 25.5 lists the essential settings.

Table 25.5: TLS arguments common to docker and dockerd

Argument	Meaning or argument
--tlsverify	Require authentication
--tlscert [a]	Path to a signed certificate
--tlskey [a]	Path to a private key
--tlscacert [a]	Path to the certificate of a trusted authority

a. Optional. The default locations are **~/.docker/**{**cert**,**key**,**ca**}**.pem**.

Successful use of TLS relies on a mature certificate management processes. Certificate issuance, revocation, and expiration are a few of the issues that need attention. Heavy is the burden of a security-conscious administrator.

Run processes as unprivileged users

Processes in containers should run as nonroot users, just as they should on a full-fledged operating system. This practice limits an attacker's ability to launch breakout attacks. When you are writing a **Dockerfile**, use the USER instruction to run future commands in the image under the named user account.

Use a read-only root filesystem

To further restrict containers, you can specify **docker run --read-only**, thereby limiting the container to a read-only root filesystem. This works well for stateless services that never need to write. You can also mount a read/write volume that your process can modify, but leave the root filesystem read-only.

Limit capabilities

See this page for more information about Linux capabilities.

The Linux kernel defines 40 separate capabilities that can be assigned to processes. By default, Docker containers are granted a large subset of these. You can enable an even greater subset by starting a container with the **--privileged** flag. However, this option disables many of the isolation benefits of using Docker. You can tune the specific capabilities that are available to containerized processes with the **--cap-add** and **--cap-drop** arguments:

```
# docker run --cap-drop SETUID --cap-drop SETGID debian:jessie
```

You can also drop all privileges and add back just the ones you need:

```
# docker run --cap-drop ALL --cap-add NET_RAW debian:jessie
```

Secure images

The Docker content trust feature validates the authenticity and integrity of images in a registry. The publisher of the image signs it with a secret key, and the registry validates it with the corresponding public key. This process ensures that the image was produced by the expected creator. You can use content trust to sign your own images or to validate the images in a remote registry. The feature is available on Docker Hub and on some third party registries, such as Artifactory.

Unfortunately, most of the content on Docker Hub is unsigned and should be considered untrustworthy. Indeed, most images on the Hub are never patched, updated, or audited in any way.

This lack of a proper chain of trust associated with many Docker images is representative of the miserable state of security on the Internet in general. It’s quite common for software packages to depend on third party libraries with little or no concern being given to the trustworthiness of the content that’s pulled in. Some software repositories have no cryptographic signatures whatsoever. It’s also common to find articles that actively encourage disabling validation. Responsible system administrators are highly suspicious of unknown and untrusted software repositories.

Debugging and troubleshooting

Containers bring with them a particularly heinous complement of obscure debugging techniques. When an application is containerized, its symptoms become more difficult to characterize and their root causes more puzzling. Many applications can run without modification inside a container, but in some scenarios they may behave differently. You might also encounter bugs within Docker itself. This section helps navigate these treacherous waters.

Errors usually manifest themselves in log files, so that's the first place to look for information. Use the advice in *Logging* to configure logging for containers, and always review the logs when you encounter issues.

If you experience problems with a running container, try

```
docker exec -ti containername bash
```

to open an interactive shell. From there you can attempt to reproduce the problem, examine the filesystem for evidence, and search for configuration errors.

If you see errors related to the docker daemon or if you have trouble starting it, search the issues list at github.com/moby/moby. You may find others that have the same problem, and one of them may have identified a potential fix or workaround.

Docker doesn't automatically clean up images or containers. When neglected, these remnants can consume an inordinate amount of disk space. If your container workload is predictable, configure a **cron** job to clean up by running **docker system prune** and **docker image prune.**

A related annoyance are "dangling" volumes, volumes that were once attached to a container but for which the container has since been removed. Volumes are independent of containers, so any files within them will continue to consume disk space until the volumes are destroyed. You can use the following incantation to clean out orphaned volumes:

```
# docker volume ls -f dangling=true  # List dangling volumes
# docker volume rm $(docker volume ls -qf dangling=true)  # Remove 'em
```

Base images you depend on may have a VOLUME instruction in their **Dockerfile**. If you don't notice this case, you might end up with a full disk after running a few containers from that image. You can show the volumes associated with a container by running **docker inspect**:

```
# docker inspect -f '{{ .Volumes }}' container-name
```

25.4 Container clustering and management

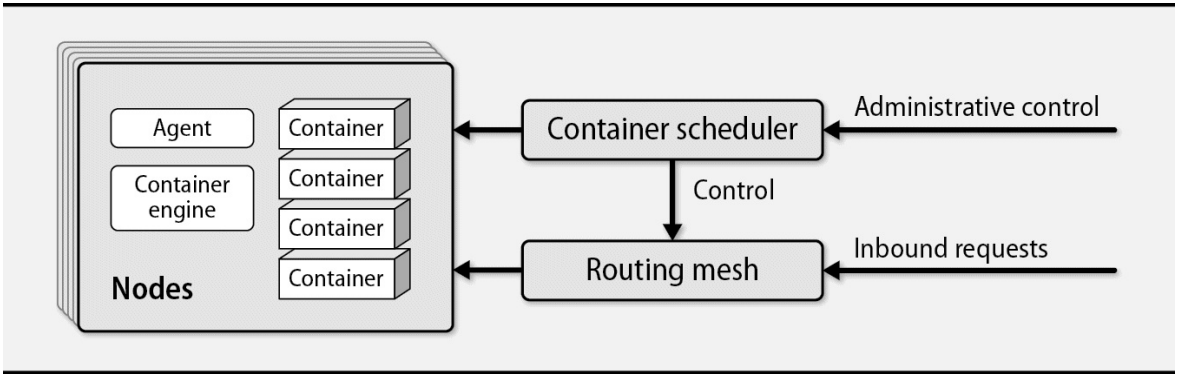
One of the great promises of containerization is the prospect of co-locating many applications on the same host while avoiding interdependencies and conflicts, thereby making more efficient use of servers. This is an appealing vision, but the Docker engine is responsible only for individual containers, not for answering the broader question of how to run many containers on distributed hosts in a highly available configuration.

Configuration management tools such as Chef, Puppet, Ansible, and Salt all support Docker. They can ensure that hosts run a certain set of containers with declared configurations. They also support image building, registry interfaces, network and volume management, and other container-related chores. These tools centralize and standardize container configuration, but they do not solve the problem of conducting the deployment of many containers across a network of servers. (Note that although configuration management systems are useful for a variety of container-related tasks, you will rarely need to use configuration management *inside* of containers.)

For network-wide container deployments, you need container orchestration software, also known as container scheduling or container management software. An entire symphony of open source and commercial tooling is available to handle large numbers of containers. Such tools are crucial for running containers at scale in a production context.

To understand how these systems work, think of the servers on the network as a farm of compute capacity. Each node in the farm offers CPU, memory, disk, and network resources to the scheduler. When the scheduler receives a request to run a container (or set of containers), it places the container on a node that has sufficient spare resources to meet the container's needs. Because the scheduler knows where containers have been placed, it can also assist in routing network requests to the correct nodes within the cluster. Administrators interact with the container management system rather than dealing with any individual container engine. Exhibit D illustrates this architecture.

Exhibit D: Basic container scheduler architecture



Container management systems supply several helpful features:

- Scheduling algorithms select the best node in light of a job's requested resources and the utilization of the cluster. For example, a job with high bandwidth requirements might be slotted onto a node with a 10 Gb/s network interface.
- Formal APIs allow programs to submit jobs to the cluster, opening the door to integration with external tools. It's easy to use container management systems in conjunction with CI/CD systems to simplify software deployments.
- Container placement can accommodate the needs of high-availability configurations. For example, an application may need to run on host nodes in several distinct geographical regions.
- Health monitoring is built in. The system can terminate and reschedule unhealthy jobs and can route jobs away from unhealthy nodes.
- It's easy to add or remove capacity. If your compute farm doesn't have enough resources available to satisfy demand, you can simply add another node. This facility is especially well suited to cloud environments.
- The container management system can interface with a load balancer to route network traffic from external clients. This facility obviates the complex administrative process of manually configuring network access to containerized applications.

One of the most challenging tasks in a distributed container system is mapping service names to containers. Remember that containers are typically ephemeral in nature and may have dynamic ports assigned. How do you map a friendly, persistent service name to multiple containers, especially when the nodes and ports change frequently? This problem is known as service discovery, and container management systems have various solutions.

It helps to be familiar with the underlying container execution engine before diving into orchestration tooling. All the container management systems we're aware of rely on Docker as the default container execution engine, although some systems also support other engines.

A synopsis of container management software

Despite their relative youth, the container management tools we discuss below are mature beyond their years and can be considered production grade. In fact, many are already used in production at high-profile, large-scale technology companies. Most are open source and have sizable user communities. Based on recent trends, we anticipate substantial development in this area in the coming years.

In the upcoming sections, we highlight the functionality and features of the most widely used systems. We also mention their integration points and common use cases.

Kubernetes

Kubernetes—sometimes shortened to “k8s” because there are eight letters between the leading “k” and the trailing “s”—has emerged as a leader in the container management space. It originated within Google and was launched by some of the same developers that worked on Borg, Google’s internal cluster manager. Kubernetes was released as an open source project in 2014 and now has more than a thousand active contributors. It has the most features and the fastest development cycle of any system we’re aware of.

Kubernetes consists of a few separate services that integrate to form a cluster. The basic building blocks include

- The API server, for operator requests
- A scheduler, for placing tasks
- A controller manager, for tracking the state of the cluster
- The Kubelet, an agent that runs on all cluster nodes
- cAdvisor, for monitoring container metrics
- A proxy, for routing incoming requests to the appropriate container

The first three items on this list run on a set of masters (which can optionally serve dual duty as nodes) for high availability. The Kubelet and cAdvisor processes run on each node, handling requests from the controller manager and reporting statistics about the health of their tasks.

In Kubernetes, containers are deployed as a “pod” which contains one or more containers. All containers in a pod are guaranteed to be co-located on the same node. Pods are assigned a cluster-wide unique IP address, and they are labeled for identification and placement purposes.

Pods are not meant to be long-lived. If a node dies, the controller schedules a replacement pod on a different node with a new IP address. Therefore, you cannot use the address of a pod as a durable name.

Services are collections of related pods with an address that is guaranteed not to change. If a pod within a service dies or fails a health check, the service removes that pod from its rotation. You can also use the built-in DNS server to assign resolvable names to services.

Kubernetes has integrated support for service discovery, secret management, deployment, and pod autoscaling. It has pluggable networking options to enable container network overlays. It can support stateful applications by migrating volumes among nodes as needed. Its CLI tool, **kubectl**, is one of the most intuitive that we’ve ever worked with. In short, it has more advanced features than we can possibly cover in this short section.

Although Kubernetes has the most active and engaged community and the most advanced features, those assets are accompanied by a steep learning curve. Recent versions have improved the experience for first-time users, but a full-fledged, customized Kubernetes deployment is not for the timid. Production k8s deployments impose a substantial administrative and operational burden.

The Google Container Engine service is implemented with Kubernetes, and it offers one of the best experiences for teams that want to run containerized workloads without the operational overhead of cluster management.

Mesos and Marathon

Mesos is an entirely different breed. It was conceived at the University of California at Berkeley around 2009 as a generic cluster manager. It quickly made its way to Twitter, where it now runs on thousands of nodes. Today, Mesos is a top-level project from the Apache Foundation and boasts a large number of enterprise users.

The major conceptual entities in Mesos are masters, agents, and frameworks. A master is a proxy between agents and frameworks. Masters relay offers of system resources from agents to frameworks. If a framework has a task to run, it chooses an offer and instructs the master to run the task. The master sends along the task details to the agent.

Marathon is a Mesos framework that deploys and manages containers. It includes a handsome user interface for managing applications and a simple, RESTful API. To run an application, you write a request definition in JSON format and submit it to Marathon through the API or the UI. Because it's an external framework, the deployment of Marathon is flexible. Marathon can run on the same node as the master for convenience, or it can run externally.

Support for multiple, coexisting frameworks is Mesos's biggest differentiator. Apache Spark, the big-data processing tool, and Apache Cassandra, a NoSQL database, both offer Mesos frameworks, thus allowing you to use Mesos agents as nodes in a Spark or Cassandra cluster. Chronos is a framework for scheduled jobs, rather like a version of **cron** that runs on a cluster instead of an individual machine. The ability to run so many frameworks on the same set of nodes is a nice feature and helps create a unified and centralized experience for administrators.

Unlike Kubernetes, Mesos does not come with batteries included. For example, load balancing and traffic routing are pluggable options that depend on your preferred solution. Marathon includes a tool, the Marathon-lb, that implements this service, or you can choose your own. We've had success using HashiCorp's Consul and HAProxy. Designing and implementing an exact solution is left as an exercise for the administrator.

Like Kubernetes, Mesos requires some contemplation to understand and use. Mesos and most of its frameworks rely on Apache Zookeeper for cluster coordination. Zookeeper is somewhat difficult to administer and is known for complex failure cases. In addition, a high-availability Mesos cluster requires a minimum of three nodes, which may be an onerous burden at some sites.

Docker Swarm

Not to be left behind, Docker offers Swarm, a container cluster manager built directly into Docker. The current incarnation of Swarm emerged in 2016 as an answer to the growing popularity of Mesos, Kubernetes, and other cluster managers that used Docker containers under the hood. Container orchestration is now a major focus for Docker, Inc.

Swarm is easier to get started with than is Mesos or Kubernetes. Any node that runs Docker can join the swarm as a worker node, and any worker node can also be a manager. There is no need to run separate nodes as masters. (Strictly speaking, this is true for Kubernetes and Mesos as well, but we've found it to be common practice to separate masters from agents in high-availability configurations.)

Starting a swarm is as simple as running **docker swarm init**. There are no additional processes to manage and configure, and there is no state to track. It works out of the box.

You can use familiar **docker** commands to run services (as in Kubernetes, collections of containers) on the swarm. You declare the state you want to achieve ("three containers running my web application") and Swarm schedules the tasks on the cluster. It automatically handles failure states and zero-downtime updates.

Swarm has a built-in load balancer that adjusts automatically as containers are added or removed. The Swarm load balancer is not as full-featured as tools such as NGINX or HAProxy, but on the other hand, it doesn't require any administrative attention.

Swarm supplies a secure Docker experience by default. All connections between nodes in a swarm are TLS-encrypted, and no configuration is required on the part of the administrator. This is a major differentiator for Swarm when compared to its competitors.

AWS EC2 Container Service

AWS offers ECS, a container management service designed for EC2 instances (AWS's native virtual servers). In a manner reminiscent of many Amazon services, AWS launched ECS with minimal functionality but has steadily enhanced it over time. ECS has matured into a fine choice for sites that are already invested in AWS and want to stick to E-Z mode.

ECS is a "mostly managed" service. The cluster manager components are operated by AWS. Users run EC2 instances that have Docker and the ECS agent installed. The agent connects to the central ECS API and registers its resource availability. To run a task on your ECS cluster, you submit a task definition in JSON format through the API. ECS then schedules the task on one of your nodes.

Because the service is mostly managed, the barrier to entry is low. You can get started with ECS in just a few minutes. The service scales well to at least hundreds of nodes and thousands of concurrent tasks.

ECS integrates with other AWS services. For example, load balancing among multiple tasks, along with the requisite service discovery, are handled by the Application Load Balancer service. You can add resource capacity to your ECS cluster by taking advantage of EC2 autoscaling. ECS also integrates with AWS's Identity and Access Manager service to grant permissions for your container tasks to interact with other services.

One of the most polished parts of ECS is the included Docker image registry. You can upload Docker images to the EC2 Container Registry, where they're stored and made available to any Docker client, whether it's running on ECS or not. If you're running containers on AWS, use the container registry in the same region as your instances. You'll achieve far better reliability and performance than with any other registry.

The ECS user interface, although functional, shares the limitations of other AWS interfaces. The AWS CLI tool has complete support for the ECS API. For management of applications on ECS, we recommend turning to third party, open source tools such as Empire (github.com/remind101/empire) or Convox (convox.com) for a more streamlined experience.

25.5 Recommended reading

Docker, Inc. *Official Docker Documentation*. docs.docker.com. Docker has good documentation. It's comprehensive and usually up to date.

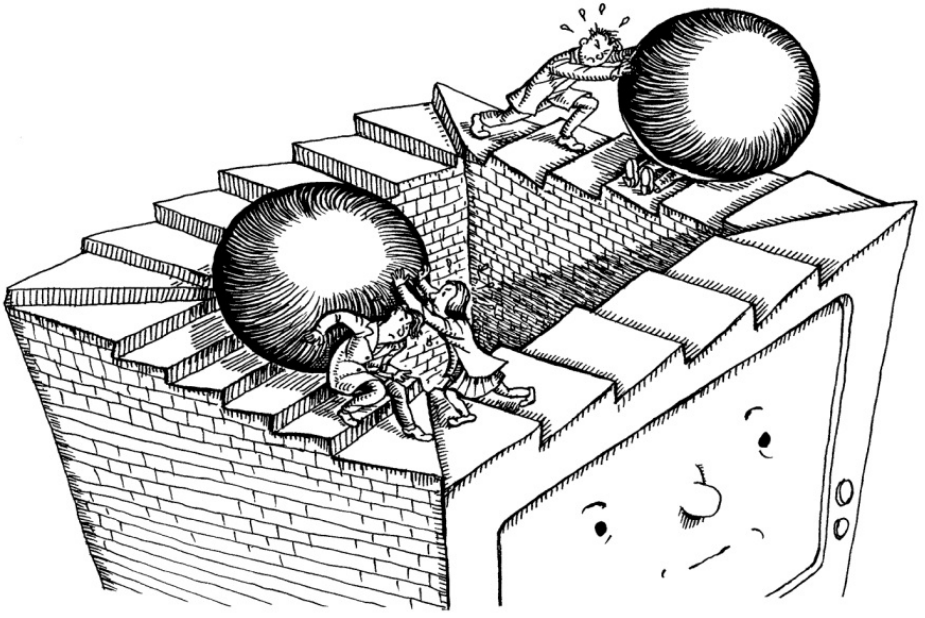
Matthias, Karl, and Sean Kane. *Docker Up & Running*. Sebastopol, CA: O'Reilly Media, 2015. This book focuses on running Docker containers in production environments.

Mouat, Adrian. *Using Docker: Developing and Deploying software with Containers*. Sebastopol, CA: O'Reilly Media, 2016. This book covers topics from basic to advanced and includes plenty of examples.

Turnbull, James. *The Docker Book*. www.dockerbook.com.

The Container Solutions blog at container-solutions.com/blog includes technical HOWTOs, best practices, and interviews with experts in the container space.

26 Continuous Integration and Delivery



Until the past decade or so, updating software was a hair-pulling, time-consuming exercise in frustration. Release processes typically involved ad hoc, home-grown scripts that were invoked in enigmatic order and saddled with outdated and incomplete documentation. Testing—if it existed at all—was performed by a quality assurance team that was far removed from the development cycle and often became a major obstacle to shipping code. Administrators, developers, and project managers would plan days-long marathons for the final stages of releasing updates to live users. Service outages were often scheduled weeks in advance.

Given this unsavory context, it should come as no surprise that some very smart people were working diligently to improve the situation. After all, where some see only problems, others see opportunity.

At top of mind is Martin Fowler, an oracle of the software industry and chief scientist of the influential development shop ThoughtWorks. In an insightful article (goo.gl/Y2lisI), Fowler describes continuous integration as “a software development practice where members of a team integrate their work frequently,” thus removing one of the major pain points of software work: the tiresome task of reconciling code fragments that have diverged dramatically over a long period of independent development. The practice of continuous integration is now ubiquitous among software development teams.

Hot on the heels of this innovation came continuous delivery, which is similar in concept but which targets a separate goal: reliably deploying updated software to live systems. Continuous delivery embraces the release of small, incremental changes to IT infrastructure. If something breaks (that is, if a "regression" is introduced), it becomes straightforward to isolate and resolve the issue because the changes between versions are small. At the extreme end, some sites aim to deploy new code to users multiple times per day. Bugs and security issues can be resolved in hours rather than weeks.

In combination, continuous integration and continuous delivery (henceforth denoted CI/CD) encompass the tools and processes needed to facilitate frequent, incremental software and configuration updates.

See this page for more comments on DevOps.

CI/CD is a pillar of the DevOps philosophy. It's the glue that holds together developers and operations folks. It is as much a business asset as a technical innovation. Once introduced, CI/CD becomes the bedrock of an IT organization because it imposes logic and organization on release processes that were previously chaotic.

Sysadmins are central to the design, implementation, and ongoing maintenance of CI/CD systems. Administrators install, configure, and operate the tools that make CI/CD function. They are responsible for ensuring that software build processes are fast and reliable.

Testing is an important element of CI/CD, and although administrators may not write the tests (though they sometimes do!), they are often responsible for setting up the infrastructure and the systems on which the tests are performed. Perhaps most importantly, it is ultimately system administrators who are responsible for deployments, the "delivery" component of CI/CD.

An effective CI/CD system is implemented not with a solitary tool but rather with a collection of software that works in unison to form a cohesive environment. Myriad open source and commercial tools are available to coordinate the various elements of CI/CD. These coordination tools rely on other software packages to do the actual work (e.g., compiling code or setting up servers in a particular configuration). Indeed, there are so many options that the initial approach to CI/CD can be overwhelming. If nothing else, the recent proliferation of tools in this space is evidence of CI/CD's growing importance to the industry.

In this chapter we attempt to navigate the maze of CI/CD concepts, terminology, and tools. We cover the basics of a CI/CD pipeline, the various types of testing and their relevance to CI/CD, the practice of running multiple environments in parallel, and some of the most popular open source tools. At the end of the chapter, we dissect an example CI/CD pipeline that uses some of the most popular tools. When you're finished with this chapter, you should understand some of the principles and techniques that go into creating a powerful, flexible CI/CD system.

26.1 CI/CD ESSENTIALS

Many terms related to CI/CD sound similar and have overlapping meanings. So let's first take a closer look at the differences between continuous integration, delivery, and deployment:

- *Continuous integration* is the process of collaborating on a shared code base, merging disparate code changes into a version control system, and automatically creating and testing builds.
- *Continuous delivery* is the process of automatically deploying builds to nonproduction environments after the continuous integration process completes.
- *Continuous deployment* closes the loop by deploying to live systems that serve real users without any operator intervention.

Continuous deployment without any human supervision can be intimidating, but that's precisely the point: the idea is to reduce the fear factor by deploying as often as possible, eliminating more and more issues until the team has enough confidence in the testing and tooling to enable automatic releases.

Continuous deployment need not be the ultimate goal of all sites. There may be compliance or risk reasons to pause at any point in the pipeline. If that's the case, you can still benefit from making each stage of the process as simple as possible for the human who pushes the final button. Every organization should choose its own boundaries.

Principles and practices

Business agility is one of the key benefits of CI/CD. Continuous deployment facilitates the release of well-tested features to production in minutes or hours instead of weeks or months. Because every change is built, tested, and deployed immediately, the delta between versions is much smaller. And that decreases the risk of deployment and helps narrow the range of possible root causes if something goes wrong. Rather than staging a small number of big-bang deployments per year, you might find yourself releasing new code multiple times per week or even per day.

CI/CD stresses the release of more features, more often. This goal is achievable only when developers write and commit code in smaller chunks. To realize continuous integration, developers need to push code changes at least once per day after running tests locally.

For administrators, CI/CD processes greatly reduce the amount of time spent preparing for and implementing releases. They also reduce the time spent debugging problems when deployments inevitably fail. Few things are more satisfying than watching a new feature release itself to production without any human intervention.

The next sections cover some basic rules of thumb to keep in mind as you develop your CI/CD processes.

Use revision control

All code should be tracked in a source control system. We recommend Git, but there are lots of options. Most software development teams use source control as a matter of course.

For sites that embrace the infrastructure-as-code concept (as we demonstrate in the section *CI/CD in practice*), you can track infrastructure-related code alongside your applications. You can even store documentation and configuration settings in source control.

Make sure version control is the single source of truth. *Nothing* can be managed manually or off the record.

Build once, deploy often

A CI/CD pipeline begins with a build. The output of the build (the “artifact”) is used from that point forward for testing and deployment. The only way to confirm that a specific build is ready to go to production is to run all tests against that build. Deploy the same artifact to at least one or two environments that match production as closely as possible.

Automate end-to-end

Building, testing, and deploying code without manual intervention is the key to reliable and reproducible updates. Even if you’re not planning to deploy code continuously to production, the final production deployment step should run fully unattended after being triggered by a human.

Build every integration commit

An integration merges changes made by multiple developers or teams of developers. The product is a composite code base that incorporates everyone's updates.

An integration does not randomly snatch work in progress out of developers' hands and stick it in the mainline code base; that's a recipe for disaster. Individual developers are responsible for managing their own development stream. When they're ready, they initiate an integration. Integrations occur as frequently as possible.

Integrations are performed through the source control system. The exact workflow varies. Individual developers might be responsible for merging their work back to the trunk, or a designated release overseer might integrate several developers or teams at once. The merge process can be largely automated, but there's always the potential for two sets of changes to conflict. That situation requires human intervention.

The idea behind continuous integration is that commits to the revision control system's integration branch automatically result in a build. The "integration branch" part is important because source control serves several purposes. In addition to being a vehicle for collaboration and integration, it's also useful as a backup system, as a checkpoint for work in progress, and as a system that lets developers work on several updates while keeping the changes related to those updates logically separate. Therefore, only commits to the integration branch result in a build.

Frequent integrations make it easy to trace a broken build back to the exact lines of code that caused the problem. The revision control system can then determine the identity of the responsible developer. But note: a broken build should carry little or no stigma. The goal is just to get the build running again. Encourage a blame-free culture within your teams.

Share responsibility

When something goes wrong, the pipeline needs to be fixed. No new code can be pushed until the previous problem has been resolved. It's the equivalent of halting the assembly line in a factory. It is the responsibility of the entire team to fix the build before resuming development work.

CI/CD shouldn't be a mysterious system that runs in the background and occasionally sends email when something is broken. Every team member should have access to the CI/CD interface to view dashboards and logs. Some sites create humorous widgets such as RGB lighting fixtures that act as a visual indicator of the pipeline's current status.

Build fast, fix fast

CI/CD is designed to yield feedback as quickly as possible, ideally within minutes after pushing code to source control. This rapid response guarantees that developers pay attention to the result. If the build fails, the developers will likely be able to fix the problem quickly because the changes they just committed are fresh in their minds. Slow build processes are

counterproductive. Strive to eliminate redundant and time-consuming steps. Ensure that your build system has enough agents, and that the agents have sufficient system resources to build quickly.

Audit and verify

Part of the CI/CD system includes a detailed history of every software release, including its progression from development to production. This auditability can be useful to ensure that only authorized builds are deployed. The settings and event timelines related to each environment can be irrefutably verified.

Environments

Applications do not run in isolation. They depend on external resources such as databases, caches, network filesystems, DNS records, remote HTTP APIs, other applications, and external network services. An execution environment includes all these resources and anything else that the application needs so it can run. Building and maintaining such environments is a target of substantial administrative attention.

Most sites run at least three environments, listed here in ascending order of importance:

- Development ("dev" for short), for integrating updates from multiple developers, testing infrastructure changes, and checking for obvious failures. Development is used mostly by the technical staff and not by business types or end users. In the context of CI/CD, the development environment may be created and reset multiple times per day.
- Staging (or "stage"), for manual and automated testing and for further vetting of changes and software updates. Some organizations call this the "test" environment. Testers, product owners, and other business stakeholders use the staging environment to review new features and bug fixes. Staging can also be used for penetration testing and other security checks.
- Production ("prod"), for implementing service for real users. The production environment usually includes extensive measures to ensure high performance and strong security. An outage in production is an all-hands-on-deck emergency that must be resolved immediately.

A typical CI/CD system promotes software through each of these environments in succession, filtering out errors and software defects along the way. You can deploy to production with confidence because you know that changes have already been tested in two other environments.

Environment parity is a subject of some complexity for administrators. The purpose of the nonproduction or "lower" environments is to prepare and scrutinize changes of all types before they are made in production. Substantive differences among environments can result in unforeseen incompatibilities that might ultimately cause degraded performance, downtime, or even the destruction of data.

For example, imagine that the development and staging environments have undergone an operating system upgrade, but production still runs the older OS version. It's time for a software deployment. The new software is thoroughly tested in dev and stage, and it seems to work fine. However, an unexpected incompatibility becomes evident during the production rollout because the older version of a certain library is missing functionality used by the new code.

This scenario is quite common, and it's one reason why administrators must be vigilant about keeping environments in sync. The closer that lower environments match production, the higher

your chances of maintaining high availability and delivering software successfully.

Running multiple environments at full capacity can be expensive and time-consuming. Because production serves far more users than the lower environments, it's usually necessary to run a larger number of more expensive systems in that environment. Production data sets tend to be larger, and the provisioned disk space and server size are beefed up to compensate.

Even this type of difference among environments can cause unanticipated problems. A load balancer misconfiguration that didn't matter in dev or stage may reveal a defect, or a database query that runs quickly in dev and stage might turn out to be far slower when applied to production-scale data.

Matching production capacity in lower environments is a tricky problem. Strive to have at least one lower environment that has redundancy in all the same places that production does (e.g., multiple web servers, fully replicated databases, and matching failover strategies for any clustered systems). It's fine for the staging servers to be smaller in size, although any tests you run to check performance will not reflect production numbers.

For best results, data sets in lower environments should be similar in size and content to those of production. One strategy is to create nightly snapshots of all relevant production data and copy it to the lower environment. For compliance and good security hygiene, sensitive user data must be anonymized before it's used this way. For truly massive data sets that are not practical to copy, import a smaller but still meaningful sample.

Despite your best efforts, lower environments will never be exactly like the production environment. Some configuration settings (such as credentials, URLs, addresses, and hostnames) will differ. Use configuration management to track these configuration items among environments. When the CI/CD system runs a deployment, consult your source of truth to find the relevant configuration for that environment and make sure that all environments are deployed in the same way.

Feature flags

A feature flag enables or disables an application feature depending on the value of a configuration setting. Developers can build support for feature flags into their software. You can use feature flags to enable certain features in specific environments. For example, you can enable a feature for the staging environment while keeping it disabled in production until it’s fully tested and ready for the user base.

For instance, consider an e-commerce application that has a shopping cart. The business wants to run a promotion that requires some changes to the code. The development team can build the feature and release it to all three environments in advance, but enable it only on dev and stage. When the business is ready to advertise and activate the promotion, enabling the feature becomes a simple, low-risk configuration change rather than a software release. If the feature has a bug, it’s easy to disable it without updating the software.

26.2 Pipelines

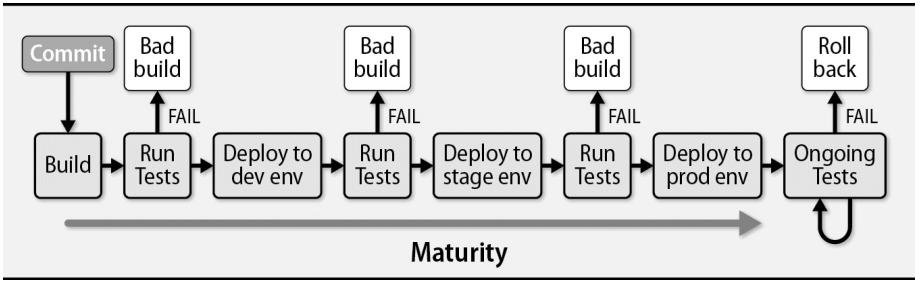
A CI/CD pipeline is a series of steps, called “stages,” that run in sequence. Each stage is essentially a script that performs tasks specific to your software project.

At the most basic level, a CI/CD pipeline

- Reliably builds and packages software
- Runs a series of automated tests to search for bugs and configuration errors
- Deploys code to one or more environments, culminating in production

Exhibit A illustrates the stages in a simple (yet mature) CI/CD pipeline.

Exhibit A: A basic CI/CD pipeline



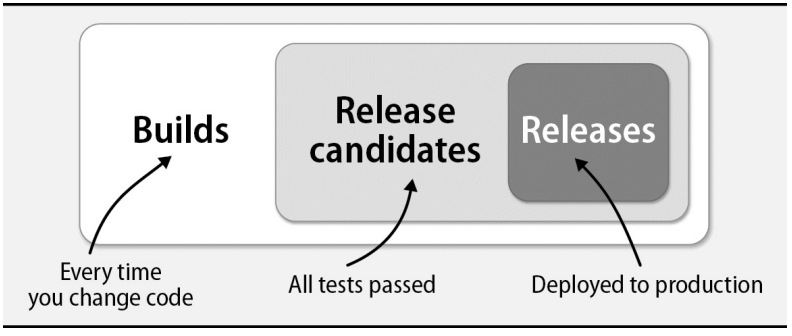
The following sections break down the three stages in further detail.

The build process

A build is a snapshot of the current status of a software project. It's typically the first stage of any CI/CD pipeline, possibly after a code analysis stage that monitors code quality and searches for security risks. The build step transforms the code into an installable piece of software. Builds can be triggered by a commit to the integration branch of the code repository or they can run on a regular schedule or on demand.

Every pipeline run starts with a build, but not every build reaches production. Once a build passes testing, it becomes a "release candidate." If the release candidate is actually deployed to production, it becomes a "release." If you do continuous deployment, every release candidate is also a release. Exhibit B illustrates these categories.

Exhibit B: Builds, release candidates, and releases



The precise steps of the build process depend on the language and software. For a program in C, C++, or Go, the build process is a compilation, often initiated by **make**, that results in an executable binary. For languages that do not require compilation, such as Python or Ruby, the build stage might involve packaging the project with all relevant dependencies and assets, including libraries, images, templates, and markup files. Some builds might involve only configuration changes.

The output of the build stage is a "build artifact." The nature of that artifact depends on the software and the configuration of the rest of the pipeline. Table 26.1 lists some of the common types of artifacts. Whatever the format, the artifact is the basis for deployments throughout the rest of the pipeline.

Table 26.1: Common types of build artifacts

Type	What it's for
.jar or **.war** file	Java archive or Java web application archive
Static binary	Statically compiled programs, commonly C or Go
.rpm or **.deb** file	OS-native software packages for Red Hat or Debian
pip or **gem** package	Packaged Python or Ruby applications
Container image	Applications that run under Docker
Machine image	Virtual servers, especially for public or private clouds
.exe file	Windows executable

Build artifacts are saved to an artifact repository. The type of repository depends on the type of artifact. At its simplest, a repository can be a directory on a remote server that's accessible through SFTP or NFS. It can also be a **yum** or APT repository, a Docker image repository, or, in the cloud, an object store such as an AWS S3 bucket. The repository must be available to all the systems that need to download and install the artifact during a deployment.

Testing

Each stage in a CI/CD pipeline runs tests to catch buggy code and bad builds so that the code that makes it through to production is free of defects (or at least, as free as possible). Testing is the linchpin of this process. It engenders trust that a release is ready to deploy.

If a build fails any test, the remaining stages of the pipeline are pointless. The team must determine why the build failed and address the underlying issue. Because builds are created for every code push, it's easy to isolate the problem to the latest commit. The fewer lines of code changed between builds, the easier the problem's isolation.

Failures do not always stem from software bugs. They can occur because of network conditions or infrastructure errors that require administrative attention. If the application depends on outside resources, such as third party APIs, there can be upstream failures in the external resource. Some tests can run in isolation, but other tests require the same infrastructure and data that will be present in production.

Consider adding each of the following types of tests to your CI/CD pipeline:

- *Static code analysis* examines code for syntax errors, duplication, violations of coding guidelines, security problems, or excessive code complexity. These checks are fast and do not involve executing the actual code.
- *Unit tests* are written by the same developers who write the application code. They reflect the developer's view of how the code is supposed to function. The intent is to test the input and output of every method and function (unit) in the code. "Code coverage" is a (sometimes misleading) metric that describes what portion of the code is being unit-tested.
- *Integration tests* take unit tests one step further by running the application in its intended execution environment. Integration tests run the application with its underlying frameworks and with external dependencies such as outside APIs, databases, queues, and caches.
- *Acceptance tests* simulate typical use. In contrast to unit tests, acceptance tests reflect a user's point of view. For web-based software, this stage might include remote-controlling browser page loads through tools such as Selenium. For mobile software, the build artifact might go to a device farm that runs the app on many different mobile devices. Different browsers and versions make acceptance tests challenging to create, but in the end, these tests have meaningful results.
- *Performance tests* search for performance problems introduced by the latest code. To identify bottlenecks, these so-called stress tests should invoke the application within a perfect clone of your production environment, with real traffic patterns. Tools such as JMeter or Gatling can simulate thousands of concurrent users interacting with an

application in a programmed pattern. To gain the most from performance testing, ensure that monitoring and graphing instrumentation are in place. Those tools clarify both the application's typical performance and its behavior under a new build.

- *Infrastructure tests* go hand-in-hand with programmatically provisioned cloud infrastructure. If you create temporary cloud infrastructure as part of your CI/CD pipeline, you can write test cases to verify the proper configuration and operation of the infrastructure itself. Does the system run through configuration management successfully? Are only the expected daemons running? Serverspec (serverspec.org) is one interesting tool in this space.

Depending on the characteristics of your project, some tests are more important than others. For example, software that implements a REST API has no need for browser-based acceptance tests. Instead, you'll likely focus on integration tests. On the other hand, for shopping cart software, browser tests for all the important user paths (catalog, product pages, cart, checkout) are mandatory. Consider the needs of your project and implement testing accordingly.

Sometimes the code that's difficult to test is also the most likely to have defects. Your code may have 85% code coverage through unit tests (which is quite high by industry standards), but if the most complex code isn't tested, bugs might be missed. Code coverage alone is not an adequate measure of code quality.

Testing workflows don't have to be linear. Actually, because one of the goals is to get feedback as quickly as possible, it's a good idea to run as much of the testing as possible in parallel. But keep in mind that some tests might depend on the results of other tests; others might potentially interfere with each other. (Ideally, tests should have no cross-dependencies.)

Avoid the temptation to ignore or overlook broken tests. It's easy to get in the habit of understanding the reason for a failure, considering it to be harmless or inapplicable, and suppressing the test. However, this thinking is dangerous and can lead to a less reliable testing system. Keep in mind the golden rule of CI/CD: fixing a broken pipeline is the top priority.

To reinforce this tenet, make it difficult to ignore failed tests. It should be a technical requirement, enforced through the CI/CD software, that production deployments cannot occur if there are any broken tests.

Deployment

Deployment is the act of installing software and preparing it for use within a server environment. The specifics of how this is done depend on the technology stack. A deployment system must understand how to retrieve the build artifact (e.g., from a package repository or container image registry), how to install it on the server, and what setup steps, if any, are necessary. A deployment concludes when a new version of software is running and the old version has been disabled.

A deployment might be as simple as updating some HTML files on disk. No restart or further configuration required! But for most cases, a deployment involves at least installing a package and restarting an application. Complex, large-scale production deployments might involve installing code on multiple systems while serving live traffic, without pausing for a service outage.

System administrators play an important role in the deployment process. They are usually responsible for creating deployment scripts, monitoring important application health indicators during deployments, and ensuring that the infrastructure and configuration needs of other team members are met.

The following list describes just a few of the possible ways to deploy software:

- Run a basic shell script that invokes **ssh** to log in to each system, downloads and installs the build artifact, and then restarts the application. These types of scripts are usually home grown and do not scale to more than a handful of systems.
- Use a configuration management tool to orchestrate the installation process across a managed set of systems. This strategy is more organized and scalable than the use of shell scripts. Most configuration management systems are not designed specifically to facilitate deployments, although they can be used for this purpose.
- If the build artifact is a container image and the application runs on a container management platform such as Kubernetes, Docker Swarm, or AWS ECS, the deployment might be nothing more than a quick API call to the container manager. The container service manages the rest of the deployment process on its own. See *Containers and CI/CD*.
- A few open source projects standardize and streamline deployment. Capistrano (capistranorb.com) is a Ruby-based deployment tool that extends Ruby's Rake system to run commands on remote systems. Fabric (fabfile.org) is a similar tool written in Python. These tools, by developers for developers, are essentially elaborate shell scripts.
- Software deployment is a well-explored problem for users of public clouds. Most cloud ecosystems include both integrated and third party deployment services that

can be used from a CI/CD pipeline. Some examples include Google Deployment Manager, AWS CodeDeploy, and Heroku.

Tailor the deployment technique to your site's technology stack and service needs. If you have a simple environment with a few servers and a small handful of applications, a configuration management tool might be appropriate. At sites with a large number of servers spread among data centers, a specialized deployment tool is called for.

An "immutable" deployment codifies the principle that servers should never be modified (or "mutated") once they've been initialized. To deploy a new release, the CI/CD tooling creates entirely new servers with the updated build artifact included in the image. In this model, servers are considered disposable and temporary. This strategy is predicated on a programmable infrastructure such as a public or private cloud, where instances can be allocated through an API call. Some of the largest users of the public cloud embrace immutable deployments.

In *CI/CD in practice*, we walk through a sample immutable deployment that uses HashiCorp's Terraform tool to create and update infrastructure.

Zero-downtime deployment techniques

At some sites, services must continue to run even while they are being upgraded or redeployed, either because an outage poses unacceptable risk (health care, government services) or because it might have substantial financial costs (high-volume e-commerce or financial services). Updating live software without service interruptions is the Xanadu of software deployments, and it's the source of much anxiety and yak-shaving.

One common way to achieve a zero-downtime release is a "blue/green" deployment. The concept is straightforward: stage the new software on a standby system (or set of systems), run tests to confirm its functionality, then switch traffic from the live system to the standby once the tests are complete.

See this page for more information about load balancers.

This strategy works particularly well when traffic is proxied through a load balancer. The live systems handle all the user connections while the standby systems are being prepared. When the time is right, the standby systems can be added to the load balancer and the previously live systems removed. The deployment is complete when all the old systems are out of the rotation and all the transactions they were handling have concluded.

A "rolling" deployment updates existing systems in a stepwise fashion, modifying software on one system at a time. Each system is removed from the load balancer, updated, then added back to the rotation to accept user traffic. This type of deployment can cause problems if the application cannot tolerate two different versions running simultaneously.

Both the blue/green and rolling deployment strategies can accommodate a "canary," akin to the hapless canary in a coal mine. You first allocate a small amount of traffic to a single system (or small percentage of systems) that runs the new release. If the new release has problems, you back it out and correct the problem, having impacted only a handful of users. Of course, the canary systems need precise telemetry and monitoring so that you can determine whether problems have been introduced.

26.3 Jenkins: the open source automation server

Jenkins is an automation server written in Java. It's by far the most popular software used to implement CI/CD. With broad adoption and an extensive ecosystem of plug-ins, Jenkins is well suited to a variety of use cases.

It's easy to dabble with Jenkins by running it in a Docker container:

```
$ docker run -p 8080:8080 --name jenkins jenkinsci/jenkins
```

Once the container starts, you can access the Jenkins user interface in a web browser on port 8080. You'll find the initial administrator password buried in the container output. In practice, you would need to change that password immediately!

A single-container configuration is fine for learning the ropes, but you'll likely need a more robust solution in production environments. The Jenkins download page (jenkins.io/download) has installation instructions that we needn't rehash here. Refer to those docs for installation on Linux and FreeBSD. CloudBees, the maker of Jenkins, also offers a high-availability version called Jenkins Enterprise.

Jenkins has plug-ins for every conceivable task. Use plug-ins to outsource builds to different types of agents, send notifications, coordinate releases, and execute scheduled jobs. Plug-ins integrate with open source tools and with all the major cloud platforms and external SaaS providers. Plug-ins give Jenkins superpowers.

Most Jenkins configuration is done through the web UI, and being merciful to your attention span, we don't attempt to cover the UI's nooks and crannies. Instead, we introduce the fundamentals of Jenkins along with some of its most important features.

Basic Jenkins concepts

At its core, Jenkins is a coordination server that links a series of tools into a chain—or, to use CI/CD terminology, a pipeline. Jenkins is an organizer and facilitator; all actual work depends on outside services such as source code repositories, compilers, build tools, testing harnesses, and deployment systems.

A Jenkins job, or project, is a collection of linked stages. Creating a project is the first order of business for a new installation. You can link the project's steps together so that they run in sequence or in parallel. You can even set up conditional steps that do different things depending on the results of previous steps.

Every project should be connected to a source code repository. Jenkins has native support for pretty much every version control system: Git, Subversion, Mercurial, even ancient systems such as CVS. There are also integration plug-ins for higher-level version control services such as GitHub, GitLab, and BitBucket. You'll need to give Jenkins the appropriate credentials to allow it to download code from your repository.

The "build context" is the current working directory on the Jenkins system that's executing a build. Source code is copied into the build context along with any supporting files that are needed for the build.

Once you've wired up Jenkins to a version control repository, you can create a build trigger. This is the signal for Jenkins to copy the current source code and start the build process. Jenkins can poll the source repository for new commits and initiate a build whenever it finds one. It can also start builds on a schedule or be triggered by a web hook, a feature supported by GitHub.

After setting up the trigger, create the build steps, that is, the specific tasks that will create a build. Steps can be code-base-specific, or they can be generic shell scripts. For example, Java projects are usually built with a tool called Maven. A Jenkins plug-in supports Maven directly, so you can simply add a Maven build step. For a project written in C, the first build step might just be a shell script that runs **make**.

The remaining build steps depend on your goals for the project. The most common builds include steps that initiate the testing tasks discussed in *Testing*. You may need a step to create a custom build artifact such as a tarball, OS package, or container image. You can also include steps that trigger administrator notifications, take deployment-related actions, or coordinate with outside tooling.

For a CI/CD project, the build steps can address all the stages of a pipeline: build the code, run tests, upload the artifact to a repository, and kick off a deployment. Each stage of the pipeline is a just a build step within the Jenkins project. The Jenkins interface presents an overview of the status of each step, so it's easy to see at a glance what's happening in the pipeline.

Sites that have many applications should have separate Jenkins projects for each. Each project will have a distinct code repository and build steps. The Jenkins system needs all the tools and dependencies to run a build for any of its projects. For example, if you have configured both a Java project and a C project, your Jenkins system must have both Maven and **make** installed.

Projects can depend on other projects. Use this to your advantage by structuring projects as generic, inheritable templates. For example, if you have a variety of applications that are built differently but deployed in the same way (e.g., as containers running on a server cluster), you can create a generic “deploy” project that manages the common deployment stage. Individual application projects can execute the deploy project, thereby eliminating a now-redundant build step.

Distributed builds

At sites that support dozens of applications, each with its own dependencies and build steps, it's easy to inadvertently create dependency conflicts and bottlenecks because too many pipelines are running at once. To compensate, Jenkins lets you graduate to a distributed build architecture. This mode of operation uses a "build master," a central system that keeps track of all the projects and their current state, and "build agents," which run the actual build steps for a project. If you use Jenkins a lot, you'll move to this configuration pretty quickly.

Build agents run on hosts that are separate from the build master. The Jenkins master logs in to the slaves (usually through SSH) to start the agent process and to add labels that document the slaves' capabilities. For example, you might distinguish your Java-capable agents from your C-capable agents by applying appropriate labels.

For best results, run agents in containers, remote VMs, or ephemeral cloud instances that scale out and back on demand. If you have a container cluster, you can use Jenkins plug-ins to run agents in the cluster through a container management system.

Pipeline as code

Thus far, we've described the process of setting up Jenkins projects by stringing together individual build steps in the web UI. This is the quickest way to get started with Jenkins, but from an infrastructure perspective it's also a bit opaque. The "code"—in this context, the contents of each build step—is managed by Jenkins. You can't check graphical build steps into a code repository, and if you lose the Jenkins server, there's no easy way to replace it; you'll need to restore your projects from a recent backup.

Jenkins version 2 introduced a major new feature, called the Pipeline, that affords first-class support for CI/CD pipelines. A Jenkins pipeline codifies the steps of a project in a declarative, domain-specific language that's based on the Groovy programming language. You can commit the Jenkins pipeline code, called a **Jenkinsfile**, alongside the code that's associated with the pipeline.

The following **Jenkinsfile** demonstrates a basic build/test/deploy cycle:

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                sh 'make'
            }
        }
        stage('Test') {
            steps {
                sh 'make test'
            }
        }
        stage('Deploy') {
            steps {
                sh 'deploy.sh'
            }
        }
    }
}
```

The `agent any` notation instructs Jenkins to prepare a workspace for this pipeline on any available build agent. A workspace is the same as a build context: a location on the agent's local disk that contains all the files needed by the build, including the source code and dependencies. Every build has a private workspace.

The Build, Test, and Deploy stages parallel the conceptual stages of a CI/CD pipeline. In our example, each stage has a single step that invokes a shell (**sh)** to execute a command.

The Deploy stage runs a custom script, **deploy.sh**, that handles the entire deployment, including copying the build artifact (generated by the Build stage) to a set of servers and restarting server processes. In practice, deployment would usually be divided into multiple stages to afford better visibility and control over the full process.

26.4 CI/CD IN PRACTICE

We now turn to a contrived example to illustrate the concepts presented so far. We've concocted a simple application, UlsahGo, that's a lot more basic than anything you might need to manage in the real world. It's entirely self-contained and has no dependencies on other applications.

Our example includes the following elements:

- The UlsahGo web application, with just one small feature
- Unit tests for the application
- A virtual machine image for DigitalOcean, which contains the application
- A single-server development environment, created on demand
- A load-balanced, multiserver staging environment, created on demand
- A CI/CD pipeline that ties all these parts together

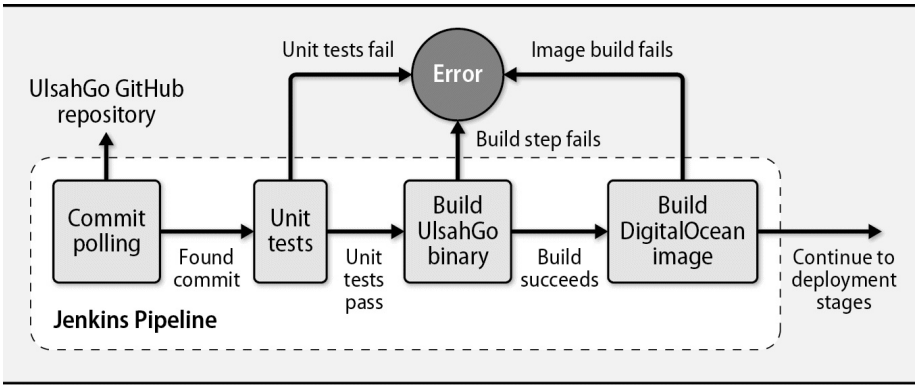
We use several popular tools and services in this example:

- GitHub as the code repository
- DigitalOcean virtual machines and load balancers
- HashiCorp's Packer, for provisioning the DigitalOcean image
- HashiCorp's Terraform, to create deployment environments
- Jenkins, to manage the CI/CD pipeline

Your applications might use a different technology stack, but the general concepts are similar, regardless of the tooling.

Exhibit C depicts the first several stages of the example pipeline. The diagram shows the pipeline polling GitHub for new commits to the UlsahGo project. When a commit is found, Jenkins runs the unit test suite. If the tests pass, Jenkins builds the binary. If the binary builds successfully, the pipeline continues to create the deployment artifact, a DigitalOcean machine image that includes the binary. If any of the stages fail, the pipeline reports an error.

Exhibit C: Demonstration pipeline (part one)



We describe the deployment stages in detail later. But first we should review these initial stages.

UlsahGo, a trivial web application

Our example application is a web service with a single feature. It returns, as JSON, the authors associated with a specified edition of this book. For example, the following query shows the authors for this edition:

```
$ curl ulsahgo.admin.com/?edition=5
{
    "authors": [
        "Evi",
        "Garth",
        "Trent",
        "Ben",
        "Dan"
    ],
    "number": 5
}
```

We do some sanity checking to make sure users aren't getting too carried away, for example, by requesting implausible editions:

```
$ curl -vs ulsahgo.admin.com/?edition=6
< HTTP/1.1 404 Not Found
< Content-Type: application/json
{
    "error": "6th edition is invalid."
}
```

Our application also has a health-check endpoint. Health checks are an easy way for monitoring systems to ask the application, "Hey, are you working OK?"

```
$ curl ulsahgo.admin.com/healthy
{
    "healthy": "true"
}
```

Developers typically work closely with administrators to create the build and test stages of a CI/CD pipeline. In this case, since the application is written in Go, we can use the standard Go tools (**go build** and **go test**) in our pipeline.

Unit testing UlsahGo

Unit tests are the first test suite to run because they operate at the source code level. Unit tests involve testing the application's functionality at the finest possible granularity: its functions and methods. Most languages have testing frameworks that offer native support for unit tests.

Let's examine one unit test for UlsahGo. Consider the following function:

```
func ordinal(n int) string {
    suffix := "th"
    switch n {
    case 1:
        suffix = "st"
    case 2:
        suffix = "nd"
    case 3:
        suffix = "rd"
    }
    return strconv.Itoa(n) + suffix
}
```

The function takes an integer as input and determines the corresponding ordinal expression. For example, when passed a 1, the function returns "1st." UlsahGo uses this function to format the text in the error message for invalid editions.

Unit tests try to prove that given some input, the function returns the expected output. Here's a unit test that exercises this function:

```
func TestOrdinal(t *testing.T) {
   ord := ordinal(1)
   exp := "1st"
   if ord != exp {
      t.Error("expected %s, got %s", exp, ord)
   }

   ord = ordinal(10)
   exp = "10th"
   if ord != exp {
      t.Error("expected %s, got %s", exp, ord)
   }
}
```

This simplified unit test runs the function on two values, 1 and 10, and confirms that the actual response matches the expectation. (The `ordinal()` function implements three special cases and a general case. A full set of unit tests would exercise each of these possible paths through the code.)

We can run the tests through Go's built-in testing framework:

```
$ go test
PASS
ok  github.com/bwhaley/ulsahgo  0.006s
```

If some part of the application changes in the future—for example, if updates are made to the `ordinal()` function—the tests report any divergence from the expected output. Developers are responsible for updating unit tests as they adjust the code. Experienced developers by design write code that is easy to test. They aim to have complete coverage of each method and function.

Taking first steps with the Jenkins Pipeline

With the code ready to ship and the unit tests in place, the first step in our CI/CD journey is to configure the project in Jenkins. The GUI interface walks us through the process. Here are our selections:

- Our new project is a Pipeline project, defined by code as opposed to a traditional "freestyle" project with build steps mostly defined through user interface elements.
- We want to track our pipeline alongside our source code repository in a **Jenkinsfile**, so we choose "Pipeline script from SCM" for the Pipeline definition.
- We trigger the build by polling GitHub for commits. We add credentials so that Jenkins can access the UlsahGo repository and configure Jenkins to poll GitHub for changes every five minutes.

The initial setup takes just a few moments. In real life, we would use a GitHub web hook to notify Jenkins that a new commit was available, thus avoiding polling and sparing GitHub from our unnecessary calls to their API.

With this setup, Jenkins executes the pipeline described by the **Jenkinsfile** in the repository whenever a new commit is pushed to GitHub.

Consider how your repositories are organized. In this project we chose to combine CI/CD and application code in the same repository, with all CI/CD-related files being kept in the **pipeline** subdirectory. The UlsahGo repository is laid out as follows:

```
$ tree ulsahgo
ulsahgo
├── pipeline
│   ├── Jenkinsfile
│   ├── packer
│   │   ├── provisioner.sh
│   │   ├── ulsahgo.json
│   │   └── ulsahgo.service
│   ├── production
│   │   ├── tf_prod.sh
│   │   └── ulsahgo.tf
│   └── testing
│       ├── tf_testing.sh
│       └── ulsahgo.tf
├── ulsahgo.go
└── ulsahgo_test.go
```

An integrated structure works well for a small project like this one. Jenkins, Packer, Terraform, and other tools can look in the **pipeline** subdirectory for their configuration files. Modifying the deployment pipeline is a simple matter of updating the repository. For more complex

environments where multiple projects share a common infrastructure, it makes sense to have a dedicated infrastructure repository.

With the project in place, we can commit our first **Jenkinsfile**. The first step in any pipeline is to check out the source code. Here's a complete **Jenkinsfile** pipeline script that does just that:

```
pipeline {
  agent any
  stages {
    stage('Checkout') {
      steps {
        checkout scm
      }
    }
  }
}
```

The `checkout scm` line instructs Jenkins to check out the code from "software configuration management," a generic industry term for source control.

With Jenkins polling GitHub and the checkout stage complete, we can move on to setting up the test and build stages. Our Go project has no external dependencies. The only requirement for building and testing our code is the **go** binary. We have already installed **go** on our Jenkins system (with **apt-get -y install golang-go**) so we need only add the test and build stages to the **Jenkinsfile**:

```
stage('Unit tests') {
  steps {
    sh 'go test'
  }
}
stage('Build') {
  steps {
    sh 'go build'
  }
}
```

After we commit the changes, Jenkins discovers the new commit and executes the pipeline. Jenkins emits friendly log output indicating that it has done so:

```
Mar 30, 2017 4:35:00 PM hudson.triggers.SCMTrigger$Runner run
INFO: SCM changes detected in UlsahGo. Triggering  #4
Mar 30, 2017 4:35:11 PM org.jenkinsci.plugins.workflow.job.WorkflowRun
    finish
INFO: UlsahGo #4 completed: SUCCESS
```

The Jenkins GUI uses a weather metaphor to indicate the health of recent builds. A sun icon represents a project that is building successfully, and a stormy cloud icon represents failures. You

can debug build failures by inspecting the console output, which is found under the build details. It shows the STDOUT printed by any part of the build.

Here's a snippet from the **go test** and **go build** steps of our pipeline:

```
[Pipeline] stage
[Pipeline] { (Unit Tests)
[Pipeline] sh
[UlsahGo] Running shell script
+ go test
PASS
ok  _/var/jenkins_home/workspace/UlsahGo 0.006s
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Build)
[Pipeline] sh
[UlsahGo] Running shell script
+ go build
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

You can normally pinpoint the cause of a failed build by reviewing the log. Look for error messages that identify the failed step. You can also add your own log messages to supply clues about the state of the system, such as the values of variables or the contents of a script at a given point in the execution. Writing output for the purpose of debugging is a time-honored programming tradition.

The output of our build is a single binary file, **ulsahgo**, which contains our entire application. (This, incidentally, is one of the primary benefits of Go programs and one of the reasons Go is popular with sysadmins: it's easy to create static binaries that run on multiple architectures and have no external dependencies. Installing a Go application is often as simple as copying it to the system.)

Building a DigitalOcean image

With **ulsahgo** ready to ship, we next build a virtual machine image for the DigitalOcean cloud. We start with a vanilla Ubuntu 16.04 image, install the latest updates, and then install **ulsahgo**. The resulting image becomes the deployment artifact for the remaining stages of the pipeline.

If you're unfamiliar with the tool **packer**, which creates virtual machine images, refer to the section *Packer* before continuing.

packer reads its image configuration from a template that has two primary sections: builders, which interact with remote APIs to create machines and images, and optional provisioners, which run custom configuration steps.

The template for our UlsahGo image has only one builder:

```
"builders": [{
    "type": "digitalocean",
    "api_token": "rj8FsrMI17vqTlB8qqBn9f7xQedJkkZJ7cqJcB1O5nmO6ihz",
    "region": "sfo2",
    "size": "512mb",
    "image": "ubuntu-16-04-x64",
    "snapshot_name": "ulsahgo-latest",
    "ssh_username": "root"
}]
```

The builder tells **packer** which platform to build the image on and how to authenticate to the API, among other provider-specific details.

Three provisioning steps follow:

```
"provisioners": [
    {
        "type": "file",
        "source": "ulsahgo",
        "destination": "/tmp/ulsahgo"
    },{
        "type": "file",
        "source": "pipeline/packer/ulsahgo.service",
        "destination": "/etc/systemd/system/ulsahgo.service"
    },{
        "type": "shell",
        "script": "pipeline/packer/provisioner.sh"
    }
]
```

---

*See this page for more information about **systemd** unit files.*

The first two provisioning steps add files to the image. The first file is the application itself, **ulsahgo**, which is uploaded to **/tmp** for later use. The second is a **systemd** drop-in unit file that manages the service.

The last provisioner executes a custom shell script on the remote system. The script, **provisioner.sh**, updates the system and then sets up the application:

```
#!/usr/bin/env bash
app=ulsahgo

# Update the OS and add a user
apt-get update && apt-get -y upgrade
/usr/sbin/useradd -s /usr/sbin/nologin $app

# Set up the working directory and app
mkdir /opt/$app && chown $app /opt/$app
cp /tmp/$app /opt/$app/$app
chown $app /opt/$app/$app && chmod 700 /opt/$app/$app

# Enable the systemd unit
systemctl enable $app
```

In addition to shell scripts, **packer** lets you use all the popular configuration management tools as provisioning steps. Call out to Puppet, Chef, Ansible, or Salt to provision your images in a more structured and scalable manner.

Finally, we can add an image-building stage to our **Jenkinsfile**:

```
stage('Build image') {
  steps {
    sh 'packer build pipeline/packer/ulsahgo.json > packer.txt'
    sh 'grep ID: packer.txt  | grep -E -o \'[0-9]{8}\' > do_image.txt'
  }
}
```

The first step invokes **packer** and saves the output to **packer.txt** in the build's working directory. The tail end of that output includes an ID for the new image:

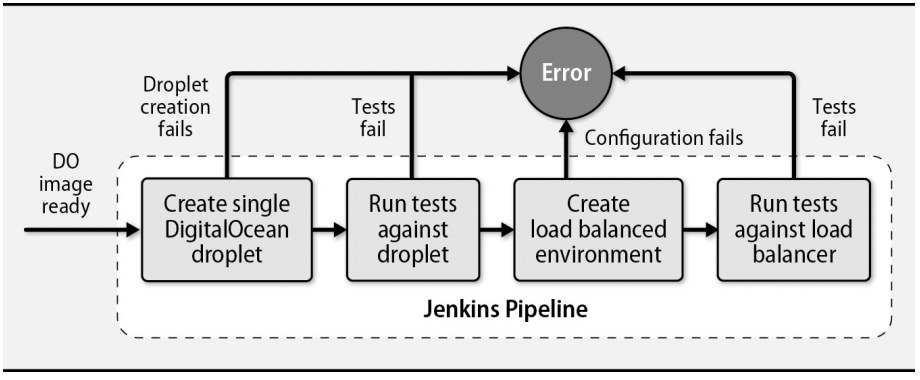
```
==> digitalocean: Gracefully shutting down droplet...
==> digitalocean: Creating snapshot: ulsahgo-latest
==> digitalocean: Waiting for snapshot to complete...
==> digitalocean: Destroying droplet...
==> digitalocean: Deleting temporary ssh key...
Build 'digitalocean' finished.
==> Builds finished. The artifacts of successful builds are:
--> digitalocean: A snapshot was created:  (ID: 23838540)
```

The second step **grep**s the ID from **packer.txt** and saves it to a new file in the build context. Because the image is the deployment artifact, we will need to refer to its ID from later stages of the pipeline.

Provisioning a single system for testing

At this point we have a process for continuously running unit tests, building the application, and creating a virtual machine image as a build artifact. The remaining build stages focus on deploying the artifact and testing it in the wild. Exhibit D picks up where Exhibit C leaves off.

Exhibit D: Demonstration pipeline (part two)



We've chosen to use **terraform**, another gem from HashiCorp, to create and manage the UlsahGo infrastructure. **terraform** reads its configuration from "plans," JSON-like configuration files that describe a desired infrastructure configuration. It then creates the cloud resources described in the plan by making an appropriate series of API calls. **terraform** supports dozens of cloud providers and a wide variety of services.

The following **terraform** configuration, **ulsahgo.tf**, requisitions a single DigitalOcean droplet running the image we created in the previous stage of the pipeline:

```
variable "do_token" {}
variable "ssh_fingerprint" {}
variable "do_image" {}

provider "digitalocean" {
  token = "${var.do_token}"
}

resource "digitalocean_droplet" "ulsahgo-latest" {
    image = "${var.do_image}"
    name = "ulsahgo-latest"
    region = "sfo2"
    size = "512mb"
    ssh_keys = ["${var.ssh_fingerprint}"]
}
```

Most of this is self-explanatory: use DigitalOcean as the provider, and authenticate with the provided token. Create the droplet in the sfo2 region from the specified image ID.

In the Packer template on this page, we directly embedded parameters such as the API token in the builder configuration. One (big!) problem with that approach is that the API key is saved in the source code repository even though it's supposedly secret. The key gives access to the cloud provider's API and hence would be dangerous in the wrong hands. Keeping secrets in revision control is a security antipattern for reasons we describe in more detail here.

In this example, we instead read the parameters as variables. The three variables are

- The DigitalOcean API token
- The fingerprint of an SSH key that will be permitted to access the droplet
- The ID of the image to use for the new system, which we captured during the previous stage of the pipeline.

Jenkins can store secrets such as the API token in its "credential store," an encrypted area that's intended for exactly this kind of sensitive data. The pipeline can read values from the credential store and save them as environment variables. The values then become accessible throughout the pipeline without being saved in the version control system.

Here's how we set this up in the **Jenkinsfile:**

```
pipeline {
  environment {
    DO_TOKEN = credentials('do-token')
    SSH_FINGERPRINT = credentials('ssh-fingerprint')
  }
...
```

Recall that we saved the ID of the DigitalOcean machine image to a file within the build area, **do_image.txt**. We need that ID in our new pipeline stage, which creates the actual DigitalOcean droplet. The code for the new stage just runs a script from the project repository:

```
stage('Create droplet') {
  steps {
    sh 'bash pipeline/testing/tf_testing.sh'
  }
}
```

It's easier and more maintainable to separate the code of complex scripts from the rest of the pipeline, as we've done here. **tf_testing.sh** contains the following lines:

```
cp do_image.txt pipeline/testing
cd pipeline/testing
terraform apply \
  -var do_image="$(<do_image.txt)" \
  -var do_token="${DO_TOKEN}" \
  -var ssh_fingerprint="${SSH_FINGERPRINT}"
terraform show terraform.tfstate \
  | grep ipv4_address | awk "{print $3}" > ../../do_ip.txt
```

This script copies the saved image ID to a temporary directory, **pipeline/testing**, then runs **terraform** from that directory. **terraform** looks for files in the current directory that have **.tf** extensions, so we don't have to explicitly name the plan file. (It's the same **ulsahgo.tf** file that we looked at on this page.)

A few explanations:

- The DO_TOKEN and SSH_FINGERPRINT environment variables are available to any shell commands in the pipeline. The environment clause shown above can appear either at the level of the overall pipeline or within a particular stage, depending on the scope you want.
- $(<do_image.txt) reads the contents of the saved DigitalOcean image ID from the text file saved in the previous stage.
- The final line of the **tf_testing.sh** script inspects the **terraform**-created droplet, obtains its IP address, and saves the address to a text file for use in the next stage. The **terraform.tfstate** file is **terraform**'s snapshot of the system state. It's how **terraform** keeps track of resources.

Like **packer**, **terraform** sends useful output to the Jenkins console output page. Here are the relevant bits from the **terraform apply** command:

```
[Pipeline] { (Create droplet)
[Pipeline] sh
[UlsahGo] Running shell script
digitalocean_droplet.ulsahgo-latest: Creating...
  disk:                 "" => "<computed>"
  image:                "" => "23888047"
  ipv4_address:         "" => "<computed>"
  ipv4_address_private: "" => "<computed>"
  name:                 "" => "ulsahgo-latest"
  region:               "" => "sfo2"
  resize_disk:          "" => "true"
  size:                 "" => "512mb"
  ssh_keys.#:           "" => "1"
  ssh_keys.0:           "" => "****"
  status:               "" => "<computed>"
digitalocean_droplet.ulsahgo-latest: Still creating... (10s elapsed)
digitalocean_droplet.ulsahgo-latest: Still creating... (20s elapsed)
digitalocean_droplet.ulsahgo-latest: Still creating... (30s elapsed)
digitalocean_droplet.ulsahgo-latest: Creation complete (ID: 44486631)
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

When this stage completes, the droplet is up and running with UlsahGo.

Testing the droplet

We have some confidence that the code is functional because it passed the unit testing step. However, we also need to make sure that it runs successfully as part of a DigitalOcean droplet. Testing at this level is considered a form of integration test. We want the integration tests to run each time a new image is created, so we add a new stage to the **Jenkinsfile**:

```
stage('Test and destroy the droplet') {
  steps {
    sh '''#!/bin/bash -l
    curl -D - -v \$(<do_ip.txt):8000/?edition=5 | grep "HTTP/1.1 200"
    curl -D - -v \$(<do_ip.txt):8000/?edition=6 | grep "HTTP/1.1 404"
    terraform destroy -force
    '''
  }
}
```

Sometimes a blunt, heavy object is the right tool for the job. This pair of **curl** commands queries **ulsahgo** on the remote droplet's port 8000, where **ulsahgo** runs by default. We check that a query for the fifth edition returns an HTTP code 200 (success) and that a query for the sixth edition returns an HTTP 404 (failure). We know to expect these specific status codes only because of our familiarity with the application.

At the conclusion of the tests, we destroy the droplet because it's no longer needed. The droplet is created, tested, and destroyed each time the pipeline runs.

Deploying UlsahGo to a pair of droplets and a load balancer

The final pipeline task is to deploy to our (mock) production environment, which consists of two DigitalOcean droplets and a load balancer. Once again, Terraform is up to the task.

We can reuse some of the configuration from the single-droplet **terraform** plan file. We still need the same variables and the droplet resource. This time, we add a second droplet resource:

```
resource "digitalocean_droplet" "ulsahgo-b" {
    name      = "ulsahgo-b"
    size      = "512mb"
    image     = "${var.do_image}"
    ssh_keys  = ["${var.ssh_fingerprint}"]
    region    = "sfo2"
}
```

We also add a load balancer resource:

```
resource "digitalocean_loadbalancer" "public"
{
    name = "ulsahgo-lb"
    region = "sfo2"

    forwarding_rule {
        entry_port = 80
        entry_protocol = "http"
        target_port = 8000
        target_protocol = "http"
    }

    healthcheck {
        port = 8000
        protocol = "http"
        path = "/healthy"
    }

    droplet_ids = [
        "${digitalocean_droplet.ulsahgo-a.id}",
        "${digitalocean_droplet.ulsahgo-b.id}"
    ]
}
```

The load balancer listens on port 80 and forwards requests to each of the droplets on port 8000, where **ulsahgo** is listening. We tell the load balancer to use the **/healthy** endpoint to confirm that each copy of the service is running. The load balancer adds a droplet to the rotation if it receives a 200 status code when it queries this endpoint.

Now we can add the production configuration as a new stage in the pipeline:

```
stage('Create LB') {
  steps {
    sh 'bash pipeline/production/tf_prod.sh'
  }
}
```

The load balancer stage is more or less identical to the single instance stage. Even the external script is pretty much the same, so we omit its contents here. We could easily refactor these scripts so that a single version could handle both environments, but for now, we've kept the scripts separate.

We can add a testing stage as well, this time running against the load balancer's IP address:

```
stage('Test load balancer') {
  steps {
    sh '''#!/bin/bash -l
      curl -D - -v -s \$(<do_lb_ip.txt)/?edition=5 | grep "HTTP/1.1 200"
      curl -D - -v -s \$(<do_lb_ip.txt)/?edition=6 | grep "HTTP/1.1 404"
    '''
  }
}
```

The **curl** commands are similar to the previous set, but they target port 80, where the load balancer listens.

Concluding the demonstration pipeline

This demonstration CI/CD implementation captures several of the key elements of a real-world pipeline:

- The first two stages (unit testing and building) demonstrate continuous integration. Each time a developer commits code, Jenkins runs unit tests and tries to build the project.
- The third stage (creating a DigitalOcean image as a build artifact) is the beginning of continuous delivery. We can use the same image when deploying to each environment.
- Deployment to a single droplet is considered a “development” or “testing” environment.
- The final stage deploys **ulsahgo** to a high-availability, production-like environment, thereby closing the loop on a continuous deployment pipeline.
- If any stage in the pipeline fails, the subsequent stages are skipped. In that case, console output is available to help debug the problem.

This pipeline relies on open source tools throughout. All the deployment code is captured in just a few text files that are kept in the same repository as the application’s source code.

Astute readers will think of a host of improvements that could be made to these steps. To name just a few:

- A blue/green deployment to ensure no downtime in the production stage
- Status notifications to email or chat rooms for each stage
- Hooks to help monitoring systems note that a new deployment has occurred
- A better method of propagating data, such as the image ID, between stages

Continuous improvement is integral to CI/CD (and to system administration in general). Over time, a chain of incremental improvements results in a highly efficient and automated software delivery system.

26.5 Containers and CI/CD

Most software relies on outside dependencies such as third party libraries, a particular filesystem layout, the availability of certain environment variables, and other localizations. Conflicts among required dependencies often make it hard to run multiple applications on a single virtual machine.

To further complicate matters, building an application requires resources different from those running it. For example, the build process might require a compiler and a test suite, but these extras are not needed at run time.

See Chapter 25 for more information about containers.

Containers offer an elegant solution to these problems. From an operations viewpoint, the environment needs only the capability to run containers. You can activate any given container on any container-compatible system without further configuration effort because all dependencies and localizations for an app are housed within its container. Multiple containers can run on the same system simultaneously without conflict.

You can use containers to simplify your CI/CD environment in several ways:

- By running the CI/CD system itself within a container
- By building applications inside containers
- By using container images as build artifacts for deployment

The first point is rather obvious: you can run your CI/CD software (including both the master and any agents) in containers, thereby avoiding the overhead of having to dedicate systems to the CI/CD infrastructure.

The other two scenarios require a bit more explication. We look at them in more detail in the next sections.

Containers as a build environment

The exact environment needed to build an application is project-specific and sometimes quite complex. Rather than installing all the necessary tools, build software, and dependencies directly on your CI/CD agent systems, you can build your software within containers and leave the CI/CD agents in a clean and generic state. The build process then becomes portable and independent of the specific CI/CD agent.

Consider a typical application that depends on a PostgreSQL database and a Redis key/value store. To build and test the application in a traditional setting, you'd need separate servers for each component: the application itself, the Redis daemon, and PostgreSQL. In a pinch, you might run all these components on one system, but you probably wouldn't use that same server to build and test another service that had different dependencies.

Instead, you can use short-lived containers for each component. One container can build and run the application. It can connect to separate containers (on the same host or a different host) for PostgreSQL and Redis. Once the build process is complete, the containers can be stopped and discarded. You can use the same CI/CD agent to build, with no risk of conflicts, applications that have other dependencies.

The container image used to build software should be distinct from the container image that runs it. The build image is normally larger than the run-time image because it includes extra components such as compilers and testing tools.

Most current CI/CD tools include native support for containers. Jenkins has a Docker plug-in that integrates nicely with the pipeline. Also check out Drone (try.drone.io), a CI/CD platform designed around containers.

Container images as build artifacts

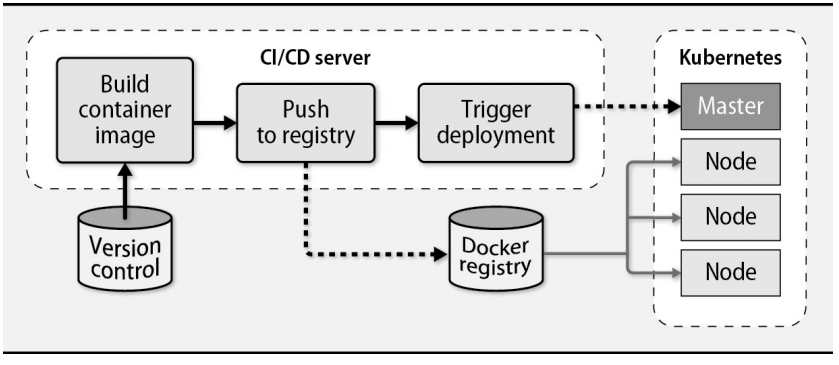
The product of a build can be a container image deployable through a container orchestration system. Containers are lightweight and highly portable. Moving container images among systems by way of an image registry is easy and fast. Any CI/CD tool can adopt the strategy of producing containers.

The basic workflow becomes:

1. Build your application inside a build-specific container.
2. Create a container image that includes the application and its dependencies.
3. Push the image to a registry.
4. Deploy that image to a container-ready execution environment.

It's generally best to use a container management platform such as Docker Swarm, Mesos/Marathon, Kubernetes, or AWS EC2 Container Service to deploy images into production. Your pipeline's deployment stage can call the appropriate APIs and let the platform handle the specifics. Exhibit E illustrates the procedure.

Exhibit E: Container-based deployment process



We've found containers to be an excellent match for mature CI/CD pipelines. Their extremely fast cycle time makes it easy both to deploy new code and to revert to a previous version in the event of a problem. Both virtual machines and configuration management systems are an order of magnitude slower.

26.6 Recommended reading

Beck, Kent, et al. *Manifesto for Agile Software Development*. agilemanifesto.org

Duvall, Paul M., Steve Matyas, and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Upper Saddle River, NJ: Addison-Wesley, 2007.

Farcic, Viktor. *The DevOps 2.0 Toolkit: Automating the Continuous Deployment Pipeline with Containerized Microservices*. Seattle, WA: Amazon Digital Services LLC, 2016.

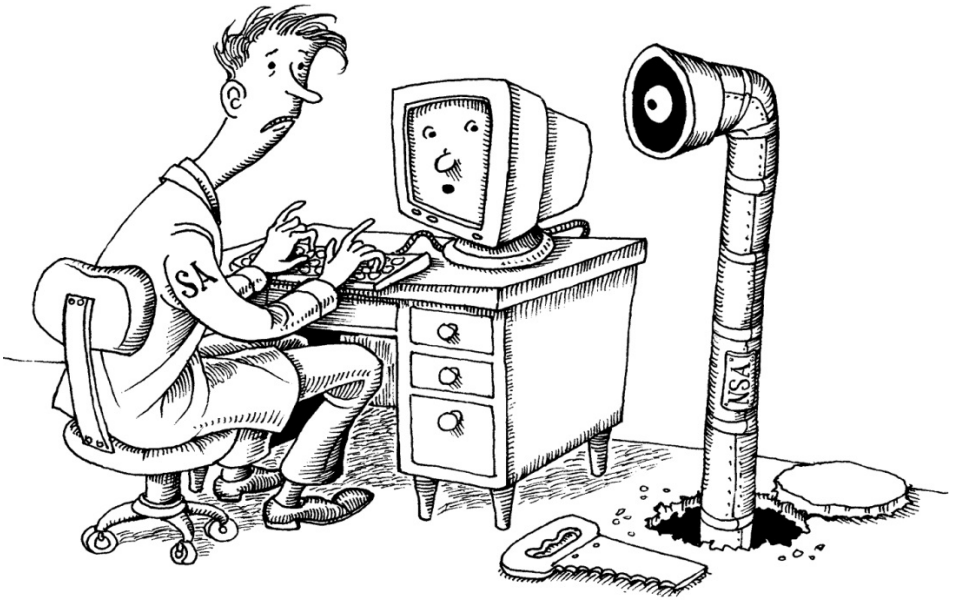
Fowler, Martin. *Continuous Integration*. goo.gl/Y2lisI (martinfowler.com)

Humble, Jez, and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Upper Saddle River, NJ: Addison-Wesley, 2010.

Morris, Kief. *Infrastructure as Code: Managing Servers in the Cloud*. Sebastopol, CA: O'Reilly Media, 2016.

jenkinsci-docs@googlegroups.com. *Jenkins User Handbook*. jenkins.io/doc/book

27 Security



Computer security is in a sorry state. In contrast to the progress seen in virtually every other area of computing, security flaws have become increasingly dire and the consequences of inadequate security more severe. Computer security issues directly influence and threaten societies around the world.

If you're tempted to skip over this chapter, permit us to pique your curiosity by reminding you of a few computer security events that have occurred since the last edition of this book:

- The sophisticated Stuxnet worm, discovered in 2010, attacked Iran's nuclear program by damaging centrifuges at a uranium enrichment plant.
- In 2013, Edward Snowden exposed the massive NSA surveillance machine, revealing that some major Internet companies were complicit in allowing the government to spy on U.S. citizens.
- Around 2013, a new type of attack known as ransomware came to prominence. Attackers compromise a target system and encrypt its data, holding it hostage. Victims must pay a ransom for recovery. They often do.
- In 2015, the U.S. Office of Personnel Management was breached, compromising the sensitive and private details of more than 21 million U.S. citizens, many of whom had security clearances.

- In 2016, Russian state-sponsored hackers allegedly mounted a campaign to influence the outcome of the U.S. presidential election.
- In 2017, a ransomware attack of unprecedented scale took over Windows systems in more than 150 countries. The attack used an exploit developed by the NSA.

The stakes have never been higher. We think it will get worse before it gets better.

Part of the challenge is that security problems are not purely technical. You cannot solve them by buying a particular product or service from a third party. Achieving an acceptable level of security requires patience, vigilance, knowledge, and persistence—not just from you and other sysadmins, but from your entire user and management communities.

As a system administrator you bear a heavy burden. You must push an agenda that secures your organization's systems and networks, ensures that they are vigilantly monitored, and properly educates your users and your staff. Familiarize yourself with current security technology and work with experts to identify and resolve vulnerabilities at your site. Security considerations should be part of every decision.

Strike a balance between security and usability. Remember that

$$\text{Security} = \frac{1}{(1.072)(\text{Convenience})}$$

The more security measures you introduce, the more constrained you and your users will be. Implement the security measures suggested in this chapter only after carefully considering the implications for your users.

Is UNIX secure? Of course not. UNIX and Linux are not secure, nor is any other operating system that communicates on a network. If you must have absolute, total, unbreachable security, then you need a measurable air gap between your computer and any other device. Sometimes even an air gap isn't enough. In a 2014 paper, Genkin, Shamir, and Tromer described a technique to extract RSA encryption keys from laptops by analyzing the high-pitched frequencies they emit when decrypting a file.

Some people argue that you also need to enclose your computer in a special room that blocks electromagnetic radiation (look up "Faraday cage"). How fun is that?

This chapter examines the complex field of computer security: the sources of attacks, the basic ways to secure systems, the tools of the trade, and sources of additional information.

27.1 Elements of security

The field of information security is quite broad, but it is often best described by the “CIA triad.” This acronym stands for

- Confidentiality
- Integrity
- Availability

Confidentiality concerns the privacy of data. Access to information should be limited to those who are authorized to have it. Authentication, access control, and encryption are a few of the subcomponents of confidentiality. If a hacker breaks into a system and steals a database of customer contact information, a compromise of confidentiality has occurred.

Integrity relates to the authenticity of information. Data integrity technology ensures that information is valid and has not been altered in unauthorized ways. It also addresses the trustworthiness of information sources. When a secure web site presents a signed TLS certificate, it is proving to the user not only that the information it is sending is encrypted but also that a trusted certificate authority (such as VeriSign or Equifax) has verified the identity of the source. Technologies such as PGP also offer some assurance of data integrity.

Availability expresses the idea that information must be accessible to authorized users when they need it. Otherwise, the data has no value. Outages not caused by intruders (e.g., those caused by administrative errors or power outages) also fall into the category of availability problems. Unfortunately, availability is often ignored until something goes wrong.

Consider the CIA principles as you design, implement, and maintain systems and networks. As the old security adage goes, “security is a process.”

27.2 How security is compromised

In this section we take a general look at how real-world security problems tend to occur. Most security lapses fit into one of the following categories.

Social engineering

Human users (and administrators) of a computer system are the weakest links in the chain of security. Even in today's world of heightened security awareness, unsuspecting users with good intentions are easily convinced to give away sensitive information. No amount of technology can protect against the user element—you must ensure that your user community has a high awareness of security threats so that they can be part of the defense.

This problem manifests itself in many forms. Attackers cold-call their victims and pose as legitimately confused users in an attempt to get help accessing the system. Someone unintentionally posts sensitive information on a public forum while troubleshooting problems. Physical compromises occur when seemingly legitimate maintenance personnel show up to rewire the network closet.

The term "phishing" describes attempts to collect information from users or to coax them to into doing something foolish, such as installing malware. Phishing begins with deceptive emails, instant messages, text messages, or social media contacts. Targeted attacks (so called "spear phishing") can be especially hard to defend against because the communication often includes victim-specific information that lends an appearance of authenticity.

Social engineering is a powerful hacking technique and is one of the most difficult threats to neutralize. Your site security policy should include training for new employees. Regular, organization-wide communications are an effective way to inform staff about social media threats, physical security, email phishing, multifactor authentication, and good password selection.

To gauge your organization's resistance to social engineering, you might find it informative to attempt some social engineering attacks of your own. Be sure you have explicit permission to do this from your own managers, however. These exploits look very suspicious if they are performed without a clear mandate! They're also a form of internal spying, so they have the potential to generate resentment if they're not handled forthrightly.

Many organizations find it useful to communicate to users that administrators will never request their passwords. Tell users to report any such password requests to the IT department immediately.

Software vulnerabilities

Over the years, countless security-sapping bugs have been discovered in computer software. By exploiting subtle programming errors or context dependencies, hackers have been able to manipulate systems into a variety of compromising positions.

Buffer overflows are an example of a programming error with complex security implications. Developers often allocate a predetermined amount of temporary memory space, called a buffer, to store a particular piece of information. If the code isn't careful about checking the size of the data against the size of the container that's supposed to hold it, the memory adjacent to the allocated space is at risk of being overwritten. Crafty hackers can input carefully composed data that crashes the program or, in the worst case, executes arbitrary code.

Buffer overflows are a subcategory of a larger class of software security bugs known as input validation vulnerabilities. Nearly all programs accept some type of input from users (e.g., command-line arguments, parameters for an HTTP request). If the code processes such data without rigorously checking it for appropriate format and content, bad things can happen.

In some ways, open source operating systems have a leg up on security. The source code for Linux and FreeBSD is available to everyone, and thousands of people can (and do) scrutinize each line of code for possible security threats. This arrangement is widely believed to result in better security than that of closed operating systems, where a limited number of people have the opportunity to examine the code for weaknesses.

See Chapter 25 for more information about containers.

What can you as an administrator do to prevent this type of attack? It depends on the application, but one obvious approach is to reduce the privileges that your applications run with to minimize the impact of security bugs. A process running as an unprivileged user can do less damage than one that runs as root. For the paranoid, this approach can include a mandatory access control system such as SELinux. Containers with limited capabilities can also play a role here.

Over time, the open source community has developed a standard process for addressing software vulnerabilities. Initial reports should go directly to the software developers so that patches to address the issue can be developed and released before hackers formulate methods to exploit it. Later, details of the security issue are released publicly so that administrators become aware of it and so that the issue and the patches can receive public scrutiny. For this reason, keeping up with patches and security bulletins is an important part of most administrators' job. Fortunately, modern operating systems strive to make software updates straightforward and easy to automate.

Distributed denial-of-service attacks (DDoS)

A DDoS attack aims to interrupt a service or adversely impact its performance, making the service unavailable to users. It's usually achieved by flooding a site with network traffic, thereby consuming all the site's available bandwidth or system resources. DDoS attacks can be financially motivated (in which case the attacker holds the site for ransom), or they can be either political or retaliatory.

To conduct an attack, attackers plant malicious code on unprotected devices outside the victim's network. This code lets the attackers remotely command these intermediary systems, forming a "botnet." In the most common DDoS scenario, the minions of the botnet are instructed to pelt the victim with network traffic.

In recent years, botnets have been assembled from Internet-connected devices such as IP cameras, printers, and even baby monitors. These devices have essentially no security, and the owners usually remain unaware that their devices have been compromised. Sophisticated command-and-control tools for managing botnets are available on the dark web for anyone to purchase. Some of them even include free customer service!

In the fall of 2016, the Mirai botnet targeted security researcher and blogger Brian Krebs, slamming his site with 620 Gb/s of traffic from tens of thousands of source IP addresses. Naturally, his hosting provider asked him to kindly move somewhere else. The Mirai botnet code has since been open sourced.

Most of the responsibility for preventing and mitigating DDoS attacks falls to the network management layer. Software and hardware are available to detect attacks and shut them down while keeping legitimate services on-line. Public cloud providers and some co-location facilities are equipped with this technology. However, the mitigations aren't perfect and the threats are constantly shifting.

Insider abuse

Employees, contractors, and consultants are trusted agents of an organization and are granted special privileges. Sometimes these privileges are abused. Insiders can steal or reveal data, disrupt systems for financial gain, or create havoc for political reasons.

This type of attack is often the hardest of all to detect. Most security measures guard against external threats, so they aren’t effective against users who have been granted access. Insiders are typically not under suspicion in the first place; only the most rigorous organizations systematically monitor their own employees.

System administrators must never knowingly install back doors in the environment for their own use. Such facilities are too easily misinterpreted or exploited by others.

Network, system, or application configuration errors

Software can be configured securely or not-so-securely. Software is developed to be useful instead of annoying, hence not-so-securely is too often the default. Hackers frequently gain access by exploiting features that would be considered helpful and convenient in less treacherous circumstances: accounts without passwords, firewalls with overly relaxed rules, and unprotected databases, to name a few.

A typical example of a host configuration vulnerability is the standard practice of allowing Linux systems to boot without requiring a boot loader password. GRUB can be configured at installation to require a password, but administrators almost never activate this option. This omission leaves the system open to physical attack.

However, it's also a perfect example of the need to balance security against usability. Requiring a password means that if the system were unintentionally rebooted (e.g., after a power outage), an administrator would have to be physically present to get the machine running again.

One of the most important steps in securing a system is simply making sure that you haven't inadvertently put out a welcome mat for hackers. Problems in this category are the easiest to find and fix, although there are potentially a lot of them, and it's not always obvious what to check for. The port and vulnerability scanning tools covered later in this chapter can help a motivated administrator identify problems before they're exploited.

27.3 Basic security measures

Most systems do not come secured out of the box. Customizations made during and after installation change the security profile for new systems. Administrators should take steps to harden new systems, integrate them into the local environment, and plan for their long-term security maintenance.

When the auditors come knocking, it's useful to be able to prove that you have followed some kind of standard procedure, especially if that procedure conforms to external recommendations and best practices for your industry. Refer to *Sources of security information* for recommendations on selecting a system-hardening standard.

At the highest level, you can improve your site's security by keeping in mind a few rules of thumb:

- Apply the principle of least privilege by allocating only the minimum privileges needed by each entity, person, or role. This principle applies to firewall rules, user permissions, file permissions, and any other situation where access controls are used.
- Layer security measures to achieve defense in depth. For example, don't rely solely on your external firewall for network protection. Otherwise, you are simply building a structure like a Tootsie Pop: a hard, crunchy outside and a soft, chewy center.
- Minimize the attack surface. The fewer interfaces, exposed systems, unnecessary services, and unused or underused systems, the lower the potential for vulnerabilities and security weaknesses.

Automation is a close ally in the security war. Use configuration management and scripting to create repeatably secure systems and applications. The more security steps you automate, the less room is available for human error.

Software updates

Keeping systems updated with the latest patches is an administrator's highest-value security chore. Most systems are preconfigured to point at the vendor's package repository, which makes applying patches as simple as running a few commands. Larger sites can use a local repository that mirrors that of the vendor, thus saving external bandwidth and speeding updates.

A reasonable approach to patching should include the following elements:

- A regular schedule for installing routine patches that is followed diligently. Consider the impact of patches on users when designing this schedule. Monthly updates are usually sufficient, but be prepared to apply critical patches on short notice.
- A change plan that documents the impact of each set of patches, outlines post-installation testing steps, and describes how to back out the changes in the event of problems. Communicate this change plan to all relevant parties.
- An understanding of which patches pertain to the environment. Administrators should subscribe to vendor-specific security mailing lists and blogs, as well as to generalized security discussion forums such as Bugtraq.
- An accurate inventory of applications and operating systems used in your environment. This census helps ensure complete coverage. Use reporting software to keep track of your installed base.

Software patches sometimes introduce novel security problems and weaknesses of their own. However, most exploits target older vulnerabilities that are widely known. Statistically speaking, you are much better off with systems that are regularly updated. Make sure it's done methodically and consistently.

Unnecessary services

Stock systems come with lots of services running by default. Disable (and possibly remove) those that are unnecessary, especially if they are network daemons. One way to see which services are using the network is to use the **netstat** command. Here's partial output from a FreeBSD system:

```
freebsd$ netstat -an | grep LISTEN
tcp6       0      0  *.22           *.*            LISTEN
tcp6       0      0  *.2049         *.*            LISTEN
tcp6       0      0  *.989          *.*            LISTEN
tcp6       0      0  *.111          *.*            LISTEN
```



Linux is transitioning to the **ss** command for this purpose, but **netstat** still works there, too.

A variety of commands can help pinpoint the service that's using a port. For example, you can use **lsof**:

```
freebsd$ sudo lsof -i:22
COMMAND PID    USER  FD   TYPE SIZE/OFF NODE NAME
sshd    701    root  3u  IPv6       0t0  TCP *:ssh (LISTEN)
sshd    701    root  4u  IPv4       0t0  TCP *:ssh (LISTEN)
sshd    815    root  3u  IPv4       0t0  TCP 10.0.2.15:ssh->10.0.2.2:54834 (ESTABLISHED)
sshd    817 vagrant  3u  IPv4       0t0  TCP 10.0.2.15:ssh->10.0.2.2:54834 (ESTABLISHED)
```

See Chapter 2 for more about starting processes at boot time.

Once you have the PIDs, you can then use **ps** to identify specific processes. If a service is unneeded, stop it and make sure that it won't be restarted at boot time. You can also use the tools **fuser** or **netstat -p** if **lsof** is not available.

Limit your systems' overall footprint. The fewer packages, the less vulnerable software. The industry as a whole is beginning to address this issue by reducing the number of packages included in a default installation. Some specialized distributions such as CoreOS take this to the extreme and force nearly everything to run in a container.

Remote event logging

See Chapter 10 for more information about logging.

The syslog service forwards log information to files, lists of users, or other hosts on your network. Consider setting up a secure host to act as a central logging machine that parses forwarded events and responds appropriately. A single centralized log aggregator can capture logs from a variety of devices and alert administrators whenever meaningful events occur. Remote logging also prevents hackers from covering their tracks by rewriting or erasing log files on systems that have been compromised.

Most systems come configured to use syslog by default, but you will need to customize the configuration to set up remote logging.

Backups

Regular, tested system backups are an essential part of any site security plan. They fall into the “availability” bucket of the CIA triad. Make sure that all filesystems are regularly replicated and that you store some backups off-site. If a significant security incident occurs, you’ll then have an uncontaminated checkpoint from which to restore.

However, backups can also be a security hazard. Protect your backups by limiting (and monitoring) access and by encrypting backup files.

Viruses and worms

UNIX and Linux have been mostly immune from viruses. Only a handful exist (most of which are academic in nature), and none have wreaked the kind of costly havoc that has become commonplace in the Windows world. Nonetheless, this fact hasn't stopped certain antivirus vendors from predicting the demise of the platform from malware—unless you purchase their antivirus product at a special low price, of course.

The exact reason for the lack of malicious software is unclear. Some claim that UNIX simply has less market share than its desktop competitors and is therefore not an interesting target for virus authors. Others insist that UNIX's access-controlled environment limits the damage from self-propagating worms and viruses.

The latter argument has some validity. Because UNIX restricts write access to system executables at the filesystem level, unprivileged user accounts cannot infect the rest of the environment. Unless virus code is being run by root, the scope of infection is significantly limited. The moral, then, is not to use the root account for day-to-day activities. See this page for more comments on this issue.

See Chapter 18 for more information about email content scanning.

Perhaps counterintuitively, one valid reason to run antivirus software on UNIX servers is to protect your site's Windows systems from Windows-specific viruses. A mail server can scan incoming email attachments for viruses, and a file server can scan shared files for infection.

ClamAV by Tomasz Kojm is a popular, free antivirus product for UNIX and Linux. This widely used GPL tool is a complete antivirus tool kit that includes signatures for thousands of viruses. You can download the latest version from clamav.net.

Of course, one school of thought argues that antivirus software is itself counterproductive. Its detection and prevention rates are mediocre, and the cost of licensing and management are burdensome. All too frequently, antivirus software breaks other aspects of a system, resulting in a variety of tech support problems. Some compromises have even resulted from attacks on the antivirus infrastructure itself.

Recent versions of Microsoft Windows include a basic antivirus tool called Windows Defender. It's not the quickest to detect new forms of malware, but it's effective and relatively unlikely to interfere with other aspects of the system.

Root kits

The craftiest hackers try to cover their tracks and avoid detection. Often, they hope to continue using your system to distribute software illegally, probe other networks, or launch attacks against other systems. They often use “root kits” to help them remain undetected. Sony is notorious for having included root-kit-like capabilities in the copy protection software included on millions of music CDs.

Root kits are programs and patches that hide important system information such as process, disk, or network activity. They come in many flavors and vary in sophistication from simple application replacements (such as hacked versions of **ls** and **ps**) to kernel modules that are nearly impossible to detect.

Host-based intrusion detection software such as OSSEC is an effective way to monitor systems for the presence of root kits. File integrity monitoring tools, such as AIDE for Linux, can alert you to files that have changed unexpectedly. Root-kit-finding scripts (such as **chkrootkit**, chkrootkit.org) can identify known kits.

Although programs are available to help administrators remove root kits from a compromised system, the time it takes to perform a thorough cleaning would probably be better spent saving data and wiping the system. The most advanced root kits are aware of common removal programs and make attempts to subvert them.

Packet filtering

If you're connecting a system to a network that has Internet access, you *must* install a packet-filtering router or firewall between the system and the outside world. The packet filter should pass only traffic for services that you specifically want to offer from that system. Limiting the public exposure of your systems is a first-line defense. Many systems do not need to be directly accessible to the public Internet.

In addition to firewalling systems at the Internet gateway, you can double up with host-based packet filters such as **ipfw** for FreeBSD and **iptables** (or **ufw**) on Linux. Determine which services run on the host, and open ports only for those services. In some cases, you can also limit which source addresses are allowed to connect to each port. Many systems need only one or two ports to be accessible.

If your systems are in the cloud, you can use security groups rather than physical firewalls. When designing security group rules, be as granular as possible. Consider adding outbound rules as well, to limit an attacker's ability to make outbound connections from your hosts. See the platform-specific sections in *Cloud networking* for additional discussion of this topic.

Passwords and multifactor authentication

We're simple people with simple rules. Here's one: every account must have a password, and it needs to be something that can't easily be guessed. Password complexity rules may be a hassle, but they exist for a reason. Guessable passwords are one of the leading sources of compromise.

One of our favorite trends from recent years is the proliferation of support for multifactor authentication (MFA) systems. These schemes validate your identity both through something you *know* (a password or passphrase) and something you *have*, such as a physical device, commonly a phone. Almost any interface can be protected with MFA, from UNIX shell accounts to bank accounts. Enabling MFA is an easy and powerful security win.

For a variety of reasons, MFA is now an absolute minimum requirement for any Internet-facing portal that gives access to administrative privileges. That includes VPNs, SSH access, and administrative interfaces to web applications. An argument can be made that single-factor (password-only) authentication is not acceptable for *any* user authentication, but you *must* secure *at least* all administrative interfaces with MFA. Fortunately, excellent cloud-based MFA services are available, such as Google Authenticator and Duo (duo.com).

Vigilance

To ensure the security of your system, regularly monitor its health, network connections, process table, and overall status (usually, daily). Do regular self-assessments with the power tools discussed later in this chapter. Security compromises tend to start with a small foothold and expand, so the earlier you identify an anomaly, the better off you'll be. This is much easier said than done.

You might find it beneficial to work with an external firm to perform a comprehensive vulnerability analysis. These projects can draw your attention to issues that you hadn't previously considered. At a minimum, they establish a baseline understanding of the areas in which you're most exposed. Such engagements often reveal that hackers have already been nesting in the client's network.

Application penetration testing

Applications that are exposed to the Internet need their own security precautions in addition to general system and network hygiene. Because of the widespread proliferation of vulnerability data and exploit tools, it's a good idea to have all applications penetration tested to verify that they've been designed with security in mind and have appropriate controls in place.

Security is only as strong as the weakest link in the chain. If you have a secure network and system infrastructure, but an application running on that infrastructure allows access to sensitive data without a password (for example), you have won the battle but lost the war.

Penetration testing is a poorly defined discipline. Many companies that tout their penetration testing services focus mostly on smoke and mirrors. The Hollywood scenes of adolescent kids in windowless basements filled with 1980s-era terminals aren't entirely inaccurate. Buyer beware.

Fortunately, the Open Web Application Security Project (OWASP) publishes information about common application vulnerabilities and methods for probing applications for these issues. Our recommendation is that you have a professional third party (who specializes in application penetration testing) perform a penetration test at launch and periodically throughout the life of an application. Make sure they adhere to the OWASP methodology.

27.4 Passwords and user accounts

See this page for more information about password cracking.

In addition to securing all Internet-facing privileged access through multifactor authentication, it's important to select and manage passwords securely. In the world of **sudo**, administrators' personal passwords are just as important as root passwords. More so, in fact: the more frequently a password is used, the more opportunities there are for it to be compromised through methods other than brute-force decryption.

From a narrowly technical perspective, the most secure password of a given length consists of a random sequence of letters, punctuation, and digits. Years of propaganda and picky web site password forms have convinced most people that this is the sort of password they ought to be using. But of course, they never do, unless they use a password vault to remember passwords on their behalf. Random passwords are simply impractical to commit to memory at the lengths needed to withstand brute-force attacks (12 characters or longer).

Because password security increases exponentially with length, your best bet is to use a very long password (a "passphrase") that is unlikely to appear elsewhere but is easy to remember. You can throw in a misspelling or modified character for extra credit, but the general idea is to let the length do the heavy lifting for you.

For example, "six guests drank Evi's poisoned wine" is an excellent passphrase. (Or at least, it was until it appeared in this book.) That's true despite the fact that it consists mostly of common, lowercase, dictionary words, and despite the fact that the words are logically related and grammatically ordered.

The other core concept that all administrators and users must keep in mind is that a given passphrase should never be used for more than one purpose. It is all too common that a large breach occurs and usernames with passwords are exposed. If those usernames and passwords were used elsewhere, all those accounts are compromised now, too. Never use the same password across administrative boundaries (e.g., your personal banking site vs. social media).

Password changes

Change root and administrator passwords

- At least every six months
- Every time someone who had access to them leaves your site
- Whenever you wonder whether security might have been compromised

In the past, the conventional wisdom has been that passwords should be changed frequently to guard against the possibility of undetected compromises. However, password updates have their own risks, and they disrupt life for administrators. Competent hackers install backup access mechanisms as soon as they penetrate a site, so password changes are less helpful than they might initially seem.

It’s still advisable to make regularly scheduled changes, but don’t go overboard. If you really want to increase security, you’re better off obsessing about password quality.

Password vaults and password escrow

It's often said that passwords "should never be written down," but it's perhaps more accurate to say that they should never be left accessible to the wrong people. As security maven Bruce Schneier has noted, a scrap of paper in an administrator's wallet is relatively secure in comparison to most forms of Internet-connected storage.

A password vault is a piece of software (or a combination of software and hardware) that stores passwords for your organization in a more secure fashion than "Would you like Windows to remember this password for you?"

Several developments have made a password vault almost a necessity:

- The proliferation of passwords needed not just to log in to computers, but also to access web pages, configure routers and firewalls, and administer remote services
- The increasing need for strong passwords as computers get so fast that weak passwords are easily broken
- Regulations that require access to certain data to be traceable to a single person—no shared logins such as root

Password management systems became more popular in the wake of U.S. legislation that imposed additional requirements on sectors such as government, finance, and health care. In some cases, this legislation requires multifactor authentication.

Password vaults are also a great boon for sysadmin support companies who must securely and traceably manage passwords not only for their own machines but also for their customers' machines.

Password vaults encrypt the passwords they store. Typically, every user has a separate vault password. (Just when you thought your password travails were over, now you have even more passwords to manage and worry about!)

Many password vault implementations are available. Free ones for individuals (e.g., KeePass) store passwords locally, give all-or-nothing access to the password database, and do no logging. Appliances suitable for huge enterprises (e.g., CyberArk) can cost tens of thousands of dollars. Many of the commercial offerings charge either by the user or by the number of passwords they remember.

The vault system we particularly like is 1Password from AgileBits (1password.com). 1Password comes from the mass-market world, so it includes polished, cross-platform UIs and integration with common web browsers. 1Password has a "teams" layer that extends this foundation of personal password management into the domain of organizational secrets.

Another system worth evaluating is Secret Server from Thycotic (thycotic.com). This system is browser-based and was designed from the ground up to serve the needs of organizations. It includes extensive management and auditing features along with role-based access control (see this page) and fine-grained permission options.

One useful feature to look for in a password management system is a "break the glass" option, so named in honor of the hotel fire alarm stations that tell you to break the glass and pull the big red lever in the event of an emergency. In this case, "breaking the glass" means obtaining a password that you wouldn't normally have access to, with loud alarms being forwarded to other administrators. It's a nice compromise between parsimonious password sharing (a normal best practice) and the realities of emergency fire fighting.

---

*See this page for more information about the **passwd** file.*

Poor password management is a common security weakness. By default, the contents of the **/etc/passwd** and **/etc/shadow** files (or on FreeBSD, the **/etc/master.passwd** file) determine who can log in, so these files are the system's first line of defense against intruders. They must be scrupulously maintained and free of errors, security hazards, and historical baggage.

UNIX allows users to choose their own passwords, and although this is a great convenience, it leads to many security problems. The comments in the section *Passwords and user accounts* apply equally to user passwords.

It is important to regularly verify (preferably daily) that every login has a password. Entries in the **/etc/shadow** file that describe pseudo-users such as "daemon" who own files but never log in should have a star or an exclamation point in their encrypted password fields. These do not match any password and thus prevent use of the account.

At sites that use a centralized authentication scheme such as LDAP or Active Directory, the same logic applies. Enforce password complexity requirements and lock out accounts after a few failed login attempts.

Password aging

Most systems that have shadow passwords also let you compel users to change their passwords periodically, a facility known as password aging. This feature may seem appealing at first glance, but it has several problems. Users often resent having to change their passwords, and since they don't want to forget the new password, they choose something simple that is easy to type and remember. Many users switch between two passwords each time they are forced to change, or increment a digit in the password, defeating the purpose of password aging. PAM modules (see this page) can help enforce strong passwords to avoid this pitfall.



On Linux systems, the **chage** program controls password aging. Using **chage**, administrators can enforce minimum and maximum times between password changes, password expiration dates, the number of days to warn users before their passwords expire, the number of days of inactivity that are permissible before accounts are automatically locked, and more. The following command sets the minimum number of days between password changes to 2, sets the maximum number to 90, sets the expiration date to July 31, 2017, and warns the user for 14 days that the expiration date is approaching:

```
linux$ sudo chage -m 2 -M 90 -E 2017-07-31 -W 14 ben
```



Under FreeBSD, the **pw** command manages password aging parameters. This example sets the password validity period to 90 days and sets the expiration date to September 25, 2017.

```
freebsd$ sudo pw user mod trent -p  2017-09-25 -e 90
```

Group logins and shared logins

Any login that is used by more than one person is bad news. Group logins (e.g., “guest” or “demo”) are sure terrain for hackers to homestead and are prohibited in many contexts by federal regulations such as HIPAA. Don’t allow them at your site. However, technical controls can’t prevent users from sharing passwords, so education is the best enforcement tactic.

User shells

In theory, you can set the shell for a user account to be just about any program, including a custom script. In practice, the use of shells other than standards such as **bash** and **tcsh** is a dangerous practice. If you find yourself tempted to create such a login, you might consider a passphrase-less SSH key pair instead.

Rootly entries

The only distinguishing feature of the root login is its UID of zero. Since there can be more than one entry in the **/etc/passwd** file that uses this UID, there can be more than one way to log in as root.

A common way for a hacker to install a back door after having obtained a root shell is to edit new root logins into **/etc/passwd**. Programs such as **who** and **w** refer to the name stored in **utmp** rather than the UID that owns the login shell, so they cannot expose hackers that appear to be innocent users but are really logged in as UID 0.

Don't allow root to log in remotely, even through the standard root account. Under OpenSSH, you can set the PermitRootLogin configuration option to No in the **/etc/ssh/sshd_config** file to enforce this restriction.

Because of **sudo** (see this page), it's rare that you'll ever need to log in as root, even on the system console.

27.5 Security power tools

Some of the time-consuming chores mentioned in the previous sections can be automated with freely available tools. Here are a few of the tools you'll want to look at.

Nmap: network port scanner

Nmap's main function is to check a set of target hosts to see which TCP and UDP ports have servers listening on them. (As described here, a port is a numbered communication channel. An IP address identifies an entire machine, and an IP address + port number identifies a specific server or network conversation on that machine.) Since most network services are associated with "well known" port numbers, this information tells you quite a lot about the software a machine is running.

Running Nmap is a great way to find out what a system looks like to someone on the outside who is trying to break in. For example, here's a report from a production Ubuntu system:

```
ubuntu$ nmap -sT ubuntu.booklab.atrust.com

Starting Nmap 7.40 ( http://insecure.org ) at 2017-03-01 12:31 MST
Interesting ports on ubuntu.booklab.atrust.com (192.168.20.25):
Not shown: 1691 closed ports
PORT        STATE   SERVICE
25/tcp      open    smtp
80/tcp      open    http
111/tcp     open    rpcbind
139/tcp     open    netbios-ssn
445/tcp     open    microsoft-ds
3306/tcp    open    mysql

Nmap finished: 1 IP address (1 host up) scanned in 0.186 seconds
```

By default, **nmap** includes the **-sT** argument to try to connect to each privileged or well-known TCP port on the target host in the normal way. Use the **-p** option to explicitly specify a range of ports to scan.

Once a connection has been established, **nmap** immediately disconnects, which is impolite but not harmful to a properly written network server.

From the example above, we can see that the host ubuntu is running two services that are likely to be unused and that have historically been associated with security problems: **portmap** (rpcbind) and an email server (smtp). An attacker would most likely probe those ports for more information as a next step in the information-gathering process.

The STATE column in **nmap**'s output shows open for ports that have servers listening, closed for ports with no server, unfiltered for ports in an unknown state, and filtered for ports that cannot be probed because of an intervening packet filter. **nmap** does not classify ports as unfiltered unless it is running an ACK scan. Here are results from a more secure server, secure.booklab.atrust.com:

```
ubuntu$ nmap -sT secure.booklab.atrust.com

Starting Nmap 7.40 ( http://insecure.org ) at 2017-03-01 12:42 MST
Interesting ports on secure.booklab.atrust.com (192.168.20.35):
Not shown: 1691 closed ports
PORT       STATE   SERVICE
25/tcp     open    smtp
80/tcp     open    http

Nmap finished: 1 IP address (1 host up) scanned in 0.143 seconds
```

In this case, it's clear that the host is set up to allow SMTP (email) and an HTTP server. A firewall blocks access to other ports.

In addition to straightforward TCP and UDP probes, **nmap** also has a repertoire of sneaky ways to probe ports without initiating an actual connection. In most cases, **nmap** probes with packets that look like they come from the middle of a TCP conversation (rather than the beginning) and waits for diagnostic packets to be sent back. These stealth probes may be effective at getting past a firewall or at avoiding detection by a network security monitor on the lookout for port scanners. If your site uses a firewall (see *Firewalls*), it's a good idea to probe it with these alternative scanning modes to see what they turn up.

nmap has the magical and useful ability to guess what operating system a remote system is running by looking at the particulars of its implementation of TCP/IP. It can sometimes even identify the software that's running on an open port. The **-O** and **-sV** options, respectively, turn on this behavior. For example:

```
ubuntu$ sudo nmap -sV -O secure.booklab.atrust.com

Starting Nmap 7.40 ( http://insecure.org ) at 2017-03-01 12:44 MST
Interesting ports on secure.booklab.atrust.com (192.168.20.35):
Not shown: 1691 closed ports
PORT     STATE   SERVICE  VERSION
25/tcp   open    smtp     Postfix smtpd
80/tcp   open    http     lighttpd 1.4.13
Device type: general purpose
Running: Linux 2.4.X|2.5.X|2.6.X
OS details: Linux 2.6.16 - 2.6.24

Nmap finished: 1 IP address (1 host up) scanned in 8.095 seconds
```

This feature can be useful for taking an inventory of a local network. Unfortunately, it is also useful to hackers, who can base their attacks on known weaknesses of the target OSes and servers.

Keep in mind that most administrators don't appreciate your efforts to scan their network and point out its vulnerabilities, however well-intentioned your motive. Do not run **nmap** on someone else's network without permission from one of that network's administrators.

Nessus: next-generation network scanner

Nessus, originally released by Renaud Deraison in 1998, is a powerful and useful software vulnerability scanner. At this point, it uses more than 31,000 plug-ins to check for both local and remote security flaws. Although it is now a closed source, proprietary product, it is still freely available, and new plug-ins are released regularly. It is the most widely accepted and complete vulnerability scanner available.

Nessus prides itself on being the security scanner that takes nothing for granted. Instead of assuming that all web servers run on port 80, for instance, it scans for web servers running on any port and checks them for vulnerabilities. Instead of relying on the version numbers reported by the service it has connected to, Nessus can attempt to exploit known vulnerabilities to see if the service is susceptible.

Although a substantial amount of setup time is required to get Nessus running (it requires several packages that aren't installed on a typical system), it's well worth the effort. The Nessus system includes a client and a server. The server acts as a database and the client handles the GUI presentation. Nessus servers and clients exist for both Windows and UNIX platforms.

One of the great advantages of Nessus is the system's modular design, which makes it easy for third parties to add new security checks. Thanks to an active user community, Nessus is likely to be a useful tool for years to come.

Metasploit: penetration testing software

Penetration testing is the act of breaking into a computer network with the owner's permission for the purpose of discovering security weaknesses. Metasploit is an open source software package written in Ruby that automates this process.

Metasploit is controlled by the U.S.-based security company Rapid7, but its GitHub project has hundreds of contributors. Metasploit includes a database of hundreds of ready-made exploits for known software vulnerabilities. For those that have the desire and the skill, it's possible to write custom exploit plug-ins to add to the database.

Metasploit uses the following basic workflow:

1. Scan remote systems to discover information about them.
2. Select and execute exploits according to the information found.
3. If a target is penetrated, use included tools to pivot from the compromised system to other hosts on the remote network.
4. Run reports to document the results.
5. Clean up and revert all changes to the remote system.

Metasploit has several interfaces: a command line, a web interface, and a full GUI client. Choose the format that you like the best; they have equivalent functionality. Learn more at metasploit.com.

Lynis: on-box security auditing

If you were faced with finding holes in the walls of an old wooden barn, you might first walk around the outside of the barn and look for the large, gaping holes. Network-based vulnerability scanning tools like Nessus give you this view of a system's security profile. Walking inside the barn on a sunny day highlights the pinpoint-sized holes in walls. To get this same level of inspection of a system, you need a tool like Lynis that runs on the system itself.

Although unfortunately named, this security power tool performs both one-time and scheduled audits of a system's configuration, patching, and hardening state. This open source tool runs on Linux and FreeBSD systems and performs hundreds of automated compliance checks. Download it from cisofy.com/lynis.

John the Ripper: finder of insecure passwords

One way to thwart poor password choices is to try to break the passwords yourself and to force users to change passwords that you have broken. John the Ripper is a sophisticated tool by Solar Designer that implements various password-cracking algorithms in a single tool. It replaces the tool **crack**, which was covered in previous editions of this book.

Even though most systems use a shadow password file to hide encrypted passwords from public view, it's still wise to verify that your users' passwords are crack resistant, especially the passwords of system administrators who have **sudo** privileges.

Knowing a user's password can be useful because people tend to use the same password over and over again. A single password might grant access to another system, decrypt files stored in a user's home directory, and allow access to financial accounts on the web. (Needless to say, it's not security-smart to reuse a password this way. But nobody wants to remember hundreds of passwords.)

Considering its internal complexity, John the Ripper is an extremely simple program to use. Direct **john** to the file to be cracked, most often **/etc/shadow**, and watch the magic happen:

```
$ sudo ./john /etc/shadow
Loaded 25 password hashes with 25 different salts (FreeBSD MD5 [32/32])
password    (jsmith)
badpass     (tjones)
```

In this example, 25 unique passwords were read from the shadow file. As passwords are cracked, **john** prints them to the screen and saves them to a file called **john.pot**. The output contains the password in the left column with the login in parentheses in the right column. To reprint passwords after **john** has completed, run the same command with the **-show** argument.

As of this writing, the most recent stable version of John the Ripper is 1.8.0. It's available from openwall.com/john. Since John the Ripper's output contains the passwords it has broken, carefully protect this output and delete it as soon as you are done checking to see which users' passwords are insecure.

As with most security monitoring techniques, it's important to obtain explicit management approval before cracking passwords with John the Ripper.

Bro: the programmable network intrusion detection system

Bro is an open source network intrusion detection system (NIDS) that monitors network traffic and looks for suspicious activity. It was originally written by Vern Paxson and is available from bro.org.

Bro inspects all traffic flowing into and out of a network. It can operate in passive mode, in which it generates alerts for suspicious activity, or in active mode, in which it injects traffic to disrupt malicious activity. Both modes likely require modification of your site's network configuration.

Unlike other NIDSs, Bro monitors traffic flows rather than just matching patterns inside individual packets. This method of operation means that Bro can detect suspicious activity by observing who talks to whom, even without matching any particular string or pattern. For example, Bro can

- Detect systems used as "stepping stones" by correlating inbound and outbound traffic
- Detect a server that has a back door installed by watching for unexpected outbound connections immediately after an inbound one
- Detect protocols running on nonstandard ports
- Report correctly guessed passwords

Some of these features require substantial system resources, but Bro includes clustering support to help you manage a group of sensor machines.

The configuration language for Bro is complex and requires significant coding experience to use. Unfortunately, there is no simple default configuration for a novice to install. Most sites require a moderate level of customization.

Bro is supported to some extent by the Networking Research Group of the International Computer Science Institute (ICSI), but it's mostly maintained by the community of Bro users. If you are looking for a turnkey commercial NIDS, you will probably be disappointed by Bro. However, Bro can do things that no commercial NIDS can do, and it can either supplement or replace a commercial solution in your network.

Snort: the popular network intrusion detection system

Snort (snort.org) is an open source network intrusion prevention and detection system originally written by Marty Roesch and now maintained by Cisco, a commercial entity. It has become the de facto standard for home-grown NIDS deployments and is also the basis of many commercial and “managed services” NIDS implementations.

Snort itself is distributed for free as an open source package. However, Cisco charges a subscription fee for access to the most recent set of detection rules.

A number of third party platforms incorporate or extend Snort, and some of those projects are open source. One excellent example is Aanval (aanval.com), which aggregates data from multiple Snort sensors in a web console.

Snort captures raw packets off the network wire and compares them with a set of rules, aka signatures. When Snort detects an event that’s been defined as interesting, it can alert a system administrator or contact a network device to block the undesired traffic, among other actions.

Although Bro is a much more powerful system, Snort is a lot simpler and easier to configure, attributes that make it a good choice as a “starter” NIDS platform.

OSSEC: host-based intrusion detection

OSSEC is free software and is available as source code under the GNU General Public License. OSSEC serves up the following:

- Root kit detection
- Filesystem integrity checks
- Log file analysis
- Time-based alerting
- Active responses

OSSEC runs on the systems of interest and monitors their activity. It can send alerts or take action according to a set of rules that you configure. For example, OSSEC can monitor systems for the addition of unauthorized files and send email notifications like this one:

```
Subject: OSSEC Notification - courtesy - Alert level 7
Date: Fri, 03 Feb 2017 14:53:04 -0700
From: OSSEC HIDS <ossecm@courtesy.atrust.com>
To: <courtesy-admin@atrust.com>

OSSEC HIDS Notification.
2017 Feb 03 14:52:52

Received From: courtesy->syscheck
Rule: 554 fired (level 7) -> "File added to the system."
Portion of the log(s):

New file
'/courtesy/httpd/barkingseal.com/html/wp-content/uploads/2017/02/
    hbird.jpg'
added to the file system.

 --END OF NOTIFICATION
```

In this way, OSSEC acts as your 24/7 eyes and ears on the system. We recommend running OSSEC on every production system.

OSSEC basic concepts

OSSEC has two primary components: the manager (server) and the agents (clients). You need one manager on your network, and you should install that component first. The manager stores the file-integrity-checking databases, logs, events, rules, decoders, major configuration options, and system auditing entries for the entire network. A manager can connect to any OSSEC agent, regardless of its operating system. The manager can also monitor certain devices that do not have a dedicated OSSEC agent.

Agents run on the systems you want to monitor and report back to the manager. By design, they have a small footprint and operate with a minimal set of privileges. Most of the agent's

configuration is obtained from the manager. Communication between the server and the agent is encrypted and authenticated. You need to create an authentication key for each agent on the manager.

OSSEC classifies alerts by severity at levels 0 to 15; 15 is the highest severity.

OSSEC installation

OSSEC packages for most distributions are available at ossec.github.io.

Install the server on the system you want to be your OSSEC manager and then install the agent on that and all other systems you want to monitor. The install script asks some additional questions, such as to what email address alerts should be sent and which monitoring modules should be enabled.

Once the installation has finished, start OSSEC with

```
server$ sudo /var/ossec/bin/ossec-control start
```

Next, register each agent with the manager. On the server, run

```
server$ sudo /var/ossec/bin/manage_agents
```

You'll see a menu that looks something like this:

```
****************************************
* OSSEC HIDS v2.8 Agent manager.
* The following options are available:
****************************************
    (A)dd an agent (A).
    (E)xtract key for an agent (E).
    (L)ist already added agents (L).
    (R)emove an agent (R).
    (Q)uit.
Choose your action: A,E,L,R or Q:
```

Select option **A** to add an agent, and then type in the name and IP address of the agent. Next, select option **E** to extract the agent's key. Here's what that looks like:

```
Available agents:
    ID: 001, Name: linuxclient1, IP: 192.168.74.3
Provide the ID of the agent to extract the key (or '\q' to quit): 001
Agent key information for '001' is:
MDAyIGxpbnV4Y2xpZW50MSAxOTIuMTY4Ljc0LjMgZjk4YjMyYzlkMjg5MWJlMT
...
```

Finally, log in to the agent system and run **manage_agents** there:

```
agent$ sudo /var/ossec/bin/manage_agents
```

On the client, you will see that the menu has somewhat different options.

```
****************************************
* OSSEC HIDS v2.8 Agent manager.
* The following options are available:
****************************************
    (I)mport key from the server (I).
    (Q)uit.
Choose your action: I or Q:
```

Select option **I** and then cut and paste the key you extracted above. After you have added an agent, you must restart the OSSEC server. Repeat the process of key generation, extraction, and installation for each agent you want to connect.

OSSEC configuration

Once OSSEC is installed and running, you'll want to tweak it so that it gives you just enough information, but not too much. The majority of the configuration is stored on the server in the **/var/ossec/etc/ossec.conf** file. This XML-style file is well commented and fairly intuitive, but it contains dozens of options.

A common item you might want to configure is the list of files to ignore when checking file integrity. For example, if you have a custom application that writes its log file to **/var/log/customapp.log**, you can add the following line to the <syscheck> section of the file:

```
<syscheck>
    <ignore>/var/log/customapp.log</ignore>
</syscheck>
```

After you've made this change and restarted the OSSEC server, OSSEC will stop alerting you every time the log file changes. The many OSSEC configuration options are documented at ossec.net/main/manual/configuration-options.

It takes time and effort to get any HIDS system running and tuned. But after a few weeks, you'll have filtered out the noise, and the system will start to generate valuable information about changing conditions in your environment.

Fail2Ban: brute-force attack response system

Fail2Ban is a Python script that monitors log files such as **/var/log/auth.log** and **/var/log/apache2/error.log**. It looks for suspicious occurrences such as multiple failed login attempts or queries to unusually long URLs. Fail2Ban then takes action to address the threat. For example, it might temporarily block network traffic from a particular IP address or send email to an incident response team. Learn more at fail2ban.org.

27.6 Cryptography primer

Most software is designed with security in mind, and that implies a strong dose of cryptography. Security standards and regulations are opinionated about the selection of cryptographic algorithms and the type of data that must be protected with cryptography. Nearly all network protocols in modern use rely on cryptography for security. In short, cryptography is a pillar of computer security and sysadmins encounter it every day. It's well worth your time to understand the basics.

Cryptography applies mathematics to the problem of securing communications. A cryptographic algorithm, called a cipher, is the set of mathematical steps taken to secure a message. Such algorithms are designed by committees of experts who represent academic, government, and research interests from around the world. Acceptance of a new algorithm is a lengthy and tedious process. By the time it makes its way to the masses, it has been thoroughly vetted.

Encryption is the process of using a cipher to convert plain text messages to unreadable ciphertext. Decryption is the reverse of that process. Cryptographic messages (ciphertext) exhibit several advantageous properties:

- *Confidentiality:* messages are impossible to read for everyone except the intended recipients.
- *Integrity*: it is impossible to modify the contents without detection.
- *Non-repudiation:* the authenticity of the message can be validated.

In other words, cryptography lets you communicate secretly over unsecured channels with the added benefit of being able to prove the correctness of the message and the identity of sender. Very valuable indeed.

Some ciphers offer only a subset of these features. Often, multiple ciphers are used together to complete the full set, thus forming a hybrid cryptosystem.

Mathematics shows that strong cryptographic algorithms are reliably secure. However, software that implements the algorithms might have weaknesses, and the security of systems that guard cryptographic secrets might also be vulnerable, rendering the algorithms impotent. Protecting your secrets and choosing well-designed, easily updated cryptography software is therefore paramount.

Cryptographers have traditional names for three subjects who participate in a simple message exchange: Alice and Bob, who wish to communicate privately, and Mallory, a bad actor who wants to compromise their secrets, disrupt their communication, or impersonate one of the other principals. We've adopted this convention.

The upcoming subsections introduce several cryptographic primitives, the associated ciphers, and some common use cases for each.

Symmetric key cryptography

Symmetric key cryptography is sometimes called “conventional” or “classic” cryptography because the ideas behind it have been around for a long time. It’s simple: Alice and Bob share a secret key that they use to encrypt and decrypt messages. They must find a way to exchange the shared secret privately. Once they both know the key, they can reuse it as long as they wish. Mallory can only inspect (or interfere with) messages if she also has the key.

Symmetric keys are relatively efficient in terms of CPU usage and the size of the encrypted payloads. As a result, symmetric cryptography is often used in applications where efficient encryption and decryption are necessary. However, the need to distribute the shared key in advance is a serious impediment to many use cases.

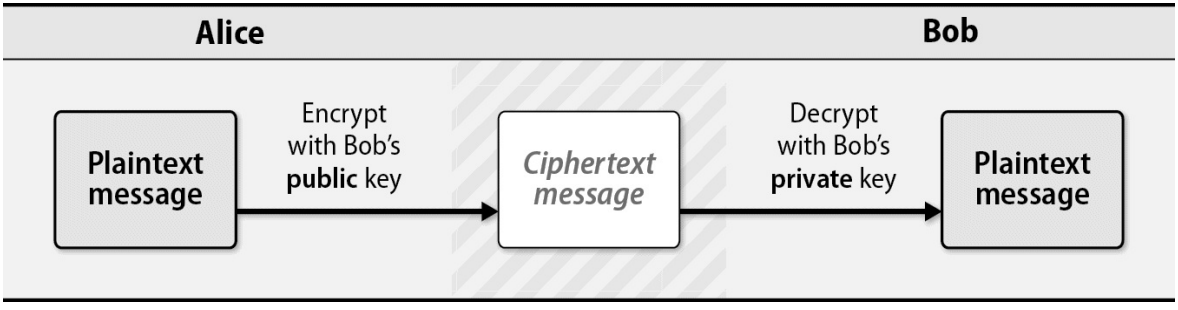
AES, the Advanced Encryption Standard from the United States National Institute of Standards and Technology (NIST), is perhaps the most widely used symmetric key algorithm. Twofish and its predecessor, Blowfish, designed by cryptographer and security expert Bruce Schneier, are also options. These algorithms play a role in the security of every network protocol you can shake your fist at, including SSH, TLS, IPsec VPNs, PGP, and many others.

Public key cryptography

A limitation of symmetric keys is the need to securely exchange the secret key in advance. The only way to do so with complete security is to meet in person without interference, a major inconvenience. For centuries, this requirement limited the practical utility of cryptography. The invention of public key cryptography, which addresses this problem, was therefore an extraordinary breakthrough when it occurred in the 1970s.

The scheme works as follows. Alice generates a pair of keys. The private key remains a secret, but the public key can be widely known. Bob similarly generates a key pair and publishes his public key. When Alice wants to send Bob a message, she encrypts it with Bob's public key. Bob, who holds the private key, is the only one who can decrypt the message.

Exhibit A: Sending a ciphertext message with public key cryptography



Alice can also sign the message with her private key. Bob can use Alice's signature and her public key to validate its authenticity. This process (simplified here for clarity) is known as a digital signature. It proves that Alice, not Mallory, sent the message.

The Diffie-Hellman-Merkle key exchange method was the first publicly available public key cryptosystem. Shortly thereafter, the RSA public key cryptosystem was circulated by the now-famous team of Ron Rivest, Adi Shamir, and Leonard Adleman. These techniques are the foundation of modern network security.

Public key ciphers, also called asymmetric ciphers, rely on the mathematical concept of trapdoor functions, in which a value is easy to compute, and yet it is difficult and expensive to derive the steps that produced that value. The performance characteristics of asymmetric ciphers generally render them impractical for encrypting large quantities of data. They are often paired with symmetric ciphers to realize the benefits of both: public keys establish a session and share a symmetric key, and the symmetric key encrypts the ongoing conversation.

Public key infrastructure

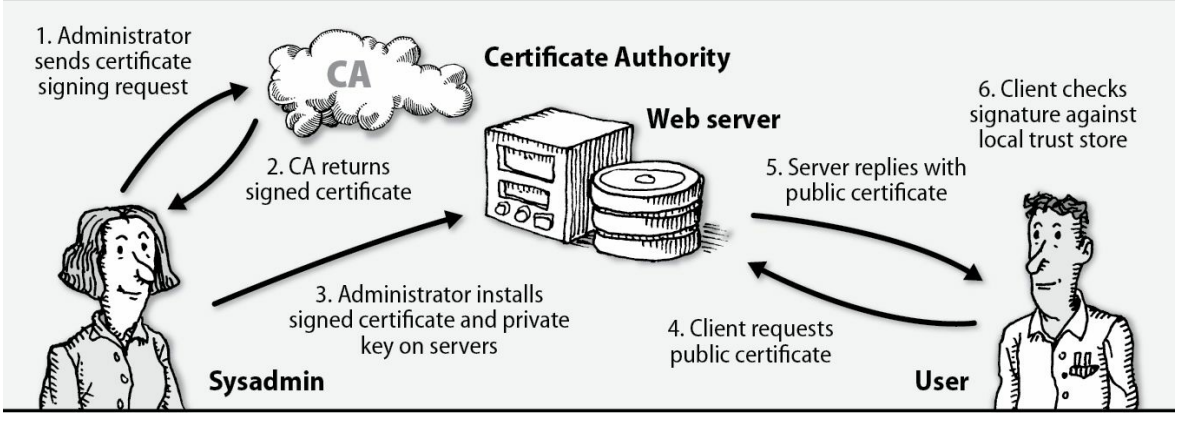
Organizing a trustworthy and reliable way to record and distribute public keys is a messy business. If Alice wants to send Bob a private message, she must trust that the public key she has for Bob is in fact his and not Mallory's. Validating the authenticity of public keys at Internet scale is a formidable challenge.

One solution, adopted by PGP, is a so-called web of trust. It boils down to a network of entities who trust each other to varying degrees. By following indirect chains of trust outside your personal network, you can establish that a public key is trustworthy with a reasonable degree of confidence. Unfortunately, the general public's interest in attending key-signing parties and cultivating a network of cryptofriends has been, shall we say, less than enthusiastic, as evidenced by PGP's continuing obscurity.

The Public Key Infrastructure, used to implement TLS on the web, addresses this problem by trusting a third party known as a Certificate Authority (CA) to vouch for public keys. Alice and Bob may not know each other, but they both trust the CA and know the CA's public key. The CA signs certificates for Alice and Bob's public keys with its own private key. Alice and Bob can then check the CA's endorsements to be sure the keys are legitimate.

The certificates of major CAs such as GeoTrust and VeriSign are bundled with operating system distributions. When a client begins an encrypted session, it will see that the peer's certificate has been signed by an authority already listed in the client's local trust store. Hence the client can trust the CA's signature and can trust that the peer's public key is valid. The scheme is depicted in Exhibit B.

Exhibit B: Public key infrastructure process for the web



Certificate authorities charge a fee for signing services, the price of which is set according to the reputation of the CA, market conditions, and various features of the certificate. Some variations,

such as so-called wild card certificates for entire subdomains or “extended validation certificates” with a more rigorous background check for the requesting entity, are more expensive.

The CA is implicitly trusted in this system. Initially, there were only a few trusted CAs, but many more have been added over time. Modern desktop and mobile operating systems trust hundreds of certificate authorities by default. The CAs themselves are therefore high-value targets for attackers, who would like to use the CA’s private key to sign certificates of their own devising.

When an authority is hacked, the entire system of trust is broken. Several CAs are known to have been compromised by attackers, and in other widely discussed incidents, CAs are known to have conspired with governments. We encourage readers to choose issuing CAs carefully when purchasing signing services.

In 2016, Let’s Encrypt was launched as a free service (sponsored by organizations such as the Electronic Frontier Foundation, the Mozilla Foundation, Cisco Systems, Stanford Law School, and the Linux Foundation) that issues certificates through an automated system. By the end of 2016, this service had issued over 24 million certificates. Given the well-publicized operational issues at some of the commercial CAs, we recommend Let’s Encrypt as a “probably just as secure” free alternative.

It’s also easy to act as your own certificate authority. You can create a CA with OpenSSL, import the CA’s certificate to the trust store throughout your site, and then issue certificates against that authority. This is a common practice for securing services on an intranet where the organization has full control over the trusted certificate store. See this page for more details.

Organizations should be careful when deciding to implement their own trusted authority on company-issued machines. Unless you have the same rigorous and audited security in place that the professional CAs do, you might just be creating a gaping vulnerability in your environment. As a corollary, if you work for an organization that installs its own certificate in your computers’ trusted store, suspect that your own security may be compromised and act accordingly.

Transport Layer Security

Transport Layer Security (TLS) uses public key cryptography and PKI to secure messages between nodes on a network. It is the successor to SSL, the Secure Sockets Layer, and you'll commonly see the acronyms SSL and TLS used interchangeably even though the old SSL is obsolete and deprecated. TLS paired with HTTP is known as HTTPS.

TLS runs as a separate layer that wraps TCP connections. It supplies only the security for the connection and does not involve itself in the HTTP transaction. Because of this hygienic architecture, TLS can secure not only HTTP but also other protocols such as SMTP.

Once a client and server have established a TLS connection, the contents of the exchange, including the URL and all headers, are protected by encryption. Only the host and port can be determined by an attacker since those details are visible through the encapsulating TCP connection. In the OSI model, TLS lies somewhere between layers 4 and 7.

Although the typical use case is one-way TLS encryption, in which the client validates the server, it is possible and increasingly common to use two-way TLS, sometimes known as mutual authentication. In this scheme, the client must present to the server a certificate that proves its own identity. This is, for example, how Netflix clients (set-top boxes and anything else that streams video from Netflix) are authenticated to the Netflix API.

The latest revision of TLS is 1.2. Disable all versions of SSL, along with TLS version 1.0, because of known weaknesses. TLS 1.3 is under active development and introduces major changes that will have significant implications for some industries.

A representative from the financial services industry attempted to influence a technical decision on the TLS development mailing list, but he was about two years too late. The concern was summarily rejected in an entertaining email thread. See the thread at goo.gl/uAEwPN.

Cryptographic hash functions

A hash function accepts input data of any length and generates a small, fixed-length value that is somehow derived from that data. The output value is variously referred to as a hash value, hash, summary, digest, checksum, or fingerprint. Hash functions are deterministic, so if you run a particular hash function on a particular input, you will always generate the same hash value.

Because hashes have a fixed length, only a finite number of possible output values exist. For example, an 8-bit hash has only 2^8 (that is, 256) possible outputs. Therefore, some inputs necessarily generate the same hash value, an event known as a collision. Longer hash values reduce the frequency of collisions but can never eliminate them entirely.

Hundreds of different hash functions are used in software, but the subset known as cryptographic hash functions are of particular interest to sysadmins and mathematicians. In this context, "cryptographic" means "real good." These hash functions are designed to have pretty much every desirable property you could want from a hash function, including the following:

- *Entanglement:* every bit of the hash value depends on every bit of the input data. On average, changing one bit of input should cause 50% of the hash bits to change.
- *Pseudo-randomness:* hash values should be indistinguishable from random data. Of course, hash values are *not* random; they are generated deterministically and reproducibly from input data. But they should still *look* like random data: they should have no detectable internal structure, should have no apparent relationship to the input data, and should pass all known statistical tests of randomness.
- *Nonreversibility:* given a hash value, it should be computationally infeasible to discover another input that generates the same hash value.

With a sufficiently high-quality hashing algorithm and a sufficiently long hash value length, we can make the leap of faith of assuming that two inputs that generate the same hash value are in fact the same input. Of course, that can't ever be theoretically certain, because all hashes have collisions. However, it can be made likely to any desired level of statistical proof by increasing the length of the hash value.

Cryptographic hashes verify the integrity of things. They can certify that a given configuration file or command binary has not been tampered with, or that a message signed by an email correspondent has not been modified in transit. For example, to verify that a FreeBSD system and a Linux system are using identical **sshd_config** files, we can use the following commands:

```
freebsd$ sha256 /etc/ssh/sshd_config
SHA256 (/etc/ssh/sshd_config) = 3ef2d95099363d...8c14f63c5b9f741ea8d5

linux$ sha256sum /etc/ssh/sshd_config
3ef2d95099363d...8c14f63c5b9f741ea8d5 /etc/ssh/sshd_config
```

We've elided part of the hash values for simplicity. As is typical for most use cases, the output values are shown here in hexadecimal notation. But keep in mind that the actual hash values are just bags of binary data and that this data can be represented in multiple ways.

Many cryptographic hash algorithms exist, but the only ones recommended for general use at this point are the SHA-2 and SHA-3 (Secure Hash Algorithm) families, which were selected through an extensive review process by NIST. The older SHA-1 has been compromised and should no longer be used.

Each of these algorithms exists in a range of variants with different hash value lengths. For example, SHA3-512 is the SHA-3 algorithm configured to generate a 512-bit hash value. A SHA algorithm without a version number, e.g., SHA-256, always refers to a member of the SHA-2 family.

Another common cryptographic hash algorithm, MD5, remains widely supported by cryptographic software. However, it's known to be vulnerable to engineered collisions, in which multiple inputs yield the same hash value. Although MD5 is no longer considered safe for use in cryptography, it's still a well-behaved hash function and is theoretically OK to use for low-security applications. But why bother? Just use SHA.

Open source software projects often publish hashes of the files they release to the community. The OpenSSH project, for example, distributes PGP signatures (which rely on cryptographic hash functions) of its tarballs for verification. To verify the authenticity and integrity of a download, you calculate the hash value of the file you actually downloaded and compare it to the published hash value, thus ensuring that you've received a complete and unmolested copy with no bit errors.

Hash functions are also used as a component of message authentication codes, aka MACs. The hash value inside a MAC is signed with a private key. The process of validating the MAC checks both the authenticity of the MAC itself (by decrypting it with the corresponding public key) and the integrity of the content (by checking it against the content hash). MAC schemes often play an important role in web application security.

Random number generation

Cryptographic systems need a source of random numbers from which to generate keys. But algorithms aren't known for their random and unpredictable behavior. What to do?

The gold standard for randomness is data from physically random processes such as radioactive decay and RF noise from the galactic core. These sources do exist: see random.org for access to some actual random data and an explanation of how it's derived. Interesting, but unfortunately not directly helpful for day-to-day cryptography.

Traditional "pseudo-random" number generators use methods similar to those of hash functions to generate sequences of random-looking data. However, the process is deterministic. Once you know the internal state of the random number generator, you can reproduce the output sequence exactly. Ergo, this is usually a poor option for cryptography. When you generate a random 2048-bit key, you want 2048 bits' worth of randomness, not 128 bits of number generator state that's been algorithmically massaged into occupying 2048 bits.

Fortunately, kernel developers have put considerable effort into recording subtle variations in system behavior and using these as sources of randomness. Sources include everything from the timing of packets seen on a network to the timing of hardware interrupts to the vagaries of communication with hardware devices such as disk drives. Even on virtual and cloud servers, there's still enough entropy available in the environment to generate reasonably random numbers.

All these sources feed forward into a secondary pseudo-random number generator that ensures the output stream of random data will have reasonable statistical properties. That data stream is then made available through a device driver. In Linux and FreeBSD, it's presented as **/dev/random** and **/dev/urandom**.

Two main things to know about random numbers:

- Nothing that runs in user space can compete with the quality of the kernel's random number generator. Never allow cryptographic software to generate its own random data; always make sure it uses random data from **/dev/random** or **/dev/urandom**. Most software does this by default.
- The choice of **/dev/random** vs. **/dev/urandom** is a matter of dispute, and unfortunately, the arguments are too subtle and mathematical to summarize here. The short version is that **/dev/random** on Linux is not guaranteed to generate data at all if the kernel feels that the system has not been accumulating enough entropy. Either get educated and pick one side or the other, or just use **/dev/urandom** and don't worry your pretty little head about this issue. Most experts seem to recommend the latter approach. FreeBSD users are excused from battle, as **/dev/random** and **/dev/urandom** on the BSD kernel are identical.

Cryptographic software selection

There is good reason to be highly suspicious of all security software, and the packages that provide cryptographic services most of all. Major international governments are rumored to have attempted to influence the design phases of cryptographic protocols and algorithms. It seems safe to assume that several well-funded groups are eager to compromise any cryptographic project that is not fully nailed down.

That said, we trust open source software more than closed. Projects such as OpenSSL have a history of serious vulnerabilities, but those problems are disclosed, mitigated, and released in a transparent, open forum. The project history and source code are examined by thousands of people.

Never rely on home-grown cryptography of any sort. It is difficult enough just to use libraries correctly! Bespoke cryptosystems are doomed to vulnerability.

The openssl command

openssl is an administrator's TLS multitool. You can use it to generate public/private key pairs, encrypt and decrypt files, examine the cryptographic properties of remote systems, create certificate authorities, convert among file formats, and myriad other cryptographic operations.

Preparing keys and certificates

One of the most common administrative functions of **openssl** is to prepare certificates for signing by a CA. Start by creating a 2048-bit private key:

```
$ openssl genrsa -out admin.com.key 2048
```

Use the private key to create a certificate signing request. **openssl** prompts for metadata known as the Distinguished Name (DN) to include with the request. It's also possible to present this information in an answers file instead of in-line, as shown below.

```
$ openssl req -new -sha256 -key admin.com.key -out admin.com.csr
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:Oregon
Locality Name (eg, city) []:Portland
Organization Name (eg, company) [Internet Widgits Pty Ltd]:ULSAH5E
Organizational Unit Name (eg, section) []:Crypto division
Common Name (e.g. server FQDN or YOUR name) []:server.admin.com
```

Submit the contents of **admin.com.csr** to the CA. The CA will perform a validation process to confirm that you are associated with the domain for which you're obtaining a certificate (usually by sending email to an address within that domain), and will subsequently return a signed certificate. You can then use **admin.com.key** and the CA-signed certificate in your web server configuration.

Most of these fields are fairly arbitrary, but the Common Name is important. It must match the name of the subdomain you want to serve. If, for instance, you want to serve TLS for www.admin.com, make that your Common Name. You can request multiple names for a single certificate or a wild card that matches all the names in a subdomain; for example, *.admin.com.

Once you have the certificate, you can examine its properties. Here are some of the details of a wild card certificate for *.google.com:

```
$ openssl x509 -noout -text -in google.com.pem
depth=2 /C=US/O=GeoTrust Inc./CN=GeoTrust Global CA
...
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: C=US, O=Google Inc, CN=Google Internet Authority G2
        Validity
            Not Before: Dec 15 13:48:27 2016 GMT
            Not After : Mar  9 13:35:00 2017 GMT
        Subject: C=US, ST=California, L=Mountain View, O=Google Inc,
    CN=*.google.com
```

The validity period is from Dec 15, 2016 through March 9, 2017. Clients who connect outside of this window will see error messages that the certificate is no longer valid. Tracking and managing certificate expiration dates is a common sysadmin responsibility.

Debugging TLS servers

Use **openssl s_client** to examine the TLS details of a remote server. This information can be quite useful when you are debugging web servers having certificate problems. For example, to examine the TLS properties of google.com (output truncated for brevity):

```
$ openssl s_client -connect google.com:443
---
New, TLSv1/SSLv3, Cipher is AES128-SHA
Server public key is 2048 bit
Secure Renegotiation IS supported
Compression: NONE
Expansion: NONE
SSL-Session:
    Protocol  : TLSv1
    Cipher    : AES128-SHA
    Session-ID: 4F72DC56EE4E80568F7E0EF9F59C8D7855C87F366B49BF1D9808...
    Session-ID-ctx:
    Master-Key: 095C6D8AF9B6B81E3E16BA05C0C9ACFACD72EF3335A32B86F3D3...
    Key-Arg   : None
    Start Time: 1484163220
    Timeout   : 300 (sec)
    Verify return code: 0 (ok)
---
```

You can use **openssl s_client** to check which versions of the TLS protocol a server supports. See also **openssl s_server**, which starts a generic TLS server. That can be handy for testing and debugging clients.

PGP: Pretty Good Privacy

See this page for more information about email privacy.

Phil Zimmermann's PGP package provides a tool chest of bread-and-butter cryptographic utilities focused primarily on email security. It can encrypt data, generate signatures, and verify the origin of files and messages.

PGP has an interesting history that includes lawsuits, criminal prosecutions, and the privatization of portions of the original PGP suite. Recently, PGP has been heavily criticized for exposing too much metadata in its most common usage modes. Exposed message length, recipients, and clear-text draft storage (among other things) are weaknesses that could potentially be exploited by attackers, especially nation-state actors with generous resources. That said, PGP is still significantly better than sending information in plain text.

PGP's file formats and protocols are being standardized by the IETF under the name OpenPGP, and multiple implementations of the proposed standard exist. The GNU project provides an excellent, free, and widely used implementation known as GnuPG at gnupg.org. For clarity, we refer to the systems collectively as PGP even though individual implementations have their own names.

Unfortunately, the UNIX and Linux versions are nuts-and-bolts enough that you have to understand a fair amount of cryptographic background to use them. Although you may find PGP useful in your own work, we don't recommend that you support it for users because it has been known to spark many puzzled questions. We have found the Windows version to be considerably easier to use than the **gpg** command with its dozens of different operating modes.

Software packages on the Internet are often distributed with a PGP signature file that purports to guarantee the origin and purity of the software. However, it is difficult or impossible for people who are not die-hard PGP users to validate these signatures. Users must have collected a library of public keys from people whose identities they have personally verified. Downloading a single public key along with a signature file and software distribution is approximately as secure as downloading the distribution alone.

Some email clients add on a simple GUI for encrypted incoming and outgoing messages. Google Chrome users can install the "end to end" extension to incorporate PGP support for Gmail.

Kerberos: a unified approach to network security

The Kerberos system, designed at MIT, attempts to address some of the issues of network security in a consistent and extensible way. Kerberos is an authentication system, a facility that "guarantees" that users and services are in fact who they claim to be. It does not afford any additional security or encryption beyond that.

Kerberos uses symmetric and asymmetric cryptography to construct nested sets of credentials called "tickets." Tickets are passed around the network to certify your identity and to give you access to network services. Each Kerberos site must maintain at least one physically secure machine (called the authentication server) on which to run the Kerberos daemon. This daemon issues tickets to users or services that present credentials (such as passwords) when they request authentication.

In essence, Kerberos improves on traditional password security in only two ways: it never transmits unencrypted passwords on the network, and it relieves users from having to type passwords repeatedly, making password protection of network services somewhat more palatable.

The Kerberos community boasts one of the most lucid and enjoyable documents ever written about a cryptosystem, Bill Bryant's "Designing an Authentication System: a Dialogue in Four Scenes." Despite its age it remains required reading for anyone interested in cryptography and is available at

web.mit.edu/kerberos/www/dialogue.html

Kerberos offers a better network security model than does the "ignoring network security entirely" model, but it is neither perfectly secure nor painless to install and run. It does not supersede the other security measures described in this chapter.

Unfortunately (and perhaps predictably), the Kerberos system distributed as part of Windows' Active Directory uses proprietary, undocumented extensions to the protocols. As a result, it does not interoperate well with distributions based on the MIT code. Fortunately, the **sssd** daemon lets UNIX and Linux systems interact with Active Directory's version of Kerberos. See the sections starting on this page for more information.

27.7 SSH, the Secure SHell

The SSH system, invented by Tatu Ylönen, is a protocol for remote logins and for securing network services on an insecure network. SSH's capabilities include remote command execution, shell access, file transfer, port forwarding, network proxy services, and even VPN tunneling. It is an indispensable tool, a veritable Swiss Army knife for system administrators.

SSH is a client/server protocol that uses cryptography for authentication, confidentiality, and integrity of communications between two hosts. It is designed for algorithmic flexibility, allowing the underlying cryptographic protocols to be updated and deprecated as the industry evolves. SSH is documented as a group of related protocols in RFCs 4250 through 4256.

In this section we discuss OpenSSH, the open source SSH implementation that is included and enabled by default on nearly every version of UNIX and Linux. We also mention a few alternative solutions for the adventurous and open-minded.

OpenSSH essentials

OpenSSH was developed by the OpenBSD project in 1999 and has since been maintained by that organization. The software suite consists of several commands:

- **ssh**, the client
- **sshd**, the server daemon
- **ssh-keygen**, for generating public/private key pairs
- **ssh-add** and **ssh-agent**, tools for managing authentication keys
- **ssh-keyscan**, for retrieving public keys from servers
- **sftp-server,** the server process for file transfer over SFTP
- **sftp** and **scp**, file transfer client utilities

In the most common and basic usage, a client establishes a connection to the server, authenticates itself, and subsequently opens a shell to execute commands. Authentication methods are negotiated according to mutual support and the preferences of the client and server. Many users can log in simultaneously. A pseudo-terminal is allocated for each, connecting their input and output to the remote system.

To initiate this process, a user invokes **ssh** with the remote host as the first argument:

```
$ ssh server.admin.com
```

ssh attempts a TCP connection on port 22, the standard SSH port assigned by IANA. When the connection is established, the server sends its public key for verification. If the server isn't already known and trusted, **ssh** prompts the user to confirm the server by presenting a hash of the server's public key called the key fingerprint:

```
The authenticity of host 'server.admin.com' can't be established.
ECDSA key fingerprint is SHA256:quLdFoXBI6OpU6HwnUy/K50cR9UuU.
Are you sure you want to continue connecting (yes/no)?
```

The intent is that a server administrator can communicate the host key to users in advance. Users can then compare the information they received from the administrator to the server's proffered fingerprint when they first connect. If the two match, the host's identity is proved.

Once the user accepts the key, the fingerprint is added to **~/.ssh/known_hosts** for future use. **ssh** won't mention the server's key again unless the key changes, in which case **ssh** displays a nasty warning message that the server's identity has changed.

In practice, this server verification dance is routinely ignored. Administrators rarely send the host key to users, and users blindly accept the host key without verification. This rubber-stamping of new host keys subjects users to man-in-the-middle attacks. Fortunately, the process can be automated and streamlined. We discuss this issue in *Host key verification with SSHFP*.

Once the host key has been accepted, the server lists the authentication methods it supports. OpenSSH implements all the methods described by the SSH RFCs, including simple UNIX password authentication, trusted hosts, public keys, GSSAPI for integration with Kerberos, and a flexible challenge/response scheme to support PAM and one-time passwords. Of these, public key authentication is the most commonly used and is the method we recommend for most sites. It offers the best balance of strong security and convenience. We discuss the use of public keys with SSH in more detail in *Public key authentication*.

ssh and **sshd** can be tuned for varying needs and security types. Configuration is found in the **/etc/ssh** directory, an uncharacteristically standard location among all flavors of UNIX and Linux. Table 27.1 enumerates the files found in that directory.

Table 27.1: Configuration files found in /etc/ssh

File	Permissions	Contents
ssh_config	0644	Site-wide client configuration
sshd_config	0644	Server configuration
moduli	0644	Prime numbers and generators for the DH key exchange
***_key**	0600	Private keys for every algorithm supported by the server
***_key.pub**	0644	A public key to match each private key

In addition to **/etc/ssh**, OpenSSH uses **~/.ssh** for storing public and private keys, for per-user client configuration overrides, and for a few other purposes. The **~/.ssh** directory is ignored unless its permissions are set to 0700.

OpenSSH has a respectable though not impeccable track record for security vulnerabilities. According to the CVE database (cve.mitre.org), several critical vulnerabilities were discovered in early versions. The last of these was documented in 2006. Occasional denial-of-service and bypass vulnerabilities continue to be announced, but most of them are considered relatively low risk. Still, it’s wise to update the OpenSSH packages as part of your regular patching schedule.

The ssh client

Getting started with **ssh** is straightforward, but its power and versatility lie in its many options. Through configuration you can choose cryptographic algorithms and ciphers, create convenient host aliases, set up port forwarding, and much more.

The basic syntax is

```
ssh [options] [username@]host [command]
```

For example, to check the disk space of **/var/log**:

```
$ ssh server.admin.com "df -h /var/log"
```

If you specify a *command*, **ssh** authenticates itself to the host, runs the command, and exits without opening an interactive shell. If you do not specify a *username*, **ssh** uses your local username on the remote host.

ssh reads configuration settings from the site-wide file **/etc/ssh/ssh_config** and processes additional options and overrides on a per-user basis from **~/.ssh/config**. Table 27.2 lists some of the more interesting options that you can tune in these files. We discuss some of these options in more detail later in this chapter.

Table 27.2: Useful SSH client configuration options

Option	Meaning	Default
AddKeysToAgent	Automatically add keys to **ssh-agent**	no
ConnectTimeout	Connection timeout in seconds	varies[a]
ControlMaster	Allow connection multiplexing	no
DynamicForward	Set up a SOCKS4 or SOCKS5 proxy	–
ForwardAgent	Enable **ssh-agent** forwarding	no
Host	Marker for a new host alias	–
IdentityFile	Path to an authentication private key	**~/.ssh/id_rsa**[b]
Port	Port to connect on	22
RequestTTY	Specify whether a TTY is needed	auto
ServerAliveInterval	Pings for connections to the server	0 (disabled)
StrictHostKeyChecking	Require (yes) or ignore (no) host keys	ask

a. The default is determined by the kernel's TCP defaults, which vary widely.
b. The precise name depends on the authentication algorithm. By default, all keys begin with **id_**.

When **ssh** assembles a final configuration, command-line arguments take precedence over entries in **~/.ssh/config**. The global configuration set in **/etc/ssh/ssh_config** is the lowest-priority source of configuration options.

ssh sends the current username as the login name if another value is not specified. You can supply a different username with the **-l** flag or the **@** syntax:

```
$ ssh -l hsolo server.admin.com
$ ssh hsolo@server.admin.com
```

Client options that are not available as direct arguments to **ssh** can still be set on the command line with the **-o** flag. For example, you could disable host checks for a server:

```
$ ssh -o StrictHostKeyChecking=no server.admin.com
```

The **-v** option prints debug messages. Specify it multiple times (maximum of three) to increase verbosity. You'll find this flag to be invaluable when debugging authentication problems.

For convenience, **ssh** returns the exit status of the remote command. Use this behavior to check for error conditions when invoking **ssh** from scripts.

Consult **man ssh** and **man ssh_config** to familiarize yourself with available options and features. Run **ssh -h** for a succinct summary.

Public key authentication

OpenSSH (and the SSH protocol generally) can use public key cryptography to authenticate users to remote systems. As a user, you start by creating a public/private key pair. You give the public key to the server administrator, who adds it to the server in the file **~/.ssh/authorized_keys**. You can then log in to the remote server by running **ssh** with the remote username and matching private key.

```
$ ssh -i ~/.ssh/id_ecdsa hsolo@server.admin.dom
```

Use **ssh-keygen** to generate a key pair. You can specify which cryptographic algorithm to use, as well as bit length and other characteristics. For example, to generate an ECDSA key pair with a 384-bit elliptic curve size:

```
$ ssh-keygen -t ecdsa -b 384
Generating public/private ecdsa key pair.
Enter file in which to save the key (/home/ben/.ssh/id_ecdsa): <return>
Enter passphrase (empty for no passphrase): <return>
Enter same passphrase again: <return>
Your identification has been saved in /home/ben/.ssh/id_ecdsa.
Your public key has been saved in /home/ben/.ssh/id_ecdsa.pub.
The key fingerprint is:
SHA256:VRh6raUfpn3YdtMm7GURbIoyfcp/npbwhsmvsdrlhK4 ben
```

The public key (**~/.ssh/id_ecdsa.pub**) and private key (**~/.ssh/id_ecdsa**) files are base64-encoded ASCII files. Never share the private key! **ssh-keygen** sets the permissions on the public and private key correctly as 0644 and 0600, respectively. This example uses ECDSA, but it's also fine to use **-t rsa** with 2048 or 4096 bits.

ssh-keygen prompts for an optional passphrase to encrypt the private key. If you use a passphrase, you must type it to decrypt the key before **ssh** can read it. A passphrase improves security because the authentication process gains an additional verification step: you must both have the key file and know the passphrase that decrypts it before you can authenticate.

We suggest setting a passphrase on all privileged accounts (that is, those with **sudo** privileges). If you need to use a key without a passphrase to enable an automated process, limit the corresponding server account's permissions.

If you're the server administrator and you need to add a public key for a new user, follow these steps:

1. Ensure that the user has an active account with a valid shell.
2. Get a copy of the user's public key from the user.
3. Create the user's **.ssh** directory with permissions 0700.

4. Add the public key to *~user*/**.ssh/authorized_keys** and set the permissions of that file to 0600.

For example, if user hsolo's public key were saved in **/tmp/hsolo.pub**, the process would look like this:

```
$ grep hsolo /etc/passwd
hsolo:x:503:503:Han Solo:/home/hsolo:/bin/bash
$ mkdir -p ~hsolo/.ssh && chmod 0700 ~hsolo/.ssh
$ cat /tmp/hsolo.pub >> ~hsolo/.ssh/authorized_keys
$ chmod 0600 ~hsolo/.ssh/authorized_keys
```

If you do this more than once you'll almost certainly find it prudent to script the procedure. Configuration management systems like Ansible and Chef handle this task cleanly.

The ssh-agent

The **ssh-agent** daemon caches decrypted private keys. You load your private keys into the agent, and **ssh** then automatically offers those keys when it connects to new servers, simplifying the process of connecting.

Use the **ssh-add** command to load a new key**.** If the key requires a passphrase, you'll be prompted to enter it. To list the currently loaded keys, type **ssh-agent -l**:

```
$ ssh-add ~/.ssh/id_ecdsa
Enter passphrase for ~/.ssh/id_ecdsa: <passphrase>
Identity added: ~/.ssh/id_ecdsa (~/.ssh/id_ecdsa)
$ ssh-add -l
384 SHA256:VRbIoyfcp/npbwhsmvsdrlhK4 ~/.ssh/id_ecdsa (ECDSA)
```

You can have many keys active at once. Remove a key with **ssh-add -d** *path*, or purge all loaded keys with **ssh-add -D**.

Oddly, to remove the private key from the agent, the public key must be in the same directory and have the same filename but with a **.pub** extension. If the public key is not available, you might receive a confusing error message that the key does not exist:

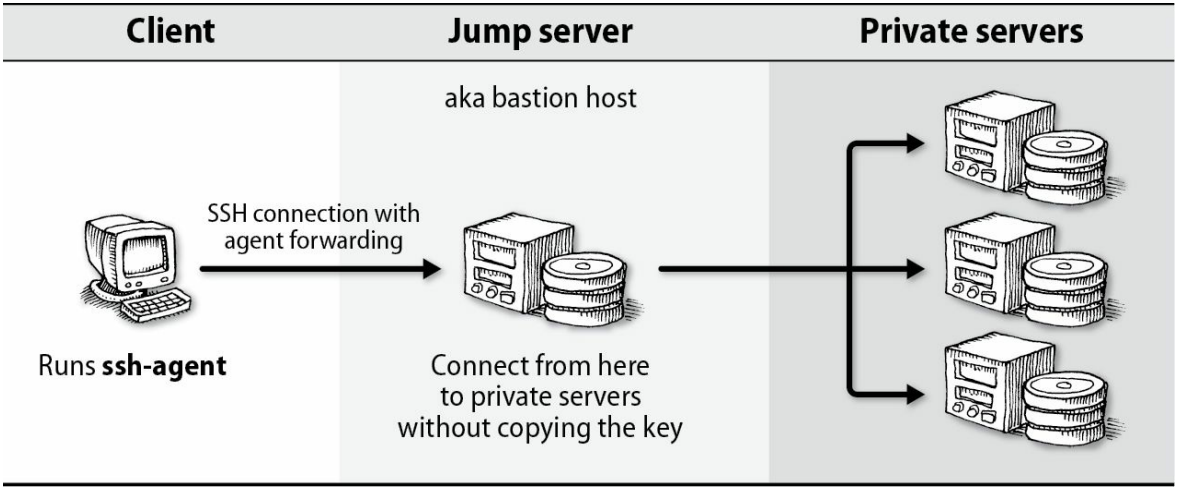
```
$ ssh-add -d ~/.ssh/id_ecdsa
Bad key file /home/ben/.ssh/id_ecdsa: No such file or directory
```

You can easily fix this problem by extracting the public key with **ssh-keygen** and saving it to the expected filename. (This extraction is possible because the private key file contains a copy of the public key as well as the private key.)

```
$ key=/home/ben/.ssh/id_ecdsa
$ ssh-keygen -yf $key > $key.pub
Enter passphrase: <passphrase>
```

ssh-agent is even more useful when you leverage its key forwarding feature, which makes the loaded keys available to remote hosts while you are logged in to them through **ssh**. You can use this feature to jump from one server to another without copying your private key to remote systems. See Exhibit C.

Exhibit C: ssh-agent forwarding



To enable agent forwarding, either add `ForwardAgent yes` to your **~/.ssh.config** file or use **ssh -A**.

Use key forwarding only on servers that you trust. Anyone in control of the server you've forwarded to can assume your identity and access remote systems. They cannot read your private keys directly, but they can use any that are available through the forwarding agent.

Host aliases in ~/.ssh/config

You'll undoubtedly encounter many different SSH configurations if you interact with or administer a large number of servers. To simplify your life, the **~/.ssh/config** file lets you set up aliases and overrides for individual hosts.

For example, consider two systems. The first is a web server with IP address 54.84.253.153 where **sshd** listens on port 2222. Your username on that server is han and you have a private key for authentication. The other is debian.admin.com, where your username is hsolo. You'd prefer to disable password authentication entirely, but the Debian server requires it.

To connect to these servers from the command line you could use option-larded commands such as these:

```
$ ssh -l han -p 2222 -i /home/han/.ssh/id_ecdsa 54.84.253.153
$ ssh -l hsolo debian.admin.com
```

Your client in this case must leave password authentication enabled (the default) because it's a hassle to type **-o PasswordAuthentication=no** all the time.

The following **~/.ssh/config** sets up aliases for these hosts and has the added benefit of disabling password authentication by default:

```
PasswordAuthentication no
Host web
   HostName 54.84.253.153
   User han
   IdentityFile /home/han/.ssh/id_ecdsa
   ForwardAgent yes
   Port 2222
Host debian
   Hostname debian.admin.com
   User hsolo
   PasswordAuthentication yes
```

Now you can use the much simpler commands **ssh web** and **ssh debian** to reach these hosts. The client reads the aliases and sets options automatically for each system.

ssh also understands some basic patterns for matching hosts. For example:

```
Host *
    ServerAliveInterval 30m
    ServerAliveCountMax 1
Host 172.20.*
    User luke
```

This example tells **ssh** to keep idle connections open for 30 minutes on all servers. It also sets username “luke” when connecting to hosts on the 172.20/16 network.

Host aliases become more powerful than you can possibly imagine when combined with other tricks of the OpenSSH trade.

Connection multiplexing

ControlMaster is a nifty **ssh** feature that enables connection multiplexing, thus considerably improving SSH performance over WAN links. When enabled, the first connection to a host creates a socket that can be reused. Subsequent connections share the socket but require separate authentication.

Turn on multiplexing with the ControlMaster, ControlPath, and ControlPersist options in a Host alias:

```
Host web
   HostName 54.84.253.153
   User han
   Port 2222
   ControlMaster auto
   ControlPath ~/.ssh/cm_socket/%r@%h:%p
   ControlPersist 30m
```

ControlMaster auto enables the feature. ControlPath creates a socket at the designated location. See **man ssh_config** for the substitutions that can be used in the ControlPath filename. In this case, the file is named according to the remote login username, host IP address, and port. Connecting to this host results in a socket like this one:

```
$ ls -l ~/.ssh/cm_socket/
srw-------  1 ben  ben  0 Jan  2 15:22 han@54.84.253.153:22
```

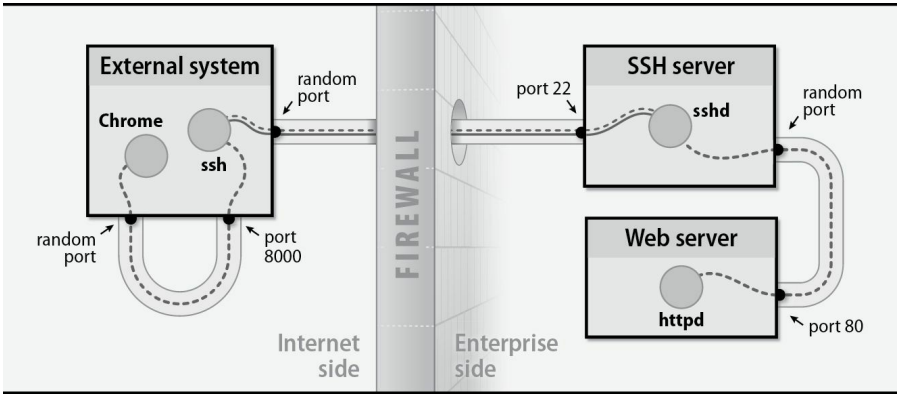
Such a pattern guarantees a unique filename for each socket. ControlPersist saves the socket for the specified period of time even if the first connection (the “master”) disconnects.

Spend the 30 seconds it takes to set this up, then make a donation to the OpenBSD foundation to thank them for implementing multiplexing and saving you time.

Port forwarding

Another useful ancillary feature of SSH is its ability to tunnel TCP connections securely through an encrypted channel, thereby allowing connectivity to insecure or firewalled services at remote sites. Exhibit D shows a typical use of an SSH tunnel and should help clarify how it works.

Exhibit D: An SSH tunnel for HTTP



In this scenario, a remote user—let's call her Alice—wants to establish an HTTP connection to a web server on an enterprise network. Access to that host or to port 80 is blocked by the firewall, but having SSH access, Alice can route the connection through the SSH server.

To set this up, Alice logs in to the remote SSH server with **ssh**. On the **ssh** command line, she specifies an arbitrary (but specific; in this case, 8000) local port that **ssh** should forward through the secure tunnel to the remote web server's port 80.

```
$ ssh -L 8000:webserver:80 server.admin.com
```

All source ports in this example are marked as random since programs choose an arbitrary port from which to initiate connections.

To access the web server, Alice can now connect to port 8000 on her own machine. The local **ssh** receives the connection and tunnels Alice's traffic over the existing SSH connection to the remote **sshd**. In turn, **sshd** forwards the connection to the web server on port 80.

Of course, tunnels such as these can be intentional or unintentional back doors as well. Use tunnels with caution and also watch for unauthorized use of this facility by users. You can disable port forwarding in **sshd** with the AllowTCPForwarding no configuration option.

sshd: the OpenSSH server

The OpenSSH server daemon, **sshd**, listens on port 22 (by default) for connections from clients. Its configuration file, **/etc/ssh/sshd_config**, boasts myriad options, some of which may need to be tuned for your site.

sshd runs as root. It forks an unprivileged child process for each connected client with the same permissions as the connecting user. If you make changes to the **sshd_config** file, you can force **sshd** to reload by sending a HUP signal to the parent process.

```
$ sudo kill -HUP $(sudo cat /var/run/sshd.pid)
```

In Linux, you can also run **sudo systemctl reload sshd**. The changes take effect for new connections. Existing connections are preserved without interruption but continue to use their previous settings.

The following example **sshd_config** includes some commonly adjusted options configured to balance server security with users' convenience:

```
# Set to inet for IPv4-only or inet6 for IPv6-only
AddressFamily any

# Allows only the named users and groups to log in
# Somewhat draconian. Adding/removing users requires reload
AllowUsers foo bar hsolo
AllowGroups admins

# TCP forwarding is convenient but can be abused
AllowTcpForwarding yes

# Display a message before users authenticate
# Important for inane legal reasons and compliance requirements
Banner /etc/banner

# We prefer to allow public key authentication only
ChallengeResponseAuthentication no
PasswordAuthentication no
RSAAuthentication no
GSSAPIAuthentication no
HostbasedAuthentication no
PubkeyAuthentication yes

# Disconnect inactive clients after 5 minutes
ClientAliveInterval 300
ClientAliveCountMax 1
```

```
# Allow compression at all times
Compression yes

# Do not allow remote hosts to use forwarded ports
GatewayPorts no

# Record failed login attempts
LogLevel VERBOSE

# Reduced from the default of 6
MaxAuthTries 3

# Do not allow root to log in (encourages use of sudo)
PermitRootLogin no

# Prevent users from setting their environment in an authorized_keys file
PermitUserEnvironment no

# Use the "auth" facility for syslog messages
SyslogFacility AUTH

# Kill the session if a TCP  connection is lost
TCPKeepAlive no

# Do not allow X forwarding if your site does not use X
X11Forwarding no
```

We encourage you to list the acceptable ciphers and key exchange algorithms explicitly. We don’t include the details here because the names are quite long and are a moving target anyway. Follow Mozilla’s OpenSSH configuration guidelines, which can be found at goo.gl/Xxgx7H (deep link into wiki.mozilla.org).

Host key verification with SSHFP

Recall from earlier in this section that SSH server host keys are routinely ignored by server administrators and users alike. Cloud instances exacerbate the problem because even the administrator has no knowledge of the host key before logging in.

Fortunately, a DNS record known as SSHFP has been developed to address this issue. The premise is that the server's key is stored as a DNS record. When a client connects to an unknown system, SSH looks up the SSHFP record to verify the server's key rather than asking the user to verify it.

The **sshfp** utility, available from github.com/xelerance/sshfp, generates SSHFP DNS resource records either by scanning a remote server (the **-s** flag) or by parsing a previously accepted key from a **known_hosts** file (the **-k** flag; this is also the default). Of course, either choice assumes that the source of the key is known to be correct.

For example, the following command generates a BIND-compatible SSHFP record for server.admin.com:

```
$ sshfp server.admin.com
server.admin.com IN  SSHFP 1 1 94a26278ee713a37f6a78110f1ad9bd...
server.admin.com IN  SSHFP 2 1 7cf72d02e3d3fa947712bc56fd0e0a3i...
```

Add these records to the domain's zone file (be mindful of the names and the $ORIGIN), reload the domain, and use **dig** to verify the key:

```
$ dig server.admin.com. IN SSHFP | grep SSHFP
; <<>> DiG 9.5.1-P2 <<>> server.admin.com. IN SSHFP
; server.admin.com. IN SSHFP
server.admin.com.    38400   IN  SSHFP 1 1 94a26278ee713a37f6a78110f...
server.admin.com.    38400   IN  SSHFP 2 1 7cf72d02e3d3fa947712bc56f...
```

ssh does not consult SSHFP records by default. Add the VerifyHostKeyDNS option to **/etc/ssh/ssh_config** to enable checking. As with most SSH client options, you can also pass **-o VerifyHostKeyDNS=yes** on the **ssh** command line when you first access a new system.

You can automate this process by generating the SSHFP record in the server's initialization scripts. Use dynamic DNS or your favorite DNS provider's API to create the record.

File transfers

OpenSSH has two utilities for transferring files: **scp** and **sftp**. On the server side, **sshd** runs a separate process called **sftp-server** to handle file transfers. SFTP has no relationship to the older and insecure File Transfer Protocol, FTP.

You can use **scp** to copy files from your system to a remote host, from a remote host to your system, or between remote hosts. The syntax mirrors that of **cp** with some extra decorations to designate hosts and usernames.

```
$ scp ./file server.admin.com:
$ scp server.admin.com:file ./file
$ scp server1.admin.com:file server2.admincom:file
```

sftp is an interactive experience similar to a traditional FTP client. You can also find graphical SFTP interfaces for most desktop operating systems.

Alternatives for secure logins

Most systems and sites rely on OpenSSH for secure remote access, but it is not the only choice.

Dropbear is an SSH implementation with a focus on maintaining a compact footprint. It compiles to a statically linked 110KiB binary, perfect for consumer-grade routers and other embedded devices. It includes some of the same features as OpenSSH, such as public key authentication and agent forwarding.

Gravitational's Teleport is another alternative SSH server that offers several advantages. Its authentication model relies on expiring certificates, which eliminates the problem of distributing and configuring users' public keys. Among Teleport's impressive features are an optional audit trail for each connection and a nifty collaboration system that lets multiple users share a session. Compared to OpenSSH, Teleport is relatively new and unproven, but to date there have been no reported vulnerabilities. We expect Gravitational to continue their rapid pace of development.

Mosh, developed by a brilliant team at MIT, is a replacement for SSH. Unlike SSH, Mosh operates on encrypted and authenticated UDP datagrams. It is designed for better performance over WAN connections and for roaming. For example, you can resume connections if you move from one IP address to another or if your connection drops. First released in 2012, Mosh has a much shorter history than OpenSSH, but in its first few years it has had no reported security vulnerabilities. Like Dropbear, it has a much smaller footprint than OpenSSH.

27.8 Firewalls

In addition to protecting individual machines, you can also implement security precautions at the network level. The basic tool of network security is the firewall, a device or piece of software that prevents unwanted packets from accessing networks and systems. Firewalls are ubiquitous today and are found in devices ranging from desktop systems and servers to consumer routers and enterprise-grade network appliances.

Packet-filtering firewalls

A packet-filtering firewall limits the types of traffic that can pass through your Internet gateway (or through an internal gateway that separates domains within your organization) according to information in the packet header. It's much like driving your car through a customs checkpoint at an international border crossing. You specify which destination addresses, port numbers, and protocol types are acceptable, and the gateway simply discards (and in some cases, logs) packets that don't meet the profile.

Packet-filtering software is included in Linux systems in the form of **iptables** (and its easier-to-use front end, **ufw**) and on FreeBSD as **ipfw**. See the details beginning here for more information.

Although these tools are capable of sophisticated filtering and bring a welcome extra dose of security, we generally discourage the use of UNIX and Linux systems as network routers and, most especially, as enterprise firewall routers. The complexity of general-purpose operating systems makes them inherently less secure and less reliable than task-specific devices. Dedicated firewall appliances such as those made by Check Point and Cisco are a better option for site-wide network protection.

Filtering of services

Most well-known services are associated with a network port in the **/etc/services** file or its vendor-specific equivalent. The daemons responsible for these services bind to the appropriate ports and wait for connections from remote sites. Most of the well-known service ports are “privileged,” meaning that their port numbers are in the range 1 to 1023. These ports can be used only by a process running as root (or with an appropriate Linux capability). Port numbers 1024 and higher are referred to as nonprivileged ports.

Service-specific filtering is predicated on the assumption that the client (the machine that initiates a TCP or UDP conversation) uses a non-privileged port to contact a privileged port on the server. For example, if you wanted to allow only inbound HTTP connections to a machine with the address 192.108.21.200, you would install a filter that allowed TCP packets destined for port 80 at that address and that permitted outbound TCP packets from that address to anywhere. (Port 80 is the standard HTTP port as defined in **/etc/services**.) The exact way that such a filter is installed depends on the kind of router or filtering system you are using.

Modern security-conscious sites use a two-stage filtering scheme. One filter is a gateway to the Internet, and a second filter lies between the outer gateway and the rest of the local network. The idea is to terminate all inbound Internet connections on systems that lie between these two filters. If these systems are administratively separate from the rest of the network, they can handle a variety of services for the Internet with reduced risk. The partially secured network is usually called the demilitarized zone or DMZ.

The most secure way to use a packet filter is to start with a configuration that allows no inbound connections. You can then liberalize the filter bit by bit as you discover useful things that don’t work and, hopefully, move any Internet-accessible services onto systems in the DMZ.

Stateful inspection firewalls

The theory behind stateful inspection firewalls is that if you could carefully listen to and understand all the conversations (in all languages) that were taking place in a crowded airport, you could make sure that someone wasn't planning to bomb a plane later that day. Stateful inspection firewalls are designed to inspect the traffic that flows through them and compare the actual network activity to what "should" be happening.

For example, if the packets exchanged in an H.323 video sequence name a port to be used later for a data connection, the firewall should expect a data connection to occur only on that port. Attempts by the remote site to connect to other ports are presumably bogus and should be dropped.

So what are vendors really selling when they claim to deliver stateful inspection? Their products either monitor a limited number of connections or protocols or they search for a particular set of "bad" situations. Not that there's anything wrong with that; clearly, some benefit is derived from any technology that can detect traffic anomalies. In this particular case, however, remember that the claims are *mostly* marketing hype.

Firewalls: safe?

A firewall should not be your only means of defense against intruders. It's only one component of what ought to be a carefully considered, multilayered security strategy. Firewalls often confer a false sense of security. If a firewall lulls you into relaxing other safeguards, it will have had a *negative* effect on the security of your site.

Every host within your organization should be individually patched, hardened, and monitored with tools such as Bro, Snort**,** Nmap, Nessus, and OSSEC. Likewise, your entire user community needs to be educated about basic security hygiene.

Ideally, local users should be able to connect to any Internet service they like, but machines on the Internet should only be able to connect to a limited set of local services hosted within your DMZ. For example, you might want to allow SFTP access to a local archive server and allow SMTP connections to a server that receives incoming email.

To maximize the value of your Internet connection, we recommend that you emphasize convenience and accessibility when deciding how to set up your network. At the end of the day, it's the system administrator's vigilance that makes a network secure, not a fancy piece of firewall hardware.

27.9 Virtual private networks (VPNs)

In its simplest form, a VPN is a connection that makes a remote network appear as if it were directly connected, even if it is physically thousands of miles and many router hops away. For increased security, the connection is not only authenticated in some way (usually with a "shared secret" such as a passphrase), but the end-to-end traffic is also encrypted. Such an arrangement is usually referred to as a "secure tunnel."

Here's a good example of the kind of situation in which a VPN is handy. Suppose that a company has offices in Chicago, Boulder, and Miami. If each office has a connection to a local ISP, the company can use VPNs to transparently (and, for the most part, securely) connect the offices across the untrusted Internet. The company could achieve a similar result by leasing dedicated lines to connect the three offices, but that would be considerably more expensive.

Another good example is a company whose employees telecommute from their homes. VPNs let those users reap the benefits of their high-speed and inexpensive cable modem service while making it appear that they are directly connected to the corporate network.

Because of the convenience and popularity of this functionality, everybody is offering some type of VPN solution. You can buy it from your router vendor as a plug-in for your operating system or even as a dedicated VPN device for your network. Depending on your budget and scalability needs, you may want to consider one of the many commercial VPN solutions.

If you're without a budget and looking for a quick fix, SSH can do secure tunneling for you. See *Port forwarding*.

IPsec tunnels

If you're a fan of IETF standards (or of saving money) and need a real VPN solution, take a look at IPsec (Internet Protocol security). IPsec was originally developed for IPv6, but it has also been widely implemented for IPv4. IPsec is an IETF-approved, end-to-end authentication and encryption system. Almost all serious VPN vendors ship a product that has at least an IPsec compatibility mode. Linux and FreeBSD include native kernel support for IPsec.

IPsec uses strong cryptography to implement both authentication and encryption services. Authentication ensures that packets are from the right sender and have not been altered in transit, and encryption prevents the unauthorized examination of packet contents.

In tunnel mode, IPsec encrypts the transport layer header, which includes the source and destination port numbers. Unfortunately, this scheme conflicts with most firewalls. For this reason, most modern implementations default to transport mode, in which only the payloads of packets (the data being transported) are encrypted.

There's a gotcha involving IPsec tunnels and MTU size. You must ensure that once a packet has been encrypted by IPsec, nothing fragments it along the network path the tunnel traverses. To achieve this feat, you might have to lower the MTU on the devices in front of the tunnel. (In the real world, 1,400 bytes usually works.) See this page in the TCP chapter for more information about MTU size.

All I need is a VPN, right?

Sadly, there's a downside to VPNs. Although they do build a (mostly) secure tunnel across the untrusted network between the two endpoints, they don't usually address the security of the endpoints themselves. For example, if you set up a VPN between your corporate backbone and your CEO's home, you may inadvertently be creating a path for your CEO's 15-year-old daughter to have direct access to everything on your network.

Bottom line: you need to treat connections from VPN tunnels as external connections and grant them additional privileges only as necessary and only after careful consideration. Think about adding a special section to your site security policy to cover the rules applying to VPN connections.

27.10 Certifications and standards

If the subject matter of this chapter seems daunting to you, don’t fret. Computer security is a complicated and vast topic, as countless books, web sites, and magazines can attest. Fortunately, much has been done to help quantify and organize the available information. Dozens of standards and certifications exist, and mindful system administrators should reflect on their guidance.

Certifications

Large corporations often employ many full-time employees whose job is guarding information. To gain credibility in the field and keep their knowledge current, these professionals attend training courses and obtain certifications. Prepare yourself for acronym-fu as we work through a few of the most popular certifications.

One of the most widely recognized security certifications is the CISSP, or Certified Information Systems Security Professional. It is administered by $(ISC)^2$, the International Information Systems Security Certification Consortium (say that ten times fast!). One of the primary draws of the CISSP is $(ISC)^2$'s notion of a "common body of knowledge" (CBK), essentially an industry-wide best practices guide for information security. The CBK covers law, cryptography, authentication, physical security, and much more. It's an incredible reference for security folks.

One criticism of the CISSP has been its concentration on breadth and consequent lack of depth. So many topics in the CBK, and so little time! To address this, $(ISC)^2$ has issued CISSP concentration programs that focus on architecture, engineering, and management. These specialized certifications add depth to the more general CISSP certification.

The SANS Institute created the Global Information Assurance Certification (GIAC) suite of certifications in 1999. Three dozen separate exams cover the realm of information security with tests divided into five categories. The certifications range in difficulty from the moderate two-exam GISF to the 23-hour, expert-level GSE. The GSE is notorious as one of the most difficult certifications in the industry. Many of the exams focus on technical specifics and require quite a bit of experience.

Finally, the Certified Information Systems Auditor (CISA) credential is an audit and process certification. It focuses on business continuity, procedures, monitoring, and other management content. Some consider the CISA an intermediate certification that is appropriate for an organization's security officer role. One of its most attractive aspects is that it involves only a single exam.

Although certifications are a personal endeavor, their application to business is undeniable. More and more companies now recognize certifications as the mark of an expert. Many businesses offer higher pay and promotions to certified employees. If you decide to pursue a certification, work closely with your organization to have it help pay for the associated costs.

Security standards

Because of the ever-increasing reliance on data systems, laws and regulations have been created to govern the management of sensitive, business-critical information. Major pieces of U.S. legislation such as HIPAA, FISMA, NERC CIP, and the Sarbanes-Oxley Act (SOX) have all included sections on IT security. Although the requirements are sometimes expensive to implement, they have helped give the appropriate level of focus to a once-ignored aspect of technology.

For a broader discussion of industry and legal standards that affect IT environments, see this page.

Unfortunately, the regulations are filled with legalese and can be difficult to interpret. Most do not contain specifics on how to achieve their requirements. As a result, standards have been developed to help administrators reach the lofty legislative requirements. These standards are not regulation-specific, but following them usually ensures compliance. It can be intimidating to confront the requirements of all the various standards at once, so it's usually best to first work through one standard in its entirety.

ISO 27001:2013

The ISO/IEC 27001 (formerly ISO 17799) standard is probably the most widely accepted in the world. First introduced in 1995 as a British standard, it is 34 pages long and is divided into 11 sections that run the gamut from policy through physical security to access control. Objectives within each section define specific requirements, and controls under each objective describe the suggested best practice solutions. The document costs about $200.

The requirements are nontechnical and can be fulfilled by any organization in a way that best fits its needs. On the downside, the general wording of the standard leaves the reader with a sense of broad flexibility. Critics complain that the lack of specifics leaves organizations open to attack.

Nonetheless, this standard is one of the most valuable documents available to the information security industry. It bridges an often tangible gap between management and engineering and helps focus both parties on minimizing risk.

PCI DSS

The Payment Card Industry Data Security Standard (PCI DSS) is a different beast entirely. It arose from the perceived need to improve security in the credit card processing industry following a series of dramatic exposures. For example, in 2013, the U.S. government revealed the exposure of 160 million credit card numbers by various Visa licensees, including JCPenney. This is the largest cybercrime case in U.S. history; it's estimated that more than $300 million was lost.

The PCI DSS standard is the result of a joint effort between Visa and MasterCard, though it is currently maintained by Visa. Unlike ISO 27001, it is freely available for anyone to download. It focuses entirely on protecting cardholder data systems and has 12 sections that define requirements for protection.

Because PCI DSS is focused on card processors, it is not generally appropriate for businesses that don't deal with credit card data. However, for those that do, strict compliance is necessary to avoid hefty fines and possible criminal prosecution. You can find the document at pcisecuritystandards.org.

NIST 800 series

The fine folks at NIST have created the Special Publication (SP) 800 series of documents to report on their research, guidelines, and outreach efforts in computer security. These documents are most often used in connection with measuring FISMA (the Federal Information Security Management Act of 2002) compliance for those organizations that handle data for the U.S. federal government. More generally, they are publicly available standards with excellent content and have been widely adopted by industry.

The SP 800 series includes more than 100 documents. All of them are available from csrc.nist.gov/publications/PubsSPs.html. Table 27.3 lists a few that you might want to consider starting with.

Table 27.3: Recommended publications in the NIST SP 800 series

Pub	Title
800-12	An Introduction to Computer Security: The NIST Handbook
800-14	Generally Accepted Principles and Practices for Securing IT Systems
800-34 R1	Contingency Planning Guide for Information Technology Systems
800-39	Managing Risk from Information Systems: An Organizational Perspective
800-53 R4	Recommended Security Controls for Federal IT and Organizations
800-123	Guide to General Server Security

The Common Criteria

The Common Criteria for Information Technology Security Evaluation (commonly known as the "Common Criteria") is a standard for evaluating the security level of IT products. These guidelines were established by an international committee consisting of members from various manufacturers and industries. See commoncriteriaportal.org to learn more about the standard.

OWASP: the Open Web Application Security Project

OWASP is a nonprofit world-wide organization focused on improving the security of application software. It is best known for its "top 10" list of web application security risks, which helps

remind all of us where to focus our energies when securing applications. Find the current list and a bunch of other great material at owasp.org.

CIS: the Center for Internet Security

CIS has excellent resources for administrators. Perhaps the most valuable are the CIS benchmarks, a collection of technical configuration recommendations for securing operating systems. You can find benchmarks for each of our example UNIX and Linux systems. CIS also has benchmarks for cloud providers, mobile devices, desktop software, network devices, and more. Learn more at cisecurity.org.

27.11 Sources of security information

Half the battle of keeping your systems secure consists of staying abreast of security-related developments in the world at large. If your site is broken into, the break-in probably won't happen through the use of a novel technique. More likely, the chink in your armor will turn out to have been a known vulnerability that has been widely discussed in vendor knowledge bases, on security-related newsgroups, and on mailing lists.

SecurityFocus.com, the BugTraq mailing list, and the OSS mailing list

SecurityFocus.com specializes in security-related news and information. The news includes current articles on general issues and on specific problems. The site also includes an extensive technical library of useful papers, nicely sorted by topic.

SecurityFocus's archive of security tools contains software for a variety of operating systems along with blurbs and user ratings. It is the most comprehensive and detailed source of tools that we are aware of.

The BugTraq list is a moderated forum for the discussion of security vulnerabilities and their fixes. To subscribe, visit securityfocus.com/archive. Traffic on this list can be fairly heavy, however, and the signal-to-noise ratio is poor. A database of BugTraq vulnerability reports is also available from the web site.

The oss-security mailing list (openwall.com/lists/oss-security) is an excellent source of security tidbits from the open source community.

Schneier on Security

Bruce Schneier's blog is a valuable and sometimes entertaining source of information about computer security, cryptography, and squid. Schneier is the author of the well-respected books *Applied Cryptography* and *Secrets and Lies*, among others. Information from the blog is also captured in the form of a monthly newsletter known as the Crypto-Gram. Learn more at schneier.com/crypto-gram.html.

The Verizon Data Breach Investigations Report

Released annually, this report is packed with statistics about the causes and sources of data breaches, and it's an entertaining read to boot. The 2016 edition suggests, based on an analysis of 3,141 incidents, that around 80% of data breaches are financially motivated. Espionage comes in a distant second. This publication includes a useful breakdown of the types of attacks being seen in the wild.

The SANS Institute

The SANS (SysAdmin, Audit, Network, Security) Institute is a professional organization that sponsors security-related conferences and training programs, as well as publishing a variety of security information. Their web site, sans.org, is a useful resource that occupies something of a middle ground between SecurityFocus and CERT. It's neither as frenetic as the former nor as stodgy as the latter.

SANS offers several weekly and monthly email bulletins that you can sign up for on their web site. The weekly NewsBites are nourishing, but the monthly summaries seem to contain a lot of boilerplate. Neither is a great source of late-breaking security news.

Distribution-specific security resources

Because security problems have the potential to generate a lot of bad publicity, vendors are usually eager to help customers keep their systems secure. Most large vendors have an official mailing list to which security-related bulletins are posted, and many maintain a web site about security issues as well. It's common for security-related software patches to be distributed for free, even by vendors that normally charge for software support.

Security portals on the web, such as SecurityFocus.com, contain vendor-specific information and links to the latest official vendor dogma.

Ubuntu maintains a security mailing list at

https://lists.ubuntu.com/mailman/listinfo/ubuntu-security-announce

For Red Hat security information, subscribe to the "enterprise watch" list to get announcements about the security of Red Hat's product line. Find it at

https://redhat.com/mailman/listinfo/enterprise-watch-list

Although CentOS advisories typically (always?) mirror Red Hat security advisories, it's probably worthwhile to subscribe to the CentOS list at

https://lists.centos.org/pipermail/centos-announce/

FreeBSD has an active security group with a mailing list at

https://lists.freebsd.org/mailman/listinfo/freebsd-security

Other mailing lists and web sites

The contacts listed above are just a few of the many security resources available on the net. Given the volume of information that's now available and the rapidity with which resources come and go, we thought it would be most helpful to point you toward some metaresources.

One good starting point is linuxsecurity.com, which logs several posts each day on pertinent Linux security issues. It also maintains a running collection of Linux security advisories, upcoming events, and user groups.

(IN)SECURE magazine is a free bimonthly magazine that includes news about current security trends, product announcements, and interviews with notable security professionals. Some of the articles should be read with a vial of salt nearby, and always check the bylines. In many cases, authors are just pimping their own products.

The Linux Weekly News is a tasty treat that includes regular updates about the kernel, security, distributions, and other topics. LWN's security section can be found at lwn.net/security.

27.12 When your site has been attacked

The key to handling an attack is simple: don't panic. It's very likely that by the time you discover an intrusion, most of the damage has already been done. In fact, it has probably been going on for weeks or months. The chance that you've discovered a break-in that just happened an hour ago is slim to none.

In that light, the wise owl says to take a deep breath and begin developing a carefully thought out strategy for dealing with the break-in. You need to avoid tipping off the intruder by announcing the break-in or performing any other activity that would seem abnormal to someone who may have been watching your site's operations for many weeks. Hint: performing a system backup is usually a good idea at this point and (hopefully!) will appear to be a normal activity to the intruder. (If system backups are not a "normal" activity at your site, you have much bigger problems than the security intrusion.)

This is also a good time to remind yourself that some studies have shown that 60% of security incidents involve an insider. Be very careful with whom you discuss the incident until you're sure you have all the facts.

Here's a quick 9-step plan that may assist you in your time of crisis:

1. **Don't panic.** In many cases, a problem isn't noticed until hours or days after it took place. Another few hours or days won't affect the outcome. The difference between a panicky response and a rational response will. Many recovery situations are exacerbated by the destruction of important log, state, and tracking information during an initial panic.

2. **Decide on an appropriate level of response.** No one benefits from an overhyped security incident. Proceed calmly. Identify the staff and resources that must participate, and leave others to assist with the postmortem after it's all over.

3. **Hoard all available tracking information.** Check accounting files and logs. Try to determine where the original breach occurred. Back up all your systems. Make sure that you physically write-protect removable media if you connect them to a live system.

4. **Assess your degree of exposure.** Determine what crucial information (if any) has "left" the company, and devise an appropriate mitigation strategy. Determine the level of future risk.

5. **Pull the plug.** If necessary and appropriate, disconnect compromised machines from the network. Close known holes and stop the bleeding. CERT recommends steps for analyzing an intrusion. The document can be found at cert.org/tech_tips/win-UNIX-system_compromise.html.

6. **Devise a recovery plan.** With a creative colleague, draw up a recovery plan on nearby whiteboard. This procedure is most effective when performed away from a keyboard. Focus on putting out the fire and minimizing the damage. Avoid assigning blame or creating excitement. In your plan, don't forget to address the psychological fallout your user community may experience. Users inherently trust others, and blatant violations of trust make many folks uneasy.

7. **Communicate the recovery plan.** Educate users and management about the effects of the break-in, the potential for future problems, and your preliminary recovery strategy. Be open and honest. Security incidents are part of life in a modern networked environment. They are not a reflection on your ability as a system administrator or on anything else worth being embarrassed about. Openly admitting that you have a problem is 90% of the battle, as long as you can demonstrate that you have a plan to remedy the situation.

8. **Implement the recovery plan.** You know your systems and networks better than anyone. Follow your plan and your instincts. Speak with a colleague at a similar institution (preferably one who knows you well) to keep yourself on the right track.

9. **Report the incident to authorities.** If the incident involved outside parties, report the matter to CERT. They have a hotline at (412) 268-5800 and can be reached by email at cert@cert.org. Include as much information as you can.

A standard form is available from cert.org to help jog your memory. Here are some of the more useful pieces of information you might include:

- The names, hardware, and OS versions of the compromised machines
- The list of patches that had been applied at the time of the incident
- A list of accounts that are known to have been compromised
- The names and IP addresses of any remote hosts that were involved
- Contact information (if known) for the administrators of remote sites
- Relevant log entries or audit information

If you believe that a previously undocumented software problem may have been involved, report the incident to the software vendor as well.

27.13 Recommended reading

Dykstra, Josiah. *Essential Cybersecurity Science: Build, Test, and Evaluate Secure Systems*. Sebastopol, CA: O'Reilly Media, 2016.

Fraser, B., Editor. *RFC2196: Site Security Handbook*. rfc-editor.org, 1997.

Garfinkel, Simson, Gene Spafford, and Alan Schwartz. *Practical UNIX and Internet Security (3rd Edition)*. Sebastopol, CA: O'Reilly Media, 2003.

Kerby, Fred, et al. "SANS Intrusion Detection and Response FAQ." SANS. 2009. sans.org/resources/idfaq

Lyon, Gordon "Fyodor". *Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning*. Nmap Project, 2009. How to use **nmap**, from the author of **nmap**.

Ristić, Ivan. *Bulletproof SSL and TLS: Understanding and Deploying SSL/TLS and PKI to Secure Servers and Web Applications*. London, UK: Feisty Duck, 2014.

Schneier, Bruce. *Liars and Outliers: Enabling the Trust that Society Needs to Thrive*. New York, NY: Wiley, 2012.

Thompson, Ken. "Reflections on Trusting Trust." in *ACM Turing Award Lectures: The First Twenty Years 1966-1985*. Reading, MA: ACM Press (Addison-Wesley), 1987.

28 Monitoring



A commitment to monitoring is the distinguishing characteristic of a professional system administrator. Inexperienced sysadmins often leave systems unmonitored and allow failures to be "detected" when a frustrated, angry user calls the help desk because they're unable to complete an intended task. Slightly more clued-in administrative groups set up a monitoring platform but disable after-hours notifications because they are too bothersome. In either case, fire fighting and hilarity ensue. These approaches adversely affect the enterprise, complicate recovery efforts, and give the sysadmin team a bad reputation.

Professional sysadmins adopt monitoring as their religion. Every system is added to the monitoring platform before it goes live, and the battery of checks is regularly tested and tuned. Metrics and trends are evaluated proactively so that problems can be spotted before they affect users or put data at risk.

A major on-line video streaming service you may have heard of values their telemetry system so much that they'd rather have a service outage than a monitoring outage. Without monitoring, they'd have no idea what was happening anyway.

A monitoring-first philosophy (along with its associated tools) makes you a sysadmin superhero. You develop a better understanding of your software and applications, fix small problems before they snowball into catastrophic failures, and become more effective at finding error conditions, debugging problems, and understanding the performance of complex systems. Monitoring also

improves your quality of life by letting you fix most issues at your convenience rather than at 3:00 a.m. on Thanksgiving Day.

28.1 An overview of monitoring

The goals of monitoring are to ensure that the IT infrastructure as a whole operates as expected and to compile, in an accessible and easily digested form, data that are useful for management and planning. Simple, right? But this high-level description covers a potentially vast territory.

Real-world monitoring systems vary in every possible dimension, but they all share this same basic structure:

- Raw data is harvested from systems and devices of interest.
- The monitoring platform reviews the data and determines what actions are appropriate, usually by applying administratively set rules.
- The raw data and any actions decided on by the monitoring system flow through to back ends that take appropriate action.

Real-world monitoring systems range from trivially simple to arbitrarily complex. For example, the following Perl script includes all the elements listed above:

```
#!/usr/bin/env perl

$loadavg = (split /[\s,]+/, `uptime`)[10];

# If load is greater than 5, notify sysadmin
if ($loadavg > 5.0) {
  system 'mail -s "Server load is too high" dan@admin.com < /dev/null'
}
```

The script runs the **uptime** command to obtain the system's load averages. If the one-minute load average is larger than 5.0, it sends mail to an administrator. Data, evaluation, reaction.

Once upon a time, a "fancy" monitoring setup involved collections of scripts like this that ran from **cron** and commandeered a modem to send messages to sysadmins' pagers. Today, you have multiple options available at every stage of the monitoring pipeline.

Of course, you can still write individual monitoring scripts and run them from **cron**. If this is really all you need, by all means keep things simple. But unless you are responsible for only one or two servers, this ad hoc approach is normally not sufficient.

The following sections review the stages of the pipeline in a bit more detail.

Instrumentation

A wide range of data that may prove useful to your organization includes performance figures (response time, utilization, transfer rate), availability figures (reachability and uptime), capacity, state changes, log entries, and even business metrics such as average shopping cart value or click conversion rate.

Because anything one might do on a computer is potentially of monitoring interest, monitoring systems are usually data-source agnostic. They often come with built-in support for a variety of inputs. Even data sources that lack direct support can normally be brought in with a few lines of adapter code or a separate data gateway such as StatsD (see this page).

With so much data out there begging to be collected, the hard part of designing a collection system can be knowing what to ignore. Avoid collecting data that does not have a clear and actionable purpose. Data overcollection loads down both the monitoring system and the entities being monitored. It also tends to obscure the values that are truly important, drowning them in a sea of noise.

Unfortunately, it's often not easy to distinguish useful data from dross. You must continually reevaluate what is monitored and rethink how that data will be acted on throughout a system's life.

Data types

At the highest level, monitoring data can be grouped into three general categories:

- *Real-time metrics*, which characterize the operational state of the environment. These are typically numbers or Boolean values. In general, it's the responsibility of the monitoring system to test these metrics against expectations and generate an alert if a current value exceeds a predefined range or threshold.

- *Events*, which often take the form of log file entries or "push" notifications from subsystems. These events, sometimes known as pattern-based metrics, can indicate that a state change, alarm condition, or other action has occurred. Events can be processed to form numeric metrics (e.g., a total or a rate), or they can trigger monitoring responses directly.

  Many of the data points collected by application monitoring software fall into the "event" category; sometimes they have quantitative data attached as well. Interrelationships among events (e.g., "the user looked at the Settings page but then canceled without changing anything") are often helpful to investigate. General-purpose monitoring platforms tend not to be very good at this sort of cross-referencing, which is one reason that application monitoring is a category of its own.

- *Aggregated and summarized historic trends*, which are often time-series collections of real-time metrics. They allow for analysis and visualization of changes over time.

Intake and processing

Most monitoring systems revolve around a central monitoring platform that absorbs data from monitored systems, performs appropriate processing, and applies administrative rules to determine what should happen in response.

First-generation platforms such as Nagios and Icinga focused on detecting and responding to problems as they occurred. These systems were revolutionary for their day and led us into the modern world of monitoring. Nevertheless, they have been eclipsed over time by the industry's gradual realization that all monitoring data is time-series data. If values didn't vary, you wouldn't be monitoring them.

Clearly, a more data-oriented approach was needed. However, monitoring data is usually so voluminous that you can't simply dump it all into a traditional database and allow it to accumulate. That's a recipe for poor performance and overflowing disks.

The modern approach is to organize monitoring around a data store that's specialized for handling time-series data. All data is stored for an initial period, but as the data ages, the store applies increasingly high levels of summarization to limit storage requirements. For example, the store might keep an hour's worth of data at one-second resolution, a week's worth of data at one-minute resolution, and a year's worth of data at one-hour resolution.

Historical data is useful not only for dashboard presentations, but also as a baseline for comparison. Is the current network error rate 25% or more above its historical average?

Notifications

Once you have a monitoring framework in place, put careful thought into what to do with the monitoring results. The first priority is usually to notify administrators and developers about a problem that needs attention.

Notifications must be actionable. Structure your monitoring system so that everyone who receives a given notification must potentially do something in response, even if the action is something as general as "check later to be sure this was taken care of." Notifications that are purely informational train staff to ignore notifications.

In most cases, notifications need to extend beyond email to be optimally effective. For critical issues, SMS notifications (that is, text messages) to administrators' cell phones are easy and efficient. Recipients can set their ring tones and phone volume so that they'll be awakened in the middle of the night if desired.

See this page for more comments on ChatOps.

Notifications should also be integrated with your team's ChatOps implementation. Less critical notifications (such as job statuses, login failures, and informational notices) can be sent to one or more chat rooms so that interested parties can actively receive subsets of alerts in which they might be interested.

Beyond these basic channels, the possibilities are endless. An LED lighting system that changes colors according to system status can be useful for at-a-glance status indication in a data center or network operations center, for example. Other options for responding to situations identified by the monitoring systems include

- Automated actions, such as dumping a database or rotating logs
- Calling an administrator on the phone
- Sending data to a wall board for public display
- Storing data in a time-series database for later analysis
- Doing nothing, and allowing later review through the system itself

Dashboards and UIs

Beyond alerting for clearly exceptional circumstances, one of the main goals of monitoring is to present the state of the environment in a manner that's more structured and easier to assimilate than a bunch of raw data. Such displays are generically termed "dashboards."

Dashboards are designed by administrators or by other stakeholders with an interest in particular aspects of the environment. They use several different techniques to transform raw data into infographic gold.

First, they're selective in what they present. They concentrate on the most important metrics for a given domain, the ones that indicate general states of health or performance. Second, they give context clues to the significance and import of the data that's shown. For example, problematic numbers and states are typically shown in red, and primary metrics are depicted in larger font sizes. Relationships among values are shown through grouping. Third, dashboards display data series as charts, making them easy to assess at a glance.

Of course, most data that's collected never shows up on a dashboard. It's helpful if your monitoring system also has a generalized UI that facilitates investigation and modification of the data schema, allows you to make arbitrary database queries, and charts arbitrarily defined sequences of data on the fly.

28.2 THE MONITORING CULTURE

This chapter is mostly about tools, but culture is at least as important. When you embark on a monitoring journey, embrace the following tenets:

- If someone cares about or depends on a system or service, it must be monitored. Full stop. Nothing in the environment that a service or user depends on can remain unmonitored.
- If a production device, system, or service exposes monitorable attributes, those attributes should be monitored. Don't let a server with a fancy "lights out" hardware management interface spend weeks futilely trying to notify you that a fan has failed.
- All high-availability constructs must be monitored. It would be unfortunate to learn that a primary server had failed only after the backup server failed, too.
- Monitoring is not optional. The work plans of every sysadmin, developer, ops staff member, manager, and project manager should include provisions for monitoring.
- Monitoring data (especially historical data) is useful to everyone. Make data easily accessible and visible so that everyone can use it to help with root cause analysis, planning, life cycle management, and architectural improvement opportunities. Put effort and resources into creating and promoting monitoring dashboards.
- Everyone should respond to alerts. Monitoring is not just an ops problem. All technical roles should receive notifications and work together to resolve issues. This approach encourages bona fide root cause analysis by whichever individuals are most suited to fix the underlying issue.
- Properly implemented, monitoring impacts quality of life in a positive way. A solid monitoring regimen frees you from the burden of worrying about what state your systems are in and empowers others to support you. Without monitoring and appropriate documentation, you are essentially on call 24 × 7 × 365.
- Train responders to fix alerts, not just suppress them. Evaluate false-positive or noisy alerts and tune them so that they no longer trigger inappropriately. Spurious alerts encourage everyone to ignore the monitoring system.

28.3 THE MONITORING PLATFORMS

If you plan to monitor multiple systems and more than a few metrics, it's worth investing some time into the deployment of a full-service monitoring platform. These are general-purpose systems that collect data from multiple sources, facilitate the display and summarization of status information, and establish a standard way to define actions and alerts.

The good news is that there are a variety of choices. The not-as-good news is that no single, perfect platform as yet exists. When selecting from among the available options, consider the following issues:

- *Data-gathering flexibility.* All platforms can absorb data from a variety of sources. However, that doesn't mean that all platforms are equivalent in this regard. Consider the data sources you want to actually use. Will you need to read data from an SQL database? From DNS records? From an HTTP connection?

- *User interface quality.* Many systems offer customizable GUIs or web interfaces. Most well-marketed packages today tout their ability to understand JSON templates for data presentation. A UI is not just marketing hype; you need an interface that relays information clearly, simply, and comprehensibly. Will you need different user interfaces for different groups within your organization?

- *Cost.* Some commercial management packages come at a stiff price. Many corporations find value in being able to say that their site is managed by a high-end commercial system. If that isn't so important to your organization, look at free options such as Zabbix, Sensu, Cacti, and Icinga.

- *Automated discovery.* Many systems offer to "discover" your network. Through a combination of broadcast pings, SNMP requests, ARP table lookups, and DNS queries, they identify all your local hosts and devices. All the discovery implementations we have seen work pretty well, but accuracy is lower on complex or heavily firewalled networks.

- *Reporting features.* Many products can send alert email, integrate with ChatOps, send text messages, and automatically generate tickets for popular trouble-tracking systems. Make sure that the platform you choose accommodates flexible reporting. Who knows what electronic devices you'll be dealing with in a few years?

Open source real-time platforms

Although the platforms in this section—Nagios, Icinga, and Sensu Core—do a little bit of everything, they're known for their strength in handling instantaneous (or threshold-based) metrics.

These systems have their proponents, but as first-generation monitoring tools, they're gradually losing favor to time-series systems, which we describe starting here. Most sites starting from scratch would be better advised to opt for a time-series system.

Nagios and Icinga

Nagios and Icinga specialize in real-time notification of error conditions. Although they do not help you determine how much your bandwidth utilization has increased over the last month, they can track you down when your web server goes off-line.

Nagios and Icinga were originally forks of a single source tree, but modern-day Icinga 2 has been completely rewritten. However, it remains compatible with Nagios in most respects.

Both systems include scores of scripts for monitoring services of all shapes and sizes, along with extensive SNMP monitoring capabilities. Perhaps their greatest strength is their modular and heavily customizable configuration system, which allows you to write custom scripts to monitor any conceivable metric.

You can whip up new monitors in Perl, PHP, Python, or even C if you're feeling ambitious and masochistic. Many standard notification methods are built in—email, web reports, text messages, etc. And as with monitoring plug-ins, it's easy to roll your own notification and action scripts.

Nagios and Icinga both work well for networks of fewer than a thousand hosts and devices. They are easy to customize and extend, and include powerful features such as redundancy, remote monitoring, and notification escalation.

If you are deploying new monitoring infrastructure from scratch, we recommend Icinga 2 over Nagios. Its code base is generally cleaner, and it has been rapidly accreting fans and community support. From a functional perspective, its UI is cleaner and faster, and it's able to autobuild service dependencies, which can be essential in complex environments.

Sensu

Sensu is a full-stack monitoring framework that's available both as an open source edition (Sensu Core) and with paid, commercially supported add-ons. It has an ultramodern UI and can run any legacy Nagios, Icinga, or Zabbix monitoring plug-in. It was designed as a replacement for Nagios, so plug-in compatibility is one of its most attractive features. Sensu allows for easy integration with Logstash and Slack notifications, and its installation process is particularly easy.

Open source time-series platforms

Detecting and responding to current problems is just one aspect of monitoring. It's often equally important to know how values are changing over time and how they relate to other values. Four popular time-series platforms aim to scratch this itch: Graphite, Prometheus, InfluxDB, and Munin.

These systems put the database front and center within the monitoring ecosystem. They vary in their degree of completeness as stand-alone monitoring systems, and in general are designed for a more modular world than traditional systems such as Icinga. You may need to supply some additional components to build a complete monitoring platform.

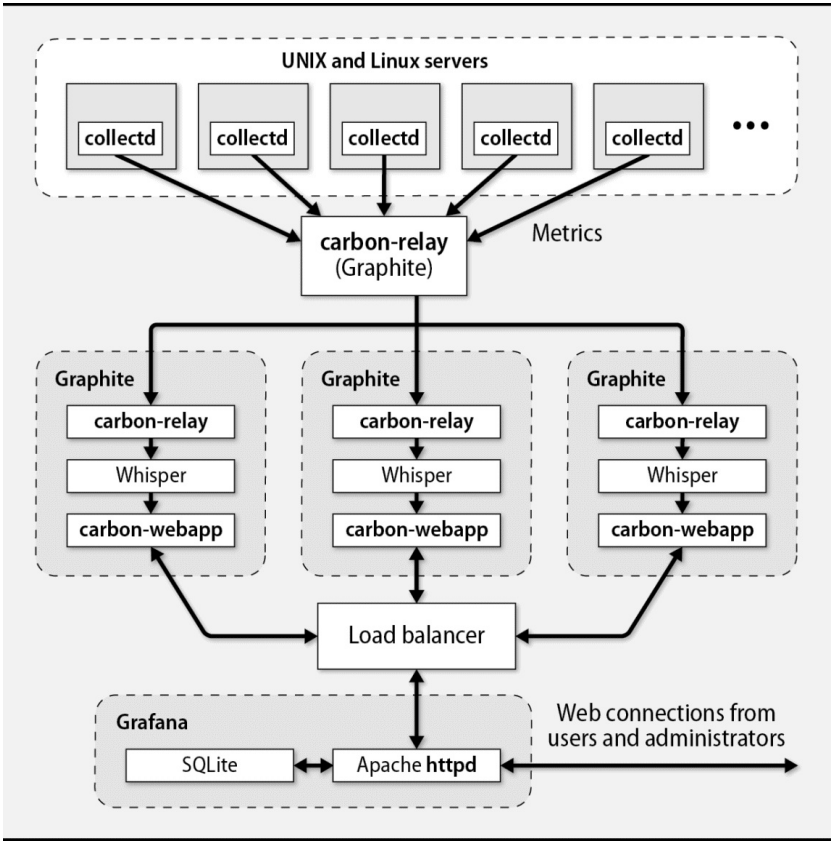
Graphite

Graphite was arguably the vanguard of the new generation of time-series monitoring platforms. At its core is a flexible time-series database with an easy-to-use query language. The reason for the #monitoringlove movement and for the enormous influence that Graphite has had on front-end UIs is the way that it aggregates and summarizes metrics. It started the move away from per-minute monitoring and toward sub-second monitoring.

As you might guess from the name, Graphite includes graphing features for web visualization. However, this aspect of the package has been somewhat eclipsed by Grafana. Graphite is better known these days for its other components, Carbon and Whisper, which form the core of the data management system.

Graphite can be combined with other tools to create a scalable, distributed, clustered, monitoring environment that is capable of absorbing and reporting on hundreds of thousands of metrics. Exhibit A shows an architectural diagram of such an implementation.

Exhibit A: Clustered Graphite architecture



Prometheus

Our favorite time-series platform today is Prometheus. It's a comprehensive platform that includes integrated collection, trending, and alerting components. The components are both sysadmin- and developer-friendly, which makes it a great choice for a DevOps shop. It does not allow for clustering, however, which may mean that it's not the right fit for sites that require high availability.

InfluxDB

InfluxDB is an extraordinarily developer-friendly time-series monitoring platform that supports a broad array of programming languages. Much like Graphite, InfluxDB is really just a time-series database engine. You'll need to complete the package with external components such as Grafana to form a complete monitoring system that includes features like alerting.

The data management features of InfluxDB are much richer than those of the alternatives listed above. However, InfluxDB's additional features also add some unwelcome complexity for typical installations.

InfluxDB has had a somewhat troubled history of bugs and incompatibilities. However, the current version appears to be stable and is probably the best current alternative to Graphite if you are seeking a stand-alone data management system.

Munin

Munin has historically been quite popular, especially in Scandinavia. It's built on a clever architecture in which the data collection plug-ins not only provide data but also tell the system how the data should be presented. Although Munin is still perfectly usable, modern alternatives such as Prometheus should be considered for new deployments. Munin is still a useful tool for application-specific monitoring in some cases; see this page.

Open source charting platforms

The two main choices for creating dashboards and charts are the graphing features built into Graphite and a newer package, Grafana.

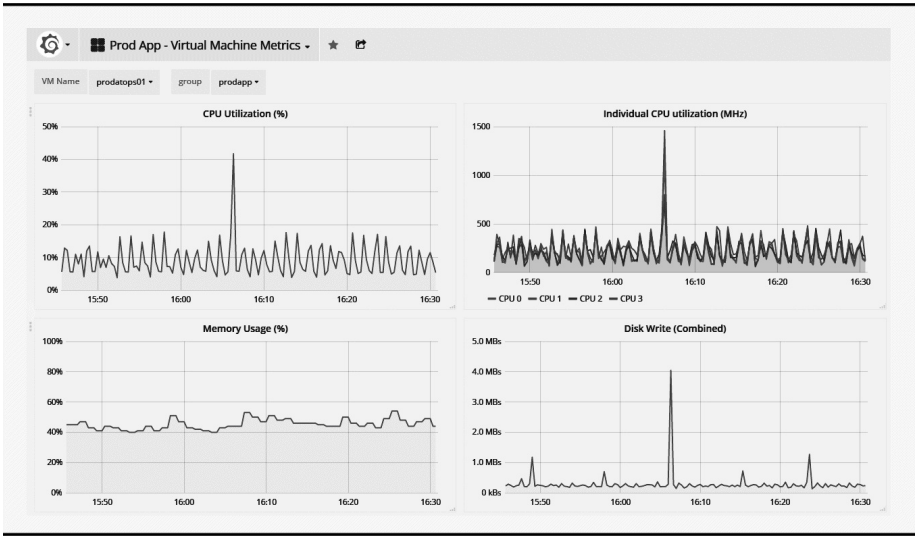
Graphite can draw data from stores other than Whisper (the native data-storage component of the Graphite package), but this isn't necessarily a well-trodden path.

As a database-agnostic package, Grafana does quite well at accommodating foreign data stores, including all those listed in the previous section. At last count, more than 30 back ends were supported. Grafana originally started out as an attempt to improve graphing for Graphite, so it's quite comfortable in a Graphite environment, too.

Both Graphite and Grafana present a dashboard-like graphing interface that can generate insight-provoking and management-pleasing visualizations. You can use them to display anything from low-level system metrics to business-level indicators. Bake-offs usually give the nod to Grafana for its superior UI and prettier graphs.

Exhibit B shows a simple Grafana dashboard.

Exhibit B: Grafana dashboard example



Commercial monitoring platforms

Hundreds of companies sell monitoring software, and new competitors enter the market every week. If you are looking for a commercial solution, you should at least consider the options listed in Table 28.1.

Table 28.1: Popular commercial monitoring platforms

Platform	URL	Comments
Datadog	datadoghq.com	Cloud-based application monitoring platform Huge list of supported systems, apps, and services
Librato	librato.com	Plug and Play with existing open source plugins
Monitus	monitus.net	E-commerce platform monitoring
Pingdom	pingdom.com	SaaS-based monitoring platform [a]
SignalFx	signalfx.com	SaaS platform with long list of cloud integrations
SolarWinds	solarwinds.com	Network monitoring stalwart
Sysdig Cloud	sysdig.com	Specialty: Docker monitoring and alerting Easy to correlate events across services
Zenoss	zenoss.com	Incredibly complex alternative to Icinga

a. No software install required. Good fit for web apps only.

Whether their systems reside in the cloud, on a data center hypervisor, or in a closet, most businesses should not be building their own monitoring stack. Outsourcing is cheaper and more reliable. Hence, consider Datadog, Librato, SignalFx, or Sysdig Cloud if you need a monitoring stack for a common set of applications or servers.

When investigating commercial monitoring platforms, you often first consider price. But don't forget to research the operational details as well:

- How easy is it to integrate into your configuration management system?
- How does the system deploy new plug-ins or checks to your hosts? Are they pushed or pulled?
- Does it integrate well with your existing notification platform if you have one?
- Does your environment allow the type of external connectivity needed to facilitate a cloud-based monitoring solution?

These are just a few of the questions you should be asking when researching platforms. In the end, the best platform for your site is one that is easily configurable, meets your budget, and is easily adopted by your users.

Hosted monitoring platforms

If you're not interested in setting up and maintaining your own network monitoring tools, you might want to consider a hosted (cloud) solution. Many free and commercial options exist, but a popular one is StatusCake, statuscake.com. An external provider's ability to see the internal details of your network is limited, but hosted options work well for validating the health of public-facing services and web sites.

A hosted monitoring provider can also liberate you from the constraints of your organization's normal Internet connection. If you rely on your upstream network to transport notifications from an internal monitoring system—as most sites ultimately do—you may want to ensure that your upstream network is itself monitored and instrumented so that staff can be marshaled in the event of trouble.

28.4 Data collection

The previous sections reviewed a variety of packages that can serve as a site's central monitoring engine. However, selecting and deploying one of these systems is only the first part of the setup process. You must now make sure that the data and events you're interested in monitoring make their way into the central monitoring platform.

The details of this instrumentation process depend on the systems you want to monitor and the philosophy of your monitoring platform. In many cases, you'll need to write some simple glue scripts to convert status information into a form that your monitoring platform can understand. Some platforms, such as Icinga, come with a wide variety of plug-ins that harvest standard metrics from commonly monitored systems. Others, such as Graphite and InfluxDB, make no real provision for data input at all and must be supplemented by a front end that handles this role.

In the following sections, we first look at StatsD, a general-purpose data collection front end, then review some tools and techniques for instrumenting some commonly monitored systems.

StatsD: generic data submission protocol

StatsD was written by engineers at Etsy as a way to track anything and everything within their own environment. It's a UDP-based front-end proxy that dumps any data you throw at it into a monitoring platform for consumption, calculation, and display. StatsD's superpower is its ability to ingest and perform calculations on arbitrary statistics.

Etsy's StatsD daemon was written in Node.js. But these days, "StatsD" refers more to the protocol than to any one of the many software packages that implement it. (Truth be told, even Etsy's version is not the original; it was inspired by a similarly named project at Flickr.) Implementations have been written in many different languages, but we focus on the Etsy release here.

StatsD depends on Node.js, so make sure that package has been installed and configured appropriately before you move on to installing StatsD. The Etsy implementation isn't included in most OS vendors' package repositories, though other versions of StatsD often are; make sure you don't confuse them. It's easiest to clone the Etsy version directly from GitHub:

```
$ git clone https://github.com/etsy/statsd
```

StatsD is incredibly modular and can feed the incoming data to a variety of back ends and clients. Let's look at a simple example that uses Graphite as the back end.

To ensure that Graphite and StatsD communicate correctly, you must modify Carbon, Graphite's storage component. Edit **/etc/carbon/storage-schemas.conf** and add a stanza similar to the following:

```
[stats]
pattern = ^stats.*
retentions = 10s:12h,1min:7d,10min:1y
```

This configuration tells Carbon to keep 12 hours of data at 10-second intervals. Carbon summarizes expiring data at 1-minute intervals and keeps that summary information for an additional 7 days. Similarly, data at 10-minute granularity is maintained for a full year. There's nothing magic about these choices; you'll need to determine what's appropriate for your organization's retention needs and the data being collected.

The exact definition of what it means to "summarize" time-series data varies according to the type of data. If you're counting network errors, for example, you probably want to summarize by adding up values. If you're looking at metrics that represent load or utilization, you probably want an average. You might also need to specify appropriate ways of handling missing data.

These policies are specified in the file **/etc/carbon/storage-aggregation.conf**. If you don't already have a working Graphite installation, you might find Graphite's sample configuration useful as a starting point:

/usr/share/doc/graphite-carbon/examples/storage-aggregation.conf.example

Below are some reasonable defaults to include in the **storage-aggregation.conf** file.

```
[min]
pattern = \.lower$
xFilesFactor = 0.1
aggregationMethod = min

[max]
pattern = \.upper(_\d+)?$
xFilesFactor = 0.1
aggregationMethod = max

[sum]
pattern = \.sum$
xFilesFactor = 0
aggregationMethod = sum


[count]
pattern = \.count$
xFilesFactor = 0
aggregationMethod = sum

[count_legacy]
pattern = ^stats_counts.*
xFilesFactor = 0
aggregationMethod = sum

[default_average]
pattern = .*
xFilesFactor = 0.3
aggregationMethod = average
```

Note that every configuration block has a regular expression pattern that attempts to match the names of data series. Blocks are read in order, and the first matching block becomes the controlling specification for each data series. For example, a series named sample.count would match the pattern for the [count] block. The values would be rolled up by adding up data points (aggregationMethod = sum).

The xFilesFactor setting determines the minimum number of samples needed to meaningfully downsample your metric. It's expressed as a number between 0 and 1 that represents the percentage of non-null values that must exist at the more granular layer in order for the rollup layer to have a non-null value. For example, the xFilesFactor setting for [min] and [max] above is 10%, so even a single data value will satisfy this criterion, given our settings in the **storage-schema.conf** file. The default is 50%. If the numbers aren't thoughtfully set, you'll get inaccurate or missing data!

We can send some test data to StatsD with netcat (**nc**):

```
$ echo "sample.count:1|c" | nc -u -w0 statsd.admin.com 8125
```

This command submits a value of 1 as a count metric (as indicated by the **c**) to the sample.count data set. The packet goes to port 8125 on statsd.admin.com; this is the port on which **statsd** listens by default. If this datum shows up in your Graphite dashboard, you're ready to collect all kinds of monitoring data through one of the many StatsD clients. See the StatsD GitHub wiki page (github.com/etsy/statsd/wiki) for a list of some of the clients that can communicate with StatsD. Or write your own! The protocol is simple and the possibilities are endless.

Data harvesting from command output

If you can investigate something from the command line, you can track it in your monitoring platform. All you need is a few lines of scripting glue to extract the data nuggets you're interested in, which you then massage into a format your monitoring platform can accept.

For example, **uptime** shows the length of time that the system has been up, the number of logged-in users, and the load averages over the last 1, 5, and 15 minutes.

```
$ uptime
07:11:50 up 22 days, 10:13,  2 users,  load average: 1.20, 1.41, 1.88
```

As a human, you can parse the output at a glance and see that the current load average is 1.20. If you want to write a script to check that value regularly or to feed it to another monitoring process, you can use text manipulation commands to isolate the desired value:

```
$ uptime | perl -anF'[\s,]+' -e 'print $F[10]'
1.20
```

Here, we use Perl to split the output wherever there's a sequence of spaces and commas and to print the contents of the tenth field (the 1-minute load average). Voilà!

Although Perl has been eclipsed by modern languages like Python and Ruby in most domains, it remains the king of quick-and-dirty text wrangling. It's probably not worth learning Perl solely for this use, but Perl's ability to phrase sophisticated text transformations as one-line commands does come in handy.

We can easily expand this one-liner into a short script that determines the load average and submits it to StatsD:

```
#!/usr/bin/env perl

use Net::Statsd;
use Sys::Hostname;

$Net::Statsd::HOST = 'statsd.admin.com';

$loadavg = (split /[\s,]+/, `uptime`)[10];
Net::Statsd::gauge(hostname . '.loadAverage' => $loadavg);
```

Compare this script with our one-line StatsD test command from this page and our one-line parsing of **uptime** output above. Here, Perl has to run the **uptime** command and process its output as a string, so that portion looks somewhat different from its one-liner equivalent. (The one-liner relies on Perl's autosplit mode.)

Instead of using **nc** to handle the network transmission of data to StatsD, we use a simple StatsD wrapper that we downloaded from the Comprehensive Perl Archive Network at cpan.org. That's

generally the preferred approach; libraries are less brittle than hacks, and they clarify the code's intent.

Many commands can generate more than one output format. Check the man page for the command to see what options are available before you attempt to parse its output. Some formats are much easier to deal with than others.

A few commands support an output format that specifically facilitates parsing. Others have configurable output systems in which you ask for only the fields that you really want. Yet another common option is a flag that suppresses descriptive header lines in the output.

28.5 NETWORK MONITORING

Network status monitoring has traditionally been many sites' first foray into the wider world of monitoring and dashboards, so it's the first of several types of monitoring that we look at in a bit more depth. In subsequent sections, we also look at OS-level monitoring, application and service monitoring, and security monitoring.

The basic unit of network monitoring is the network ping, also known as an ICMP Echo Request packet. We discuss the technical details more thoroughly starting here, along with the **ping** and **ping6** commands, which initiate pings from the command line.

The concept is simple: you send an echo request packet to another host on the network, and that host's IP implementation returns a packet to you in response. If you receive a response to your probe, you know that all the network gateways and devices that lie between you and the target host are operational. You also know that the target host is powered on and that its kernel is up and running. However, because pings are handled within the TCP/IP protocol stack, they don't guarantee anything about the state of higher-level software that might be running on the target host.

Pings don't impose much overhead on the network, so it's OK to send them frequently; say, every ten seconds. Design your pinging strategy thoughtfully, so that it covers all important gateways and networks. Keep in mind that if a ping can't get through a gateway, neither can monitoring data that reports the failure of pings. You'll want at least one set of pings to originate on the central monitoring host itself.

Network gateways aren't required to answer ping packets, so pings might be dropped by a busy gateway. Even a properly functioning network loses a packet now and then. Ergo, don't set off alarms at the first sign of trouble. It makes sense to collect ping data as binary event records (got through/didn't get through) and roll it up into aggregate measures of percentage packet loss over longer terms.

You might also find it interesting to measure throughput between two points on the network. That can be done with iPerf; see this page for details.

Most network devices support the Simple Network Management Protocol (SNMP), an industry-standard way of naming and collecting operational data. Although SNMP has metastasized far beyond its networking roots, we consider it obsolete for purposes other than basic network monitoring.

SNMP is a rather large topic of its own, so we defer further discussion of it until later in this chapter. See *SNMP: the Simple Network Management Protocol* for details.

28.6 SYSTEMS MONITORING

Since the kernel controls a system's CPU, memory, I/O, and devices, most of the interesting system-level state information you might want to monitor lives somewhere inside the kernel. Whether you're investigating a particular system by hand or setting up an automated monitoring platform, you need the right tools to extract and expose this state information. Most kernels define formal channels through which such information is exported.

Unfortunately, kernels are like other types of software; error checking, instrumentation, and debugging features are often something of an afterthought. Although recent years have brought improvements in transparency, identifying and understanding the exact parameter you might want to monitor can be challenging and sometimes impossible.

---

*See this page for more information about the **/proc** filesystem.*

A particular value can often be obtained in more than one way. In the case of load averages, for example, you can read the values directly from **/proc/loadavg** on Linux systems or with **sysctl -n vm.loadavg** on FreeBSD. Load averages are also included in the output of the **uptime**, **w**, **sar**, and **top** commands (though **top** would be a poor choice for noninteractive use). It's generally easiest and most efficient to access values directly from the kernel (through **sysctl** or **/proc**) if you can.

Monitoring platforms such as Nagios and Icinga include a rich set of community-developed monitoring plug-ins that you can use to get your hands on commonly monitored elements. They, too, are often simply scripts that run commands and parse the resulting output, but they come already tested and debugged, and they often work on multiple platforms. If you can't find a plug-in that yields the value you're interested in, you can write your own.

Commands for systems monitoring

Table 28.2 lists some command that are commonly used in monitoring. Many of these commands yield wildly different output depending on the command-line options you supply, so check the man pages for details.

Table 28.2: Commands that yield commonly monitored parameters

Command	Available information
df	Free and used disk space and inodes
du	Directory sizes
free	Free, used, and swap (virtual) memory
iostat	Disk performance and throughput
lsof	Open files and network ports
mpstat	Per-processor utilization on multiprocessor systems
vmstat	Process, CPU, and memory statistics

The Swiss Army knife of command-line data extraction is **sar** (short for "system activity report"). This command has a sordid history, having originally been introduced in System V UNIX in the 1980s. Old-school sysadmins are often identifiable by their fluency in **sar**.

The primary attraction of this command is that it has been implemented on a wide variety of systems, so it enhances the portability of both scripts and sysadmins. Sadly, the BSD port is no longer maintained.

The example below requests reports every two seconds for a period of one minute (i.e., 30 reports). The **DEV** argument is a literal keyword, not a placeholder for a device or interface name.

```
$ sar -n DEV 2 30
17:50:43  IFACE rxpck/s txpck/s rxbyt/s txbyt/s rxcmp/s txcmp/s rxmcst/s
17:50:45  lo       3.61    3.61  263.40  263.40    0.00    0.00     0.00
17:50:45  eth0    18.56   11.86 1364.43 1494.33    0.00    0.00     0.52
17:50:45  eth1     0.00    0.00    0.00    0.00    0.00    0.00     0.00
```

This example is from a Linux machine with two network interfaces. The output includes both instantaneous and average readings of interface utilization in units of both bytes and packets. The second interface (eth1) is clearly not in use.

collectd: generalized system data harvester

As the work of system administration has evolved from wrangling individual systems to managing fleets of virtualized instances, simple command-line tools have started to create a lot of friction in the monitoring world. Although writing scripts to collect and analyze parameters is a utilitarian and flexible approach, maintaining the consistency of that code base across multiple systems quickly becomes cumbersome. Modern tools such as **collectd**, **sysdig**, and **dtrace** offer a more scalable approach to collecting this type of data.

Collection of system statistics should be a continuous process, and the UNIX solution to an ongoing task is to create a daemon to handle it. Enter **collectd**, the system statistics collection daemon.

This popular and mature tool runs on both Linux and FreeBSD. Typically, **collectd** runs on the local system, collects metrics at specified intervals, and stores the resulting values. You can also configure **collectd** to run in client/server mode, where one or more **collectd** instances aggregate data from a group of other servers.

Specification of the metrics to be collected and the destinations to which they are saved is flexible; over 100 plug-ins are available to suit your exact needs. Once **collectd** is running, it can be queried by a platform such as Icinga or Nagios for instantaneous monitoring or can forward data to a platform such as Graphite or InfluxDB for time-series analysis.

An example **collectd** configuration file is shown below.

```
## /etc/collectd/collectd.conf

Hostname client1.admin.com
FQDNLookup false
Interval 30
LoadPlugin syslog
<Plugin syslog>
    LogLevel info
</Plugin>

LoadPlugin cpu
LoadPlugin df
LoadPlugin disk
LoadPlugin interface
LoadPlugin load
LoadPlugin memory
LoadPlugin processes
LoadPlugin rrdtool

<Plugin rrdtool>
    DataDir "/var/lib/collectd/rrd"
</Plugin>
```

This basic configuration collects a variety of interesting system statistics every 30 seconds and writes RRDtool-compatible data files in **/var/lib/collectd/rrd**.

sysdig and dtrace: execution tracers

sysdig (Linux) and **dtrace** (BSD) comprehensively instrument both kernel and user process activity. They include components that are inserted into the kernel itself, exposing not only deep kernel parameters but also per-process system calls and other performance statistics. These tools are sometimes described as “Wireshark for the kernel and processes.”

See this page for more information about Wireshark.

Both of these tools are complex. However, they are well worth tackling. A weekend spent learning either one will give you amazing new superpowers and ensure your status as an A-list guest on the sysadmin cocktail circuit.

See Chapter 25 for more information about Docker and containers.

---

sysdig is container-aware, so it confers extraordinary visibility into environments where tools such as Docker and LXC are in use. **sysdig** is distributed as open source, and you can integrate it with other monitoring tools such as Nagios or Icinga. The developers also offer a commercial monitoring service (Sysdig Cloud) that has full monitoring and alerting capability.

28.7 Application monitoring

At the top of the software ziggurat, we find the holy grail: application monitoring. This type of monitoring is rather vaguely defined, but the general idea is that it attempts to validate the status and performance of particular pieces of software rather than systems or networks as a whole. In many cases, application monitoring can reach into those systems and profile their internal operations.

To make sure you're monitoring the right things, you need business units and developers to join the party and tell you more about their interests and concerns. If you have a web site that runs on the LAMP stack, for example, you'll probably want to make sure you're monitoring page load times, flagging critical PHP errors, keeping tabs on the MySQL database, and monitoring for specific issues such as excessive connection attempts.

Although monitoring for this layer can be complex, this domain is also where monitoring gets sexy. Imagine monitoring (and pulling into your beautiful Grafana dashboard) the number of widgets you've sold in the past hour or the average length of time that an item stays in a shopping cart. If you show your application developers and process owners that level of functionality, you usually get immediate buy-in to add more monitoring and may even get some help implementing it. Eventually, this layer of monitoring becomes invaluable to the business, and you start to be viewed as the champion of monitoring, metrics, and data analysis.

Application-level monitoring can yield additional insight into other events within your environment. For example, if widget sales decrease quickly, that might be an indication that one of your advertisement networks is down.

Log monitoring

In its most basic form, log monitoring involves grepping through log files to find interesting data you'd like to monitor, pulling out that data, and processing it into a form that's usable for analysis, display, and alerting. Since log messages consist of free-form text, implementation of this pipeline can range in complexity from trivial to challenging.

Logs are typically best managed with a comprehensive aggregation system designed for that purpose. We address such systems in the section *Management of logs at scale*. Although these systems focus primarily on centralizing log data and making it easy to search and review, most aggregation systems also support threshold, alarm, and reporting functionality.

If you need automated log review for a few specific purposes and are reluctant to commit to a more general log management solution, we recommend a couple of smaller-scale tools: **logwatch** and OSSEC.

logwatch is a flexible, batch-oriented log summarizer. Its primary use is to create daily summaries of events reported in the logs. You can run **logwatch** more often than once a day, but it isn't designed for real-time monitoring. For that, you would probably want to take a look at OSSEC, which we discuss here. OSSEC is promoted as a security tool, but its architecture is general enough that it's useful for other kinds of monitoring as well.

Supervisor + Munin: a simple option for limited domains

An all-singing, all-dancing platform such as Icinga or Prometheus might be overkill for your needs or your environment. What if you're only interested in monitoring one particular application process and don't want the headache of a full-fledged monitoring platform? Consider combining Munin and Supervisor. They're easy to install, require little configuration, and work well together.

Supervisor and its server process **supervisord** help you monitor processes and generate events or notifications when the processes exit or throw an exception. The system is similar in spirit to Upstart or to the process-management portions of **systemd**.

As mentioned on this page, Munin is a general monitoring platform with particular strengths in application monitoring. It's written in Perl and requires an agent known as a Munin Node to be running on all the systems you want to monitor. Setting up a new Node is easy: just install the **munin-node** package, edit **munin-node.conf** to point to the master machine, and you're good to go.

Munin defaults to creating RRDtool graphs with the data that it collects, so it's a nice way to get some graphical feedback without much configuration. More than 300 plug-ins are distributed with Munin, and nearly 200 others are available as contributed libraries. It's likely that you can find an existing plug-in that meets your needs. If not, it's easy to write a new script for **munin-node** to execute.

Commercial application monitoring tools

If you search Google for “application monitoring tool,” you’ll discover many pages of commercial offerings to evaluate. For gold-star due diligence, you’ll also need to scrub through layers of recent discussions regarding application performance monitoring (APM).

See this page for more information about DevOps.

You’ll see many references to DevOps in these venues, and for good reason: application monitoring and APM are key tenets of DevOps. They supply quantitative metrics that teams can use to decide which areas of the stack would most benefit from efforts to improve performance and stability.

We think New Relic (newrelic.com) and AppDynamics (appdynamics.com) are standouts in this area. These systems’ capabilities overlap in many ways, but AppDynamics typically targets more a “full stack” monitoring solution, whereas New Relic deals more with profiling behavior inside the application layer itself.

Regardless of how you monitor your applications, it’s crucial to keep the development team involved in the process. They’ll help ensure that all important metrics are being monitored. Close cooperation on monitoring fosters the relationship between teams and limits duplication of effort.

28.8 Security monitoring

Security monitoring is a universe of its own. This area of operational practice is sometimes known as security operations or SecOps.

Dozens of open source and commercial tools and services can be enlisted to help monitor an environment's security. Third parties, sometimes called managed security service providers (MSSPs), render outsourced services. Despite all these options, security breaches remain common and often go undetected for months or years.

It always sounds attractive to outsource security operations; then it becomes someone else's problem to make sure your environment is secure. But think of it this way: would you be comfortable paying someone to watch your cash-filled wallet sit on a table with 10,000 other wallets in a busy train station? If so, an MSSP might be a good fit for you!

Perhaps the most important thing to know about security monitoring is that no automated tool or service is enough. You *must* implement a comprehensive security program that includes standards for user behavior, data storage, and incident response procedures, just to name a few elements. Chapter 27, *Security*, covers these basics.

Two core security functions should be integrated with your automated, continuous monitoring strategy: system integrity verification and intrusion detection.

System integrity verification

System integrity verification (often called file integrity monitoring or FIM) is the validation of the current state of the system against a known-good baseline. Most often, this validation compares the contents of the system files (kernel, executable commands, config files) with a cryptographically sound checksum such as SHA-512. If the checksum value of the file in the running system is different from that of the baseline version, a sysadmin is notified. Of course, regular maintenance activities such as planned changes, updates, and patches must be taken into account; not all changes are suspicious.

Acceptable hashing algorithms change over time. For example, MD5 is no longer considered cryptographically secure and should no longer be used.

The most commonly deployed FIM platforms are Tripwire and OSSEC; the latter is described in more detail starting here. The Linux version of AIDE also includes file integrity monitoring, but unfortunately, the FreeBSD version lacks this component.

Simpler is often better. A great bare-bones FIM option is **mtree**, which is native to FreeBSD and has recently been ported to Linux. **mtree** is an easy way to monitor file state and content changes, and it's easily integrated into your monitoring scripts. Here's an example of a quick script that uses **mtree**:

```
#!/bin/bash
if [ $# -eq 0 ]; then
    echo "mtree-check.sh [-bv]"
    echo "-b = create baseline"
    echo "-v = verify against baseline"
    exit
fi

## seed
KEY=93948764681464

## baseline directory
DIR=/usr/local/lib/mtree-check

if [ $1 = "-b" ]; then
    rm -rf $DIR/mtree_*
    cd $DIR
    mtree-c -K sha512 -s $KEY -p /sbin > mtree_sbin
fi

if [ $1 = "-v" ]; then
    cd $DIR
    mtree -s $KEY -p /sbin < mtree_sbin | \
        mail -s "`hostname` mtree integrity check" dan@admin.com
fi
```

With the **-b** flag, this script creates and stores the baseline. When run again with the **-v** flag, it validates the current contents of **/sbin** against the baseline.

As with so many aspects of system administration, setting up a FIM platform and operating it over time are wildly different propositions. You'll need a defined process in place to maintain the FIM data and respond to FIM alerts. We suggest feeding information from your FIM platform into your monitoring and alerting infrastructure so that it is not sidelined or ignored.

Intrusion detection monitoring

Two common forms of intrusion detection systems (IDSs) are in use: host-based (HIDS) and network-based (NIDS). NIDS systems examine the traffic transiting the network and attempt to identify unexpected or suspicious patterns. The most common NIDS system is based on Snort and is covered in detail starting here.

HIDS systems run as a set of processes on each system. They typically keep tabs on a variety of things, including network connections, file modification times and checksums, daemon and applications logs, use of elevated privileges, and other clues that may signal the operation of tools designed to facilitate unauthorized access ("root kits"). A HIDS is not a one-stop solution for security, but it's a valuable component of a comprehensive approach.

The two most popular open source HIDS platforms are OSSEC (Open Source SECurity) and AIDE (the Advanced Intrusion Detection Environment). In our experience, OSSEC is hands-down the better choice. Although AIDE is a nice FIM platform on Linux, OSSEC includes a broader array of functions. It can even be used in a client/server mode that supports non-UNIX clients such as Microsoft Windows and a variety of network infrastructure devices.

Much like FIM alerts, HIDS data is only as useful as the attention that's paid to it. HIDS is not a "set it and forget it" subsystem, and you will need to integrate HIDS alerts with your overall monitoring system. The most effective strategy we've found for addressing this issue is to auto-open tickets for HIDS alerts in your trouble ticketing system. You can then add a monitoring check that alerts on any unresolved HIDS tickets.

28.9 SNMP: THE SIMPLE NETWORK MANAGEMENT PROTOCOL

Years ago, the networking industry decided it would be helpful to create a standard protocol for the collection of monitoring data. Hence, the Simple Network Management Protocol, aka SNMP.

Despite its name, SNMP is actually quite complex. It defines a hierarchical namespace of management data and methods for reading and writing that data on each network device. SNMP also defines a way for managed servers and devices (“agents”) to send event notification messages (“traps”) to management stations.

Before we delve into the arcana of SNMP, we should note that the terminology associated with it is some of the most wretched technobabble to be found in the networking arena. In many cases, the standard names for SNMP concepts and objects actively lead you away from an understanding of what’s going on.

That said, the protocol itself is simple; most of SNMP’s complexity lies above the protocol layer in the conventions for constructing the namespace and in the unnecessarily baroque vocabulary that surrounds SNMP like a protective shell. As long as you don’t think too hard about its internal mechanics, SNMP is easy to use.

SNMP was designed to be implementable by dedicated network hardware such as routers, in which context it remains a plausible option. SNMP was later extended to include monitoring of servers and desktop systems, but its fitness for this purpose has always been questionable. Today, much better alternatives (e.g., **collectd**; see this page) are available.

We suggest that you approach SNMP as a low-level data collection protocol for use with special-purpose devices that don’t support anything else. Get data out of the SNMP world as quickly as possible and turn it over to a general-purpose monitoring platform for storage and processing. SNMP can be an interesting neighborhood to visit, but you wouldn’t want to live there!

SNMP organization

SNMP data is arranged in a standardized hierarchy. The naming hierarchy is made up of "Management Information Bases" (MIBs), structured text files that describe the data accessible through SNMP. MIBs contain descriptions of specific data variables, which are referred to with names known as object identifiers, or OIDs. An OID is just a fancy way of naming a specific managed piece of information.

All current SNMP-capable devices support the structure for MIB-II defined in RFC1213. But each vendor can and does extend that MIB further to add more data and metrics.

OIDs exist within a hierarchical namespace where the nodes are numbered rather than named. However, for ease of reference, nodes also have conventional text names. The separator for pathname components is a dot. For example, the OID that refers to the uptime of a device is 1.3.6.1.2.1.1.3. This OID is also known by the human-readable (though not necessarily "human-understandable without additional documentation") name

iso.org.dod.internet.mgmt.mib-2.system.sysUpTime

Table 28.3 presents a sampling of OID nodes that might be interesting to monitor for assessing network availability.

Table 28.3: Selected OIDs from MIB-II

OID[a]	Type	Contents
system.sysDescr	string	System info: vendor, model, OS type, etc.
interfaces.ifNumber	int	Number of network interfaces present
interfaces.ifTable	table	Table of infobits about each interface
ip.ipForwarding	int	1 if system is a gateway; otherwise, 2
ip.ipAddrTable	table	Table of IP addressing data (masks, etc.)
icmp.icmpInRedirects	int	Number of ICMP redirects received
icmp.icmpInEchos	int	Number of pings received
tcp.tcpInErrs	int	Number of TCP errors received

a. Relative to iso.org.dod.internet.mgmt.mib-2.

In addition to the universally supported MIB-II, there are MIBs for various kinds of hardware interfaces and protocols, MIBs for individual vendors, MIBs for different **snmpd** server implementations, and MIBs for particular hardware products.

A MIB is only a schema for naming management data. To be useful, a MIB must be backed up with agent-side code that maps between the SNMP namespace and the device's actual state.

SNMP agents that run on UNIX, Linux, or Windows come with built-in support for MIB-II. Most can be extended to support supplemental MIBs and to interface with scripts that do the

actual work of fetching and storing these MIBs' associated data. You'll see lots of software like this that's left over from a bygone era when SNMP was the new hotness. But it's all smoke and no fire; you shouldn't be running an SNMP agent on a UNIX system at all these days, except perhaps to answer the most basic queries about network configuration.

SNMP protocol operations

Only four basic SNMP operations exist: get, get-next, set, and trap.

Get and set are the basic operations for reading and writing data to the node identified by a specific OID. Get-next steps through a MIB hierarchy and can read the contents of tables as well.

A trap is an unsolicited, asynchronous notification from server (agent) to client (manager) that reports the occurrence of an interesting event or condition. Several standard traps are defined, including “I’ve just come up” notifications, reports of failure or recovery of a network link, and announcements of various routing and authentication problems. The mechanism by which the destinations of trap messages are specified depends on the implementation of the agent.

Since SNMP messages can potentially modify configuration information, some security mechanism is needed. The simplest version of SNMP security uses the concept of an SNMP “community string,” which is really just a horribly obfuscated way of saying “password.” There’s usually one community string for read-only access and another that allows writing.

Many systems come with the default community string set to “public”. Never leave this default in place; set real passwords for both the read-only and read/write community strings.

These days it makes a lot more sense to set up the SNMPv3 management framework, which allows for more security including authorization and access control for individual users.

Net-SNMP: tools for servers

On Linux and FreeBSD, the most common package that implements SNMP is called Net-SNMP. It includes an agent (**snmpd**), some command-line tools, a server for receiving traps, and even a library for developing SNMP-aware applications.

These days, Net-SNMP is primarily of interest because of its commands and libraries rather than its agent. It has been ported to many different UNIX-like systems, so it acts as a consistent platform on top of which you can write scripts. So, most distributions just separate out the Net-SNMP agent into a package of its own, making it easier to install only the commands.



On Debian and Ubuntu, the Net-SNMP packages are called **snmp** and **snmpd**. Install only the commands with **apt-get install snmp**.

RHEL 
On Red Hat and CentOS, the analogous packages are **net-snmp** and **net-snmp-tools**. Install the commands with **yum install net-snmp-tools**.


On Linux, configuration information goes in **/etc/snmp**; take note of the **snmpd.conf** file and **snmp.d** directory in that location. Run **systemctl start snmpd** to start up the agent daemon.


On FreeBSD, everything is included in one package: **pkg install net-snmp**. Configuration information goes in **/usr/local/etc/snmp**, which you'll have to create by hand. You can start the agent by hand with **service snmpd start**, or put

```
snmpd_enable="YES"
```

in **/etc/rc.conf** to start it at boot time.

On all systems where you need to run the SNMP agent, you'll need to ensure that UDP port 162 is not blocked by a firewall.

The commands that come with Net-SNMP can familiarize you with SNMP, and they're also great for one-off checks of specific OIDs. Table 28.4 lists the most commonly used tools.

Table 28.4: Command-line tools in the Net-SNMP package

Command	Function
snmpdelta	Monitors changes in SNMP variables over time
snmpdf	Monitors disk space on a remote host through SNMP
snmpget	Gets the value of an SNMP variable from an agent
snmpgetnext	Gets the next variable in sequence
snmpset	Sets an SNMP variable on an agent
snmptable	Gets a table of SNMP variables
snmptranslate	Searches for and describes OIDs in the MIB hierarchy
snmptrap	Generates a trap alert
snmpwalk	Traverses a MIB starting at a particular OID

Basic SNMP checks generally use some combination of **snmpget** and **snmpdelta**. Other programs are helpful when you want to identify new OIDs to monitor from your fancy enterprise management tool. For example, **snmpwalk** starts at a specified OID (or at the beginning of the MIB, by default), and repeatedly makes "get next" calls to the agent. This process dumps a complete list of available OIDs and their associated values.

For example, here's a truncated sample **snmpwalk** of the host tuva, a Linux system. The community string is "secret813community," and **-v1** specifies simple authentication.

```
$ snmpwalk -c secret813community -v1 tuva
SNMPv2-MIB::sysDescr.0 = STRING: Linux tuva.atrust.com 2.6.9-11.ELsmp #1
SNMPv2-MIB::sysUpTime.0 = Timeticks: (1442) 0:00:14.42
SNMPv2-MIB::sysName.0 = STRING: tuva.atrust.com
IF-MIB::ifDescr.1 = STRING: lo
IF-MIB::ifDescr.2 = STRING: eth0
IF-MIB::ifDescr.3 = STRING: eth1
IF-MIB::ifType.1 = INTEGER: softwareLoopback(24)
IF-MIB::ifType.2 = INTEGER: ethernetCsmacd(6)
IF-MIB::ifType.3 = INTEGER: ethernetCsmacd(6)
IF-MIB::ifPhysAddress.1 = STRING:
IF-MIB::ifPhysAddress.2 = STRING: 0:11:43:d9:1e:f5
IF-MIB::ifPhysAddress.3 = STRING: 0:11:43:d9:1e:f6
IF-MIB::ifInOctets.1 = Counter32: 2605613514
IF-MIB::ifInOctets.2 = Counter32: 1543105654
IF-MIB::ifInOctets.3 = Counter32: 46312345
IF-MIB::ifInUcastPkts.1 = Counter32: 389536156
IF-MIB::ifInUcastPkts.2 = Counter32: 892959265
IF-MIB::ifInUcastPkts.3 = Counter32: 7712325
...
```

In this example, general information about the system is followed by statistics about the host's network interfaces: lo0, eth0, and eth1. Depending on the MIBs supported by the agent you are managing, a complete dump can run to hundreds of lines. In fact, a full install on an Ubuntu system configured to serve every MIB spits out over 12,000 lines!

If you looked up the MIBs for the latest version of Net-SNMP on an Ubuntu system (check out mibdepot.com or install the **snmp-mibs-downloader** package), you would see that the five-minute load average OID is 1.3.6.1.4.1.2021.10.1.3.2. If you were interested in seeing the five-minute load average for the local host (configured with a community string of “public”), you would run:

```
$ snmpget -v 2c -c public localhost .1.3.6.1.4.1.2021.10.1.3.2
iso.3.6.1.4.1.2021.10.1.3.2 = STRING: "0.08"
```

Many useful SNMP-related Perl, Ruby, and Python modules are available from these languages’ respective module repositories. Although you can write scripts in terms of Net-SNMP commands, it’s usually easier and cleaner to make use of native modules that are customized for your language of choice.

28.10 Tips and tricks for monitoring

Over the years, we've picked up a few tips on how to maximize the effectiveness of monitoring. These are the main ideas:

- Avoid monitoring burn-out. Ensure that sysadmins who receive notifications outside of regular work hours get regular breaks. This goal is best achieved with a rotation system in which teams of two or more individuals are on call for a day or a week, then the next team takes over. Failure to heed this advice results in crabby sysadmins who hate their jobs.
- Define what circumstances really require 24 × 7 attention, and make sure this information is clearly communicated to your monitoring team, your on-call teams, and the customers or business units you support. The mere fact that you're monitoring something doesn't mean that administrators should be mustered at 3:30 a.m. when the value goes out of bounds. Many issues should be addressed during normal business hours.
- Eliminate monitoring noise. If false positives or notifications for noncritical services are being generated, make time to stop and fix them. Otherwise, like the boy who cried wolf, all notifications will eventually fail to receive the necessary attention.
- Create run books for everything. Any common restart, reset, or corrective procedure should be documented in a form that allows a responder who is not intimately familiar with the system in question to take appropriate action. The costs of not having such documentation are that problems will not be fixed quickly, mistakes will be made, and additional staff will be rousted to handle emergencies. Wikis are great for maintaining this type of documentation.
- Monitor the monitoring platform. This one will seem obvious once you've missed a critical outage because the monitoring platform was also down. Learn from our mistakes and make sure something is watching your watchful eyes.
- Miss an outage because of something that wasn't monitored? Make sure it gets added so you catch the problem next time.
- Finally, and perhaps most importantly: no server or service goes into production without first being added to the monitoring system. No exceptions.

28.11 Recommended reading

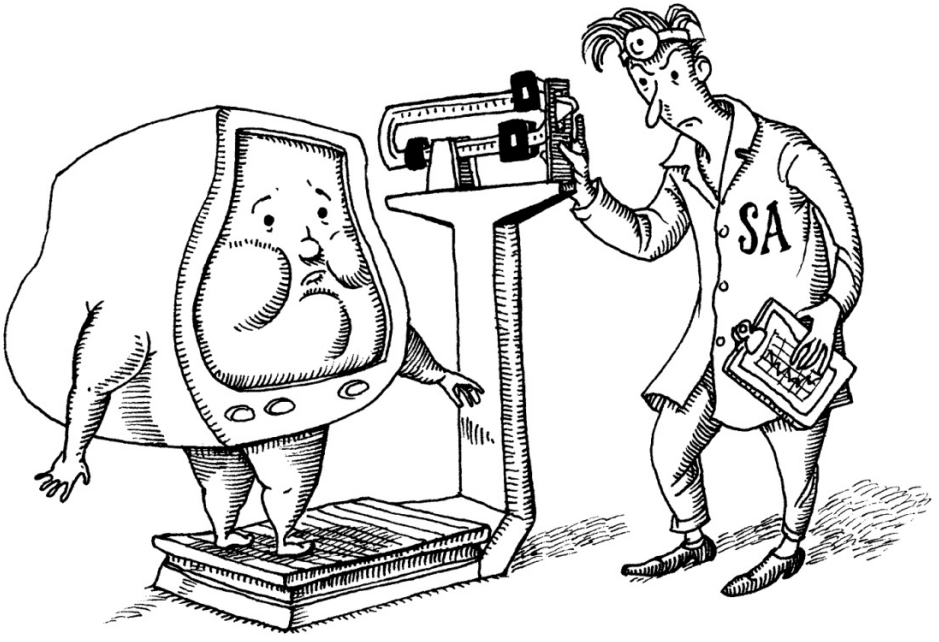
Hecht, James. *Rethinking Monitoring for Container Operations*. Great details on strategy and philosophy for monitoring containers. Find it at

http://thenewstack.io/monitoring-reset-containers/

Turnbull, James. *The Art of Monitoring*. Seattle, WA: Amazon Digital Services, 2016.

Dixon, Jason. *Monitoring with Graphite: Tracking Dynamic Host and Application Metrics at Scale.* Sebastopol, CA: O'Reilly Media, 2017.

29 Performance Analysis



Performance analysis and tuning are often treated as a form of system administration witchcraft. They're not really witchcraft, but they do qualify as both science and art. The "science" part involves making careful quantitative measurements and applying the scientific method. The "art" part relates to the need to balance resources in a practical, level-headed way, since optimizing for one application or user can result in other applications or users suffering. As with so many things in life, you will often find that it's impossible to make everyone happy.

Folks often assert that today's performance problems are somehow wildly different from those of previous decades. That claim is inaccurate. It's true that systems have become more complex, but the baseline determinants of performance and the high-level abstractions for measuring and managing it remain the same as always. Unfortunately, improvements in system performance correlate strongly with the community's ability to create new applications that suck up all available resources.

An added complexity of recent years is the many layers of abstraction that often sit between your servers and the physical infrastructure of the cloud. It's often impossible to know exactly what hardware is providing storage or CPU cycles to your server.

The magic and the challenge of the cloud are two aspects of the same form. Despite popular belief, you do not get to ignore performance considerations just because your servers are virtual. In fact, the billing models used by cloud providers create an even more direct link between

operational efficiency and server costs. Knowing how to measure and evaluate performance has become more important than ever.

This chapter focuses on the performance of systems that are used as servers. Desktop systems (and laptops) typically do not experience the same types of performance issues that servers do. The answer to the question of how to improve performance on a desktop machine is almost always, “Upgrade the hardware.” Users like this answer because it means they get fancy new systems more often.

29.1 Performance tuning philosophy

One way that UNIX and Linux differ from other mainstream operating systems is that copious data are available to characterize their inner workings. Detailed information is generated by every level of the system, and administrators control a variety of tunable parameters. Source code is usually available for review if you have trouble identifying the cause of a performance problem despite the available instrumentation. For these reasons, UNIX and Linux are typically the operating systems of choice at performance-conscious sites.

Even so, performance tuning isn't easy. Users and administrators alike often think that if they only knew the right "magic," their systems would be twice as fast. But that's rarely true.

One common fantasy involves tweaking the kernel variables that control the paging system and the buffer pools. These days, kernels are pretuned to achieve reasonable (though admittedly, not optimal) performance under a variety of load conditions. If you try to optimize the system on the basis of one particular measure of performance (e.g., buffer utilization), the chances are high that you will distort the system's behavior relative to other performance metrics and load conditions.

The most serious performance issues often lie within applications and have little to do with the underlying operating system. This chapter discusses system-level performance tuning and mostly leaves application-level tuning to others. As a system administrator, you need to be mindful that application developers are people, too. How many times have you said, or thought, that "It must be a network problem"? In a similar way, application developers often initially assume that any issues must originate in a subsystem that is someone else's responsibility.

Given the complexity of modern applications, some problems can only be resolved through collaboration among application developers, system administrators, server engineers, DBAs, storage administrators, and network architects. In this chapter, we help you determine what data and information to take back to these other folks to help them solve a performance problem—if, indeed, the problem lies in their area. This approach is far more productive than just saying, "Everything looks fine; it's not my problem."

In all cases, take everything you read on the Internet with a tablespoon of salt. In the area of system performance, you will see superficially convincing arguments on all sorts of topics. However, most of the proponents of these theories do not have the knowledge, discipline, and time required to design valid experiments. Popular support means very little; for every hare-brained proposal, you can expect to see a Greek chorus of "I increased the size of my buffer cache by a factor of ten just like Joe said, and the system feels *much, much* faster!!!" Right.

Here are some rules to keep in mind:

- Collect and review *historical* information about your system. If the system was performing fine a week ago, an examination of the aspects of the system that have changed may well lead you to a smoking gun. Keep baselines and trends in your hip

pocket to pull out in an emergency. As a first step, review log files to see if an underlying hardware problem has developed.

- Familiarize yourself with the trend collection and analysis tools discussed in Chapter 28, *Monitoring*. These tools are critical for performance assessment.
- Tune your system in a way that lets you compare the current results to the system's previous baseline.
- Don't intentionally overload your systems or your network. The kernel gives each process the illusion of infinite resources. But once 100% of the system's resources are in use, the kernel has to work hard to maintain that illusion, delaying processes and often consuming a sizable fraction of the resources itself.
- As in particle physics, the more information you collect with system monitoring utilities, the more you affect the system you are observing. It is best to rely on something simple and lightweight that runs in the background (e.g., **sar** or **vmstat**) for routine observation. If those feelers show something significant, you can investigate further with other tools.
- When making changes, change only one thing at a time, and document each change that you make. Observe, record, and ponder the results before changing anything else.
- Always make sure you have a rollback plan in case your magic fix actually makes things worse.

29.2 Ways to improve performance

Here are some specific things you can do to improve performance:

- Ensure that the system has enough memory. As we see in the next section, memory size has a major influence on performance. If you're running a system in the cloud, the amount of memory allocated to an instance is usually easy to adjust (though it's often bundled with other resource allocations into a full-system profile).
- Eliminate storage resources' dependence on mechanical operations where possible. Solid state disk drives (SSDs) are widely available and can yield big performance boosts because they don't require the physical movement of a disk or armature to read bits. SSDs are easily installed (or, in the case of cloud environments, chosen) in place of existing old-school disk drives.
- If you are using UNIX or Linux as a web server or as some other type of network application server, you may want to spread traffic among several systems with a (physical or virtual) load balancer. Such an appliance balances the load according to one of several user-selectable algorithms such as "most responsive server" or "round robin."

  These load balancers also add useful redundancy should a server go down. They're really quite necessary if your site must handle unexpected traffic spikes.
- Double-check the configuration of the system and of individual applications. Many applications can be tuned to yield tremendous performance improvements (e.g., by spreading data across disks, by not performing DNS lookups on the fly, or by running multiple instances of a server).
- Correct problems of usage, both those caused by "real work" (too many services running at once, inefficient programming practices, batch jobs run at excessive priority, and large jobs run at inappropriate times of day) and those caused by the system (such as unwanted daemons).
- Organize hard disks and filesystems so that load is evenly balanced, maximizing I/O throughput. For specific applications such as databases, you can use a fancy multidisk technology such as striped RAID to optimize data transfers. Consult your database vendor for recommendations. For Linux systems, ensure that you've selected the appropriate Linux I/O scheduler for your disk (see this page for details).

  Remember that different types of applications and databases respond differently to being spread across multiple disks. RAID comes in many forms; take time to determine which form (if any) is appropriate for your particular application.

- Monitor your network to be sure that it is not saturated with traffic and that the error rate is low. A wealth of network information is available through the **netstat** (FreeBSD) and **ss** (Linux) commands.
- Identify cases where the system is fundamentally inadequate to satisfy the demands being made of it. You cannot tune your way out of these situations.

These steps are listed in rough order of effectiveness. Adding memory, converting to SSDs, and balancing traffic across multiple servers can often make a huge difference in performance. The effectiveness of the other measures ranges from noticeable to none.

Analysis and optimization of software data structures and algorithms almost always lead to significant performance gains. But unless you have a substantial base of local software, this level of design is usually out of your control.

29.3 FACTORS THAT AFFECT PERFORMANCE

Perceived performance is determined by the basic capacities of the system's resources and by the efficiency with which those resources are allocated and shared. The exact definition of a "resource" is rather vague. It can include such items as cached contexts on the CPU chip and entries in the address table of the memory controller. However, to a first approximation, only the following four resources have much effect on performance:

- CPU utilization (and stolen cycles, see below)
- Memory
- Storage I/O
- Network I/O

If resources are still left after active processes have taken what they want, the system's performance is about as good as it can be.

If there are not enough resources to go around, processes must take turns. A process that does not have immediate access to the resources it needs has to wait around doing nothing. The amount of time spent waiting is one of the basic measures of performance degradation.

Historically, CPU utilization was one of the easiest resources to measure because a constant amount of processing power was always available. Nowadays, some virtualized or cloud environments may allocate CPU more dynamically. A process that's using more than 90% of the allocated CPU is entirely CPU bound and is consuming essentially all of the system's available computing power.

Many people assume that CPU resources are the most important factor affecting a system's overall performance. Given infinite amounts of all other resources or certain types of applications (e.g., numerical simulations), a faster CPU (or more CPU cores) *does* make a dramatic difference. But in the everyday world, CPU is actually relatively unimportant.

Disk bandwidth is a common performance bottleneck. Because traditional hard disks are mechanical systems, it takes many milliseconds to locate a disk block, fetch its contents, and wake up the process that's waiting for it. Delays of this magnitude overshadow every other source of performance degradation. Each disk access causes a stall worth millions of CPU instructions. Solid state drives (SSDs) are one tool you can use to address this problem.

Because of virtual memory, disk bandwidth and memory can be directly related if the demand for physical memory is greater than the supply. Situations in which physical memory becomes scarce often result in memory pages being written to disk so that they can be reclaimed and reused for another purpose. In these situations, using memory is just as expensive as using the disk. Avoid this trap when performance is important; make sure that every system has adequate physical memory.

Network bandwidth resembles disk bandwidth in many ways because of the latencies involved in network communication. However, networks are atypical in that they involve entire communities rather than individual computers. They are also particularly susceptible to hardware problems and overloaded servers.

29.4 Stolen CPU cycles

The promise of the cloud (and of virtualization more generally) is that your server always has the resources it needs. In reality, much of this bounty is created with smoke and mirrors. Even in a large-scale virtualization environment, resource contention can have a noticeable effect on a virtual server's performance.

CPU is the resource most commonly affected. There are two ways by which CPU cycles can be stolen from your virtual machine:

- The hypervisor running your VM enforces a CPU quota based on how much CPU power you have contracted for. Shortfalls can be addressed by allocation of more resources at the hypervisor level or by your purchasing a larger instance size from the cloud provider.
- The physical hardware is oversubscribed, and there are not enough physical CPU cycles available to meet the current needs of all VM instances, even though these instances may all be under their CPU quotas. On a cloud provider, fixing this issue may be as simple as restarting your instance so that it gets reassigned to new physical hardware. In a data center of your own, the solution may require upgrading your virtualization environment with more resources.

Although CPU stealing can happen to any operating system running on a virtualized platform, Linux gives you some visibility into this phenomenon with the `st` metric ("stolen") in **top**, **vmstat**, and **mpstat**.

Here's an example from **top**:

```
top - 18:36:42 up 3 days, 18:03,  1 user,  load average: 3.40, 2.25, 2.08
Tasks: 218 total,   4 running, 217 sleeping,   0 stopped,   0 zombie
%Cpu: 41.6 us, 42.2 sy, 0.0 ni, 0.0 id, 0.0 wa, 0.0 hi, 0.0 si, 16.2 st
```

In this example, 16.2% of the time, the system is ready to do work but is unable to run because CPU is being diverted away from the VM by the hypervisor. This time spent waiting translates directly to reduced throughput. Monitor this metric carefully on virtual servers to ensure that your workloads aren't being unintentionally starved of CPU.

29.5 Analysis of performance problems

It can be difficult to isolate performance problems in a complex system. As a sysadmin, you often receive anecdotal problem reports that suggest a particular cause or fix (e.g., “The web server has gotten painfully sluggish because of all those damn AJAX calls…”). Take note of this information, but don’t assume that it’s accurate or reliable; do your own investigation.

Rigorous, transparent application of the scientific method helps you reach conclusions that you and others in your organization can rely on. Such an approach lets others evaluate your results, increases your credibility, and raises the likelihood that your suggested changes will actually fix the problem.

“Being scientific” doesn’t mean that you have to gather all the relevant data yourself. External information usually helps a lot. Don’t spend hours doing experiments related to issues that can just as easily be looked up in a FAQ.

We suggest the following five steps:

1. *Formulate the question.* Pose a specific question in a defined functional area, or state a tentative conclusion or recommendation that you are considering. Be specific about the type of technology, the components involved, the alternatives you are considering, and the outcomes of interest.

2. *Gather and classify evidence.* Conduct a systematic search of documentation, knowledge bases, known issues, blogs, white papers, forums, and other resources to locate external evidence related to your question. On your own systems, capture telemetry data and, where necessary or possible, instrument specific system and application areas of interest.

3. *Critically appraise the data.* Review each data source for relevance and critique it for validity. Abstract key information and note the quality of the sources.

4. *Summarize the evidence both narratively and graphically.* Combine findings from multiple sources into a narrative précis and, if possible, a graphic representation. Data that appears equivocal in numeric form can often become decisive once charted.

5. *Develop a conclusion statement.* State your conclusions (i.e., the answer to your question) concisely. Assign a grade to indicate the overall strength or weakness of the evidence that supports your conclusions.

29.6 System performance checkup

Enough generalities—let’s look at some specific tools and areas of interest. Before you take measurements, you need to know what you’re looking at.

Taking stock of your equipment

Start your inquiry with an inventory of your (physical or virtual) hardware, especially CPU and memory resources. This inventory can help you interpret the information presented by other tools and can help you set realistic expectations regarding the upper bounds on performance.



On Linux systems, the **/proc** filesystem is the place to find an overview of the hardware your operating system thinks you have. (More detailed hardware information can be found in **/sys**; see this page.) Table 29.1 shows some of the key files. See this page for general information about **/proc**.

Table 29.1: Sources of hardware information on Linux

File	Contents
/proc/cpuinfo	CPU type and description
/proc/meminfo	Memory size and usage
/proc/diskstats	Disk devices and usage statistics

Four lines in **/proc/cpuinfo** help you identify the system's exact CPU: vendor_id, cpu family, model, and model name. Some of the values are cryptic; it's best to look up the decoding on-line.

The exact info contained in **/proc/cpuinfo** varies by system and processor, but here's a representative example:

```
$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 15
model name      : Intel(R) Xeon(R) CPU  E5310   @ 1.60GHz
stepping        : 11
cpu MHz         : 1600.003
cache size      : 4096 KB
physical id     : 0
cpu cores       : 2
siblings        : 2
...
```

The file contains one entry for each processor core seen by the OS. The data vary slightly by kernel version. The processor value uniquely identifies each core. physical id values are unique per CPU socket, and core id values (not shown above) are unique per core within a CPU socket. Cores that support hyperthreading (duplication of CPU contexts without duplication of other processing features) are identified by an ht in the flags field (not shown above). If hyperthreading is actually in use, the siblings field for each core shows how many contexts are available on a given core.

Another command to run for information on both FreeBSD and Linux is **dmidecode**. It dumps the system's Desktop Management Interface (DMI, aka SMBIOS) data. The most useful option is **-t** *type*; Table 29.2 shows the valid *type*s.

Table 29.2: Type values for dmidecode -t

Value	Description
1	System information
2	Base board Information
3	Chassis information
4	Processor information
7	Cache information
8	Port connector information
9	System slot information
11	OEM strings
12	System configuration options
13	BIOS language information
16	Physical memory array
17	Memory device
19	Memory array mapped address
32	System boot information
38	IPMI device information

The example below shows typical information:

```
$ sudo dmidecode -t 4
# dmidecode 2.11
SMBIOS 2.2 present.

Handle 0x0004, DMI type 4, 32 bytes.
Processor Information
    Socket Designation: PGA 370
    Type: Central Processor
    Family: Celeron
    Manufacturer: GenuineIntel
    ID: 65 06 00 00 FF F9 83 01
    Signature: Type 0, Family 6, Model 6, Stepping 5
...
```

Bits of network configuration information are scattered about the system. **ifconfig -a** (FreeBSD) and **ip a** (Linux) are the best sources of IP and MAC information for each configured interface.

Gathering performance data

Most performance analysis tools tell you what's going on at a particular point. However, the number and character of loads probably change throughout the day. Be sure to gather a cross-section of data before taking action. The best information on system performance often becomes clear only after a long period (a month or more) of data collection. It is particularly important to collect data during periods of peak use. Resource limitations and system misconfigurations are often visible only when the machine is under heavy load. See Chapter 28, *Monitoring*, for more information about collecting and analyzing data.

Analyzing CPU usage

You will probably want to gather three kinds of CPU data: overall utilization, load averages, and per-process CPU consumption. Overall utilization can help identify systems on which the CPU's speed is itself the bottleneck. Load averages profile overall system performance. Per-process CPU consumption data can identify specific processes that are hogging resources.

You can obtain summary information with the **vmstat** command. **vmstat** takes two arguments: the number of seconds to monitor the system for each line of output and the number of reports to print. If you don't specify the number of reports, **vmstat** runs until you press <Control-C>. For example:

```
$ vmstat 5 5
procs ----------memory----------  -swap-  ---io--  -system--  ----cpu----
 r b  swpd    free   buff  cache  si  so    bi bo    in   cs  us sy id wa
 1 0   820 2606356 428776 487092   0   0  4741 65  1063 4857  25  1 73  0
 1 0   820 2570324 428812 510196   0   0  4613 11  1054 4732  25  1 74  0
 1 0   820 2539028 428852 535636   0   0  5099 13  1057 5219  90  1  9  0
 1 0   820 2472340 428920 581588   0   0  4536 10  1056 4686  87  3 10  0
 3 0   820 2440276 428960 605728   0   0  4818 21  1060 4943  20  3 77  0
```

Although exact columns vary among systems, CPU utilization statistics are fairly consistent across platforms. User time, system (kernel) time, idle time, and time waiting for I/O are shown in the us, sy, id, and wa columns on the far right. CPU numbers that are heavy on user time generally indicate computation, and high system numbers indicate that processes are making a lot of system calls or are performing lots of I/O.

A rule of thumb for general-purpose compute servers that has served us well over the years is this: the system should spend approximately 50% of its non-idle time in user space and 50% in system space; the overall idle percentage should be nonzero. If you are dedicating a server to a single, CPU-intensive application, the majority of time should be spent in user space.

The cs column shows context switches per interval (that is, the number of times that the kernel changed which process was running). The number of interrupts per interval (usually generated by hardware devices or components of the kernel) is shown in the in column. Extremely high cs or in values typically indicate a misbehaving or misconfigured hardware device. The other columns are useful for memory and disk analysis, which we discuss later in this chapter.

Long-term averages of the CPU statistics let you determine whether there is fundamentally enough CPU power to go around. If the CPU usually spends part of its time in the idle state, there are cycles to spare. Upgrading to a faster CPU won't do much to improve the overall throughput of the system, although it may speed up individual operations.

As you can see from this example, the CPU generally flip-flops back and forth between heavy use and idleness. Therefore, be sure to observe these numbers as an average over time. The

smaller the monitoring interval, the less consistent the results.

On multiprocessor machines, most tools present an average of processor statistics across all processors. On Linux, the **mpstat** command generates **vmstat**-like output for each individual processor. The **-P** flag lets you specify a specific processor to report on. **mpstat** is useful for debugging software that supports symmetric multiprocessing—it's also enlightening to see how (in)efficiently your system uses multiple processors. Here's an example that shows the status of each of four processors:

```
linux$ mpstat -P ALL
08:13:38 PM CPU %user %nice %sys %iowait %irq %soft %idle intr/s
08:13:38 PM   0  1.02  0.00 0.49    1.29 0.04  0.38 96.79 473.93
08:13:38 PM   1  0.28  0.00 0.22    0.71 0.00  0.01 98.76 232.86
08:13:38 PM   2  0.42  0.00 0.36    1.32 0.00  0.05 97.84 293.85
08:13:38 PM   3  0.38  0.00 0.30    0.94 0.01  0.05 98.32 295.02
```

On a workstation with only one user, the CPU generally spends most of its time idle. Then when you render a web page or switch windows, the CPU is used heavily for a short period. In this situation, information about long-term average CPU usage is not meaningful.

The second CPU statistic that's useful for characterizing the burden on your system is the "load average," which represents the average number of runnable processes. It gives you a good idea of how many pieces the CPU pie is being divided into. The load average is obtained with the **uptime** command:

```
$ uptime
11:10am  up 34 days, 18:42, 5 users, load average: 0.95, 0.38, 0.31
```

Three values are given, corresponding to the 1, 5, and 15-minute averages. In general, the higher the load average, the more important the system's aggregate performance becomes. If there is only one runnable process, that process is usually bound by a single resource (commonly disk bandwidth or CPU). The peak demand for that one resource becomes the determining factor in performance.

When more processes share the system, loads may or may not be more evenly distributed. If the processes on the system all consume a mixture of CPU, disk, and memory, the performance of the system is less likely to be dominated by constraints on a single resource. In this situation, it becomes most important to look at average measures of consumption, such as total CPU utilization.

See this page for more information about priorities.

The system load average is an excellent metric to track as part of a system baseline. If you know your system's load average on a normal day and it is in that same range on a bad day, this is a hint that you should look elsewhere (such as the network) for performance problems. A load

average above the expected norm suggests that you should look at the processes running on the system itself.

Another way to view CPU usage is to run the **ps -aux** command to see how much of the CPU each process is using. On a busy system, at least 70% of the CPU is often consumed by just one or two processes. Deferring the execution of the CPU hogs or reducing their priority makes the CPU more available to other processes.

---

*See this page for more information about **top**.*

An excellent alternative to **ps** is a program called **top**. It presents about the same information as **ps**, but in a live, regularly updated format that shows the status of the system over time. Refreshing **top**'s output too rapidly can itself be quite a CPU hog, so be judicious in your use of **top**.

Understanding how the system manages memory

The kernel manages memory in units called pages that are usually 4KiB or larger. It allocates virtual pages to processes as they request memory. Each virtual page is mapped to real storage, either to RAM or to "backing store" on disk. The kernel uses a "page table" to keep track of the mappings between these made-up virtual pages and real pages of memory.

The kernel can effectively allocate as much memory as processes ask for by augmenting real RAM with swap space. Since processes expect their virtual pages to map to real memory, the kernel may have to constantly shuffle pages between RAM and swap as different pages are accessed. This activity is known as paging. Ages ago, a second process known as "swapping" could occur in which all of a process's pages were pushed out to disk at the same time. Today, demand paging is used in all cases.

The kernel tries to manage the system's memory so pages that have been recently accessed are kept in memory and less active pages are paged out to disk. This scheme is known as an LRU system since the least recently used pages are the ones that get shunted to disk.

It would be inefficient for the kernel to keep track of all memory references, so it uses a cache-like algorithm to decide which pages to keep in memory. The exact algorithm varies by system, but the concept is similar across platforms. This system is cheaper than a true LRU system and produces comparable results.

When memory is low, the kernel tries to guess which pages on the inactive list were least recently used. If those pages have been modified by a process, they are considered "dirty" and must be paged out to disk before the memory can be reused. Pages that have been laundered in this fashion (or that were never dirty to begin with) are "clean" and can be recycled for use elsewhere.

When a process refers to a page on the inactive list, the kernel returns the page's memory mapping to the page table, resets the page's age, and transfers the page from the inactive list to the active list. Pages that have been written to disk must be paged in before they can be reactivated if the page in memory has been remapped. A "soft fault" occurs when a process references an in-memory inactive page, and a "hard fault" results from a reference to a nonresident (paged-out) page. In other words, a hard fault requires a page to be read from disk and a soft fault does not.

The kernel tries to stay ahead of the system's demand for memory, so there is not necessarily a one-to-one correspondence between page-out events and page allocations by running processes. The goal of the system is to keep enough free memory handy that processes don't have to actually wait for a page-out each time they make a new allocation. If paging increases dramatically when the system is busy, it would probably benefit from more RAM.



You can tune the kernel's "swappiness" parameter (**/proc/sys/vm/swappiness**) to tell the kernel how to balance between reclaiming swap-backed and file-backed pages. Setting

swappiness to zero focuses reclamations entirely on file-backed pages; a swappiness of 100 makes an equal balance between the two. By default, the swappiness parameter has a value of 60. (If you find yourself tempted to modify this parameter, it's probably time to buy more RAM for the system.)

If the kernel is unable to reclaim pages, Linux uses an "out-of-memory killer" to handle this condition. The killer function selects and kills a process to free up memory. Although the kernel attempts to kill off the least important user process on your system, running out of memory is always something to avoid. In this situation, it's likely that a substantial portion of the system's resources are being devoted to memory housekeeping rather than to useful work.

Analyzing memory usage

Two numbers summarize memory activity: the total amount of active virtual memory and the current paging rate. The first number tells you the total demand for memory, and the second suggests the proportion of that memory that is actively used. Your goal is to reduce activity or increase memory until paging remains at an acceptable level. Occasional paging is inevitable; don't try to eliminate it completely.

Run **swapon -s** to determine the amount of paging (swap) space that's currently in use:

```
linux$ swapon -s
Filename        Type       Size     Used  Priority
/dev/hdb1       partition  4096532  0     -1
/dev/hda2       partition  4096564  0     -2
```

swapon reports usage in kilobytes. The sizes quoted by these programs do not include the contents of core memory, so you have to compute the total amount of virtual memory yourself:

VM = size of real memory + amount of swap space used

On FreeBSD systems, paging statistics obtained with **vmstat** look like this:

```
freebsd$ vmstat 5 5
procs  memory       page                    disks     faults
r b w  avm   fre   flt  re  pi  po    fr   sr da0 cd0   in    sy
0 0 0 412M  1.8G    97   0   1   0   200    6   0   0   51   359
2 0 0 412M  1.8G     1   0   0   0     0    7   0   0    5    27
2 0 0 412M  1.8G     0   0   0   0     0    7   0   0    4    25
1 0 0 412M  1.8G     0   0   0   0     0    6   0   0    4    25
0 0 0 412M  1.8G     0   0   0   0     0    7   0   0    6    26
```

CPU information has been removed from this example. Under the procs heading are shown the number of processes that are immediately runnable, blocked on I/O, and runnable but swapped. If the value in the w column is ever nonzero, it is likely that the system's memory is pitifully inadequate relative to the current load.

Under the memory heading, you can see both the active virtual memory (avm) and free virtual memory (fre). The columns under the page heading give information about paging activity. All columns represent average values per second. Table 29.3 shows their meanings.

Table 29.3: Decoding guide for FreeBSD vmstat paging statistics

Column	Meaning
flt	Total number of page faults
re	Number of pages reclaimed (rescued from the free list)
pi	Number of kilobytes paged in
po	Number of kilobytes paged out
fr	Number of kilobytes placed on the free list
sr	Number of pages scanned by the clock algorithm

On Linux systems, paging statistics obtained with **vmstat** look like this:

```
linux$ vmstat 5 5
procs ---------memory-------- -swap- ---io-- -system-  -----cpu------
 r b  swpd  free  buff  cache si  so  bi  bo   in  cs  us sy id wa st
 5 0     0 66488 40328 597972  0   0 252  45 1042 278   3  4 93  1  0
 0 0     0 66364 40336 597972  0   0   0  37 1009 264   0  1 98  0  0
 0 0     0 66364 40344 597972  0   0   0  5  1011 252   1  1 98  0  0
 0 0     0 66364 40352 597972  0   0   0  3  1020 311   1  1 98  0  0
 0 0     0 66364 40360 597972  0   0   0  21 1067 507   1  3 96  0  0
```

As in the FreeBSD output, the number of processes that are immediately runnable and that are blocked on I/O are shown under the procs heading. Paging statistics are condensed to two columns, si and so, which represent pages swapped in and out, respectively.

Any apparent inconsistencies among the memory-related columns are for the most part illusory. Some columns count pages and others count kilobytes. All values are rounded averages. Furthermore, some are averages of scalar quantities and others are average deltas.

Use the si and so fields to evaluate the system's swapping behavior. A page-in (si) represents a page being recovered from the swap area. A page-out (so) represents data being written to the swap area after being forcibly ejected by the kernel.

If your system has a constant stream of page-outs, it's likely that you would benefit from more physical memory. But if paging happens only occasionally and does not produce annoying hiccups or user complaints, you can ignore it. If your system falls somewhere in the middle, further analysis should depend on whether you are trying to optimize for interactive performance (e.g., a workstation) or for a more server-like workload.

Analyzing disk I/O

You can monitor disk performance with the **iostat** command. Like **vmstat**, it accepts optional arguments to specify an interval in seconds and a repetition count, and its first line of output is a summary since boot. **iostat** output on Linux looks like this:

```
linux$ iostat
...
Device:       tps  kB_read/s  kB_wrtn/s  kB_read  kB_wrtn
sda          0.41       8.20       1.39   810865   137476
dm-0         0.39       7.87       1.27   778168   125776
dm-1         0.02       0.03       0.04     3220     3964
dm-2         0.01       0.23       0.06    22828     5652
```

Each hard disk has the columns tps, kB_read/s, kB_wrtn/s, kB_read, and kB_wrtn, indicating transfers per second, kilobytes read per second, kilobytes written per second, total kilobytes read, and total kilobytes written.

The cost of seeking is the most important factor affecting mechanical disk drive performance. To a first approximation, the rotational speed of the disk and the speed of the bus to which the disk is connected have relatively little impact. Modern mechanical disks can transfer hundreds of megabytes of data per second if they are read from contiguous sectors, but they can only perform about 100 to 300 seeks per second. If you transfer one sector per seek, you can easily realize less than 5% of the drive's peak throughput. SSD disks have a significant advantage over their mechanical predecessors because their performance is not tied to platter rotation or head movement.

Whether you're using mechanical or SSD disks, you should put filesystems that are used together on separate disks to maximize performance. Although bus architectures and device drivers influence efficiency, most computers can manage multiple disks independently, thereby increasing throughput. For example, it's often worthwhile to segregate frequently accessed web server data and web server logs onto different disks.

It's especially important to split the paging (swap) area among several disks if possible, since paging tends to slow down the entire system. Many systems can use both dedicated swap partitions and swap files on a formatted filesystem.

---

*See this page for more information about **lsof** and **fuser**.*

The **lsof** command, which lists open files, and the **fuser** command, which shows the processes that are using a filesystem, can be helpful for isolating disk I/O performance issues. These commands show interactions between processes and filesystems, some of which may be unintended. For example, if an application is writing its log to the same device used for database logs, a disk bottleneck may result.

fio: testing storage subsystem performance

fio (github.com/axboe/fio) is available for both Linux and FreeBSD. Use it to test the performance of the storage subsystem. It's particularly helpful in large environments where shared storage resources (such as a Storage Area Network) are deployed. If you find yourself in a situation where storage performance is a concern, it's often valuable to determine quantitative values for the following:

- Throughput in I/O operations per second (IOPS) (read, write, and mixed)
- Average latency (read and write)
- Maximum latency (peak read or write latency)

As part of the **fio** distribution, config (**.fio**) files for common tests such as these are included in the **examples** subdirectory. Here's an example of a simple read/write test:

```
$ fio read-write.fio
ReadWriteTest: (g=0): rw=rw, bs=4K-4K/4K-4K/4K-4K, eng=posixaio, depth=1
fio-2.18
Starting 1 thread
Jobs: 1 (f=1): [M] [100.0% done] [110.3MB/112.1MB/0KB /s]
    [22.1K/28.4K/0 iops] [eta 00m:00s]
  read : io=1024.7MB, bw=91326KB/s, iops=20601, runt= 9213msec
    slat (usec): min=0, max=73, avg= 2.30, stdev= 0.23
    clat (usec): min=0, max=2214, avg=20.30, stdev=101.20
     lat (usec): min=5, max=2116, avg=22.21, stdev=101.21
    clat percentiles (usec):
    |  1.00th=[    4],  5.00th=[    6], 10.00th=[    7], 20.00th=[    7],
    | 30.00th=[    6], 40.00th=[    7], 50.00th=[    7], 60.00th=[    7],
    | 70.00th=[    8], 80.00th=[    8], 90.00th=[    8], 95.00th=[   10],
    | 99.00th=[  668], 99.50th=[ 1096], 99.90th=[ 1208], 99.95th=[ 1208],
    | 99.99th=[ 1256]
...
  READ: io=1024.7MB, aggrb=91326KB/s, minb=91326B/s, maxb=91326KB/s,
    mint=10671msec, maxt=10671msec
  WRITE: io=1023.4MB, aggrb=98202KB/s, minb=98202KB/s, maxb=98202KB/s,
    mint=10671msec, maxt=10671msec
```

As with so many performance-related metrics, there is no universally correct value for any of these measures. It's best to establish a benchmark, make adjustments, and remeasure.

sar: collecting and reporting statistics over time

The **sar** command is a performance monitoring tool that has lingered through multiple UNIX and Linux epochs despite its somewhat obscure command-line syntax. The original command had its roots in early AT&T UNIX.

At first glance, **sar** seems to display much the same information as **vmstat** and **iostat**. However, there's one important difference: **sar** can report on historical as well as current data.

---

*The Linux package that contains **sar** is called sysstat.*

Without options, the **sar** command reports CPU utilization for the day at 10-minute intervals since midnight, as shown below. This historical data collection is made possible by the **sal** script, which is part of the **sar** toolset and must be set up to run from **cron** at periodic intervals. **sar** stores the data it collects underneath the **/var/log** directory in a binary format.

```
linux$ sar
Linux 4.4.0-66-generic (nuerbull)  03/19/17  _x86_64_  (4 CPU)
19:10:01   CPU   %user  %nice  %system  %iowait  %steal  %idle
19:12:01   all    0.02   0.00     0.01     0.00    0.00  99.97
19:14:01   all    0.01   0.00     0.01     0.00    0.00  99.98
```

In addition to CPU information, **sar** can also report on metrics such as disk and network activity. Use **sar -d** for a summary of this day's disk activity or **sar -n DEV** for network interface statistics. **sar -A** reports all available information.

See [this page] for more information about Grafana.

---

sar has some limitations, but it's a good bet for quick-and-dirty historical information. If you're serious about making a long-term commitment to performance monitoring, we suggest that you set up a data collection and graphing platform such as Grafana.

Choosing a Linux I/O scheduler


Linux systems use an I/O scheduling algorithm to mediate among processes competing to perform disk I/O. The I/O scheduler massages the order and timing of disk requests to achieve the best possible overall I/O performance for a given application or situation.

Three different scheduling algorithms are available in current Linux kernels:

- Completely Fair Queuing: This is the default algorithm and is usually the best choice for mechanical hard disks on general-purpose servers. It tries to evenly distribute access to I/O bandwidth. (If nothing else, the algorithm surely deserves an award for marketing: who could ever say no to a completely fair scheduler?)
- Deadline: This algorithm tries to minimize the latency for each request. It reorders requests to increase performance.
- NOOP: This algorithm implements a simple FIFO queue. It assumes that I/O requests have already been optimized or reordered by the driver, or that they will be optimized or reordered by the device (as might be done by an intelligent controller). This option may be the best choice in some SAN environments and is the best choice for SSD drives (because SSD drives don't have variable retrieval latencies).

You can view or set the algorithm in use for any particular device through the file **/sys/block/***disk***/queue/scheduler**. The active scheduler is enclosed in brackets.

```
$ cat /sys/block/sda/queue/scheduler
noop deadline [cfq]
$ sudo sh -c "echo noop > /sys/block/sda/queue/scheduler"
$ cat /sys/block/sda/queue/scheduler
[noop] deadline cfq
```

By determining which scheduling algorithm is most appropriate for your environment (you may need to run trials with each scheduler) you may be able to improve I/O performance.

See this page for more information about GRUB.

Unfortunately, the scheduling algorithm does not persist across reboots when set in this manner. You can set it for all devices at boot time with the **elevator**=*algorithm* kernel argument. That configuration is usually set in the GRUB boot loader's configuration file, **grub.conf**.

perf: profiling Linux systems in detail

Linux kernel versions 2.6 and higher include a perf_events interface that affords user-level access to the kernel's performance metric event stream. The **perf** command is a powerful, integrated system profiler that reads and analyzes information from this stream. All components of a system can be profiled: hardware, kernel modules, the kernel itself, shared libraries, and applications.



To get started with **perf**, you'll need to get a full set of the **linux-tools** packages:

```
$ sudo apt-get install linux-tools-common linux-tools-generic
    linux-tools-`uname -r`
```

Once you've installed the software, check out the tutorial at goo.gl/f88mt for examples and use cases. (This is a deep link into perf.wiki.kernel.org.)

The TL;DR path to getting started is to try **perf top**, which is a **top**-like display of system-wide CPU use. Of course, the simple example below only scratches the surface of **perf**'s capabilities.

```
$ sudo perf top
Samples: 161K of event 'cpu-clock', Event count (approx.): 21695432426
Overhead  Shared Object            Symbol
   4.63%  [kernel]                 [k] 0x00007fff8183d3b5
   2.15%  [kernel]                 [k] finish_task_switch
   2.04%  [kernel]                 [k] entry_SYSCALL_64_after_swapgs
   2.03%  [kernel]                 [k] str2hashbuf_signed
   2.00%  [kernel]                 [k] half_md4_transform
   1.44%  find                     [.] 0x0000000000016a01
   1.41%  [kernel]                 [k] ext4_htree_store_dirent
   1.21%  libc-2.23.so             [.] strlen
   1.19%  [kernel]                 [k] __d_lookup_rcu
   1.14%  find                     [.] 0x00000000000169f0
   1.12%  [kernel]                 [k] copy_user_generic_unrolled
   1.06%  [kernel]                 [k] kfree
   1.06%  [kernel]                 [k] _raw_spin_lock
   1.03%  find                     [.] 0x00000000000169fa
   1.01%  find                     [.] 0x0000000000016a05
   0.86%  find                     [.] fts_read
   0.73%  [kernel]                 [k] __kmalloc
   0.71%  [kernel]                 [k] ext4_readdir
   0.69%  libc-2.23.so             [.] malloc
   0.65%  libc-2.23.so             [.] fcntl
   0.64%  [kernel]                 [k] __ext4_check_dir_entry
```

The Overhead column shows the percentage of the time the CPU was in the corresponding function when it was sampled. The Shared Object column is the component (e.g., the kernel), shared library, or process in which the function resides, and the Symbol column is the name of the function (in cases where symbol information hasn't been stripped).

29.7 Help! My server just got really slow!

In previous sections, we've talked mostly about issues that relate to the average performance of a system. Solutions to these long-term concerns generally take the form of configuration adjustments or upgrades.

However, you will find that even properly configured systems are sometimes more sluggish than usual. Luckily, transient problems are often easy to diagnose. Most of the time, they are caused by a greedy process that is simply consuming so much CPU power, disk, or network bandwidth that other processes are affected. On occasion, malicious processes hog available resources to intentionally slow a system or network, a scheme known as a "denial of service" (DOS) attack.

The first step in diagnosis is to run **ps auxww** or **top** to look for obvious runaway processes. Any process that's using more than 50% of the CPU is likely to be at fault. If no single process is getting an inordinate share of the CPU, check to see how many processes are getting at least 10%. If you snag more than two or three (don't count **ps** itself), the load average is likely to be quite high. This is, in itself, a cause of poor performance. Check the load average with **uptime**, and use **vmstat** or **top** to check whether the CPU is ever idle.

If no CPU contention is evident, run **vmstat** to see how much paging is going on. All disk activity is interesting: a lot of page-outs may indicate contention for memory, and disk traffic in the absence of paging may mean that a process is monopolizing the disk by constantly reading or writing files.

There's no direct way to tie disk operations to processes, but **ps** can narrow down the possible suspects for you. Any process that is generating disk traffic must be using some amount of CPU time. You can usually make an educated guess about which of the active processes is the true culprit. Use **kill -STOP** to suspend the process and test your theory.

A large virtual address space or resident set used to be a suspicious sign, but shared libraries have made these numbers less useful. **ps** is not very smart about separating system-wide shared library overhead from the address spaces of individual processes. Many processes wrongly appear to have tens or hundreds of megabytes of active memory.

Suppose you do find that a particular process is at fault—what should you do? Usually, nothing. Some operations just require a lot of resources and are bound to slow down the system. It doesn't necessarily mean that they're illegitimate. It is sometimes useful, however, to **renice** an obtrusive process that is CPU-bound.

Sometimes, application tuning can dramatically reduce a program's demand for CPU resources; this effect is especially visible with custom network server software such as web applications.

Processes that are disk or memory hogs often can't be dealt with so easily. **renice** generally does not help. You do have the option of killing or stopping such processes, but we recommend

against this if the situation does not constitute an emergency. As with CPU pigs, you can use the low-tech solution of asking the owner to run the process later.


Linux has a handy option for dealing with processes that consume excessive disk bandwidth in the form of the **ionice** command. This command sets a process's I/O scheduling class; at least one of the available classes supports numeric I/O priorities (which can be set through **ionice** as well). The most useful invocation to remember is **ionice -c 3 -p** *pid*, which allows the named process to perform I/O only if no other processes wants to.

The kernel allows a process to restrict its own use of physical memory by calling the **setrlimit** system call. This facility is also available in the shell through the built-in **ulimit** command (**limits** on FreeBSD). For example, the command

```
$ ulimit -m 32000000
```

causes all subsequent commands that the user runs to have their use of physical memory limited to 32MB. This feature is roughly equivalent to **renice** for memory-bound processes. More granular resource management can be achieved through the Class-based Kernel Resource Management functionality; see ckrm.sourceforge.net.

If a runaway process doesn't seem to be the source of poor performance, investigate two other possible causes. The first is an overloaded network. Many programs are so intimately bound up with the network that it's hard to tell where system performance ends and network performance begins.

Some network overloading problems are hard to diagnose because they come and go very quickly. For example, if every machine on the network runs a network-related program out of **cron** at a particular time each day, there will often be a brief but dramatic glitch. Every machine on the net will hang for five seconds, and then the problem will disappear as quickly as it came.

Server-related delays are another possible cause of performance crises. UNIX and Linux systems are constantly consulting remote servers for NFS, Kerberos, DNS, and any of a dozen other facilities. If a server is dead or some other problem makes the server expensive to communicate with, the effects ripple back through client systems.

For example, on a busy system, some process may use the **gethostent** library routine every few seconds or so. If a DNS glitch makes this routine take two seconds to complete, you will likely perceive a difference in overall performance. DNS forward and reverse lookup configuration problems are responsible for a surprising number of server performance issues.

29.8 Recommended reading

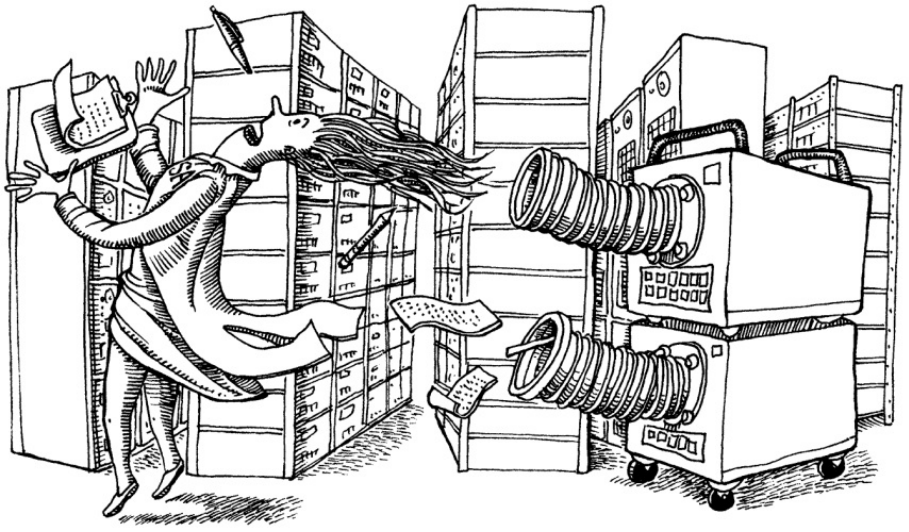
Drepper, Ulrich. *What Every Programmer Should Know about Memory*. You can find this article on-line at lwn.net/Articles/250967.

Ezolt, Phillip G. *Optimizing Linux Performance*. Upper Saddle River, NJ: Prentice Hall PTR, 2005.

Gregg, Brendan. *Systems Performance: Enterprise and the Cloud*. Upper Saddle River, NJ: Prentice Hall PTR, 2013.

Koziol, Prabhat, and Quincey Koziol. *High Performance Parallel I/O*. London: CRC Press, 2014.

30 Data Center Basics



A service is only as reliable as the data center that houses it. For those with hands-on experience, that's just common sense. (Of course, it's possible to distribute a service among multiple data centers, thereby limiting the impact of a failure in any one center.)

Proponents of cloud computing sometimes seem to imply that the cloud can magically break the chains that join the physical and virtual worlds. But although cloud providers offer a variety of services that help boost resilience and availability, every cloud resource ultimately lives somewhere in mundane reality.

Understanding where your data actually lives is an important part of being a system administrator. If you are involved in selecting third party cloud providers, evaluate vendors and their facilities quantitatively. You might also find yourself in positions where security, data sovereignty, or political concerns dictate that you build and maintain your own data center.

A data center is composed of

- A physically safe and secure space
- Racks that hold computer, networking, and storage devices
- Electric power (and standby power) sufficient to operate the installed devices
- Cooling, to keep the devices within their operating temperature ranges
- Network connectivity throughout the data center, and to places beyond (enterprise network, partners, vendors, Internet)

- On-site operational staff to support the equipment and infrastructure

Certain aspects of data centers—such as their physical layout, power, and cooling—were traditionally designed and maintained by “facilities” or “physical plant” staff. However, the fast-moving pace of IT technology and the increasingly low tolerance for downtime have forced a shotgun marriage of IT and facilities staff as partners in the planning and operation of data centers.

30.1 Racks

The days of the traditional raised-floor data center—in which power, cooling, network connections, and telecommunications lines are all hidden underneath the floor—are over. Have you ever tried to trace a cable that runs under the floor of one of these labyrinths? Our experience is that although it looks nice through glass, a "classic" raised-floor machine room is really just a hidden rat's nest. Today, you should use a raised floor to hide electrical power feeds, to distribute cooled air, and for *nothing else*. Network cabling (both copper and fiber) should be routed through overhead raceways designed specifically for this purpose. (These days, electrical feeds are often found overhead as well.)

In a dedicated data center, storing equipment in racks (as opposed to, say, setting it on tables or on the floor) is the only maintainable, professional choice. The best storage schemes use racks that are interconnected with an overhead track system through which cables can be routed. This approach confers that irresistible high-tech feel without sacrificing organization or maintainability.

The best overhead track system that we know of is manufactured by Chatsworth Products. You can construct homes for both shelf-mounted and rack-mounted servers from standard 19" single-rail telco racks. Two back-to-back 19" telco racks make a high-tech-looking "traditional" rack for situations in which you need to attach rack hardware both in front of and in back of equipment. Chatsworth provides the racks, cable races, and cable management doodads, as well as all the hardware necessary to mount them in your building. Since the cables lie in visible tracks, they are easy to trace and you will naturally be motivated to keep them tidy.

30.2 Power

Several strategies may be needed to provide a data center with clean, stable, fault-tolerant power. Common options include

- **Uninterruptible power supplies (UPSs)** – UPSs provide power when the normal long-term power source (e.g., the commercial power grid) becomes unavailable. Depending on size and capacity, a UPS can provide anywhere from a few minutes to a couple of hours of power. UPSs alone cannot support a site in the event of a long-term outage.
- **On-site power generation** – If the commercial grid is unavailable, on-site standby generators can provide long-term power. Generators are usually fueled by diesel, LP gas, or natural gas and can support the site as long as fuel is available. It is customary to store at least 72 hours of fuel on-site and to arrange to buy fuel from multiple providers.
- **Redundant power feeds** – In some locations, it may be possible to obtain more than one power feed from the commercial power grid (possibly from different power generators).

See this page for more information about shutdown procedures.

In all cases, servers and network infrastructure equipment should at least be put on uninterruptible power supplies. Good UPSs have an Ethernet or USB interface that can be attached either to the machine to which they supply power or to a centralized monitoring infrastructure that can elicit a higher-level response. Such connections let the UPS warn computers or operators that power has failed and that a clean shutdown should be performed before the batteries run out.

UPSs are available in various sizes and capacities, but even the largest ones cannot provide long-term backup power. If your facility must operate on standby power for longer than a UPS can handle, you need a local generator in addition to a UPS.

A large selection of standby power generators is available, ranging in capacity from 5 kW to more than 2,500 kW. The gold standard is the family of generators made by Cummins Onan (power.cummins.com). Most organizations select diesel as their fuel type. If you're in a cold climate, make sure you fill the tank with "winter mix diesel" or substitute Jet A-1 aircraft fuel to prevent gelling. Diesel is chemically stable but can grow algae, so consider adding an algicide to diesel that you plan to store for an extended period.

Generators and the infrastructure to support them are expensive, but they can save money in some ways, too. If you install a standby generator, your UPSs need only be large enough to cover

the short gap between the power going out and your generator coming on-line.

If UPSs or generators are part of your power strategy, it is extremely important to have a periodic test plan in place. We recommend that you test all components of your standby power system at least every 6 months. In addition, you (or your vendor) should perform preventive maintenance on standby power components at least annually.

Rack power requirements

Planning the power for a data center is a difficult challenge. Typically, the opportunity to build a new data center (or to significantly remodel an existing one) comes up only every decade or so. So it's important to look far down the road when planning power systems.

Most architects are biased toward calculating the amount of power needed in a data center by multiplying the center's square footage by a magic number. This approach proves to be ineffective in most real-world cases because the size of the data center alone tells you little about the types of equipment it might eventually house. Our recommendation is to use a per-rack power consumption model and to ignore the amount of floor space.

Historically, data centers have been designed to provide between 1.5 kW and 3 kW to each rack. But now that server manufacturers have started squeezing servers into 1U of rack space and building blade server chassis that hold 20 or more blades, the power needed to support a full rack of modern gear has skyrocketed.

One approach to solving the power density problem is to put only a handful of 1U servers in each rack, leaving the rest of the rack empty. Although this technique eliminates the need to provide more power to the rack, it's a prodigious waste of space. A better strategy is to develop a realistic projection of the power that might be needed by each rack and to provision power accordingly.

Equipment varies in its power requirements, and it's hard to predict exactly what the future will hold. A good approach is to create a system of power consumption tiers that allocates the same amount of power to all racks in a particular tier. This scheme is useful not only for meeting current equipment needs but also for planning future use. Table 30.1 outlines some basic starting points for tier definitions.

Table 30.1: Power-tier estimates for racks in a data center

Power tier	Watts/rack
Insanely high density or "custom"	40 kW
Ultra high density	25 kW
Very high density (e.g., blade servers)	20 kW
High density (e.g., 1U servers)	16 kW
Storage equipment	12 kW
Network switching equipment	8 kW
Normal density	6 kW

Once you've defined your power tiers, estimate your need for racks in each tier. On the floor plan, put racks from the same tier together. Such zoning concentrates the high-power racks and lets you plan cooling resources accordingly.

kVA vs. kW

One of the many common disconnects between IT folks, facilities folks, and UPS engineers is that each of these groups uses different units for power. The amount of power a UPS can provide is typically labeled in kVA (kilovolt-amperes). But computer equipment and the electrical engineers that support your data center usually express power in watts (W) or kilowatts (kW). You might remember from fourth grade science class that watts = volts × amps. Unfortunately, your fourth grade teacher probably failed to mention that watts is a vector value, which for AC power includes a "power factor" (pf) in addition to volts and amps.

If you are designing a bottle-filling line at a brewery that involves lots of large motors and other heavy equipment, ignore this section and hire a qualified engineer to determine the correct power factor for use in your calculations. But for modern-day computer equipment, you can cheat and use a constant for a "probably good enough" conversion between kVA and kW:

kVA = kW / 0.85

See this page for some additional tips on measuring power consumption.

A final point to note is that when estimating the amount of power you need in a data center (or to size a UPS), you should measure devices' actual power consumption rather than relying on manufacturers' stated values as shown on equipment labels. Label values typically represent the maximum possible power consumption and are therefore misleading.

Energy efficiency

Energy efficiency has become a popular operational metric for evaluating data centers. The industry has standardized on a simple ratio known as the power usage effectiveness (PUE) as a way of expressing a plant's overall efficiency:

$$\text{PUE} = \frac{\text{Total power consumed by facility}}{\text{Total power consumed by IT equipment}}$$

A hypothetically perfect data center would have a PUE of 1.0; that is, it would consume exactly the amount of power needed by IT gear, with no overhead. Of course, this goal is unreachable in practical terms. Equipment generates heat that must be removed, human operators need lighting and other environmental accommodations, etc. The higher the PUE value, the less energy efficient (and more expensive) a data center is to operate.

Modern-day data centers that are reasonably energy efficient generally have a PUE ratio of 1.4 or less. For reference, data centers from a decade ago typically had PUE ratios in the 2.0–3.0 range. Google, which has made energy efficiency a focus, regularly publishes its PUE ratios and as of 2016 has achieved an average PUE of 1.12 across its data centers.

Metering

You get what you measure. If you are serious about energy efficiency, it's important to understand which devices are actually consuming the most energy. Although the PUE ratio gives you a general impression of the amount of energy consumed as non-IT overhead, it says very little about the power efficiency of the actual servers. (In fact, replacing servers with more power-efficient models will increase the PUE rather than decreasing it.)

It's up to the data center administrator to select components that use the minimum amount of energy. One obvious enabling technology is power consumption metering at the aisle, rack, and device level. Select or build data centers that can easily provide this critical usage data.

Cost

Once upon a time, the cost of power was more or less the same across data centers in different locations. These days, the hyperscale cloud industry (Amazon, Google, Microsoft, and others) sends data center designers hunting for potential cost efficiencies in every corner of the world. One successful strategy has been to locate large data centers near sources of inexpensive power such as hydroelectric power plants.

When deciding whether to operate your own data center, be sure to factor the cost of power into your assessment. Chances are that the big guys have a built-in cost advantage in this aspect of operations (and others). Widespread fiber and bandwidth availability have largely rendered obsolete the traditional advice to locate your data center near your team.

Remote control

You might occasionally find yourself needing to regularly power-cycle a server because of a kernel or hardware glitch. (Or, perhaps you have non-Linux servers in your data center that are more prone to this type of problem.) In either case, you can consider installing a system that lets you power-cycle problem servers by remote control.

A reasonable family of solutions is manufactured by American Power Conversion (APC). Their remotely manageable products are conceptually similar to power strips, except that they can be controlled by a web browser that reaches the power distribution unit through a built-in Ethernet port.

30.3 Cooling and environment

Just like humans, computers work better and live longer if they're happy in their environment. Maintenance of a safe operating temperature is a prerequisite for this happiness.

The American Society of Heating, Refrigerating and Air-conditioning Engineers (ASHRAE) traditionally recommended data center temperatures (measured at server inlets) in the range of 68° to 77°F (20° to 25°C). To support organizations' attempts to reduce energy consumption, ASHRAE released guidance in 2012 that suggests a more lenient temperature range to 64.4° to 80.6°F (18° to 27°C). Although this range seems unhelpfully broad, it does suggest that today's hardware can flourish in a wide range of environments.

Cooling load estimation

Temperature maintenance starts with an accurate estimate of your cooling load. Traditional textbook models for data center cooling (even those from the 2000s) can be off from the realities of today's high-density blade server chassis by up to an order of magnitude. Hence, we have found that it's a good idea to double-check the cooling load estimates produced by your HVAC folks.

You need to determine the heat load contributed by the following components:

- Roof, walls, and windows
- Electronic gear
- Light fixtures
- Operators (people)

Of these, only the first should be left to your HVAC folks. The other components can be assessed by the HVAC team, but you should do your own calculations as well. Make sure that any discrepancies between your results and those of the HVAC team are fully explained before construction starts.

Roof, walls, and windows

Your roof, walls, and windows (don't forget solar load) contribute to your environment's cooling load. HVAC engineers usually have a lot of experience with these elements and should be able to give you good estimates.

Electronic gear

You can estimate the heat load produced by your servers (and other electronic gear) by determining their power consumption. In practical terms, all electric power that is consumed eventually ends up as heat.

As when planning for power-handling capacity, direct measurement of power consumption is by far the best way to obtain this information. Your friendly neighborhood electrician can help, or you can purchase an inexpensive meter and do it yourself. The Kill A Watt meter made by P3 is a popular choice at around $20, but it's limited to small loads (15 amps) that plug in to a standard wall outlet. For larger loads or nonstandard connectors, use a clamp-on ammeter such as the Fluke 902 (also known as a "current clamp") to make these measurements.

Most equipment is labeled with its maximum power consumption in watts. However, typical consumption tends to be significantly less than the maximum.

You can convert power consumption to the standard heat unit, BTUH (British thermal units per hour), by multiplying by 3.413 BTUH/watt. For example, if you wanted to build a data center that would house 25 servers rated at 450 watts each, the calculation would be

$$\left(25 \text{ servers}\right)\left(\frac{450 \text{ watts}}{\text{server}}\right)\left(\frac{3.412 \text{ BTUH}}{\text{watt}}\right) = 38{,}385 \text{ BTUH}$$

Light fixtures

As with electronic gear, you can estimate light fixture heat load from power consumption. Typical office light fixtures contain four 40-watt fluorescent tubes. If your new data center had six of these fixtures, the calculation would be

$$\left(6 \text{ fixtures}\right)\left(\frac{160 \text{ watts}}{\text{fixture}}\right)\left(\frac{3.412 \text{ BTUH}}{\text{watt}}\right) = 3{,}276 \text{ BTUH}$$

Operators

At one time or another, humans will need to enter the data center to service something. Allow 300 BTUH for each occupant. To allow for four humans in the data center at the same time:

$$\left(4 \text{ humans}\right)\left(\frac{300 \text{ BTUH}}{\text{human}}\right) = 1{,}200 \text{ BTUH}$$

Total heat load

Once you have calculated the heat load for each component, sum the results to determine your total heat load. For this example, let's assume that our HVAC engineer estimated the load from the roof, walls, and windows to be 20,000 BTUH.

20,000	BTUH for roof, walls, and windows
38,385	BTUH for servers and other electronic gear
3,276	BTUH for light fixtures
1,200	BTUH for operators
62,861	BTUH total

Cooling system capacity is typically expressed in tons. You can convert BTUH to tons by dividing by 12,000 BTUH/ton. You should also allow at least a 50% slop factor to account for errors and future growth.

$$\left(62{,}681 \text{ BTUH}\right)\left(\frac{1 \text{ ton}}{12{,}000 \text{ BTUH}}\right)\left(1.5\right) = 7.86 \text{ tons of cooling required}$$

See how your estimate matches up with the one from your HVAC folks.

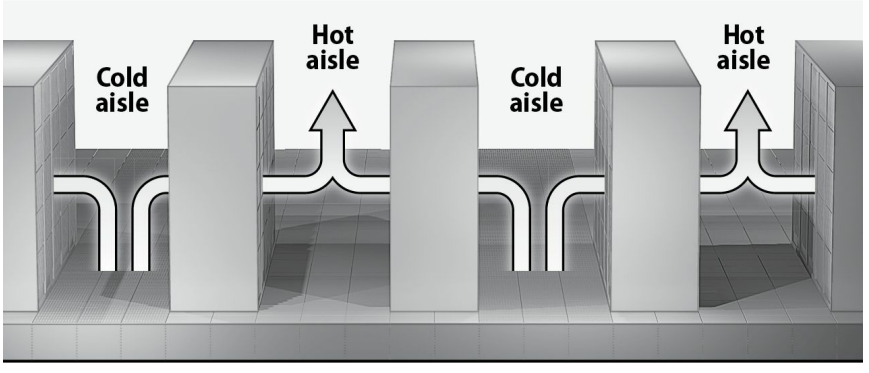
Hot aisles and cold aisles

You can dramatically reduce your data center's cooling difficulties by putting some thought into its physical layout. The most common and effective strategy is to alternate hot and cold aisles.

Facilities that have a raised floor and are cooled by a traditional CRAC (computer room air conditioner) unit are often set up so that cool air enters the space under the floor, rises up through holes in the perforated floor tiles, cools the equipment, and then rises to the top of the room as warm air, where it is sucked into return air ducts. Traditionally, racks and perforated tiles have been placed "randomly" about the data center, a configuration that results in relatively even temperature distribution. The result is an environment that is comfortable for humans but not really optimized for computers.

A better strategy is to lay out alternating hot and cold aisles between racks. Cold aisles have perforated cooling tiles and hot aisles do not. Racks are arranged so that equipment draws in air from a cold aisle and exhausts it to a hot aisle; the exhaust sides of two adjacent racks are therefore back to back. Exhibit A illustrates this basic concept.

Exhibit A: Hot and cold aisles, raised floor

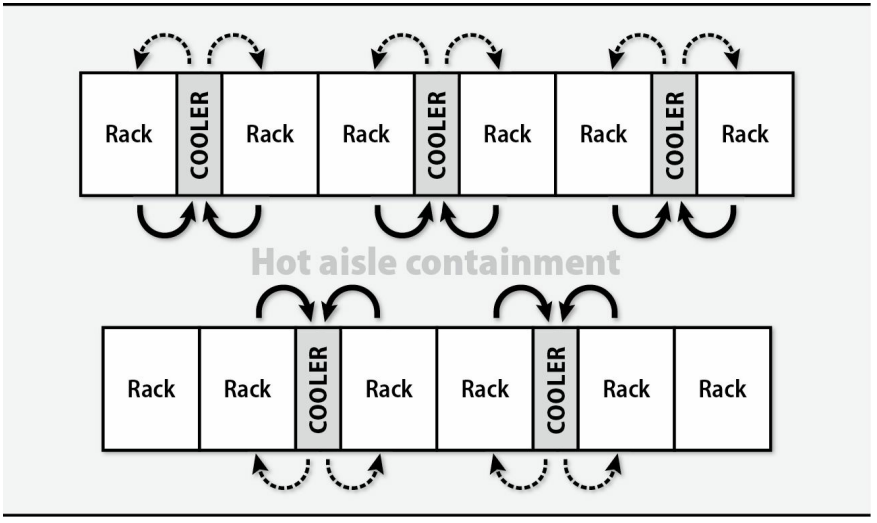


This arrangement optimizes the flow of cooling so that air inlets always breathe cool air rather than another server's hot exhaust. Properly implemented, the alternating row strategy results in aisles that are noticeably cold and hot. You can measure your cooling success with an infrared thermometer such as the Fluke 62, which is an indispensable tool of the modern system administrator. This point-and-shoot $100 device instantly measures the temperature of anything you aim it at, up to six feet away. Don't take it out to the bars.

If you *must* run cabling under the floor, run power under cold aisles and network cabling under hot aisles.

Facilities without a raised floor can use in-row cooling units such as those manufactured by APC (apc.com). These units are skinny and sit between racks. Exhibit B shows how this system works.

Exhibit B: Hot and cold aisles with in-row cooling (bird's-eye view)



Both CRAC and in-row cooling units need a way to dissipate heat outside the data center. This requirement is typically satisfied with a loop of liquid refrigerant (such as chilled water, Puron/R410A, or R22) that carries the heat outdoors. We omitted the refrigerant loops from Exhibit A and Exhibit B for simplicity, but most installations will require them.

Humidity

According to the 2012 ASHRAE guidelines, data center humidity should be kept between 8% and 60%. If the humidity is too low, static electricity becomes a problem. Recent testing has shown that there is little operational difference between 8% and the previous standard of 25%, so the minimum humidity standard was adjusted accordingly.

If humidity is too high, condensation can form on circuit boards and cause short circuits and oxidation.

Depending on your geographic location, you might need either humidification or dehumidification equipment to maintain a proper level of humidity.

Environmental monitoring

If you are supporting a mission-critical computing environment, it's a good idea to monitor the temperature (and other environmental factors, such as noise and power) in the data center even when you are not there. It can be disappointing to arrive on Monday morning and find a pool of melted plastic on your data center floor.

Fortunately, automated data center monitors can watch the goods while you are away. We use and recommend the Sensaphone product family. These inexpensive boxes monitor environmental variables such as temperature, noise, and power, and they phone or text you when they detect a problem.

30.4 Data center reliability tiers

The Uptime Institute is a commercial entity that certifies data centers. They have developed a four-tier system for classifying the reliability of data centers, which we summarize in Table 30.2. In this table, N means that you have just enough of something (e.g., UPSs or generators) to meet normal needs. N+1 means that you have one spare; 2N means that each device has its own spare.

Table 30.2: Uptime Institute availability classification system

Tier	Generators	UPSs	Power feeds	HVAC	Availability
1	None	N	Single	N	99.671%
2	N	N+1 [a]	Single	N+1	99.741%
3	N+1	N+1 [a]	Dual, switchable	N+1	99.982%
4	2N	2N	Dual, simultaneous	2N	99.995%

a. With redundant components

Centers in the highest tier must be "compartmentalized," which means that groups of systems are powered and cooled in such a way that the failure of one group has no effect on other groups.

Even 99.671% availability may look pretty good at first glance, but it works out to nearly 29 hours of downtime per year. 99.995% availability corresponds to 26 minutes of downtime per year.

Of course, no amount of redundant power or cooling will keep an application available if it's administered poorly or is improperly architected. The data center is a foundational building block, necessary but not sufficient to ensure overall availability from the end user's perspective.

You can learn more about the Uptime Institute's certification standards (which include certification of design, construction, and operational phases) from their web site, uptimeinstitute.org. In some cases, organizations use the concept of these tiers without paying the Uptime Institute's hefty certification fees. The important part is not the framed plaque but the use of a common vocabulary and assessment methodology to compare data centers.

30.5 Data center security

Perhaps it goes without saying, but the physical security of a data center is at least as important as its environmental attributes. Make sure that threats of both natural (e.g., fire, flood, earthquake) and human (e.g., competitors and criminals) origin have been carefully considered. A layered approach to security is the best way to ensure that a single mistake or lapse will not lead to a catastrophic outcome.

Location

Whenever possible, data centers should not be located in areas that are prone to forest fires, tornadoes, hurricanes, earthquakes, or floods. For similar reasons, it's advisable to avoid man-made hazard zones such as airports, freeways, refineries, and tank farms.

Because the data center you select (or build) will likely be your home for a long time, it's worthwhile to invest some time in researching the available risk data when making a site selection. The U.S. Geological Survey (usgs.gov) publishes statistics such as earthquake probability, and the Uptime Institute produces a composite map of data center location risks.

Perimeter

To reduce the risk of a targeted attack, a data center should be surrounded by a fence that is at least 25 feet from the building on all sides. Access to the inside of the fence perimeter should be controlled by a security guard or a multifactor badge access system. Vehicles allowed within the fence perimeter should not be permitted within 25 feet of the building.

Continuous video monitoring must cover 100% of the external perimeter, including all gates, access driveways, parking lots, and roofs.

Buildings should be unmarked. No signage should indicate what company the building belongs to or mention that it houses a data center.

Facility access

Access to the data center itself should be controlled by a security guard and a multifactor badge system, possibly one that incorporates biometric factors. Ideally, authorized parties should be enrolled in the physical access-control system before their first visit to the data center. If this is not possible, on-site security guards should follow a vetting process that includes confirming each individual's identity and authorized actions.

One of the trickiest situations in training security guards is properly handling the appearance of "vendors" who claim they have come to fix some part of the infrastructure, such as the air conditioning. Make no mistake: unless the guard can confirm that someone authorized or requested this vendor visit, such visitors must be turned away.

Rack access

Large data centers are often shared with other parties. This is a cost-effective approach, but it comes with the added responsibility of securing each rack (or “cage of racks”). This is another case in which a multifactor access control system (such as a card reader plus a fingerprint reader) is needed to ensure that only authorized parties have access to your equipment. Each rack should also be individually monitored by video.

30.6 Tools

A well-outfitted sysadmin is an effective sysadmin. Having a dedicated tool box is an important key to minimizing downtime in an emergency. Table 30.3 lists some items to keep in your tool box, or at least within easy reach.

Table 30.3: A system administrator's tool box

General tools	
Hex (Allen) wrench kit	Ball-peen hammer, 4 oz.
Scissors	Electrician's knife or Swiss Army knife
Small LED flashlight	Phillips-head screwdrivers: #0, #1, and #2
Socket wrench kit	Pliers, both flat-needlenose and regular
Stud finder	Ridgid SeeSnake micro inspection camera
Tape measure	Slot-head screwdrivers: 1/8", 3/16", and 5/16"
Torx wrench kit	Teensy tiny jeweler's screwdrivers
Tweezers	
Computer-related specialty items	
PC screw kit	Cable ties (and their Velcro cousins)
Infrared thermometer	Digital multimeter (DMM)
RJ-45 end crimper	Portable network analyzer/laptop
SCSI terminators	Spare Category 5 and 6A RJ-45 crossover cables
Spare power cord	Spare RJ-45 connectors (solid core and stranded)
Static grounding strap	Wire stripper (with an integrated wire cutter)
Miscellaneous	
Q-Tips	Telescoping magnetic pickup wand
Cellular telephone	First-aid kit, including ibuprofen and acetaminophen
Electrical tape	Home phone and pager #s of on-call support staff
Can of compressed air	List of emergency maintenance contacts[a]
Dentist's mirror	Six-pack of good microbrew beer (suggested minimum)

a. And maintenance contract numbers, if applicable

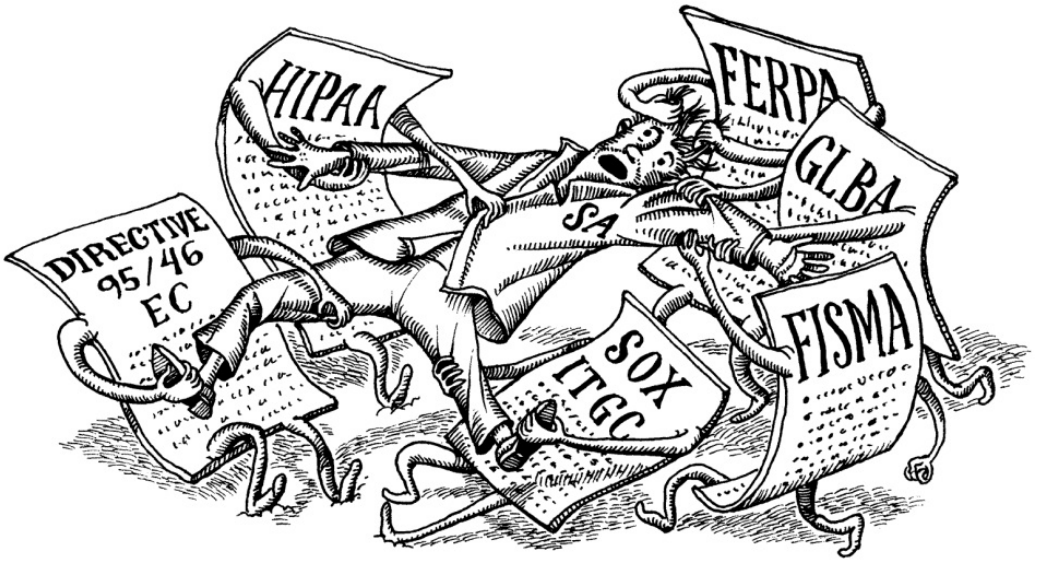
30.7 Recommended reading

ASHRAE, Inc. *ASHRAE Thermal Guidelines for Data Processing Environments (3rd edition)*. Atlanta, GA: ASHRAE, Inc., 2012.

Telecommunications Infrastructure Standard for Data Centers. ANSI/TIA/EIA 942.

A variety of useful information and standards related to energy efficiency can be found at the Center of Expertise for Energy Efficiency in Data Centers web site at datacenters.lbl.gov.

31 Methodology, Policy, and Politics



During the past four decades, the role of information technology in business and daily life has changed dramatically. It's hard to imagine a world without the instant gratification of Internet search.

For most of this period, the predominant philosophy of IT management was to increase stability by minimizing change. In many cases, hundreds or thousands of users depended on a single system. If a failure occurred, hardware often had to be express shipped for repair, or hours of downtime were needed to reinstall software and restore state. IT teams lived in fear that something would break and that they wouldn't be able to fix it.

Change minimization has undesirable side effects. IT departments often became stuck in the past and failed to keep pace with business needs. "Technical debt" accumulated in the form of systems and applications in desperate need of upgrade or replacement that everyone was afraid to touch for fear of breaking something. IT staff became the butt of jokes and the least popular folks everywhere from board rooms to holiday parties.

Thankfully, those times are behind us. The advents of cloud infrastructure, virtualization, automation tools, and broadband communication have greatly reduced the need for one-off systems. Such servers have been replaced by armies of clones that are managed as battalions. In turn, these technical factors have enabled the evolution of a service philosophy known as DevOps which lets IT organizations drive and encourage change rather than resisting it. The DevOps name is a portmanteau of development and operations, the two traditional disciplines it combines.

An IT organization is more than a group of technical folks who set up Wi-Fi hot spots and computers. From a strategic perspective, IT is a collection of people and roles that use technology to accelerate and support the organization's mission. Never forget the golden rule of system administration: enterprise needs drive IT activities, not the other way around.

In this chapter, we discuss the nontechnical aspects of running a successful IT organization that uses DevOps as its overarching schema. Most of the topics and ideas presented in this chapter are not specific to any particular environment. They apply equally to a part-time system administrator or to a large group of full-time professionals. Like green vegetables, they're good for you no matter what size meal you're preparing.

31.1 The grand unified theory: DevOps

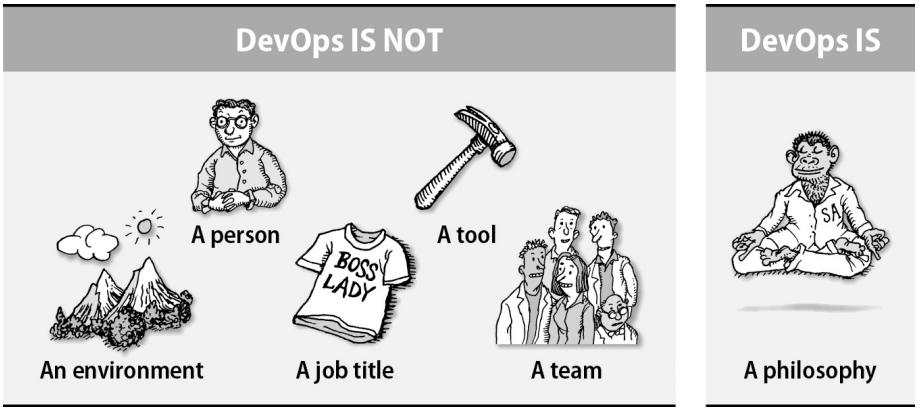
System administration and other operational roles within IT have traditionally been separate from domains such as application development and project management. The theory was that app developers were specialists who would push products forward with new features and enhancements. Meanwhile, the stolid and change-resistant operations team would provide 24 × 7 management of the production environment. This arrangement usually creates tremendous internal conflict and ultimately fails to meet the needs of the business and its clients.

Exhibit A: Courtesy of Dave Roth



The DevOps approach mingles developers (programmers, application analysts, application owners, project managers) with IT operations staff (system and network administrators, security monitors, data center staff, database administrators) in a tightly integrated way. This philosophy is rooted in the belief that working together as a collaborative team breaks down barriers, reduces finger pointing, and produces better results. Exhibit B summarizes a few of the main concepts.

Exhibit B: What is DevOps?



DevOps is a relatively new development in IT management. The early 2000s brought change to the development side of the house, which moved from “waterfall” release cycles to agile approaches that featured iterative development. This system increased the speed at which products, features, and fixes could be created, but deployment of those enhancements often stalled because the operations side wasn’t prepared to move as quickly as the development side. Hitching up the development and operations groups allowed everyone to accelerate down the road at the same pace, and DevOps was born.

DevOps is CLAMS

The tenets of DevOps philosophy are most easily described with the acronym CLAMS: Culture, Lean, Automation, Measurement, and Sharing.

Culture

People are the ultimate drivers of any successful team, so the cultural aspects of DevOps are the most important. Although DevOps has its own canon of cultural tips and tricks, the main goal is to get everyone working together and focused on the overall picture.

Under DevOps, all disciplines work together to support a common business driver (product, objective, community, etc.) through all phases of its life cycle. Achieving this goal may ultimately require changes in reporting structure (no more isolated application development groups), seating layout, and even job responsibilities. These days, good system administrators occasionally write code (often automation or deployment scripts), and good application developers regularly examine and manage infrastructure performance metrics.

Here are some typical features of a DevOps culture:

- Both developers (Dev) and operations (Ops) have 24 × 7, simultaneous ("everyone gets paged"), on-call responsibility for the complete environment. This rule has the wonderful side effect that root causes can be addressed wherever they occur. (The first six weeks or so of a shared on-call model is painful. Then suddenly it turns around. Trust us.)
- No application or service can launch without automated testing and monitoring being in place at both the system and application level. This rule seals in functionality and creates a contract between Dev and Ops. Likewise, Dev and Ops must sign off on any launch before it happens.
- All production environments are mirrored by identical development environments. This rule creates a runway for testing and reduces accidents in production.
- Dev teams do regular code reviews to which Ops is invited. Code architecture and functionality are no longer just Dev functions. Likewise, Ops performs regular infrastructure reviews in which Dev is involved. Dev must be aware of—and contribute to—decisions about underlying infrastructure.
- Dev and Ops have regular, joint stand-up meetings. In general, meetings should be minimized, but joint stand-ups serve as a useful stopgap to foster communication.
- Dev and Ops should all sit in a common chat room dedicated to discussion of both strategic (architecture, direction, sizing) and operational issues. This communication channel is often known as ChatOps, and several amazing platforms are available to support it. Check out HipChat, Slack, MatterMost, and Zulip, to name a few.

A successful DevOps culture pushes Dev and Ops so close that their scopes interpenetrate, and everyone learns to be comfortable with that. The optimal level of overlap is probably higher than most people would naturally prefer in the absence of cultural indoctrination. Team members must learn to respond gracefully to queries and feedback about their work from colleagues who may be formally trained in other disciplines.

Lean

The easiest way to explain the lean aspect of DevOps is to note that if you schedule a recurring weekly meeting at your organization to discuss your DevOps implementation plan, you have instantly failed.

DevOps is about real-time interaction and communication among people, processes, and systems. Use real-time tools (like ChatOps) to communicate wherever possible, and focus on solving component problems one at a time. Always ask "what can we do *today*" to make progress on an issue. Avoid the temptation to boil the ocean.

Automation

Automation is the most universally recognized aspect of DevOps. The two golden rules of automation are these:

- If you need to perform a task more than twice, it should be automated.
- Don't automate what you don't understand.

Automation brings many advantages:

- It prevents staff from being trapped performing mundane tasks. Staff brainpower and creativity can be used to solve new and more difficult challenges.
- It reduces the risk of human error.
- It captures infrastructure in the form of code, allowing versions and outcomes to be tracked.
- It facilitates evolution while also reducing risk. If a change fails, automated rollback is (well, should be) easy.
- It facilitates the use of virtualized or cloud resources to achieve scale and redundancy. Need more? Spin some up. Need less? Kill them off.

Tools are instrumental in the quest for automation. Systems such as Ansible, Salt, Puppet, and Chef (covered in Chapter 23, *Configuration Management*) are front and center. Continuous integration tools such as Jenkins and Bamboo (see this page) help manage repeatable or triggered tasks. Packaging and launch utilities such as Packer and Terraform automate low-level infrastructure tasks.

Depending on your environment, you might need one, some, or all (!?) of these tools. New tools and enhancements are being developed rapidly, so focus on finding the tool that is a good fit for the particular function or process you are automating, as opposed to picking a tool and then looking for the questions it answers. Most importantly, reevaluate your tool set every year or two.

Your automation strategy should include at least the following elements:

- **Automated setup of new machines:** This is not just OS installation. It also includes all the additional software and local configuration necessary to allow a machine to enter production. It's inevitable that your site will need to support more than one type of configuration, so include multiple machine types in your plans from the beginning.
- **Automated configuration management:** Configuration changes should be entered in the configuration base and applied automatically to all machines of the same type. This rule helps keep the environment consistent.
- **Automated promotion of code:** Propagation of new functionality from the development environment to the test environment, and from the test environment into production, should be automated. Testing itself should be automated, with clear criteria for evaluation and promotion.
- **Systematic patching and updating of existing machines:** When you identify a problem with your setup, you need a standardized and easy way to deploy updates to all affected machines. Because servers are not turned on all the time (even if they are supposed to be), your update scheme must correctly handle machines that are not on-line when an update is initiated. You can check for updates at boot time or update on a regular schedule; see *Periodic processes* for more information.

Measurement

See Chapter 28 for more information about monitoring.

The ability to scale virtualized or cloud infrastructure (see Chapter 9, *Cloud Computing*) has pushed the world of instrumentation and measurement to new heights. Today's gold standard is the collection of sub-second measurements throughout the entire service stack (business, application, database, subsystems, servers, network, and so on). Several DevOps-y tools such as Graphite, Grafana, ELK (the Elasticsearch + Logstash + Kibana stack), plus monitoring platforms like Icinga and Zenoss, support these efforts.

Having measurement data and doing something useful with it are two different things, however. A mature DevOps shop ensures that metrics from the environment are visible and evangelized to all interested parties (both inside and outside of IT). DevOps sets nominal targets for each metric and chases down any anomalies to determine their cause.

Sharing

Collaborative work and shared development of capabilities lie at the heart of a successful DevOps effort. Staff should be encouraged and incentivized to share their work both internally (lunch-and-learn presentations, team show-and-tell, wiki how-to articles) and externally (Meetups, white papers, conferences). These efforts extend the silo-busting philosophy beyond the local workgroup and help everyone learn and grow.

System administration in a DevOps world

System administrators have always been the jacks-and-jills-of-all-trades of the IT world, and that remains true under the broader DevOps umbrella. The system administrator role oversees systems and infrastructure, typically including primary responsibility for these areas:

- Building, configuring, automating, and deploying system infrastructure
- Ensuring that the operating system and major subsystems are secure, patched, and up to date
- Deploying, supporting, and evangelizing DevOps technologies for continuous integration, continuous deployment, monitoring, measurement, containerization, virtualization, and ChatOps platforms
- Coaching other team members on infrastructure and security best practices
- Monitoring and maintaining infrastructure (physical, virtual, or cloud) to ensure that it meets performance and availability requirements
- Responding to user resource or enhancement requests
- Fixing problems with systems and infrastructure as they occur
- Planning for the future expansion of systems, infrastructure, and capacity
- Advocating cooperative interactions among team members
- Managing various outside vendors (cloud, co-location, disaster recovery, data retention, connectivity, physical plant, hardware service)
- Managing the life cycle of infrastructure components
- Maintaining an emergency stash of ibuprofen, tequila, and/or chocolate to be shared with other team members on those not-as-fresh days

This is just a subset of the breadth covered by a successful system administrator. The role is part drill sergeant, part mother hen, part EMT, and part glue that keeps everything running smoothly.

Above all, remember that DevOps is founded on overcoming one’s normal territorial impulses. If you find yourself at war with other team members, take a step back and remember that you are most effective if you are seen as a hero who helps make everyone else successful.

31.2 Ticketing and task management systems

A ticketing and task management system lies at the heart of every functioning IT group. As with all things DevOps, having one ticketing system that spans all IT disciplines is critical. In particular, enhancement requests, issue management, and software bug tracking should all be part of the same system.

A good ticketing system helps staff avoid two of the most common workflow pitfalls:

- Tasks that fall through the cracks because everyone thinks they are being taken care of by someone else
- Resources that are wasted through duplication of effort when multiple people or groups work on the same problem without coordination

Common functions of ticketing systems

A ticket system accepts requests through various interfaces (email and web being the most common) and tracks them from submission to solution. Managers can assign tickets to groups or to individual staff members. Staff can query the system to see the queue of pending tickets and perhaps resolve some of them. Users can find out the status of requests and see who is working on them. Managers can extract high-level information such as

- The number of open tickets
- The average time to close a ticket
- The productivity of staff members
- The percentage of unresolved (rotting) tickets
- Workload distribution by time to solution

The request history stored in the ticket system becomes a history of the problems with your IT infrastructure and also the solutions to those problems. If that history is easily searchable, it becomes an invaluable resource for the sysadmin staff.

Resolved tickets can be provided to novice staff members and trainees, inserted into a FAQ system, or made searchable for later discovery. New staff members can benefit from receiving copies of closed tickets because those tickets include not only technical information but also examples of the tone and communication style that are appropriate for use with customers.

Like all documents, your ticketing system's historical data can potentially be used against your organization in court. Follow the document retention guidelines set up by your legal department.

Most request tracking systems automatically confirm new requests and assign them a tracking number that submitters can use to follow up or inquire about the request's status. The automated response message should clearly state that it is just a confirmation. It should be followed promptly by a message from a real person that explains the plan for dealing with the problem or request.

Ticket ownership

Work can be shared, but in our experience, responsibility is less amenable to distribution. Every task should have a single, well-defined owner. That person need not be a supervisor or manager, just someone willing to act as a coordinator—someone willing to say, "I take responsibility for making sure this task gets done."

An important side effect of this approach is that it is implicitly clear who implemented what or who made what changes. This transparency becomes important if you want to figure out why something was done in a certain way or why something is suddenly working differently or not working anymore.

Being "responsible" for a task should not equate to being a scapegoat if problems arise. If your organization defines responsibility as blameworthiness, you may find that the number of available project owners quickly dwindles. Your goal in assigning ownership is simply to remove ambiguity about who should be addressing each problem. Don't punish staff members for requesting help.

From a customer's point of view, a good assignment system is one that routes problems to a person who is knowledgeable and can solve them quickly and completely. But from a managerial perspective, assignments must occasionally be challenging to the assignee so that staff members continue to grow and learn in the course of doing their jobs. Your task is to balance reliance on staff members' strengths against the need to challenge them, all while keeping both customers and staff members happy.

Larger tasks can be anything up to and including full-blown software engineering projects. These tasks may require the use of formal project management and software engineering tools. We don't describe those tools here; nevertheless, they're important and should not be overlooked.

Sometimes sysadmins know that a particular task needs to be done, but they don't do it because the task is unpleasant. A sysadmin who points out a neglected, unassigned, or unpopular task is likely to receive that task as an assignment. This situation creates a conflict of interest because it motivates sysadmins to remain silent regarding such situations. Don't let that happen at your site; give your sysadmins an avenue for pointing out problems. You can allow them to open up tickets without assigning an owner or associating themselves to the issue, or you can create an email alias to which issues can be sent.

User acceptance of ticketing systems

Receiving a prompt response from a real person is a critical determinant of customer satisfaction, even if the personal response contains no more information than the automated response. For most problems, it is far more important to let the submitter know that the ticket has been reviewed by a real person than it is to fix the problem immediately. Users understand that administrators receive many requests, and they're willing to wait a fair and reasonable time for your attention. But they're not willing to be ignored.

The mechanism through which users submit tickets affects their perception of the system. Make sure you understand your organization's culture and your users' preferences. Do they want a web interface? A custom application? An email alias? Maybe they're only willing to make phone calls!

It's also important that administrators take the time to make sure they understand what users are actually requesting. This point sounds obvious, but think back to the last five times you emailed a customer service or tech support alias. We'd bet there were at least a couple of cases in which the response seemed to have nothing to do with the question—not because those companies were especially incompetent, but because accurately parsing tickets is harder than it looks.

Once you've read enough of a ticket to develop an impression of what the customer is asking about, the rest of the ticket starts to look like "blah blah blah." Fight this! Clients hate waiting for a ticket to find its way to a human, only to learn that the request has been misinterpreted and must be resubmitted or restated. Back to square one.

Tickets are often vague or inaccurate because the submitter does not have the technical background needed to describe the problem in the way that a sysadmin would. That doesn't stop users from making their own guesses as to what's wrong, however. Sometimes these guesses are perfectly correct. Other times, you must first decode the ticket to determine what the user *thinks* the problem is, then trace back along the user's train of thought to intuit the underlying problem.

Sample ticketing systems

The following tables summarize the characteristics of several well-known ticketing systems. Table 31.1 shows open source systems, and Table 31.2 shows commercial systems.

Table 31.1: Open source ticket systems

Name	Input[a]	Lang	Back end[b]	Web site
Double Choco Latte	W	PHP	MP	github.com/gnuedcl/dcl
Mantis	WE	PHP	M	mantisbt.org
OTRS	WE	Perl	DMOP	otrs.org
RT: Request Tracker	WE	Perl	M	bestpractical.com
OSTicket	WE	PHP	M	osticket.com
Bugzilla	WE	Perl	MOP	bugzilla.org

a. Input types: W = web, E = email
b. Back end: D = DB2, M = MySQL, O = Oracle, P = PostgreSQL

Table 31.2 shows some of the commercial alternatives for request management. Since the web sites for commercial offerings are mostly marketing hype, details such as the implementation language and back end are not listed.

Table 31.2: Commercial ticket systems

Name	Scale	Web site
EMC Ionix (Infra)	Huge	infra-corp.com/solutions
HEAT	Medium	ticomix.com
Jira	Any	atlassian.com
Remedy (now BMC)	Huge	remedy.com
ServiceDesk	Huge	ca.com/us/service-desk.aspx
ServiceNow	Any	servicenow.com
Track-It!	Medium	trackit.com

Some of the commercial offerings are so complex that they need a person or two dedicated to maintaining, configuring, and keeping them running. Others (such as Jira and ServiceNow) are available as a “software as a service” product.

Ticket dispatching

In a large group, even one with an awesome ticketing system, one problem still remains to be solved: it is inefficient for several people to divide their attention between the task they are working on right now and the request queue, especially if requests come in by email to a personal mailbox. We have experimented with two solutions to this problem.

Our first try was to assign half-day shifts of trouble queue duty to staff members in our sysadmin group. The person on duty would try to answer as many of the incoming queries as possible during a shift. The problem with this approach was that not everybody had the skills to answer all questions and fix all problems. Answers were sometimes inappropriate because the person on duty was new and was not familiar with the customers, their environments, or the specific support contracts they were covered by. The result was that the more senior people had to keep an eye on things and so were not able to concentrate on their own work. In the end, the quality of service was worse and nothing was really gained.

After this experience, we created a “dispatcher” role that rotates monthly among a group of senior administrators. The dispatcher checks the ticketing system for new entries and farms out tasks to specific staff members. If necessary, the dispatcher contacts users to extract any additional information that is necessary for prioritizing requests. The dispatcher uses a home-grown database of staff skills to decide who on the support team has the appropriate skills and time to address a given ticket. The dispatcher also makes sure that requests are resolved in a timely manner.

31.3 LOCAL DOCUMENTATION MAINTENANCE

Just as most people accept the health benefits of exercise and leafy green vegetables, everyone appreciates good documentation and has a vague idea that it's important. Unfortunately, that doesn't necessarily mean that they'll write or update documentation without prodding. Why should we care, really?

- Documentation reduces the likelihood of a single point of failure. It's wonderful to have tools that deploy workstations in no time and distribute patches with a single command, but these tools are nearly worthless if no documentation exists and the expert is on vacation or has quit.
- Documentation aids reproducibility. When practices and procedures are not stored in institutional memory, they are unlikely to be followed consistently. When administrators can't find information about how to do something, they have to wing it.
- Documentation saves time. It doesn't feel like you're saving time as you write it, but after spending a few days re-solving a problem that has been tackled before but whose solution has been forgotten, most administrators are convinced that the time is well spent.
- Finally, and most importantly, documentation enhances the intelligibility of the system and allows subsequent modifications to be made in a manner that's consistent with the way the system is supposed to work. When modifications are made on the basis of only partial understanding, they often don't quite conform to the architecture. Entropy increases over time, and even the administrators that work on the system come to see it as a disorderly collection of hacks. The end result is often the desire to scrap everything and start again from scratch.

Local documentation should be kept in a well-defined spot such as an internal wiki or a third-party service such as Google Drive. Once you have convinced your administrators to document configurations and administration practices, it's important to protect this documentation as well. A malicious user can do a lot of damage by tampering with your organization's documentation. Make sure that people who need the documentation can find it and read it (make it searchable), and that everyone who maintains the documentation can change it. But balance accessibility with the need for protection.

Infrastructure as code

Another important form of documentation is known as “infrastructure as code.” It can take a variety of forms, but is most commonly seen in the form of configuration definitions (such as Puppet modules or Ansible playbooks) that can then be stored and tracked in a version control system such as Git. The system and its changes are well documented in the configuration files, and the environment can be built and compared against the standard on a regular basis. This approach ensures that the documentation and the environment always match and are up to date, solving the most common problem of traditional documentation. See Chapter 23, *Configuration Management*, for more information.

Documentation standards

If you must document elements manually, our experience suggests that the easiest and most effective way to maintain documentation is to standardize on short, lightweight documents. Instead of writing a system management handbook for your organization, write many one-page documents, each of which covers a single topic. Start with the big picture and then break it down into pieces that contain additional information. If you have to go into more detail somewhere, write an additional one-page document that focuses on steps that are particularly difficult or complicated.

This approach has several advantages:

- Higher management is probably only interested in the general setup of your environment. That is all that's needed to answer questions from above or to conduct a managerial discussion. Don't pour on too many details or you will just tempt your boss to interfere in them.
- The same holds true for customers.
- A new employee or someone taking on new duties within your organization needs an overview of the infrastructure to become productive. It's not helpful to bury such people in information.
- It's more efficient to use the right document than to browse through a large document.
- You can index pages to make them easy to find. The less time administrators have to spend looking for information, the better.
- It's easier to keep documentation current when you can do that by updating a single page.

This last point is particularly important. Keeping documentation up to date is a huge challenge; documentation is often is the first thing to be dropped when time is short. We have found that a couple of specific approaches keep the documentation flowing.

First, set the expectation that documentation be concise, relevant, and unpolished. Cut to the chase; the important thing is to get the information down. Nothing makes the documentation sphincter snap shut faster than the prospect of writing a dissertation on design theory. Ask for too much documentation and you might not get any. Consider developing a simple form or template for your sysadmins to use. A standard structure helps avoid blank-page anxiety and guides sysadmins to record pertinent information rather than fluff.

Second, integrate documentation into processes. Comments in configuration files are some of the best documentation of all. They're always right where you need them, and maintaining them takes virtually no time at all. Most standard configuration files allow comments, and even those

that aren't particularly comment friendly can often have some extra information sneaked into them.

Locally built tools can require documentation as part of their standard configuration information. For example, a tool that sets up a new computer can require information about the computer's owner, location, support status, and billing information, even if these facts aren't directly relevant to the machine's software configuration.

---

*See this page for more information about **cron**.*

Documentation should not create information redundancies. For example, if you maintain a site-wide master list of systems, there should be no other place where this information is updated by hand. Not only is it a waste of your time to make updates in multiple locations, but inconsistencies are also certain to creep in over time. When this information is required in other contexts and configuration files, write a script that obtains it from (or updates) the master configuration. If you cannot completely eliminate redundancies, at least be clear about which source is authoritative. And write tools to catch inconsistencies, perhaps run regularly from **cron**.

The advent of tools such as wikis, blogs, and other simple knowledge management systems has made it much easier to keep track of IT documentation. Set up a single location where all your documents can be found and updated. Don't forget to keep it organized, however. One wiki page with 200 child pages all in one list is cumbersome and difficult to use. Be sure to include a search function to get the most out of your system.

31.4 Environment separation

Organizations that write and deploy their own software need separate development, test, and production environments so that releases can be staged into general use through a structured process. Separate, that is, but identical; make sure that when development systems are updated, the changes propagate to the test and production environments as well. Of course, the configuration updates themselves should be subject to the same kind of structured release control as the code. “Configuration changes” include everything from OS patches to application updates and administrative changes.

Historically, it has been standard practice to “protect” the production environment by enforcing role separation throughout the promotion process. For example, the developers who have administrative privileges in the development environment are not the same people who have administrative and promotion privileges in other environments. The fear was that a disgruntled developer with code promotion permissions could conceivably insert malicious code at the development stage and then promote it through to production. By distributing approval and promotion duties to other people, multiple people would need to collude or make mistakes before problems could find their way into production systems.

Unfortunately, the anticipated benefits of such draconian measures are rarely realized. Code promoters often don’t have the skills or time to review code changes at a level that would actually catch intentional mischief. Instead of helping, the system creates a false sense of protection, introduces unnecessary roadblocks, and wastes resources.

In the DevOps era, this problem is solved in a different way. Rather than separate roles, the preferred approach is to track all changes “as code” in a repository (such as Git) that has an immutable audit trail. Any undesirable change is traceable back to the human that introduced it, so strict role separation is unnecessary. Because configuration changes are applied in an automated way across each environment, identical changes can be evaluated in lower environments (such as dev or test) before they are promoted to production, to ensure that no unintended consequences manifest themselves. If problems are discovered, reversion is as easy as identifying the problematic commit and temporarily bypassing it.

In a perfect world, neither developers nor ops staff would have administrative privileges in the production environment. Instead, all changes would be made through an automated, tracked process that has appropriate privileges of its own. Although this is a worthy aspirational goal, our experience has been that it is not yet realistic for most organizations. Work toward this utopian fantasy, but don’t get trapped by it.

31.5 Disaster management

Your organization depends on a working IT environment. Not only are you responsible for day-to-day operations, but you should also have plans in place to deal with any reasonably foreseeable eventuality. Preparation for such large-scale problems influences both your overall game plan and the way that you define daily operations. In this section, we look at various kinds of disasters, the data you need to recover gracefully, and the important elements of recovery plans.

Risk assessment

Before designing a disaster recovery plan, it's a good idea to pull together a risk assessment to help you understand what assets you have, what risks they face, and what mitigation steps you already have in place. The NIST 800-30 special publication details an extensive risk assessment process. You can download it from nist.gov.

Part of the risk assessment process is to make an explicit, written catalog of the potential disasters you want to protect against. Disasters are not all the same, and you may need several different plans to cover the full range of possibilities. For example, some common threat categories are

- Malicious users, both external and internal
- Floods
- Fires
- Earthquakes
- Hurricanes and tornadoes
- Electrical storms and power spikes
- Power failures, both short- and long-term
- Extreme heat or failure of cooling equipment
- ISP/Telecom/Cloud outage
- Device hardware failures (dead servers, fried hard disks)
- Terrorism
- Zombie apocalypse
- Network device failures (routers, switches, cables)
- Accidental user errors (deleted or damaged files and databases, lost configuration information, lost passwords, etc.)

Historically, about half of security breaches originate with insiders. Internal misbehavior continues to be the disaster of highest likelihood at most sites. For each potential threat, consider and write down all the possible implications of that event.

Once you understand the threats, prioritize the services within your IT environment. Build a table that lists your IT services and assigns a priority to each. For example, a "software as a service"

company might rate its external web site as a top-priority service, while an office with a simple, informational external web site might not worry about the site’s fate during a disaster.

Recovery planning

More and more, organizations are designing their critical systems to automatically fail over to secondary servers in the case of problems. This is a great idea if you have little or no tolerance for services being down. However, don't fall prey to the belief that because you are mirroring your data, you do not need off-line backups. Even if your data centers are miles apart, it is certainly possible that you could lose both of them.

Malicious hackers and ransomware can easily destroy an organization that does not maintain read-only, off-line backups. Make sure you include data backups in your disaster planning.

Read more about cloud computing in Chapter 9.

Cloud computing is often an essential element of disaster planning. Through services such as Amazon's EC2, you can get a remote site set up and functioning within minutes without having to pay for dedicated hardware. You pay only for what you use, when you use it.

A disaster recovery plan should include the following sections (derived from the NIST disaster recovery standard, 800-34):

- *Introduction* – purpose and scope of the document
- *Concept of operations* – system description, recovery objectives, information classification, line of succession, responsibilities
- *Notification and activation* – notification procedures, damage assessment procedures, plan activation
- *Recovery* – the sequence of events and procedures required to recover lost systems
- *Return to normal operation* – concurrent processing, reconstituted system testing, return to normal operation, plan deactivation

We are accustomed to communicating and accessing documents through the network. However, these facilities may be unavailable or compromised after an incident. Store all relevant contacts and procedures off-line. Know where to get recent backups and how to make use of them without reference to on-line data.

In all disaster scenarios, you will need access to both on-line and off-line copies of essential information. The on-line copies should, if possible, be kept in a self-sufficient environment, one that has a rich complement of tools, has key sysadmins' environments, runs its own name server, has a complete local **/etc/hosts** file, has no file-sharing dependencies, and so on.

Here's a list of handy data to keep in the disaster support environment:

- An outline of the recovery procedure: who to call, what to say
- Service contract phone numbers and customer numbers
- Key local phone numbers: police, fire, staff, boss
- Cloud vendor login information
- Inventory of backup media and the backup schedule that produced them
- Network maps
- Software serial numbers, licensing data, and passwords
- Copies of software installation media (can be kept as ISO files)
- Copies of your systems' service manuals
- Vendor contact information
- Administrative passwords
- Data on hardware, software, and cloud environment configurations: OS versions, patch levels, partition tables, and the like
- Startup instructions for systems that need to be brought back on-line in a particular order

Staffing for a disaster

Your disaster recovery plan should document who will be in charge in the event of a catastrophic incident. Set up a chain of command and keep the names and phone numbers of the principals off-line. We keep a little laminated card with important names and phone numbers printed in microscopic type. Handy—and it fits in your wallet.

The best person to put in charge may be a sysadmin from the trenches, not the IT director (who is usually a poor choice for this role).

The person in charge must be someone who has the authority and decisiveness to make tough decisions in the context of minimal information (e.g., a decision to disconnect an entire department from the network). The ability to make such decisions, communicate them in a sensible way, and lead the staff through the crisis are probably more important than having theoretical insight into system and network management.

An important but sometimes unspoken assumption made in most disaster plans is that sysadmin staff will be available to deal with the situation. Unfortunately, people get sick, go on vacation, leave for other jobs, and in stressful times may even turn hostile. Consider what you'd do if you needed extra emergency help. (Not having enough sysadmins around can sometimes constitute an emergency in its own right if your systems are fragile or your users unsophisticated.)

You might try forming a sort of NATO pact with a local consulting company that has sharable system administration talent. Of course, you must be willing to share back when your buddies have a problem. Most importantly, don't operate close to the wire in your daily routine. Hire enough system administrators and don't expect them to work 12-hour days.

Security incidents

System security is covered in detail in Chapter 27. However, it's worth mentioning here as well because security considerations impact the vast majority of administrative tasks. No aspect of your site's management strategy can be designed without due regard for security.

For the most part, Chapter 27 concentrates on ways of preventing security incidents from occurring. However, thinking about how you might recover from a security-related incident is an equally important part of security planning.

Having your web site hijacked is a particularly embarrassing type of break-in. For the sysadmin at a web-hosting company, a hijacking can be a calamitous event, especially when it involves sites that handle credit card data. Phone calls stream in from customers, from the media, from the company VIPs who just saw the news of the hijacking on CNN. Who will take the calls? What should that person say? Who is in charge? What role does each person play? If you are in a high-visibility business, it's definitely worth thinking through this type of scenario, coming up with some preplanned answers, and perhaps even having a practice session to work out the details.

Sites that accept credit card data have legal requirements to deal with after a hijacking. Make sure your organization's legal department is involved in security incident planning, and make sure you have relevant contact names and phone numbers to call in a time of crisis.

When CNN or Reddit announces that your web site is down, the same effect that makes highway traffic slow down to look at an accident on the side of the road causes your Internet traffic to increase enormously, often to the point of breaking whatever it was that you just fixed. If your web site cannot handle an increase in traffic of 25% or more, consider having your load-balancing device route excess connections to a server that presents a page that simply says "Sorry, we are too busy to handle your request right now." Of course, forward-thinking capacity planning that includes auto-scaling into the cloud (see Chapter 9) might avoid this situation altogether.

Develop a complete incident handling guide to take the guesswork out of managing security problems. See this page for more details on security incident management.

31.6 IT POLICIES AND PROCEDURES

Comprehensive IT policies and procedures serve as the groundwork for a modern IT organization. Policies set standards for users and administrators and foster consistency for everyone involved. More and more, policies require acknowledgement in the form of a signature or other proof that the user has agreed to abide by their contents. Although this may seem excessive to some, it is actually a great way to protect administrators in the long run.

The ISO/IEC 27001:2013 standard is a good basis for constructing your policy set. It interleaves general IT policies with other important elements such as IT security and the role of the Human Resources department. In the next few sections, we discuss the ISO/IEC 27001:2013 framework and highlight some of its most important and useful elements.

The difference between policies and procedures

Policies and procedures are two distinct things, but they are often confused, and the words are sometimes even used interchangeably. This sloppiness creates confusion, however. To be safe, think of them this way:

- Policies are documents that define requirements or rules. The requirements are usually specified at a relatively high level. An example of a policy might be that incremental backups must be performed daily, with total backups being completed each week.
- Procedures are documents that describe how a requirement or rule will be met. So, the procedure associated with the policy above might say something like "Incremental backups are performed with Backup Exec software, which is installed on the server backups01…"

This distinction is important because your policies should not change often. You should review them annually and maybe change one or two pieces. Procedures, on the other hand, evolve continuously as you change your architecture, systems, and configurations.

Some policy decisions are dictated by the software you are running or by the policies of external groups, such as ISPs. Some policies are mandatory if the privacy of your users' data is to be protected. We call these topics "nonnegotiable policy."

In particular, we believe that IP addresses, hostnames, UIDs, GIDs, and usernames should all be managed site-wide. Some sites (multinational corporations, for example) are clearly too large to implement this policy, but if you can swing it, site-wide management makes things a lot simpler. We know of a company that enforces site-wide management for 35,000 users and 100,000 machines, so the threshold at which an organization becomes too big for site-wide management must be pretty high.

Other important issues have a larger scope than just your local IT group:

- Handling of security break-ins
- Filesystem export controls
- Password selection criteria
- Removal of logins for cause
- Copyrighted material (e.g., MP3s and movies)
- Software piracy

Policy best practices

Several policy frameworks are available, and they cover roughly the same territories. The following topics are examples of those that are typically included in an IT policy set:

- Information security policy
- External party connectivity agreements
- Asset management policy
- Data classification system
- Human Resources security policy
- Physical security policy
- Access control policies
- Security standards for development, maintenance, and new systems
- Incident management policy
- Business continuity management (disaster recovery)
- Data retention standards
- Protection of user privacy
- Regulatory compliance policy

Procedures

Procedures in the form of checklists or recipes can codify existing practice. They are useful both for new sysadmins and for old hands. Better yet are procedures that include executable scripts or are captured in a configuration management tool such as Ansible, Salt, Puppet, or Chef. Over the long term, most procedures should be automated.

Several benefits accrue from standard procedures:

- Tasks are always done in the same way.
- Checklists reduce the likelihood of errors or forgotten steps.
- It's faster for the sysadmin to work from a recipe.
- Changes are self-documenting.
- Written procedures provide a measurable standard of correctness.

Here are some common tasks for which you might want to set up procedures:

- Adding a host
- Adding a user
- Localizing a machine
- Setting up backups/snapshots for a new machine
- Securing a new machine
- Removing an old machine
- Restarting a complicated piece of software
- Reviving a web site that is not responding
- Upgrading the operating system
- Patching software
- Installing a software package
- Upgrading critical software
- Backing up and restoring files
- Expiring old backups
- Performing emergency shutdowns

Many issues sit squarely between policy and procedure. For example:

- Who can have an account on your network?
- What happens when they leave?

The resolutions of such issues need to be written down so that you can stay consistent and avoid falling prey to the well-known four-year-old's ploy of "Mommy said no, let's go ask Daddy!"

31.7 Service level agreements

For the IT organization to keep users happy and meet the needs of the enterprise, the exact details of the service being provided must be negotiated, agreed to, and documented in “service level agreements” or SLAs. A good SLA is a tool that sets appropriate expectations and serves as a reference when questions arise. (But remember, IT provides solutions, not roadblocks!)

When something is broken, users want to know when it’s going to be fixed. That’s it. They don’t really care which hard disk or generator broke, or why; leave that information for your managerial reports.

From a user’s perspective, no news is good news. The system either works or it doesn’t, and if the latter, it doesn’t matter why. Our customers are happiest when they don’t even notice that we exist! Sad, but true.

An SLA helps align end users and support staff. A well-written SLA addresses each of the issues discussed in the following sections.

Scope and descriptions of services

This section is the foundation of the SLA because it describes what the organization can expect from IT. Write it in terms that can be understood by nontechnical staff. Some example services might be

- Email
- Chat
- Internet and web access
- File servers
- Business applications
- Authentication

The standards that IT will adhere to when providing these services must also be defined. For example, an availability section would define the hours of operation, the agreed-on maintenance windows, and the expectations regarding the times at which IT staff will be available to provide live support. One organization might decide that regular support should be available from 8:00 a.m. to 6:00 p.m. on weekdays but that emergency support must be available 24/7. Another organization might decide that it needs standard live support available at all times.

Here is a list of issues to consider when documenting your standards:

- Response time
- Service (and response times) during weekends and off-hours
- House calls (support for environments at home)
- Weird (unique or proprietary) hardware
- Upgrade policy (ancient hardware, software, etc.)
- Supported operating systems
- Supported cloud platforms
- Standard configurations
- Data retention
- Special-purpose software

When considering service standards, keep in mind that many users will want to customize their environments (or even their systems) if the software is not nailed down to prevent this. The stereotypical IT response is to forbid all user modifications, but although this policy makes things easier for IT, it isn’t necessarily the best policy for the organization.

Address this issue in your SLAs and try to standardize on a few specific configurations. Otherwise, your goals of easy maintenance and scaling to grow with the organization will meet some serious impediments. Encourage your creative, OS-hacking employees to suggest modifications that they need for their work, and be diligent and generous in incorporating these suggestions into your standard configurations. If you don’t, your users will work hard to subvert your rules.

Queue prioritization policies

In addition to knowing what services are provided, users must also know about the priority scheme used to manage the work queue. Priority schemes always have wiggle room, but try to design one that covers most situations with few or no exceptions. Some priority-related variables are listed below:

- The importance of the service to the overall organization
- The security impact of the situation (has there been a breach?)
- The service level the customer has paid or contracted for
- The number of users affected
- The importance of any relevant deadline
- The loudness of the affected users (squeaky wheels)
- The importance of the affected users (this is a tricky one, but let's be honest: some people in your organization have more pull than others)

Although all these factors will influence your rankings, we recommend a simple set of rules together with some common sense to deal with the exceptions. We use the following basic priorities:

- Many people cannot work
- One person cannot work
- Requests for improvements

If two or more requests have top priority and the requests cannot be worked on in parallel, we decide which problem to tackle first by assessing the severity of the issues (e.g., email not working makes almost everybody unhappy, whereas the temporary unavailability of a web service might hinder only a few people). Queues at lower priorities are usually handled in a FIFO manner.

Conformance measurements

An SLA needs to define how the organization will measure your success at fulfilling the terms of the agreement. Targets and goals allow the staff to work toward a common outcome and can lay the groundwork for cooperation throughout the organization. Of course, you must make sure you have tools in place to measure the agreed-on metrics.

At a minimum, you should track the following metrics for your IT infrastructure:

- Percentage or number of projects completed on time and on budget
- Percentage or number of SLA elements fulfilled
- Uptime percentage by system (e.g., “email 99.92% available through Q1”)
- Percentage or number of tickets that were satisfactorily resolved
- Average time to ticket resolution
- Time to provision a new system
- Percentage or number of security incidents handled according to the documented incident handling process

31.8 Compliance: regulations and standards

IT auditing and governance are big issues today. Regulations and quasi-standards for specifying, measuring, and certifying compliance have spawned myriad acronyms: SOX, ITIL, COBIT, and ISO 27001, just to name a few. Unfortunately, this alphabet soup is leaving something of a bad taste in system administrators' mouths, and quality software to implement all the controls deemed necessary by recent legislation is currently lacking.

Some of the major advisory standards, guidelines, industry frameworks, and legal requirements that might apply to system administrators are listed below. The legislative requirements are largely specific to the United States.

Typically, the standard you must use is mandated by your organization type or the data you handle. In jurisdictions outside the United States, you will need to identify the applicable regulations.

- The **CJIS (Criminal Justice Information Systems)** standard applies to organizations that track criminal information and integrate that information with the FBI's databases. Its requirements can be found on-line at fbi.gov/hq/cjisd/cjis.htm.
- **COBIT** is a voluntary framework for information management that attempts to codify industry best practices. It is developed jointly by the Information Systems Audit and Control Association (ISACA) and the IT Governance Institute (ITGI); see isaca.org for details. COBIT's mission is "to research, develop, publicize, and promote an authoritative, up-to-date, international set of generally accepted information technology control objectives for day-to-day use by business managers and auditors."

  The first edition of the framework was published in 1996, and we are now at version 5.0, published in 2012. This latest iteration was strongly influenced by the requirements of the Sarbanes-Oxley Act. It includes 37 high-level processes categorized into five domains: Align, Plan, and Organize (APO); Build, Acquire, and Implement (BAI); Deliver, Service, and Support (DSS); Monitor, Evaluate, and Assess (MEA); and Evaluate, Direct, and Monitor (EDM).
- **COPPA**, the **Children's Online Privacy Protection Act**, regulates organizations that collect or store information about children under the age of 13. Parental permission is required to gather certain information; see ftc.gov for details.
- **FERPA**, the **Family Educational Rights and Privacy Act**, applies to all institutions that are recipients of federal aid administered by the Secretary of Education. This regulation protects student information and accords students specific rights with respect to their data. For details, search for FERPA at ed.gov.

- **FISMA**, the **Federal Information Security Management Act**, applies to all government agencies and their contractors. It's a large and rather vague set of requirements that seek to enforce compliance with a variety of IT security publications from NIST, the National Institute of Standards and Technology. Whether or not your organization falls under the mandate of FISMA, the NIST documents are worth reviewing. See nist.gov for more information.

- The FTC's **Safe Harbor** framework bridges the gap between the U.S. and E.U. approaches to privacy legislation and defines a way for U.S. organizations that interface with European companies to demonstrate their data security. See export.gov/safeharbor.

- **GLBA**, the **Gramm-Leach-Bliley Act** regulates financial institutions' use of consumers' private information. If you've been wondering why the world's banks, credit card issuers, brokerages, and insurers have been pelting you with privacy notices, that's the Gramm-Leach-Bliley Act at work. See ftc.gov for details. Currently, the best GLBA information is in the business section of the Tips & Advice portion of the web site. The shortcut goo.gl/vv2011 currently works as a deep link.

- **HIPAA**, the **Health Insurance Portability and Accountability Act**, applies to organizations that transmit or store protected health information (aka PHI). It is a broad standard that was originally intended to combat waste, fraud, and abuse in health care delivery and health insurance, but it is now used to measure and improve the security of health information as well. See hhs.gov/ocr/privacy/index.html.

- **ISO 27001:2013** and **ISO 27002:2013** are a voluntary (and informative) collection of security-related best practices for IT organizations. See iso.org.

- **CIP (Critical Infrastructure Protection)** is a family of standards from the North American Electric Reliability Corporation (NERC) which promote the hardening of infrastructure systems such as power, telephone, and financial grids against risks from natural disasters and terrorism. In a textbook demonstration of the Nietzschean concept of organizational "will to power," it turns out that most of the economy falls into one of NERC's 17 "critical infrastructure and key resource" (CI/KR) sectors and is therefore richly in need of CIP guidance. Organizations within these sectors should be evaluating their systems and protecting them as appropriate. See nerc.com.

- The **Payment Card Industry Data Security Standard (PCI DSS)** was created by a consortium of payment brands including American Express, Discover, MasterCard, JCB, and Visa. It covers the management of payment card data and is relevant for any organization that accepts credit card payments. The standard comes in two flavors: a self-assessment for smaller organizations and a third-party audit for organizations that process more transactions. See pcisecuritystandards.org.

- The FTC's **Red Flag Rules** require anyone who extends credit to consumers (i.e., any organization that sends out bills) to implement a formal program to prevent and detect identity theft. The rules require credit issuers to develop heuristics for identifying suspicious account activity; hence, "red flag." Search for "red flag" at ftc.gov for details.

- In the 1990s and early 2000s, the **Information Technology Infrastructure Library (ITIL)** was a de facto standard for organizations seeking a comprehensive IT service management solution. Many large organizations deployed a formal ITIL program complete with project managers for each process, managers for the project managers, and reporting for managers of the project managers. In most cases, the results were not favorable. The heavy process focus combined with siloed functions resulted in intractable IT constipation. This red tape created opportunities for lean startups to steal market share from well-established companies, thus sending many career IT practitioners out to pasture. We hope we've seen the last of ITIL. Some say DevOps is the anti-ITIL methodology.

- Last, but certainly not least, the **IT general controls (ITGC)** portion of the **Sarbanes-Oxley Act (SOX)** applies to all public companies and is designed to protect shareholders from accounting errors and fraudulent practices. See sec.gov.

Some of these standards contain good advice even for organizations that are not required to adhere to them. It might be worth breezing through a few of them just to see if they contain any best practices you might want to adopt. If you have no other constraints, check out NERC CIP and NIST 800-53; they are our favorites with regard to thoroughness and applicability to a broad range of situations.

The National Institute for Standards and Technology (NIST) publishes a host of standards that are useful to administrators and technologists. The two most commonly used ones are mentioned below, but if you are ever bored and looking for standards, you might check out their web site. You will not be disappointed.

NIST 800-53, *Recommended Security Controls for Federal Information Systems and Organizations*, describes how to assess the security of information systems. If your organization has developed an in-house application that holds sensitive information, NIST 800-53 can help you make sure you have truly secured it. Beware, however: embarking on a NIST 800-53 compliance journey is not for the faint of heart. You are likely to end up with a document that is close to 100 pages long and that includes excruciating details. If you plan to do business with a U.S. government agency, you may be required to complete a NIST 800-53 assessment whether you want to or not.

NIST 800-34, *Contingency Planning Guide for Information Technology Systems*, is NIST's disaster recovery bible. It is directed at government agencies, but any organization can benefit from it. Following the NIST 800-34 planning process takes time, but it forces you to answer

important questions such as, “Which systems are the most critical?”, “How long can we survive without these systems?”, and “How are we going to recover if our primary data center is lost?”

31.9 Legal issues

The U.S. federal government and several states have enacted laws regarding computer crime. At the federal level, two pieces of legislation date from the early 1990s and three are more recent:

- The Electronic Communications Privacy Act
- The Computer Fraud and Abuse Act
- The No Electronic Theft Act
- The Digital Millennium Copyright Act
- The Email Privacy Act
- The Cybersecurity Act of 2015

Some major issues in the legal arena are these: liability of sysadmins, network providers, and public clouds; peer-to-peer file-sharing networks; copyright issues; and privacy issues. The topics in this section comment on these issues and a variety of other legal debacles related to system administration.

Privacy

Privacy has always been difficult to safeguard, and with the rise of the Internet, it is in more danger than ever. Medical records have been repeatedly disclosed from poorly protected systems, stolen laptops, and misplaced backup tapes. Databases full of credit card numbers are routinely compromised and sold on the black market. Web sites purporting to offer antivirus software actually install spyware when used. Fake email arrives almost daily, appearing to be from your bank and alleging that problems with your account require you to “verify” your account data.

Technical measures can never protect against these attacks because they target your site’s most vulnerable weakness: its users. Your best defense is a well-educated user base. To a first approximation, no legitimate email or web site will ever

- Suggest that you have won a prize;
- Request that you “verify” account information or passwords;
- Ask you to forward a piece of email;
- Ask you to install software you have not explicitly searched for; or
- Inform you of a virus or other security problem.

Users who have a basic understanding of these dangers are more likely to make sensible choices when a pop-up window claims they have won a free MacBook.

Policy enforcement

Log files might prove to you conclusively that person X did bad thing Y, but to a court it is all just hearsay evidence. Protect yourself with written policies. Log files sometimes include time stamps, which are useful but not necessarily admissible as evidence unless you can also prove that the computer was running the Network Time Protocol (NTP) to keep its clock synced to a reference standard.

You may need a security policy to prosecute someone for misuse. That policy should include a statement such as this: “Unauthorized use of computing systems may involve not only transgression of organizational policy but also a violation of state and federal laws. Unauthorized use is a crime and may involve criminal and civil penalties; it will be prosecuted to the full extent of the law.”

We advise you to display a splash screen that advises users of your snooping policy. You might say something like: “Activity may be monitored in the event of a real or suspected security incident.”

To ensure that users see the notification at least once, include it in the startup files you give to new users. If you require the use of SSH to log in (and you should), you can configure **/etc/ssh/sshd_config** so that SSH always shows the splash screen.

Be sure to specify that through the act of using their accounts, users acknowledge your written policy. Explain where users can get additional copies of policy documents, and post key documents on an appropriate web page. Also include the specific penalty for noncompliance (e.g., deletion of the account).

In addition to displaying the splash screen, have users sign a policy agreement before giving them access to your systems. Craft the acceptable use agreement in conjunction with your legal department. If you don’t have signed agreements from current employees, make a sweep to obtain them, then make signing the agreement a standard part of the induction process for new hires.

You might also consider periodically offering training sessions on information security. This is a great opportunity to educate users about important issues such as phishing scams, when it’s OK to install software and when it’s not, password security, and any other points that affect your environment.

Control = liability

Service providers (ISP, cloud, etc.) typically have an appropriate use policy (AUP) dictated by their upstream providers and required of their downstream customers. This "flow down" of liability assigns responsibility for users' actions to the users themselves, not to the service provider or the service provider's upstream provider. Such policies have been used to attempt spam control and to protect service providers in cases where customers have stored illegal or copyrighted material in their accounts. Check the laws in your area; your mileage may vary.

Your policies should explicitly state that users are not to use organizational resources for illegal activities. However, that's not really enough—you also need to discipline users if you find out they are misbehaving. Organizations that know about violations but do not act on them are complicit and can be prosecuted. Unenforced or inconsistent policies are worse than none, from both a practical and legal point of view.

Because of the risk of being found complicit in user activities, some sites limit the data that they log, the length of time for which log files are kept, and the amount of log file history kept on backup tapes. Some software packages help with the implementation of this policy by including levels of logging that help the sysadmin debug problems but that do not violate users' privacy. However, always be aware of what kind of logging might be required by local laws or by any regulatory standards that apply to you.

Software licenses

Many sites have paid for K copies of a software package and have N copies in daily use, where K < N. Getting caught in this situation could be damaging to the company—probably more damaging than the cost of those N-minus-K other licenses. Other sites have received a demo copy of an expensive software package and hacked it (reset the date on the machine, found a license key, etc.) to make it continue working after the expiration of the demo period. How do you as a sysadmin deal with requests to violate license agreements and make copies of software on unlicensed machines? What do you do when you find that machines for which you are responsible are running pirated software?

It's a tough call. Management will often not back you up in your requests that unlicensed copies of software be either removed or paid for. Often, it is a sysadmin who signs the agreement to remove the demo copies after a certain date, but a manager who makes the decision not to remove them.

We are aware of several cases in which a sysadmin's immediate manager would not deal with the situation and told the sysadmin not to rock the boat. The admin then wrote a memo to the boss asking for correction of the situation and documenting the number of copies of the software that were licensed and the number that were in use. The admin quoted a few phrases from the license agreement and carbon copied the president of the company and his boss's managers. In one case, this procedure worked and the sysadmin's manager was let go. In another case, the sysadmin quit when even higher management refused to do the right thing. No matter what you do in such a situation, get things in writing. Ask for a written reply; if all you get is spoken words, write a short memo documenting your understanding of your instructions and send it to the person in charge.

31.10 Organizations, conferences, and other resources

Many UNIX and Linux support groups—both general and vendor-specific—help you network with other people who are running the same software. Table 31.3 briefly lists a few such organizations, but many other national and regional groups are not listed here.

Table 31.3: UNIX and Linux organizations of interest to system administrators

Name	What it is
FSF	The Free Software Foundation, sponsor of GNU
USENIX	UNIX/Linux user group, quite technically oriented[a]
LOPSA	The League of Professional System Administrators
SANS	Sponsors sysadmin and security conferences
SAGE-AU	Australian sysadmins who hold yearly conferences in Oz
Linux Foundation	Nonprofit Linux consortium; produces LinuxCon among others
LinuxFest Northwest	Grass-roots conference with great content

a. Well-known parent organization of the LISA special interest group, which was retired in 2016

FSF, the Free Software Foundation, sponsors the GNU Project ("GNU's Not Unix," a recursive acronym). The "free" in the FSF's name is the "free" of free speech and not that of free beer. The FSF is also the origin of the GNU Public License, which now exists in several versions and covers many of the free software packages used on UNIX and Linux systems.

USENIX, an organization of users of Linux, UNIX, and other open source operating systems, holds one general conference and several specialized (smaller) conferences or workshops each year. The Annual Technical Conference (ATC) is a potpourri of in-depth UNIX and Linux topics and is a great place for networking with the community.

The League of Professional System Administrators, LOPSA, has a fairly complex and somewhat sordid history. It was originally associated with USENIX and was intended to assume the mantle of USENIX's system administration special interest group, SAGE. Unfortunately, LOPSA and USENIX parted on less than amicable terms and are now separate organizations.

Today, LOPSA sponsors a variety of sysadmin-related networking, mentorship, and educational programs, including events such as System Administrator Appreciation Day on the last Friday of July. The customary gift for this holiday is bottle of scotch.

SANS offers courses and seminars in the security space and also founded a certification program, the Global Information Assurance Certification (GIAC), which operates somewhat independently. Certifications are available in a variety of general and specific skill areas such as system administration, coding, incident handling, and forensics. See giac.org for details.

Many local areas have their own regional UNIX, Linux, or open systems user groups. Meetup.com is an excellent resource for finding relevant groups in your area. Local groups usually have regular meetings, workshops with local or visiting speakers, and often, dinner together before or after the meetings. They’re a good way to network with other sysadmins.

31.11 Recommended reading

Brooks, Frederick P., Jr. *The Mythical Man-Month: Essays on Software Engineering (2nd Edition)*. Reading, MA: Addison-Wesley, 1995.

Kim, Gene, Kevin Behr, and George Spafford. *The Phoenix Project: A Novel About IT, DevOps, and Helping Your Business Win (Revised Edition)*. Scottsdale, AZ: IT Revolution Press, 2014.

Kim, Gene, et al. *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*. Scottsdale, AZ: IT Revolution Press, 2016.

Limoncelli, Thomas A. *Time Management for System Administrators*. Sebastopol, CA: O'Reilly Media, 2005.

Machiavelli, Niccolò. *The Prince*. 1513. Available on-line from gutenberg.org.

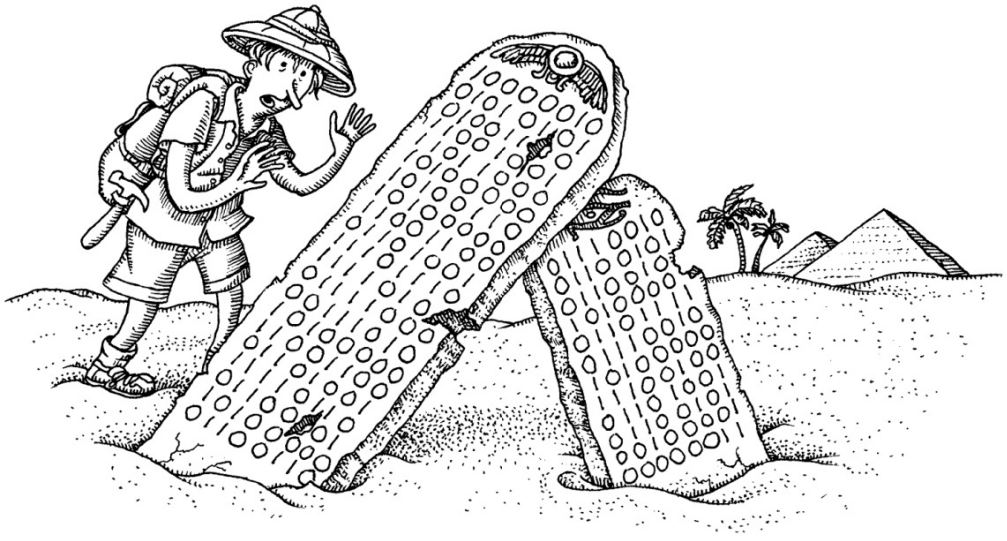
Morris, Kief. *Infrastructure as Code: Managing Servers in the Cloud*. Sebastopol, CA: O'Reilly Media, 2016. This book is a well-written 10,000-foot overview of DevOps and large-scale tools for system administration in the cloud. It includes few specifics about configuration management per se, but it's helpful for understanding how configuration management integrates into the larger scheme of DevOps and structured administration.

The site itl.nist.gov is the landing page for the NIST Information Technology Laboratory and includes lots of information about standards.

The web site of the Electronic Frontier Foundation, eff.org, is a great place to find commentary on the latest issues in privacy, cryptography, and legislation. Always interesting reading.

SANS hosts a collection of security policy templates at sans.org/resources/policies.

A Brief History of System Administration With Dr. Peter H. Salus, technology historian



In the modern age, most folks have at least a vague idea what system administrators do: work tirelessly to meet the needs of their users and organizations, plan and implement a robust computing environment, and pull proverbial rabbits out of many different hats. Although sysadmins are often viewed as underpaid and underappreciated, most users can at least identify their friendly local sysadmin—in many cases, more quickly than they can name their boss's boss.

It wasn't always this way. Over the last 50 years (and the 30-year history of this book), the role of the system administrator has evolved hand-in-hand with UNIX and Linux. A full understanding of system administration requires an understanding of how we got here and of some of the historical influences that have shaped our landscape. Join us as we reflect on the many wonderful years.

THE DAWN OF COMPUTING: SYSTEM OPERATORS (1952–1960)

The first commercial computer, the IBM 701, was completed in 1952. Before the 701, all computers had been one-offs. In 1954, a redesigned version of the 701 was announced as the IBM 704. It had 4,096 words of magnetic core memory and three index registers. It used 36-bit words (as opposed to the 701's 18-bit words) and did floating-point arithmetic. It executed 40,000 instructions every second.

But the 704 was more than just an update: it was incompatible with the 701. Although deliveries were not to begin until late 1955, the operators of the eighteen 701s in existence (the predecessors of modern system administrators) were already fretful. How would they survive this "upgrade," and what pitfalls lay ahead?

IBM itself had no solution to the upgrade and compatibility problem. It had hosted a "training class" for customers of the 701 in August 1952, but there were no textbooks. Several people who had attended the training class continued to meet informally and discuss their experiences with the system. IBM encouraged the operators to meet, to discuss their problems, and to share their solutions. IBM funded the meetings and made available to the members a library of 300 computer programs. This group, known as SHARE, is still the place (60+ years later) where IBM customers meet to exchange information. (Although SHARE was originally a vendor-sponsored organization, today it is independent.)

FROM SINGLE-PURPOSE TO TIME SHARING (1961–1969)

Early computing hardware was physically large and extraordinarily expensive. These facts encouraged buyers to think of their computer systems as tools dedicated to some single, specific mission: whatever mission was large enough and concrete enough to justify the expense and inconvenience of the computer.

If a computer were a single-purpose tool—let's say, a saw—then the staff that maintained that computer would be the operators of the saw. Early system operators were viewed more as "folks that cut lumber" than as "folks that provide what's necessary to build a house." The transition from system operator to system administrator did not start until computers began to be seen as multipurpose tools. The advent of time sharing was a major reason for this change in viewpoint.

John McCarthy had begun thinking about time sharing in the mid-1950s. But it was only at MIT (in 1961–62) that he, Jack Dennis, and Fernando Corbato talked seriously about permitting "each user of a computer to behave as though he were in sole control of a computer."

In 1964, MIT, General Electric, and Bell Labs embarked on a project to build an ambitious time-sharing system called Multics, the Multiplexed Information and Computing Service. Five years later, Multics was over budget and far behind schedule. Bell Labs pulled out of the project.

UNIX IS BORN (1969–1973)

Bell Labs' abandonment of the Multics project left several researchers in Murray Hill, NJ, with nothing to work on. Three of them—Ken Thompson, Rudd Canaday, and Dennis Ritchie—had liked certain aspects of Multics but hadn't been happy with the size and the complexity of the system, and they often gathered in front of a whiteboard to delve into design philosophy. The Labs had Multics running on its GE-645, and Thompson continued to work on it "just for fun."

Doug McIlroy, the manager of the group, said, “When Multics began to work, the very first place it worked was here. Three people could overload it.”

In the summer of 1969, Thompson became a temporary bachelor for a month when his wife, Bonnie, took their year-old son to meet his relatives on the West Coast. Thompson recalled, “I allocated a week each to the operating system, the shell, the editor, and the assembler…it was totally rewritten in a form that looked like an operating system, with tools that were sort of known; you know, assembler, editor, shell—if not maintaining itself, right on the verge of maintaining itself, to totally sever the GECOS connection…essentially one person for a month.” (GECOS was the General Electric Comprehensive Operating System.)

Steve Bourne, who joined Bell Labs the next year, described the cast-off PDP-7 used by Ritchie and Thompson: “The PDP-7 provided only an assembler and a loader. One user at a time could use the computer…The environment was crude, and parts of a single-user UNIX system were soon forthcoming…[The] assembler and rudimentary operating system kernel were written and cross-assembled for the PDP-7 on GECOS. The term UNICS was apparently coined by Peter Neumann, an inveterate punster, in 1970.” The original UNIX was a single-user system, obviously an “emasculated Multics.” But although there were aspects of UNICS/UNIX that were influenced by Multics, there were also, as Dennis Ritchie said, “profound differences.”

“We were a bit oppressed by the big system mentality,” he said. “Ken wanted to do something simple. Presumably, as important as anything was the fact that our means were much smaller. We could get only small machines with none of the fancy Multics hardware. So, UNIX wasn’t quite a reaction against Multics…Multics wasn’t there for us anymore, but we liked the feel of interactive computing that it offered. Ken had some ideas about how to do a system that he had to work out…Multics colored the UNIX approach, but it didn’t dominate it.”

Ken and Dennis’s “toy” system didn’t stay simple for long. By 1971, user commands included **as** (the assembler), **cal** (a simple calendar tool), **cat** (catenate and print), **chdir** (change working directory), **chmod** (change mode), **chown** (change owner), **cmp** (compare two files), **cp** (copy file), **date**, **dc** (desk calculator), **du** (summarize disk usage), **ed** (editor), and over two dozen others. Most of these commands are still in use.

By February 1973, there were 16 UNIX installations. Two big innovations had occurred. The first was a “new” programming language, C, based on B, which was itself a “cut-down” version of Martin Richards’ BCPL (Basic Combined Programming Language). The other innovation was the idea of a pipe.

A pipe is a simple concept: a standardized way of connecting the output of one program to the input of another. The Dartmouth Time-Sharing System had communication files, which anticipated pipes, but their use was far more specific. The notion of pipes as a general facility was Doug McIlroy’s. The implementation was Ken Thompson’s, at McIlroy’s insistence. (“It was one of the only places where I very nearly exerted managerial control over UNIX,” Doug said.)

“It’s easy to say ‘**cat** into **grep** into…’ or ‘**who** into **cat** into **grep**’ and so on,” McIlroy remarked. “It’s easy to say and it was clear from the start that it would be something you’d like to say. But there are all these side parameters… And from time to time I’d say ‘How about making something like this?’ And one day I came up with a syntax for the shell that went along with piping, and Ken said ‘I’m going to do it!’”

In an a orgy of rewriting, Thompson updated all the UNIX programs in one night. The next morning there were one-liners. This was the real beginning of the power of UNIX—not from the individual programs, but from the relationships among them. UNIX now had a language of its own as well as a philosophy:

- Write programs that do one thing and do it well.
- Write programs to work together.
- Write programs that handle text streams as a universal interface.

A general-purpose time-sharing OS had been born, but it was trapped inside Bell Labs. UNIX offered the promise of easily and seamlessly sharing computing resources among projects, groups, and organizations. But before this multipurpose tool could be used by the world, it had to escape and multiply. Katy bar the door!

UNIX HITS THE BIG TIME (1974–1990)

In October 1973, the ACM held its Symposium on Operating Systems Principles (SOSP) in the auditorium at IBM’s new T.J. Watson Research Center in Yorktown Heights, NY. Ken and Dennis submitted a paper, and on a beautiful autumn day, drove up the Hudson Valley to deliver it. (Thompson made the actual presentation.) About 200 people were in the audience, and the talk was a smash hit.

Over the next six months, the number of UNIX installations tripled. When the paper was published in the July 1974 issue of the *Communications of the ACM*, the response was overwhelming. Research labs and universities saw shared UNIX systems as a potential solution to their growing need for computing resources.

According to the terms of a 1958 antitrust settlement, the activities of AT&T (parent of Bell Labs) were restricted to running the national telephone system and to special projects undertaken on behalf of the federal government. Thus, AT&T could not sell UNIX as a product and Bell Labs had to license its technology to others. In response to requests, Ken Thompson began shipping copies of the UNIX source code. According to legend, each package included a personal note signed “love, ken.”

One person who received a tape from Ken was Professor Robert Fabry of the University of California at Berkeley. By January 1974, the seed of Berkeley UNIX had been planted.

Other computer scientists around the world also took an interest in UNIX. In 1976, John Lions (on the faculty of the University of New South Wales in Australia) published a detailed commentary on a version of the kernel called V6. This effort became the first serious documentation of the UNIX system and helped others to understand and expand on Ken and Dennis's work.

Students at Berkeley enhanced the version of UNIX they had received from Bell Labs to meet their needs. The first Berkeley tape (1BSD, short for 1st Berkeley Software Distribution) included a Pascal system and the **vi** editor for the PDP-11. The student behind the release was a grad student named Bill Joy. 2BSD came the next year, and 3BSD, the first Berkeley release for the DEC VAX, was distributed in late 1979.

In 1980, Professor Fabry struck a deal with the Defense Advanced Research Project Agency (DARPA) to continue the development of UNIX. This arrangement led to the formation of the Computer Systems Research Group (CSRG) at Berkeley. Late the next year, 4BSD was released. It became quite popular, largely because it was the only version of UNIX that ran on the DEC VAX 11/750, the commodity computing platform of the time. Another big advancement of 4BSD was the introduction of TCP/IP sockets, the generalized networking abstraction that spawned the Internet and is now used by most modern operating systems. By the mid-1980s, most major universities and research institutions were running at least one UNIX system.

In 1982, Bill Joy took the 4.2BSD tape with him to start Sun Microsystems (now part of Oracle America) and the Sun operating system (SunOS). In 1983, the court-ordered divestiture of AT&T began. One unanticipated side effect of the divestiture was that AT&T was now free to begin selling UNIX as a product. They released AT&T UNIX System V, a well-recognized albeit awkward commercial implementation of UNIX.

Now that Berkeley, AT&T, Sun, and other UNIX distributions were available to a wide variety of organizations, the foundation was laid for a general computing infrastructure built on UNIX technology. The same system that was used by the astronomy department to calculate star distances could be used by the applied math department to calculate Mandelbrot sets. And that same system was simultaneously providing email to the entire university.

THE RISE OF SYSTEM ADMINISTRATORS

The management of general-purpose computing systems demanded a different set of skills than those required just two decades earlier. Gone were the days of the system operator who focused on getting a single computer system to perform a specialized task. System administrators came into their own in the early 1980s as people who ran UNIX systems to meet the needs of a broad array of applications and users.

Because UNIX was popular at universities and because those environments included lots of students who were eager to learn the latest technology, universities were early leaders in the development of organized system administration groups. Universities such as Purdue, the

University of Utah, the University of Colorado, the University of Maryland, and the State University of New York (SUNY) Buffalo became hotbeds of system administration.

System administrators also developed an array of their own processes, standards, best practices, and tools (such as **sudo)**. Most of these products were built out of necessity; without them, unstable systems and unhappy users were the result.

Evi Nemeth became known as the "mother of system administration" by recruiting undergraduates to work as system administrators to support the Engineering College at the University of Colorado. Her close ties with folks at Berkeley, the University of Utah, and SUNY Buffalo created a system administration community that shared tips and tools. Her crew, often called the "munchkins" or "Evi slaves" attended USENIX and other conferences and worked as on-site staff in exchange for the opportunity to absorb information at the conference.

It was clear early on that system administrators had to be rabid jacks of all trades. A system administrator might start a typical day in the 1980s by using a wire-wrap tool to fix an interrupt jumper on a VAX backplane. Mid-morning tasks might include sucking spilled toner out of a malfunctioning first-generation laser printer. Lunch hour could be spent helping a grad student debug a new kernel driver, and the afternoon might consist of writing backup tapes and hassling users to clean up their home directories to make space in the filesystem. A system administrator was, and is, a fix-everything, take-no-prisoners guardian angel.

The 1980s were also a time of unreliable hardware. Rather than living on a single silicon chip, the CPUs of the 1980s were made up of several hundred chips, all of them prone to failure. It was the system administrator's job to isolate failed hardware and get it replaced, quickly. Unfortunately, these were also the days before it was common to FedEx parts on a whim, so finding the right part from a local source was often a challenge.

In one case, our beloved VAX 11/780 was down, leaving the entire campus without email. We knew there was a business down the street that packaged VAXes to be shipped to the (then cold-war) Soviet Union "for research purposes." Desperate, we showed up at their warehouse with a huge wad of cash in our pocket, and after about an hour of negotiation, we escaped with the necessary board. At the time, someone remarked that it felt more comfortable to buy drugs than VAX parts in Boulder.

System administration documentation and training

As more individuals began to identify themselves as system administrators—and as it became clear that one might make a decent living as a sysadmin—requests for documentation and training became more common. In response, folks like Tim O'Reilly and his team (then called O'Reilly and Associates, now O'Reilly Media) began to publish UNIX documentation that was based on hands-on experience and written in a straightforward way.

See Chapter 31 for more pointers to sysadmin resources.

As a vehicle for in-person interaction, the USENIX Association held its first conference focused on system administration in 1987. This Large Installation System Administration (LISA) conference catered mostly to a west coast crowd. Three years later, the SANS (SysAdmin, Audit, Network, Security) Institute was established to meet the needs of the east coast. Today, both the LISA and SANS conferences serve the entire U.S. region, and both are still going strong.

In 1989, we published the first edition of this book, then titled *UNIX System Administration Handbook*. It was quickly embraced by the community, perhaps because of the lack of alternatives. At the time, UNIX was so unfamiliar to our publisher that their production department replaced all instances of the string "etc" with "and so on," resulting in filenames such as **/and so on/passwd**. We took advantage of the situation to seize total control of the bits from cover to cover, but the publisher is admittedly much more UNIX savvy today. Our 30-year relationship with this same publisher has yielded a few other good stories, but we'll omit them out of fear of souring our otherwise amicable relationship.

UNIX HUGGED TO NEAR DEATH, LINUX IS BORN (1991–1995)

By late 1990, it seemed that UNIX was well on its way to world domination. It was unquestionably the operating system of choice for research and scientific computing, and it had been adopted by mainstream businesses such as Taco Bell and McDonald's. Berkeley's CSRG group, then consisting of Kirk McKusick, Mike Karels, Keith Bostic, and many others, had just released 4.3BSD-Reno, a pun on an earlier 4.3 release that added support for the CCI Power 6/32 (code named "Tahoe") processor.

Commercial releases of UNIX such as SunOS were also thriving, their success driven in part by the advent of the Internet and the first glimmers of e-commerce. PC hardware had become a commodity. It was reasonably reliable, inexpensive, and relatively high-performance. Although versions of UNIX that ran on PCs did exist, all the good options were commercial and closed source. The field was ripe for an open source PC UNIX.

In 1991, a group of developers that had worked together on the BSD releases (Donn Seeley, Mike Karels, Bill Jolitz, and Trent R. Hein), together with a few other BSD advocates, founded Berkeley Software Design, Inc. (BSDI). Under the leadership of Rob Kolstad, BSDI provided binaries and source code for a fully functional commercial version of BSD UNIX on the PC platform. Among other things, this project proved that inexpensive PC hardware could be used for production computing. BSDI fueled explosive growth in the early Internet as it became the operating system of choice for early Internet service providers (ISPs).

In an effort to recapture the genie that had escaped from its bottle in 1973, AT&T in 1992 filed a lawsuit against BSDI and the Regents of the University of California, alleging code copying and theft of trade secrets. It took AT&T's lawyers over two years to identify the offending code. When all was said and done, the lawsuit was settled and three files (out of more than 18,000) were removed from the BSD code base.

Unfortunately, this two-year period of uncertainty had a devastating effect on the entire UNIX world, BSD and non-BSD versions alike. Many companies jumped ship to Microsoft Windows, fearful that they would end up at the mercy of AT&T as it hugged its child to near-death. By the time the dust cleared, BSDI and the CSRG were both mortally wounded. The BSD era was coming to an end.

Meanwhile, Linus Torvalds, a Helsinki college student, had been playing with Minix (a PC-based UNIX clone developed by Andrew S. Tanenbaum) and began writing his own UNIX clone. By 1992, a variety of Linux distributions (including SuSE and Yggdrasil Linux) had emerged. 1994 saw the establishment of Red Hat and Linux Pro.

Multiple factors have contributed to the phenomenal success of Linux. The strong community support enjoyed by the system and its vast catalog of software from the GNU archive make Linux quite a powerhouse. It works well in production environments, and some folks argue that you can build a more reliable and performant system on top of Linux than you can on top of any other operating system. It's also interesting to consider that part of Linux's success may relate to the golden opportunity created for it by AT&T's action against BSDI and Berkeley. That ill-timed lawsuit struck fear into the hearts of UNIX advocates right at the dawn of e-commerce and the start of the Internet bubble.

But who cares, right? What remained constant through all these crazy changes was the need for system administrators. A UNIX system administrator's skill set is directly applicable to Linux, and most system administrators guided their users gracefully through the turbulent seas of the 1990s. That's another important characteristic of a good system administrator: calm during a storm.

A WORLD OF WINDOWS (1996–1999)

Microsoft first released Windows NT in 1993. The release of a "server" version of Windows, which had a popular user interface, generated considerable excitement just as AT&T was busy convincing the world that it might be out to fleece everyone for license fees. As a result, many organizations adopted Windows as their preferred platform for shared computing during the late 1990s. Without question, the Microsoft platform has come a long way, and for some organizations it *is* the best option.

Unfortunately, UNIX, Linux, and Windows administrators initially approached this marketplace competition in an adversarial stance. "Less filling" vs. "tastes great" arguments erupted in organizations around the world. (Just for the record, Windows is indeed less filling.) Many UNIX and Linux system administrators started learning Windows, convinced they'd be put out to pasture if they didn't. After all, Windows 2000 was on the horizon. By the close of the millennium, the future of UNIX looked grim.

UNIX AND LINUX THRIVE (2000–2009)

As the Internet bubble burst, everyone scrambled to identify what was real and what had been only a venture-capital-fueled mirage. As the smoke drifted away, it became clear that many organizations with successful technology strategies were using UNIX or Linux *along with* Windows rather than one or the other. It wasn't a war anymore.

A number of evaluations showed that the total cost of ownership (TCO) of a Linux server was significantly lower than that of a Windows server. As the impact of the 2008 economic crash hit, TCO became more important than ever. The world again steered toward open source versions of UNIX and Linux.

UNIX AND LINUX IN THE HYPERSCALE CLOUD (2010-PRESENT)

Linux and PC-based UNIX variants such as FreeBSD have continued to expand their market share, with Linux being the only operating system whose market share on servers is growing. Not to be left out, Apple's current full-size operating system, macOS, is also a variant of UNIX. (Even Apple's iPhone runs a cousin of UNIX, and Google's Android operating system derives from the Linux kernel.)

Much of the recent growth in UNIX and Linux has occurred in the context of virtualization and cloud computing. See Chapter 24, *Virtualization*, and Chapter 9, *Cloud Computing*, for more details about these technologies.

The ability to create virtual infrastructure (and entire virtual data centers) by making API calls has fundamentally shifted the course of the river once again. Gone are the days of managing physical servers by hand. Scaling infrastructure no longer means slapping down a credit card and waiting for boxes to appear on the loading dock. Thanks to services such as Google GCP, Amazon AWS, and Microsoft Azure, the era of the hyperscale cloud has arrived. Standardization, tools, and automation are not just novelties but intrinsic attributes of every computing environment.

Today, competent management of fleets of servers requires extensive knowledge and skill. System administrators must be disciplined professionals. They must know how to build and scale infrastructure, how to work collaboratively with peers in a DevOps environment, how to code simple automation and monitoring scripts, and how to remain calm when a thousand servers are down at once. (One thing hasn't changed: whiskey is still a close friend to many system administrators.)

UNIX AND LINUX TOMORROW

Where are we headed next? The lean, modular paradigm that has served UNIX so well over the last few decades is also one foundation of the up-and-coming Internet of Things (IoT). The Brookings Institution estimates that 50 billion small, distributed IoT devices will exist by 2020 (see brook.gs/2bNwbya).

It's tempting to think of these devices as we thought of the non-networked consumer appliances (e.g., toaster ovens or blenders) of yesteryear: plug them in, use them for a few years, and if they break, throw them in the landfill. They don't need "management" or central administration, right?

In fact, nothing could be further from the truth. Many of these devices handle sensitive data (e.g., audio streamed from a microphone in your living room) or perform mission-critical functions such as controlling the temperature of your house.

Some of these devices run embedded software derived from the OSS world. But regardless of what's inside the devices themselves, the majority report back to a mother ship in the cloud that runs—you guessed it—UNIX or Linux. In the early market-share land grab, many devices have already been deployed without much thought to security or to how the ecosystem will operate in the future.

The IoT craze isn't limited to the consumer market. Modern commercial buildings are riddled with networked devices and sensors for lighting, HVAC, physical security, and video, just to name a few. These devices often pop up on the network without coordination from the IT or Information Security departments. They're then forgotten without any plan for ongoing management, patching, or monitoring.

Size doesn't matter when it comes to networked systems. System administrators need to advocate for the security, performance, and availability of IoT devices (and their supporting infrastructure) regardless of size, location, or function.

System administrators hold the world's computing infrastructure together, solve the hairy problems of efficiency, scalability, and automation, and provide expert technology leadership to users and managers alike.

We are system administrators. Hear us roar!

RECOMMENDED READING

MCKUSICK, MARSHALL KIRK, KEITH BOSTIC, MICHAEL J. KARELS, AND JOHN S. QUARTERMAN. *The Design and Implementation of the 4.4BSD Operating System (2nd Edition)*. Reading, MA: Addison-Wesley, 1996.

SALUS, PETER H. *A Quarter Century of UNIX*. Reading, MA: Addison-Wesley, 1994.

SALUS, PETER H. *Casting the Net: From ARPANET to Internet and Beyond*. Reading, MA: Addison-Wesley, 1995.

SALUS, PETER H. *The Daemon, the Gnu, and the Penguin*. Marysville, WA: Reed Media Services, 2008. This book was also serialized at www.groklaw.net.

Colophon

For previous editions of this book, we used Adobe FrameMaker as our layout tool and were (more or less) happy with its features, stability, and facility with book-length writing projects. But although FrameMaker still exists, it now runs only on Windows and has become an increasingly vestigial member of the Adobe lineup.

Rather than targeting the general publishing market, Adobe seems to be pitching FrameMaker predominantly to multilingual workshops and to the poor lost souls who generate government-mandated SGML content. The product's core features haven't been updated in decades. It's like seeing a much-admired football quarterback working at the local mini mart loading hot dogs onto the fancy rolling cooker: someone has to do the job, but it seems like a waste of talent.

Ever since Adobe InDesign debuted in 2000, we've been waiting for it to grow sufficiently mature that we could contemplate switching. It was never originally intended for long documents, but over the years, the addition of features like tables of contents and indexing suggested that Adobe hoped to develop InDesign as a general solution for document publishing.

For this edition, we finally allowed ourselves to be goaded into the InDesign corral. We ran Adobe InDesign CC 2017 on macOS systems. Hilarity ensued…at least, if you find slapstick hilarious.

InDesign has many strengths and adherents, and we'll certainly reach for it the next time we need to design a six-page glossy brochure. However, we're reasonably certain that most InDesign fans aren't writing technical books.

One author said it best: "I have the impression that InDesign's book features were designed primarily to let InDesign slip past product selection committees' screening phases." To get this book into your hands, we've had to write literally thousands of lines of InDesign JavaScript code. Some of that code is ULSAH-specific, but a lot of it does basic book production chores that FrameMaker handled out of the box.

Unfortunately, our complaints with InDesign go beyond its cursory support for book production. We've also been continually surprised by its instability, obvious bugs, and nondeterministic

behaviors.

We wish we had some useful tooling advice to impart to other book authors, but we're just as puzzled as everyone else. FrameMaker has no future and InDesign has no present. Second-tier and on-line book publishing systems seem largely to target those looking for EZ-mode, book-length text editors. As far as we can tell, no modern GUI software addresses the task of creating a book like this one.

Ah well, there's always Microsoft Word. Our publisher tells us it's still the predominant format in which authors submit manuscripts.

We used the IndexUtilities add-on from Kerntiff Publishing Systems (kerntiff.co.uk) to supplement InDesign's fledgling indexing system. *Indexing from A to Z* by Hans H. Wellisch remains an invaluable reference text for the art of indexing and a resource that we recommend highly.

Lisa Haney drew the interior cartoons with a 0.05mm Staedtler pigment liner, then scanned them and converted them to 1200dpi bitmaps. The cover artwork was executed on black Ampersand Clayboard (a scratchboard) with Dr. Martin's Dyes for color. After scanning, the cover art was color-corrected in Photoshop and the layout completed in Adobe Illustrator.

The body text is Minion Pro, designed by Robert Slimbach. Headings, tables, and illustrations are set in Myriad Pro SemiCondensed by Robert Slimbach and Carol Twombly, with Fred Brady and Christopher Slye.

We've long sought a good typographical solution for "code" samples, and after trying out pretty much every font and option on the market, we've finally found our one true love: the Input font system by David Jonathan Ross, available from input.fontbureau.com. It looks great, and with 168 different variants, it's easy to match to any given body text.

Better yet, Input has both monospaced and proportionally spaced versions. You can use well-behaved proportional variants for most purposes, then switch to monospaced for tabular output. The styles intermix cleanly, so readers won't even notice a difference unless they closely scrutinize the typography.

The authors were never in the same physical location during this project. We maintained a shared tree of source files hosted on GitLab. Binary **.indd** (InDesign) files were stored in the repository and managed with the help of a horde of custom Ruby scripts. This approach was tolerable, but the inability to see per-commit diffs for text changes (because of InDesign's binary file format) limited its flexibility.

InDesign does support an XML interchange format, but unfortunately, every trip to XML resets all internal object IDs. The same file saved out twice in XML appears to have thousands of differences. Hence, no diffs, no merging, and no collaboration.

About the Contributors

James Garnett holds a PhD in Computer Science from the University of Colorado and is a Senior Software Engineer at Secure64 Software, Inc., where he develops DDoS mitigation technologies for Linux kernels. When not knee-deep in kernel code, he is usually somewhere deep in the Cascade Mountain range of Washington state.

Fabrizio Branca (@fbrnc) is the Lead System Developer at AOE. He, his wife, and their two children just returned to Germany after living in San Francisco for four years. Fabrizio has contributed to several open souce projects. He focuses on architecture, infrastructure, and high-performance applications. He promotes solid development, testing, and deployment processes for large projects.

Adrian Mouat (@adrianmouat) has been involved with containers from the early days of Docker and wrote the O'Reilly book *Using Docker* (amzn.to/2sVAIZt). He is currently Chief Scientist at Container Solutions, a pan-European company focusing on consulting and product development for microservices and containers.

About the Authors

For general comments and bug reports, please contact ulsah@book.admin.com. We regret that we are unable to answer technical questions.

Evi Nemeth retired from the Computer Science faculty at the University of Colorado in 2001. She explored the Pacific on her 40-foot sailboat named *Wonderland* for many years before becoming lost at sea in 2013 (see this page). The fourth edition of this book was her last as an active participant, but we're proud to say that we've retained her writing where possible.

Garth Snyder (@GarthSnyder) has worked at NeXT and Sun and holds a BS in Engineering from Swarthmore College and an MD and an MBA from the University of Rochester.

Trent R. Hein (@trenthein) is a serial entrepreneur who is passionate about practical cybersecurity and automation. Outside of technology, he loves hiking, skiing, fly fishing, camping, bluegrass, dogs, and the Oxford comma. Trent holds a BS in Computer Science from the University of Colorado.

Ben Whaley is the founder of WhaleTech, an independent consultancy. He was honored by Amazon as one of the first AWS Community Heroes. He obtained a BS in Computer Science from the University of Colorado at Boulder.

Dan Mackin (@dan_mackin) has a BS in Electrical and Computer Engineering from the University of Colorado at Boulder. He applies Linux and other open source technologies not only to his work, but also to automation, monitoring, and weather metrics collection projects at home. Dan loves skiing, sailing, backcountry touring, and spending time with his wife and dog.

Index

We have alphabetized files under their last components. And in most cases, *only* the last component is listed. For example, to find index entries relating to the **/etc/mail/aliases** file, look under **aliases**. Our friendly vendors have forced our hand by hiding standard files in new and inventive directories on each system.

Symbols

. directory entry 129
.. directory entry 129
#! (“shebang”) syntax 133
1Password 1011
389 Directory Server 589, 592–593
802.2* IEEE standards *see* IEEE standards

A

A DNS records 503, 524
AA (Access Agent) 607, 610
AAAA DNS records 525
acceptance tests 974
accept command 373
accept_redirects parameter 403, 426
accept router, Exim 665
accept_source_route parameter 426
Access Agent (AA) 607, 610
access control 65–68
access control lists *see* ACLs
access_db feature, **sendmail** 634

/etc/mail/access file 634
access points, wireless 474
accounts *see* user accounts
ACLs
    DNS 539, 559
    Exim 659–662
    filesystem 140–152
Active Directory *see* Microsoft Active Directory
addresses
    see also IP
    see also IPv6
    broadcast 388
    Ethernet (aka MAC) 386
    loopback 389
    multicast 389
adduser command 256, 264
/etc/adduser.conf file 263
Adleman, Leonard 1024
/var/adm directory 126
administrative privileges *see* root account
Adobe InDesign
    crash URL, frequently used 688
    experiences with 1165
AES (Advanced Encryption Standard) 1023
AFR (Annual Failure Rate) 735
AfriNIC 394
AgileBits 1011
AIDE (Advanced Intrusion Detection Environment) 1007, 1079, 1080
air conditioning *see* cooling
air plenums, wiring 467
AIX 10
Akamai Technologies 702
algorithms, cryptographic 1027
alias_database parameter, Postfix 676
/etc/mail/aliases file 619
aliases, email 619
    see also email
    see also Exim
    see also Postfix
    see also **sendmail**
    files, as alias source 621
    files, mailing to 621
    hashed database 622

loops 620
mailing lists 619
postmaster 620
programs, mailing to 622
alias_maps parameter, Postfix 676
Allman, Eric 304, 622
/var/cron/{**allow**,**deny**} file 113
Almquist shell 189
Alpine Linux 7
always_add_domain feature, **sendmail** 634
Amazon EC2 Container Registry 952
Amazon Linux 10
Amazon Web Services *see* AWS
AMD 916
American Power Conversion (APC) 1114
American Registry for Internet Numbers (ARIN) 483
AMP 483
Anixter 483
Annual Failure Rate (AFR) 735
Ansible 853, 856, 862–863, 865–883, 1127
access options, client 881–883
in AWS 871
comments on 862
comparison to Salt 907–909
and Docker 959
example 866
groups, client 870
groups, dynamic 871
iteration 875
and Jinja 875
passwords 882
playbooks 877–878
play elements 877
pros and cons 909
recommendations, configuration base 880
requirements, client 868–870
roles 878–880
securing client connections 882
security 908
setup 868–870
state parameters 874
tasks 873
templates 876–877

variables 871
ansible.cfg file 865
ansible command 865, 869
/etc/ansible directory 866
ansible-galaxy command 880
ansible-playbook command 856, 865, 878
ansible-vault command 865, 882
Anvin, H. Peter 155
anycast 388, 534
apache2.conf file 710
/var/log/apache2/* files 299
Apache Cassandra 962
apachectl command 709
Apache Directory Studio 588
Apache HTTP Server 696
Apache Software Foundation 16, 696, 962
Apache Spark 962
Apache Traffic Server 701
Apache Zookeeper 962
APC (American Power Conversion) 1114
apex zone, DNS 528
APIs (Application Programming interfaces) 704–706
APM (Application Performance Monitoring) 1076–1078
APNIC 394
AppArmor 87–89
/etc/apparmor.d directory 88
AppDynamics 1078
appendfile transport, Exim 667
application monitoring 1076–1078
Application Programming Interfaces (APIs) 704–706
Appropriate Use Policy (AUP) 1151
apropos command 15
APT (Advanced Package Tool) 167, 167–175, 169–170
apt-cache command 170
apt command 170
/var/log/apt* file 299
apt-get command 22, 170
apt-mirror command 172
Arch Linux 8
ARIN (American Registry for Internet Numbers) 394, 483
ARP (Address Resolution Protocol) 380, 403–404
ARPANET 378
arp command 404

Artifactory 952
as a service 324
ASHRAE temperature range 1115
ash shell 189
Assmann, Claus 649
AT&T 1158
attack surface 1004
AT&T UNIX System V 1159
auditd daemon 301
auditing, user access 80
authconfig command 248
/var/log/auth.log file 299
AUTH_MECHANISMS option, **sendmail** 639
~/.ssh/authorized_keys file 1037
/etc/auto.direct file 828
autofs filesystem 826
/etc/auto_master file 827
/etc/auto.master file 827
automation 1127
  code promotion 1128
  configuration management 1128
  machine setup 1127
  patching 1128
  scripts 184–243
  upgrades 1128
automount
  automatic mounts 830
  direct maps 828
  executable maps 829
  indirect maps 827–828
  on Linux 827, 831
  master maps 828
  replicated filesystems 830
  visibility 829–830
automountd daemon 826
automount utility 826
autonegotiation, Ethernet 471
/etc/auto.net file 829
autonomous system (AS) 492
autoreply transport, Exim 667
autounmountd daemon 826
availability 1000
availability zones, cloud 279–280

Avatier Identity Management Suite (AIMS) 269
AWS 25, 274, 275–276, 920
and Ansible 871
booting alternate kernel 359
CDN 703
CloudFormation 283
CloudWatch 324
CodeDeploy 976
console log 287
EBS 282, 293
EC2 275, 285–288
EC2 Container Service 963, 996
Elastic Beanstalk 278
Elastic File System (EFS) 826
emergency mode 62
event-based computing 708
firewall 287
IAM 283
instance store 784
Lambda 284
load balancing 698
NACLs 452–453
and NFS 808, 826
PaaS 707
pricing 293
quick start 284
RDS 282
Redshift 282
reserved instance 292
security groups 287, 452–453
shutting down systems 59
single-user mode 62
SQS 323
stopping instances 288
subnets 451–452
swap space 784
and Terraform 454–457
VPC 450–451
VPN 451–452
aws CLI tool 285–288

B

Backblaze 735
backing store 1098
backup, data
need for 802–803
plan 802–803
and security 1006
strategy 802–803
bad blocks, disk 747
BAD_RCPT_THROTTLE feature, **sendmail** 642
Bairavasundaram et al. 785
Bamboo 1127
Barracuda 616
.bash_profile file 70, 183, 194, 259
.bashrc file 70, 259
/etc/bashrc file 256
bash shell 6, 187, 189–198
see also **sh** shell
command editing 190
environment variables 193–194
pipes 190–192
quoting 192–193
redirection 190–192
search path 70
variables 192–193
Basic Input/Output System (BIOS) 32 *see also* UEFI
bc command 393
BCP (Best Current Practice) 379
BCPL (Basic Combined Programming Language) 1157
beer, suggested minimum 1122
Belden Cable 483
Bell Labs 1156
Berkeley *see* University of California at Berkeley
Berkeley Fast File System 776
Berkeley Internet Name Domain system *see* BIND
Berkeley Software Design, Inc. (BSDI) 1161
BGP (Border Gateway Protocol) 491, 494
bgpd daemon 497
bhyve 918, 924
BIND 530–547
see also DNS
see also **named**
see also name servers
ACLs 559–560

AXFR zone transfer 555
chrooted environment 559, 561
components 530
configuration examples 549–554
debugging 576–585
delv command 573
DNSSEC 564–576
dnssec-keygen command 562, 568
dnssec-signzone command 569
doc command 584
drill command 575
example configuration 549
forwarding zone 545
forward-only server 537
IXFR zone transfer 555
named.conf file 531
nsupdate program 557
rndc command 539, 545–546, 556, 582
/etc/rndc.conf file 546
rndc-confgen command 546
/etc/rndc.key file 546
security features 559–576
TSIG/TKEY 561–563
zone signing 569
zone transfers 555
/bin directory 125, 126
~/bin directory 183
/usr/bin directory 126
BIN_DIRECTORY variable, Exim 652
BIOS (Basic Input/Output System) 32
see also UEFI
BitBucket 978
Black Box Corporation 483
blacklist_recipients feature, **sendmail** 641
blacklists, **sendmail** 641
block device files 128, 130, 331
block size, disk 741–742
block storage 282
Blowfish hashing algorithm 249, 1023
blue/green deployment 977
/boot directory 39, 125, 126
booting
initial processes 94

logging 299, 320
and NFS filesystems 823
PXE 155
boot loader 33
GRUB 35–38
loader 38–40
password 1003
/var/log/boot.log file 299
bootstrapping
drive selection 32
failures 58–59, 59–60
firmware 32
fsck and 61
process overview 30–31
single-user mode 35, 38, 41, 60, 60–62
startup scripts 57–58
tasks 30
/boot/bootx64.efi bootstrap 39
/efi/boot/bootx64.efi bootstrap 34
Bostic, Keith 1161
botnets 1002
Bourne-again shell *see* **bash**
Bourne shell 187
Bourne, Stephen 187, 1157
break the glass 1011
broadcast
domain 465
packets 388, 465
ping 409, 425, 426
storm 415, 469
Bro network intrusion detection system 1017
Brouwer, Andries E. 757
Bryant, Bill 1032
BSDCan conference 19
bsdinstall utility 162–165
BSD UNIX 11, 1158–1159
btrfs command 797
Btrfs filesystem 753, 769, 784–786, 796–801
and Docker 946
setup 797–799
shallow copies 801
snapshots 800, 800–801
subvolumes 800

volumes 800
vs. ZFS 796–797
BugTraq 1052
Bugzilla 1132
building wiring 478
Burgess, Mark 854
bus errors 96
BUS signal 95

C

cables
10*base* 464
Category* 464, 466–467
coating 467
color-coding 468
Ethernet 464
fiber 467–468
wiring standard, UTP 467
CA (Certificate Authority) 1024
cache poisoning, DNS 536
cache, web server 699, 701
caching-only name server 509
Cacti 440–442
camcontrol command 61, 748, 749, 750
Canaday, Rudd 1156
cancel command 373
Canonical, Ltd. 9, 87
canonical name (CNAME) DNS records 527
Capistrano 854, 976
Carbon 1065, 1069
Card, Rémy 776
CAS 269
CBK (Common Body of Knowledge) 1049
ccTLDs (country code Top Level Domains) 507
CDN (Content Delivery Network) 702–703
Center for Internet Security 756
CentOS Linux 8, 10
Ceph 805
CERT 1055
Certificate Authority (CA) 1024
Certificate Signing Request (CSR) 1030

cfdisk command 731
CFEngine 854
chage command 248, 1012
chain of trust, DNSSEC 572
channels, wireless 475
character device files 128, 130, 331
ChatOps 1060, 1126
chat platforms 1126
Chatsworth Products 1110
chattr command 139
check_client_access option, Postfix 680
Check Point 410
checksum, network 384
Chef 853, 856, 861, 1127
    959
chfn command 250
chgrp command 137–138
Children's Online Privacy Protection Act (COPPA) 1147
chkrootkit command 1007
chmod command 135–137, 199
chown command 137–138, 260
Christiansen, Tom 188, 189
CIA triad 1000
CI/CD (Continuous Integration and Continuous Delivery)
    artifact 973
    auditability 969
    automation 968
    blue/green deployment 977
    build 968, 972–973
    and containers 995–997
    delivery 967
    deployment 967, 975–977
    environments 969–971
    essential concepts 967–971
    example 981–995
    feature flags 971
    integration 967, 968–969
    pipeline 971–977, 989–995
    release 972
    release candidate 972
    repository organization 985
    and revision control 968
    stages 994

testing 974–975, 992
zero downtime 977–978
CIDR (Classless Inter-Domain Routing) 381, 391, 393–394
CIFS *see* SMB (Server Message Block)
CIP (Critical Infrastructure Protection) 1148
CISA (Certified Information Systems Auditor) 1049
CIS (Center for Internet Security) 1051
Cisco Adaptive Security Appliance 411
Cisco IronPort 618
Cisco routers 498–500
Cisco Systems 483, 1025
CISSP (Certified Information Systems Security Professional) 1049
CJIS (Criminal Justice Information Systems) 1147
ClamAV 1007
CLAMS acronym 1125
C language 1157
cleanup daemon 672
/var/spool/clientmqueue directory 627
clone system call 93
CloudBees 978
cloud computing 271–294
access to 278–279
automation 283
availability zones 279–280
backup, data 802–803
booting alternate kernels 359–360
cost control 292–294
CPU stolen cycles 1092
and DevOps 273
foundations of 271
fundamentals 277
IaaS 277
identity and authorization 283
images 281
instances 281
management layers 278
networking 281–282, 450–460
PaaS 277
platforms 274–277
public, private, and hybrid 274–275
reasons for 272
regions 279–280
SaaS 277

serverless 284
storage 282
virtual private servers 281
web hosting 706–708
CloudFlare 703
CloudFront 703
cloud hosting providers 25
/var/log/cloud-init.log file 299
CNAME DNS records 527
cn LDAP attribute 591
Coarse Wavelength Division Multiplexing (CWDM) 468
Coax cable 464
Cobbler 161–163, 854
COBIT 1146, 1147
code coverage 974
code promotion 1128, 1136
collectd command 1075
etc/collectd/collectd.conf file 1075
Common Body of Knowledge (CBK) 1049
Common Criteria 1051
common name, LDAP 591
/etc/pam.d/common-passwd file 248
/compat directory 126
Computer Fraud and Abuse Act 1150
concentrators *see* Ethernet: hubs
conferences, system administration 19
confidentiality, data 1000, 1022
/etc/selinux/config file 86
configuration management
architecture 856–857
best practices 909
dangers of 846
dependency management 859–861
elements of 847–853
language comparison, platforms 857
overview 846
popular systems 853–865
rosetta stone 855
/usr/exim/configure file 656
CONFIGURE_FILE variable, Exim 652
confLOG_LEVEL option, **sendmail** 650
congestion control algorithms, TCP 418
conncontrol feature, **sendmail** 642

ConnectionRateThrottle option, **sendmail** 648
CONNECTION_RATE_THROTTLE option, **sendmail** 639
containers 918–920, 930–964
  as build artifacts 996
  capabilities 932
  and CI/CD 995–997
  clustering 959–964
  control groups 932
  core concepts 931–934
  images 932–933, 996
  management software 960–964
  namespaces 932
  networking 933
  utility of 930–931
  vs. virtualization 920
Content Delivery Network (CDN) 702–703
Continuous Integration and Delivery *see* CI/CD
control 1114
CONT signal 95, 96
cooling
  calculating load 1115–1117
  data center 1114–1119
  in-row 1118
COPPA (Children's Online Privacy Protection Act) 1147
Corbato, Fernando 1156
CoreOS Linux 7, 8
country code domains (ccTLDs) 507
Courion 269
CPAN (Comprehensive Perl Archive Network) 1072
CPU
  analyzing usage 1096–1098
  stolen cycles 1092
  utilization 1091
/proc/cpuinfo file 1094
CRAC (Computer Room Air Conditioner) 1117
/var/crash directory 363
Critical Infrastructure Protection (CIP) 1148
/etc/cron.{**allow**,**deny**} file 113
cron daemon 109–114, 1058
  log file 299
/var/log/cron file 299
crontab command 110
crontab file 110–114

cryptography 1022–1033
Diffie-Hellman key exchange 564
DNSSEC 564
public key 1023–1024
symmetric key 1023
C shell 187, 189
.cshrc file 259
csh shell 189
CSMA/CD protocol 464
CSR (Certificate Signing Request) 1030
CSRG (Computer Systems Research Group) 1159
Cummins Onan 1111
CUPS (Common UNIX Printing System) 365–369
see also printing
autoconfiguration 370
filters 368–369
instances 367
logging 373
network printing 369, 371
queue 366
restarting 373
cups-config command 373
/etc/cups/cupsd.conf file 367, 369
cupsd daemon 366
cupsdisable command 373
cupsenable command 373
curl command 24, 687, 691–692
cut command 194
CVS 978
CyberArk 1011
Cybersecurity Act of 2015 1150

D

DA (Delivery Agent) 607, 609
/var/log/daemon.log file 299
daemons 40
DARPA (Defense Advanced Research Project Agency) 1159
dash shell 189
DataBase Administrators (DBAs) 27
data center
see also cooling

availability 1119
components 1109
cooling load 1115–1117
generators 1111
hot aisle 1117–1118
humidity 1118
in-row cooling 1118
location 1120
power 1110
rack density 1112
rack power requirements 1112–1113
racks 1110
raised floor 1110, 1117
redundant power 1111, 1119
reliability tiers 1119
security 1120
temperature range 1115
toolbox 1122
track system 1110
UPSs 1111, 1119
Datadog 1067
Data Loss Prevention (DLP) 618
DBAN (Darik's Boot and Nuke) 750
DBAs (DataBase Administrators) 27
dc LDAP attribute 591
DDoS (Distributed Denial-of-Service attack) 1002
debconf utility 159
Debian/GNU Linux 7, 8, 9
debian-installer script 159
debt, technical 1123
/var/log/debug* files 299
debugging *see* troubleshooting
default route 416, 422, 428, 487, 495
DefCon conference 19
defense in depth 1004
DeHaan, Michael 161
DELAY_LA option, **sendmail** 639, 648
deleting accounts 265
Delivery Agent (DA) 607, 609
deluser command 266
delv command 513, 573
denial of service (DOS) attack 1106
DNS 512, 519, 560

email 639, 648
Dennis, Jack 1156
Deraison, Renaud 1015
DES hashing algorithm 249
/etc/devd.conf file 340
devd daemon 333, 340–341
/dev directory 125, 126, 130, 331
development environment 970
devfs filesystem 333, 340
device drivers 330–341
346–347
device files 331–332
block vs. character 331
creation of 333
for disks 746–747
management of 332–341
device names, ephemeral 747
DevOps 26, 1124
see also CI/CD
automation 1127
ChatOps 1126
chat platforms 1126
CI/CD 965–997
CLAMS acronym 1125
and cloud computing 273
code promotion 1128
and configuration management 845
culture 1125
environment separation 1137
infrastructure as code 1134
lean tenet 1127
measurement 1128
philosophy, paging 1126
sharing 1128
system administrator role 1129
tenets 1125
DevOpsDays conference 19
devtmpfs filesystem 334
df command 780, 1074
dhclient command 428
/etc/dhclient.conf file 428
dhcpd.conf file 406–408
dhcpd daemon 406–408

DHCP (Dynamic Host Configuration Protocol) 156, 404–408
556
dhcrelay daemon 408
Diffie-Hellman-Merkle key exchange 1024
dig command 513–517
Digital Millennium Copyright Act (DMCA) 1150
DigitalOcean 26, 274, 276–278, 981, 987–989
booting alternate kernel 359
networking 459–460
quick start 290–292
recovery kernel 63
directories 122, 124–127
deleting 129
search bit 133
disaster recovery 1137–1141
in the cloud 279–280
list of data to keep handy 1139
planning 1138
risk assessment 1137
staffing 1140
standards 1138
threats 1137
who to put in charge 1140
/dev/disk directory 747, 783
diskpart command 757
disks
see also filesystems
addition of 730–733
bad block management 747–748
block size 741–742
comparison of HDD and SSD 734
device files 746–747
elevator algorithm 1104–1105
failure rate 735
filesystems 775–776, 776–784
formatting 747–748
hardware 733–742
hardware attachment 745–746
hardware interfaces 742–745
HDD 734–737
hot-pluggable 745
hybrid 740
logical volumes 760

LVM 759–765
management layers 752–754
monitoring with SMART 750–751
naming standards, device 746
partitions 754–759
performance 1101–1102, 1102–1103
physical volumes 760
RAID 765–775
reliability, HDD 735
reliability, SSD 739
resizing filesystems 763–765
rewritability limit, SSD 738
scheme, partitioning 755
snapshots 762–763
speeds 734
tradeoffs of 733
types 736
usage 109
usb drive, mounting 783
volume groups 760
warranties 737
/proc/diskstats file 1094
distance-vector protocols 490–491
distinguished name, LDAP 591
Distributed Denial of Service attack (DDoS) 1002
DIX Ethernet II framing 384
DKIM (DomainKeys Identified Mail) 618
DKIM (DomainKeys Identified Mail) DNS records 530
DLP (Data Loss Prevention) 618
DMARC (Domain-based Message Authentication, Reporting, and Conformance) DNS records 530
DMCA (Digital Millennium Copyright Act) 1150
dmesg command 320, 338
/var/log/dmesg file 299
dmidecode command 1095
dmsetup command 754
DMZ (DeMilitarized Zone) 1046
dn LDAP attribute 591
DNS
see also BIND
see also name servers
see also resource records, DNS
see also zones, DNS

apex zone 528
architecture 503
Berkeley Internet Name Domain (BIND) daemon 530–547
bogus TLD 549
cache poisoning 536
caching 512–513
and CDNs 702
cloud-based 504
configuration 417
database 517
debugging 513–514
delv command 513
dynamic updates 556
EDNS0 protocol 538
efficiency 512–513
forward zones 506
in-addr.arpa zone 506, 525
ip6.arpa 526
IPv6 support 525, 526
key rollover, DNSSEC 572
KSKs (key-signing keys) 573–577
lame delegations 584
lookups 504
named.conf file 531
nameserver directive 505
name server market share 504
name servers 508
name server types 508
namespace 506
negative caching 512
NOERROR status 514
/etc/nsswitch.conf file 505
NXDOMAIN status 514, 579
open resolvers 560
primary objective 502
private addresses, queries from 547
query 503
record types 520
recursive servers 535
registering a domain name 507
/etc/resolv.conf file 504
resolver configuration 504
resource records *see* resource records, DNS

reverse mapping 506, 510, 525
root server hints 544
root servers 509, 510, 544
round robin 512–513
second-level domain name 507
security 558
SERVFAIL status 514, 576, 580, 584
service providers 504
splattercast 535
split DNS 547
subdomains 507
TTL (time to live) 512
UDP packet size 538
ZSK (Zone-Signing Keys) 573–577
DNSKEY DNS records 565
dnslookup driver, Exim 665
Dnsmasq 156
DNSSEC 530, 564–576
dnssec-keygen command 568
dnssec-signzone command 569
doc command 584
Docker 919, 930, 934–953
architecture 934–936
base images 949
bridge network 944–945
client setup 937
debugging 958–960
Dockerfile 949, 949–950, 957
docker group 937
Docker Hub 937
filesystem 933–934
image building 948–952
installation 936
interactive shell 938
logging 954
logs 940
namespaces 944–945
networking 943–946
options 947–948
overlays, network 945
registries 952–953
repository, images 937–941
rules of thumb 953

running containers 940
security 955–958
storage drivers 946–947
subcommands 936
Swarm 963
and **systemd** 947
TLS 956
volume containers 942–943
volumes 941–942
.dockercfg file 952
docker command 934–936, 937–941
dockerd daemon 934, 947–948
/var/lib/docker directory 935
DOCKER_HOST environment variable 937
Docker Hub 952
Docker, Inc. 934, 952
/var/run/docker.sock socket 948
Docker Swarm 976, 996
documentation 13–15
BCPs 379
FYIs 379
local 1134
man pages 14–16
package-specific 16
RFCs 17, 379
standards 1135
STDs 379
system-specific 16
DomainKeys Identified Mail *see* DKIM
DOMAIN macro, **sendmail** 633
“do not fragment” flag 385
DontBlameSendmail option, **sendmail** 643
DONT_BLAME_SENDMAIL option, **sendmail** 639
dot files 260
DOUBLE_BOUNCE_ADDRESS option, **sendmail** 639
Double Choco Latte 1132
double colon notation 397
dpkg command 164, 166–167
/var/log/dpkg.log file 299
dpkg-query command 21
DR *see* disaster recovery
drill command 513–515, 575

drivers *see* device drivers
Drone 996
Dropbear 1044
DS DNS records 565
dtrace command 1076
dtrace tool 1075
du command 1074
dumpon command 363
Duo 1008

E

echo command 201
EDITOR environment variable 193
editors, text 6
effective GID 67
effective UID 67
efibootmgr command 34
EFI (Extensible Firmware Interface) 758
EFI System Partition (ESP) 33
EGID (Effective GID) 92
Eich, Brendan 187
EIGRP (Enhanced Interior Gateway Routing Protocol) 491, 494
Elastic 323
Elastic Beanstalk 707
Elastic Load Balancer (ELB) 698
Elasticsearch 323, 1128
Electronic Communications Privacy Act 1150
Electronic Frontier Foundation 1025
elevator algorithm 1104–1105
ELK (Elasticsearch, Logstash, Kibana) stack 323–324, 1128
.emacs file 259
email
    see also aliases, email
    see also Exim
    see also MX DNS records
    see also Postfix
    see also **sendmail**
    access agents 610
    aliases 619
    architecture 607–610
    body 610–612

bounces 621
components 607–610
configuration 622–624
delivery agents 609
encryption 618
envelope 610
forgery 617
headers 610–612
loops 621
Maildir format 609
mailer-daemon 620
Mail Submission Agent (MSA) 608
marketshare 623
mbox format 609
message structure 610–613
MX records 526–527, 633, 637, 665
postmaster 620
privacy 618–619
spam *see* spam
transport agents 609
User Agents (UA) 607
Email Privacy Act 1150
EMC Ionix (Infra) 1133
emergency.target target 49
encryption *see* cryptography
environmental monitoring 1119
environment separation 1136
environment variables 193–194
ephemeral storage 282
equipment racks 1110
escrow, password 1010–1012
ESMTP protocol 609, 613
ESP (EFI System Partition) 33
/etc directory 125, 126
Ethernet 383, 463–473
addressing 386–387
autonegotiation 414, 423, 427, 471
broadcast domain 465
broadcast storms 469
cable characteristics 466
collisions 464
congestion 481
CSMA/CD protocol 464

framing 384
hubs 469
jumbo frames 472
loops 469
MTU 385, 472
OUIs 386
packet types 465
power over (PoE) 471
Routers 470–471
signaling 464
speeds 463–464
standards 464
switches 469
topology 465
trunking 470
VLANs 470
ethtool command 422
Etsy 1069
EUID (Effective UID) 92
event logging 1006
example systems 8–11
exec system call 93
execute bit 133
exicyclog command 655
exigrep command 655
exilog command 655
Exim 622, 651–670
see also email
access control lists (ACLs) 659–662
aliases file 666
authentication 663
blacklists 661
command-line flags 654
configuration 656–668
configuration variables 652
content scanning 660
debugging 670
.forward file 667
global options 657–659
installation of 652
lists 658
logging 669–670
macros 659

options 658
panic log 669
retry configuration file section 668
rewrite configuration file section 669
routers 664–667
security 654
transports 667–668
utilities 655
virus protection 660
exim_checkaccess command 655
exim command 654
exim_dbmbuild command 655
exim_dumpdb command 655
exim_fixdb command 655
exim_lock command 655
eximon command 655
eximstats command 655
exim_tidydb command 655
EXIM_USER variable, Exim 652
exinext command 655
exipick command 655
exiqgrep command 655
exiqsumm command 655
exiwhat command 655
expect language 6
export command 194
exportfs command 815
/etc/exports file 810, 815
exports, NFS 809
EXPOSED_USER macro, **sendmail** 636
ext4 filesytem 776–784

F

FaaS (Functions as a Service) 284
Fabric 854, 976
Fabry, Robert 1158
Fail2Ban 1021
/var/log/faillog file 299
Family Educational Rights and Privacy Act (FERPA) 1147
FAST_SPLIT option, **sendmail** 639
FAT (File Allocation Table) 33

fcntl systems call 810
fdisk command 61, 731, 757
FEATURE macro, **sendmail** 633
Federal Information Security Management Act (FISMA) 1147
Fedora Linux 8, 10
Feigenbaum, Barry 832
FERPA (Family Educational Rights and Privacy Act) 1147
Ferraiolo, David 85
fetch command 24
fiber, optical
    color-coding 468
    multimode 467
    single-mode 468
    standards 468
    types 468
Fielding, Roy 706
file attributes 132–140
    ACLs 81
    change time 134
    changing 135–137
    color-coding 135
    of device files 135
    displaying with **ls** 134–135
    encoding 136
    execute bit 133
    flags 139
    group permission 132
    inode number 135, 777
    linux bonus 139
    mnemonic syntax 136
    NFS 148
    owner permission 132
    permission bits 132–133
    setuid/setgid bits 68, 133
    sticky bit 133
file command 127
file descriptors 190
File Integrity Monitoring (FIM) 1079
filenames
    control characters in 128
    globbing 128
    length limitation of 122
    pathnames 122

pattern matching 12
spaces in 202
files
see also device files
see also directories
see also file attributes
see also filenames
access control of 66–67
block device 128, 130
character device 128, 130
default permissions 138
deleting 128, 129, 132
directory 128, 129
hard link 129
link count 134
local domain socket 128, 130–131
modes *see* file attributes
named pipe 128, 131
regular 128, 129
renaming 133
revision control 236–241
symbolic link 128
types of 127–132
file sharing 804–831, 832–843
Filesystem Hierarchy Standard 127
filesystems 775–776, 776–784, 784–786
see also partitions, disk
ACLs 81, 140–152
automatic mounting 780–783, 826–831
Btrfs 796–801
checking and repairing 61
components of 121
copy-on-write 784
error detection 785
ext4 776–784
inodes 777
journaling 776
lazy unmounting 123
lost+**found** directory 779
mounting 122–124, 780
NFS 809
organization of 124–127
pathnames 122

performance 785
polymorphism 778
processes using 123
relation to other layers 752–754
replicated 830
resizing 763–771
root 36, 38, 40, 125
SMB 839
superblock 777
terminology 777–778
UFS 776–784
union 932–934
unmounting 122–124
XFS 776–784
ZFS 784–786, 786–796
filesystem UID 67
file transfer, secure 1044
FIM (File Integrity Monitoring) 1079
find command 137, 200
fio command 1102
firewalls 1045–1047
packet-filtering 1045
safety of 1047
service filtering 1045
stateful inspection 1046
firmware, system 32
FISMA (Federal Information Security Management Act) 1049, 1147
flock system call 810
Fluke 478, 483
Fluke meter 1115, 1118
Fontana, Richard 787
fork system call 93
formatting, disk 747
.forward file 619
FORWARD chain, **iptables** 445
forwarder, DNS 508–509
.forward file 645
forward zone, DNS 506
Fowler, Martin 965
Fox, Brian 187
FQDNs (Fully Qualified Domain Names) 506
frame, network 384
FreeBSD 11

and Active Directory 597–598
adding users 264
and Kerberos 597–598
anti-virus 1007
autonegotiation 427
boot messages 356–358
building, kernel 348
configuration, kernel 347–349
default route 428
device management 340–341
disk addition recipe 732–733
firewall 447–450, 1008, 1045
idle timeout 255
installation 162–165
jails 919
kernel panics 362–363
loadable modules, kernel 351
location of kernel source 348
logging 304–320
logical volume management 765
log rotation 322–323
NAT 447–450
network hardware 427
networking 426–429
network parameters 429
paging statistics 1100
parameters, kernel 347–348
partitioning, disk 759
printing 364–375
/proc filesystem 105
removing users 264
router, use as a 486
security of 999
shadow passwords 253–255
software management 175–178
tracing 1076
versions, kernel 330
virtualization 924
VPN 1048
freebsd-update command 176
free command 1074
Free Software Foundation (FSF) 1153
fsck command 123, 776, 778, 779

FSF (Free Software Foundation) 1153
/etc/fstab file 61, 123, 732, 733, 779, 826
    on FreeBSD 781
    on Linux 782
fully qualified domain names (FQDNs) 506
Functions as a Service (FaaS) 284
fuser command 123, 1005, 1102
FYI (For Your Information) 379

G

gcloud cli tool 289–290
.gconf file 259
.gconfpath file 259
GCP (Google Cloud Platform) 25, 274, 276
    App Engine 278
    BigQuery 282
    booting alternate kernel 359
    Cloud Functions 284
    networking 457–458
    pricing 293
    quick start 289–290
GECOS field 246, 250, 1157
gem command 231
General Electric 1156
generators, standby 1111
generic top-level domains (gTLDs) 507
geom command 732
GeoTrust 1024
getent command 600
getfacl command 142–152
getpwnam() library call 245
getpwuid() library call 245
getty process 245
GIAC (Global Information Assurance Certification) 1049
gibi- prefix 12
GIDs *see* group IDs
giga- prefix 12
Git 968, 978, 1134
git command 237
.gitconfig file 259
GitHub repository 23, 236, 240–242, 978, 981

GitLab 236, 240–242, 978
Gi unit 12
GLBA (Gramm-Leach-Bliley Act) 1148
globbing 12, 128, 209
GlusterFS 805
go command 985
Go language 704, 982–983
Google 735, 769, 961
Google App Engine 707
Google Authenticator 1008
Google Cloud Platform *see* GCP
Google Compute Engine (GCE)
    booting failures 63
    serial console 63
Google Container Registry 952
Google Deployment Manager 976
Google Gmail 606
Google G Suite 616
Google Wifi 475
gpart command 37, 732, 759, 782
gparted command 731, 758, 771
gpasswd command 255
gpg command 1032
GPG encryption 618
GPT (GUID Partition Table) 33, 731, 758, 783
Grafana 1065, 1066–1067, 1128
Gramm-Leach-Bliley Act (GLBA) 261, 1148
graphical.target target 49
Graphite 1064–1065, 1065, 1066–1067, 1069, 1128
Gravitational 1044
Graylog 324
greet_pause feature, **sendmail** 642
grep command 197–198
groupadd command 256
groupdel command 256
/etc/group file 66–67, 255–256, 588, 599
group IDs 92, 92–93, 246, 250
    see also groups
    mapping names to 67
    pseudo-groups 79–80
    real, effective, and saved 67
groupmod command 256
group permission bits 132

groups
see also **/etc/group** file
see also group IDs
adding 256
docker 937
GID 255
GIDs (group IDs) 67
passwords 255
vs. RBAC 85
growfs command 765
GRUB boot loader 34, 35–38
boot password 1003
command line 37
commands 37
options 38
single-user mode 62
grub.cfg config file 36, 362
grub-mkconfig utility 36
/efi/ubuntu/grubx64.efi bootstrap 34
/etc/gshadow file 255
gTLDs (Generic Top-Level Domains) 507
GUID (globally unique identifier) 33
GUID Partition Table (GPT) 33
GVinum 765

H

h2i command 687
H2O HTTP server 696
halt command 59
halting the system 59
haproxy.cfg file 722
HAProxy load balancer 722–726
configuration of 722–723
health checks 724
statistics 724
sticky sessions 725–726
TLS terminaton 726–727
Hard Disk Drive (HDD) 734–737
hard links 129
hash, cryptographic 1026–1028
HashiCorp 283, 454, 925, 928, 981

Hazel, Philip 622, 651
HDD (Hard Disk Drive) 734–737
hdparm command 749, 750
head command 197
header, packet 383
Health Insurance Portability and Accountability Act *see* HIPAA (Health Insurance Portability and Accountability Act)
HEAT 1133
Hein, Trent R. 1161
Heroku 278, 707, 976
HGST 737
HIDS (Host-based Intrusion Detection System) 1080
hier man page 126
HIPAA (Health Insurance Portability and Accountability Act) 261, 1049, 1148
HipChat 1126
history
  of Linux 1161–1162
  of Sun Microsystems (now Oracle America) 1159
  of system administrators 1159–1160
  of UNIX 1156–1158
Hitachi 737
/home directory 126
home directory, user 251, 756
home_mailbox option, Postfix 677
host command 513
hosting recommendations 25
hostname 387, 413–414
  fully qualified 506
hostname command 413
/etc/hostname file 420
/etc/hosts file 387, 413
hot aisle cooling 1117–1118
HP-UX 10
htop command 103
htpasswd command 713
.htpasswd file 714
httpd.conf file 710
/var/log/httpd/* file 299
httpd server 696, 709–716
  applications server modules 714
  authentication, basic 713
  configuration 710–712
  log file 299

logging 715
multi-processing modules 709
TLS configuration 714
virtual hosts 712
HTTP protocol 686–694
authentication, basic 688, 713
headers 690
keep-alive 692–693
load balancers 696
over TLS 693
port 709
request methods 689
responses 689
security of 688
servers 695–696
SNI (Server Name Indication) 694
versions of 687
HTTPS protocol 688, 693
port 709
hubs, Ethernet 469
humidity 1118
HUP signal 95, 96, 306, 321
HVAC *see* cooling
HVM (Hardware Virtual Machine 916
hybrid cloud 275
hypervisors 915–918

I

IaaS (Infrastructure as a Service) 277
IAM (Identity and Access Management) 269–270, 283
IANA (Internet Assigned Numbers Authority) 507
IBM SoftLayer 274, 920
IBM T. J. Watson Research Center 622, 670
ICANN (Internet Corporation for Assigned Names and Numbers) 378, 394, 483, 507
Icinga 1060, 1063–1064, 1128
icmp_echo_ignore_broadcasts parameter 425, 426
ICMP (Internet Control Message Protocol) 380
redirects 403–405, 408
id command 813
~/.ssh/id_ecdsa.pub file 1036
identity management *see* IAM

idle timeout, FreeBSD 255
IDS (Intrusion Detection System) 1080
IEC units 13
IEEE 802.2 framing 384
IEEE standards
  802.1Q 470
  802.1x 474, 479
  802.3 464
  802.3ab 464
  802.3af 471
  802.3an 464
  802.3ba 464
  802.3bs 464
  802.3bt 471
  802.3u 464
  802.11ac 473
  802.11b 476
  802.11g 473
  802.11n 473
IETF (Internet Engineering Task Force) 378
ifconfig command 389, 414, 426–427, 474, 1096
ifdown command 418, 420, 422
ifup command 418, 420, 422
IGF (Internet Governance Forum) 378
IMAP protocol 615
IMAPS protocol 615, 619
in-addr.arpa DNS records 525
in-addr.arpa zone 506
incident handling 1054–1056, 1140
/usr/include directory 126
InfluxDB 1066
Information Systems Audit and Control Association (ISACA) 1147
Information Technology Infrastructure Library (ITIL) 1149
Infrastructure as a Service (IaaS) 277
infrastructure as code 926, 968, 1134
init process 31–32, 41–43, 57, 94
  bootstrapping and 41
  flavors of 42
  modes 41
  run levels 42, 49
  startup scripts 57
  vs. **systemd** 43
init scripts 31

Innotek GmbH 925
inode file attribute 135, 777
INPUT chain, **iptables** 445
in-row cooling 1118
(IN)SECURE magazine 1054
integration tests 974
integrity, data 1000, 1022
Intel 916
/etc/network/interfaces file 420
interfaces, network *see* networks
International Computer Science Institute (ICSI) 1018
International Organization for Standardization (ISO) 466
Internet
    documentation 378–381
    governance of 378
    history 377–380
    registries 394, 507
    standards 378–381
Internet Assigned Numbers Authority (IANA) 507
Internet Corporation for Assigned Names and Numbers (ICANN) 483, 507
Internet of Things (IoT) 1163
Internet Printing Protocol (IPP) 365
Internet protocol *see* IP
Internet Systems Consortium (ISC) 16, 530
INT signal 95, 96
ioctl system call 332
iostat command 1074, 1101–1102
IoT (Internet of Things) 1163
IP 377–461
    see also routing
    address assignment 413–414
    address classes 389–390
    addresses 387, 389–400
    allocation, address 394
    anycast 388
    ARP 403–404
    broadcast 388, 427
    broadcast ping 409, 425
    broadcast storm 415
    CIDR 393–394
    configuration 412–418, 419–420
    configuring 426–427
    debugging 429–438

default route 416, 422, 428
DHCP 404–408
directed broadcast 409
encapsulation 383–384
faster than light (FTL) 379
firewalls 410–411
forwarding 408
fragmentation 385
IPv4 vs. IPv6 381–383
layers 380
masquerading *see* NAT
MTU 385
multicast 388
netmask 415, 427
packet size 385–386
packet sniffers 435–438
packet structure 383–384
ports 387–388
private addresses 394–396
privileged ports 388
reassembly 385
redirects 408
routing 400–403
security 408–411
services 388
source routing 409
spoofing 409–410
stack 380
subnetting 390–391
time-to-live field (TTL) 433
transmission via avian carriers 379
unicast 388
uRFP 410
VPN 411–412
TCP/IP *see* IP
ip6.arpa zone 526
ipcalc tool 392
ip command 389, 401, 404, 414, 415, 419–420, 487, 1096
iperf tool 439–440
/etc/ipf/ipf.conf file 447
IPFilter 447–450
ip_forward parameter 426
ipfw command 1008, 1045

IPP (Internet Printing Protocol) 365
IPsec protocol 411, 1048
iptables command 442–447, 1008, 1045
IPv4 *see* IP
/proc/sys/net/ipv4 directory 424
IPv6 396–401, 404–408
    see also IP
    addressing 396–401
    address notation 397–398
    automatic host numbering 399
    debugging 429–438
    DNS support 525, 526
    double colon 397
    fragmentation 385
    link-local unicast 489
    MTU 385
    Neighbor Discovery (ND) 400, 403–404
    penetration of 381
    prefixes 398
    scenic routing for 379
    SLAAC 399
    tunneling 400
    vs. IPv4 381–383
/proc/sys/net/ipv6 directory 424
(ISC)2 International Information Systems Security Certification Consortium 1049
ISO 27001:2013 standard 1141, 1148
ISO 27002:2013 standard 1148
ISOC (Internet Society) 378
ISO (International Organization for Standardization) 466
ISP (Internet Service Provider) 378, 394
IT Governance Institute (ITGI) 1147
ITIL (Information Technology Infrastructure Library) 1146, 1149
iwconfig command 474
iwlist command 474

J

Jacobson, Van 436
Java language 704
JavaScript 186, 187
JavaScript Object Notation (JSON) 705, 862–865
JBOD (just a bunch of disks) 767

Jenkins 977–981, 1127
build agents 979
build context 978
build master 979
build trigger 978
and code repositories 978
concepts 978–979
distributed builds 979–980
and Docker 977
Jenkinsfile file 980–981, 984, 992
job 978
Pipeline 980–982, 984–987
project 978
Jenkins Enterprise 978
Jenkinsfile file 980–981, 984, 992
Jinja 858, 864, 874, 875, 876, 888, 891, 892, 893
Jira 1133
Jodies, Krischan 392
John the Ripper 1017
Jolitz, Bill 1161
JOSSO 269
journalctl command 56–57, 300, 320, 338
/etc/systemd/journald.conf file 56, 301
journald process 43, 56
journaling, filesystem 776
Joy, Bill 187, 1159
jq command 705
JSON (JavaScript Object Notation) 705, 863–865
jumbo frames, Ethernet 472
Juniper Networks 483

K

k8s *see* Kubernetes
Kali Linux 8
Kaminsky, Dan 536
Karels, Mike 1161
.kde/ directory 259
kdestroy command 598
[**kdump**] process 41
KeePass 1011
Kerberos 81, 268, 588, 596–598, 1032–1033

and Active Directory 596–598
and NFS 811
and NTP 597
kernel
arguments 31
booting 351
booting alternate in cloud 359–360
building, FreeBSD 348–349
building, Linux 344–346
errors 360–363
functions of 327–328
initialization of 40
loadable modules 349–351
loading 31
location of 125
options 38
panics 360–363
selection of 36, 40
single-user mode 35
source location, FreeBSD 348
source location, Linux 344
tuning, FreeBSD 347–348
tuning, Linux 341–343
version numbers 329–330
/boot/kernel directory 351
/var/log/kern.log file 299
kgdb command 363
Kibana 323, 1128
kibi- prefix 12
Kickstart 156
killall command 97
Kill A Watt meter 1115
kill command 97, 1106
KILL signal 95, 96
kilo- prefix 12
kinit command 598
Ki unit 12
kldload command 351
kldstat command 351
kldunload command 351
klist command 598
/dev/kmsg device 301
knife command 857

~/.ssh/known_hosts file 1034
Kolstad, Rob 1161
Korn shell 189
/etc/krb5.conf file 597–598
Krebs, Brian 1002
ks.cfg file 156–159
ksh shell 189
kubectl tool 961
Kubernetes 284, 961–962, 976, 996
Kuhn, Rick 85
kVA unit conversion 1113–1114
KVM 918, 923–924
    guest installation 923–924
kW unit conversion 1113–1114

L

LACNIC 394
Lambda 708
lame delegations, DNS 584
LAN (Local Area Network) 463–473
Large Installation System Administration (LISA) conference 1161
last command 300
/var/log/lastlog file 299, 300
layer 3 switches 470
LDAP 268, 589–595
    alternatives to 603–604
    attributes 590
    coverting **passwd** and **group** file 595–596
    data structure 590–591
    use with Exim 664
    LDIF 591
    querying 593–595
    uses for 590
    use with **sendmail** 635
/etc/openldap/ldap.conf file 592
LDAP_DEFAULT_SPEC option, **sendmail** 639
LDAP (Lightweight Directory Access Protocol) 588
ldap_routing feature, **sendmail** 635
ldapsearch command 593
LDIF (LDAP Data Interchange Format) 591
League of Professional System Administrators (LOPSA) 1153

least privilege, principle of 1004
LEDE 476
Lerdorf, Rasmus 187
Let’s Encrypt 1025
/lib64 directory 125
/lib directory 125, 126
/usr/lib directory 126
libpcap format 436
Librato 1067
licenses, software 1152
Lightweight Directory Access Protocol *see* LDAP
limits command 1107
link layer 384
links
    hard 129
    symbolic 128, 131–132
link-state protocols 491
lint tool 710
Linux
    account attributes 251–253
    and Active Directory 596–597
    adding users 262–263
    anti-virus 1007
    as a firewall 410, 441–450
    autonegotiation 423
    boot messages 352–356
    building, kernel 344–346
    bulk account creation 264–265
    configuration, kernel 341–347
    default route 422
    device driver, adding 346–347
    device management 334–340
    device mapper 754
    disk addition recipe 731–732
    distributions 7, 8
    errors, kernel 360–362
    filesystem, creating 778
    firewall 1008, 1045
    fstab file 782
    history of 1161–1162
    iptables 442
    and Kerberos 596–597
    loadable modules, kernel 349–351

location of kernel source 344
logging 296, 301–304
logical volume management 760
log rotation 321–322
LXC 919
NAT 446–447
network configuration 419–420
network hardware 422–424
networking 418–426
NetworkManager 418
network tuning 424–426
parameters, kernel 341–343
partitioning, disk 758
PAT 446–447
performance checkup 1094–1106
pluggable congestion control algorithms 418
printing 364–375
profiling, performance 1105–1108
RAID 769, 771–777
reasons to choose 1088
router, use as a 486
scheduler, I/O 1104–1105
security-enhanced 83, 85
security of 999
shadow passwords 251–253
stateful firewall 447–450
swap space 784
TCP/IP options 424–426
tracing 1076
versions, kernel 329–330
vga modes 362
virtualization 920–924
and viruses 1006
VPN 1048
and ZFS 787
LinuxCon conference 19
LinuxFest Northwest 1153
Linux Foundation 920, 1025, 1153
Linux installation
see also system administration
automating with **debian-installer** 159–161
automating with Kickstart 156–159
CentOS 156

Debian 159
netbooting with Cobbler 161–162
preseeding 159–163
Red Hat 156
Ubuntu 159
via PXE 154–155
Linux Mint 8
Linux package management 164–167
APT 169–170
high-level management systems 167–175
repositories 168–169
RHN 169
Lions, John 1159
LISA conference 19
LLC (Link Layer Control) layer 384
lmtp daemon 672
LMTP (Local Mail Transfer Protocol) 672
ln command 129, 131
loadable modules
in FreeBSD 351
in Linux 349–351
/proc/loadavg device 1073
load balancing
architecture 697
in AWS 698
equalization 698
with HAProxy 722
http 696
with NGINX 721
partitioning 698
round robin 512, 697
and security 698
servers 1090
loader bootstrap 38–39
commands 40
configuration 40
/boot/loader.conf configuration file 40
/boot/defaults/loader.conf file 351
Local Area Network (LAN) 463–473
local daemon 672, 677
/usr/local directory 125, 126
/etc/mail/local-host-names file 633
local_interfaces option, Exim 658

localization, software 178–181
guidelines 179
limiting active releases 180
organization 179
testing 180–181
update structure 180
Local Mail Transfer Protocol (LMTP) 672
locate command 21
lockd daemon 811
lockf system call 810
locking accounts 266–267
/var/log directory 126, 298
/var/spool/exim/log directory 669
log_file_path option, Exim 669
logger command 320
logging 295–326
see also syslog
architecture 297
at scale 323–324
boot-time 320
Docker 940, 954
for **sudo** 73
growth 299
httpd 715
kernel 320
locations 297–300
management 295–296, 321–323, 323–324
policies 324–326
rotation of files 321–323
Loggly 324
logical volume management 760
see also Btrfs filesystem
see also LVM
see also ZFS filesystem
login command 245, 600
.login_conf file 259
/etc/login.conf file 248, 254–255
logind process 43
/etc/pam.d/login file 602
.login file 259
logins *see* user accounts
login shell, user 251
log monitoring 1077

logos, example system 9
/etc/logrotate.conf file 322
logrotate utility 297, 321, 321–322
log_selector option, Exim 670
/dev/log socket 301
Logstash 323, 1128
logwatch tool 1077
loopback address 389
LOPSA (League of Professional System Administrators) 1153
lost+found directory 779
lpadmin command 373
lpc command 373
lp command 365, 373
lpinfo command 373
lpmove command 373
lpoption command 373
lpoptions command 372
lppasswd command 373
lpq command 365, 373
lpr command 365, 366, 373
lprm command 366, 373
lpstat command 366, 373
lsattr command 139
lsblk command 731, 746
ls command 134–135
LSM (Linux Security Modules) 83
lsmod command 349
lsof command 124, 1005, 1074, 1102
lsusb command 337
lvchange command 760, 761, 764
lvcreate command 760, 762, 763
lvdisplay command 760
LVM (logical volume management) 759–765
    architecture 760
    configuration phases 761–765
    relation to other layers 752–754
    snapshots 762–763
    vs. RAID 759
lvresize command 760, 764
LWN (Linux Weekly News) 1054
LXC 932
Lynis audit tool 1016

M

m4 command 624, 628–629
MAC (Mandatory Access Control) 84–85
MAC (Media Access Control) address 386
MAC (Media Access Control) layer 384
/bin/mail command 608
mail *see* email
mailbox_command option, Postfix 677
mailbox_transport option, Postfix 677
Maildir format, email 609
Maildrop 609
/var/spool/postfix/maildrop directory 672
MAILER macro, **sendmail** 632
/var/log/mail* files 299
MAIL_HUB macro, **sendmail** 637
mailq command 649, 654, 683
mail_spool_directory option, Postfix 677
Mail Submission Agent (MSA) 607, 608
Mail Transport Agent (MTA) 607, 609
Mail User Agent (MUA) 607
main.cf file, Postfix 673
major numbers, device 331
make command 973, 979
makemap command 632
makewhatis command 15
malware 616–618, 1001
man command 14
Mandatory Access Control (MAC) 84–85
mandb command 15
/usr/share/man directory 126
man pages 13–15
    sections 14
    updating keywords db 15
MANPATH environment variable 15
Mantis 1132
manualroute driver, Exim 666
/dev/mapper device 754
Marathon 962–963, 996
Mascheck, Sven 133
MASQUERADE_AS macro, **sendmail** 636
Master Boot Record (MBR) 33

MasterCard 1050
master.cf file, postfix 673
/etc/salt/master fiile 884
/etc/master.passwd file 68, 246, 253, 1011
Matsumoto, Yukihiro “Matz” 187, 223
MatterMost 1126
MaxDaemonChildren option, **sendmail** 648
MAX_DAEMON_CHILDREN option, **sendmail** 639
MaxMessageSize option, **sendmail** 648
MAX_MESSAGE_SIZE option, **sendmail** 639
MAX_MIME_HEADER_LENGTH option, **sendmail** 639
MAX_RCPTS_PER_MESSAGE feature, **sendmail** 642
MAX_RCPTS_PER_MESSAGE option, **sendmail** 639
mbox format, email 609
MBR (Master Boot Record) 33, 757
McAfee SaaS Email Protection 616
McCarthy, John 271, 1156
MCI_CACHE_SIZE option, **sendmail** 639
MCI_CACHE_TIMEOUT option, **sendmail** 639
McIlroy, Doug 1157
McKusick, Kirk 776, 1161
MD5 hashing algorithm 247, 1028
mdadm command 771
mdadm.conf file 773–774
md RAID system 769
/proc/mdstat file 772
Mean Time Between Failures (MTBF) 735
mebi- prefix 12
/media directory 126
mega- prefix 12
memcached daemon 282
/proc/meminfo file 1094
memory
    management 1098–1099
    paging 1098
    usage, analyzing 1099
Mercurial 978
Mesos 284, 962–963, 996
/var/log/messages file 299
Metasploit 1016
Metcalfe, Bob 463
MFA (MultiFactor Authentication) 1008

mfsBSD project 163
MIB (Management Information Base) 1081
microscripts 183–184
Microsoft Active Directory 588, 589
    and FreeBSD 597–598
    and Linux 596–598
Microsoft Azure 274
Microsoft Office 365 606
mii-tool command 422
Miller, Todd 70
MIN_FREE_BLOCKS option, **sendmail** 639
/etc/salt/minion file 886
minor numbers, device 331
MIN_QUEUE_AGE option, **sendmail** 639
Mirai botnet 1002
MIT 271, 588, 1156
Mi unit 12
mkdir command 129
mkfs.btrfs command 797
mkfs command 731, 778
mkisofs command 163
mknod command 130, 333
mkswap command 784
/mnt directory 126
Moby project 934
mod_cache caching module, **httpd** 701
modified EUI-64 algorithm 399
mod_passenger module, **httpd** 715
mod_perl module, **httpd** 715
mod_php module, **httpd** 715
modprobe command 350, 446
/etc/modprobe.conf file 350
mod_proxy_fcgi module, **httpd** 715
/boot/modules directory 351
/lib/modules directory 349
mod_wsgi module, **httpd** 715
Monitorama conference 19
monitoring 1057–1086
    application 1076–1078
    burn-out 1085
    charting platforms 1066–1067
    command output, harvesting 1071–1074

commercial platforms 1067–1068
culture 1061–1062
dashboards 1061
data collection 1068–1072
data types 1059
environmental 1119
events 1059
graphing 1064–1066
historical data 1060
historic trends 1059
instrumentation 1059
intrusion detection 1080–1081
log 1077
network 1072–1073
noise 1085
notifications 1060–1061
overview 1058–1061
platforms 1062–1068
push notifications 1059
and quality of life 1062
real-time metrics 1059
real-time platforms 1063–1064
run books 1085
security 1078–1080
SNMP 1080–1085
systems 1073–1076
temperature 1119
time-series platforms 1064–1066
tips and tricks 1085–1086
what to monitor 1061–1062
Monitus 1067
Mosh 1045
mount command 122–124, 778, 780, 821, 839
mountd daemon 815
mounting, filesystem 780
mount.ntfs command 123
mount_smbfs command 123
Mozilla Foundation 1025
MPLS (Multiprotocol Label Switching) 383
mpstat command 1074, 1092, 1097
/var/spool/mqueue directory 627
MSA (Mail Submission Agent) 607, 608
/var/spool/exim/msglog file 670

MSSP (Managed Security Service Provider) 1078
MTA (Mail Transport Agent) 607, 609
MTBF (Mean Time Between Failures) 735
mtree command 1079
mtr tool 435
MTU (Maximum Transfer Unit) 385–386, 1048
MUA (mail user agent) 607
multicast, IP 388, 465
multicast routing 494–495
Multics 1156
MultiFactor Authentication (MFA) 1008
multimode fiber 464, 467
Multiple-Input, Multiple-Output (MIMO) wireless 475
multiuser mode 41, 42
multi-user.target target 49
Munin 1066, 1077
munin-node command 1077
munin-node.conf file 1077
Murdock, Ian 9
MX DNS records 526–527
my_local_delivery clause, Exim 668
my_remote_delivery clause, Exim 668

N

Nagios 1060, 1063–1064
named
see also BIND
see also DNS
see also name servers
$INCLUDE directive 518
$ORIGIN directive 518, 544
$TTL directive 517, 518
acl statement 539, 559–560
allow-query-cache clause 537
allow-query clause 537, 559
allow-transfer clause 537, 559
allow-update clause 537
also-notify statement 534
avoid-v4-udp-ports clause 536
avoid-v6-udp-ports clause 536

blackhole clause 537, 559
channel clause 577
clients-per-query option 539
controls statement for **rndc** 545
datasize option 539
debug levels 581
directory statement 534
dnssec-enable option 538
dnssec-must-be-secure option 538
dnssec-validation option 538
dynamic updates 556
edns-udp-size option 537
error messages 580
files option 539
forwarders clause 537
forward option 537
freeze 583
hostname statement 534
include statement 533
key-directory statement 534
key (TSIG) statement 540
lame-ttl option 539
localhost zone 549
logging 576–580
logging clause 541, 577
log messages 580
masters statement 541
max-acache-size option 539
max-cache-size option 535, 539
max-cache-ttl option 539
max-clients-per-query option 539
max-journal-size option 539
max-ncache-ttl option 539
max-udp-size option 538
notify statement 534
options statement 533
performance tuning 539–540
query-source clause 536
query-source-v6 clause 536
recursion option 535
recursive-clients option 535

reload 583
root.cache file 545
search directive 505
server-id statement 534
server statement 540
slave servers, configuring 544
split DNS 547
statistics-channel statement 542
tcp-clients option 539
update-policy clause 558
use-v4-udp-ports clause 536
use-v6-udp-ports clause 536
version statement 534
view statement 547
zones, configuring 542
zone statement 542
zone-statistics option 539
zone transfer permissions 537
zone transfers 555
named.conf file 531
named pipes 128, 131
name servers
see also BIND
see also DNS
see also **named**
authoritative 508, 509
caching 508
caching-only 509
delegation 510
forwarder 508
master 508
nonauthoritative 508
nonrecursive 508, 509
primary 508
recursive 508, 509
root servers 509
secondary 508
slave 508
stub 508
switch file 505
Name Service Switch (NSS) 599
namespaces, Linux 82

namespaces, process 91
nano command 6
Napierała, Edward Tomasz 826
National Security Agency (NSA) 83, 85
NAT (Network Address Translation) 381, 394–396, 446–447, 447–450
nc command 1071
ND (Neighbor Discovery protocol) 400, 403–404
negative caching, DNS 512
Neighbor Discovery protocol (ND) 400, 403–404
Nemeth, Evi xxxii–xxxiii, 1160
NERC CIP 1049
NERC (North American Electric Reliability Corporation 1148
Nessus vulnerability scanner 1015–1021
netcat command 687
net command 598
Netlink sockets 332
Net-SNMP package 1083
netstat command 401, 416, 486, 1005, 1090
net-tools package 486
network administrators 27
network booting 154–155
networkd process 43
/etc/sysconfig/network file 421
Network File System *see* NFS
networking
    broadcast storm 415
    congestion control algorithms 418
    default route 428
    and Docker 943–946
    packet sniffers 435–438
    troubleshooting 429–438
    tuning 424–426
Network Intrusion Detection System (NIDS) 1017–1020, 1080
NetworkManager 418
network monitoring 1072–1073
network operations center (NOC) 27
networks
    see also Ethernet
    see also IP
    see also IPv6
    see also routing
    see also TCP/IP
    see also wireless networks

architecture 481
congestion 481
design issues 480
documentation 482
expansion 481
firewalls 1045–1047
maintenance 482
management 482
packet contents 384
port scanning 1013–1015
software-defined (SDN) 477
subnetting 390–391
success factors 463
wireless 473–476
network-scripts directory 55
Network Time Protocol (NTP) 1151
Neumann, Peter 1157
newaliases command 622, 654, 676
Newark Electronics 483
newfs command 732, 778
newgrp command 255
New Relic 1078
newsyslog utility 322–323
newusers command 264
NFS 804–831
ACLs 147–152
approach 807–814
automatic mounting 826–831
and AWS 808, 826
client-side 821–824
dedicated servers 825–826
drawbacks of 807
exports 809
hard vs. soft mounts 822, 823
history of 807–808
identity mapping 812–813, 824
and Kerberos 811
on Linux 815–818
locking, file 810–811
mounting at boot time 823
mount options 822
nobody account 813
on FreeBSD 818–819

performance 806, 814, 825–826
ports 812, 823
protocol versions 807–808
pseudo-filesystem 810
remote procedure calls 808
root access 813
RPC 808
security 806–807, 811–812, 823
server-side 814–820
statefulness 805, 809
statistics 824–825
transport protocols 808
vs. SMB 805, 834
nfsd daemon 815, 819–823
nfsstat command 824
nfsuserd daemon 824
nginx.conf file 718
nginx daemon 717
NGINX HTTP server 696, 716–722
configuration 717–720
installation of 717
load balancing 721
master process 716
signals 717
TLS 720
virtual hosts 718
worker process 716
nice command 103–104
niceness, process 93
NIDS (Network Intrusion Detection System) 1017–1020, 1080
NIS (Network Information Service) 603
NIST SP 800 series standards 1051
800-34 1138, 1149
800-53 1149
NLnet Labs DNSSEC tools 573
nmap port scanner 1013–1015
nmbd daemon 833–834
nobody account 79, 813
NOC (network operations center) 27
Node.js 704, 1069
No Electronic Theft Act 1150
nohup command 96
/var/run/nologin file 254

/bin/nologin shell 78
non-repudiation 1022
North American Electric Reliability Corporation (NERC) 1148
NoSQL database 962
NSA (National Security Agency) 998, 999
NS DNS records 524
NSEC3 DNS records 565
NSEC DNS records 565
nslookup command 513
NSS (Name Service Switch) 599
nsswitch.conf file 245, 246, 505, 588, 596, 599, 625, 836
ntpd daemon 596, 894
/dev/null file 332

O

objectClass LDAP attribute 591
object stores 282
office wiring 478
OID (Object Identifier), SNMP 1081
o LDAP attribute 591
OM1 fiber 468
OM2 fiber 468
OM3 fiber 468
OpenLDAP 589
open resolvers, DNS 560
OpenSolaris 786
OpenSSH 1036–1037
openssl selection 1029–1031
OpenStack 274, 275, 284
openSUSE Linux 8
OpenVPN 411
Open Web Application Security Project (OWASP) 1009
OpenWrt 8, 476
/opt directory 126
optical fiber 467
/etc/network/options file 420
Oracle 925
Oracle Identity Management 269
Oracle Linux 8
Oracle VirtualBox 918
O'Reilly Media 1160

O’Reilly series (books) 16
O’Reilly, Tim 1160
organizational unit, LDAP 591
orphaned processes 94
OS1 fiber 468
OSCON conference 19
ospf6d daemon 497
ospfd daemon 497
OSPF (Open Shortest Path First) protocol 491, 493
OSSEC (Open Source SECurity) 1007, 1077, 1079, 1080
OSS mailing list 1052
OSTicket 1132
OSTYPE macro, **sendmail** 633
OTRS 1132
OUI (Organizationally Unique Identifier) 386
ou LDAP attribute 591
out-of-memory killer 1099
oven, easy-bake 471
OWASP (Open Web Application Security Project) 1009, 1051
owner permission bits 132
ownership, file 137–138

P

PaaS (Platform as a Service) 277, 707
package management 21–23, 163–177
packages *see* software packages
Packer 925–928, 981, 987–989
packer command 927–928
packet encapsulation 383–384
packet filtering 1007–1008
packet-filtering firewalls 1045
packet forwarding 486–489
packets
    broadcast 465
    multicast 465
    unicast 465
packet sniffers 435–438
Padl Software 595
[**pagedaemon**] process 41
page size 90
page table 1098

PAM (Pluggable Authentication Modules) 80–81, 267, 600–603
panic, kernel 360
Papertrail 324
paravirtualization 916
parted command 731, 747, 758
partitions, disk 754–759
    see also filesystems
    relation to other layers 752–754
    scheme 755
PARTLABEL option 783
PARTUUID option 783
passphrase 1009
passwd command 68, 252, 258
/etc/passwd file 67, 245, 246–251, 588, 599, 1011, 1013
passwords 1009–1013
    aging 1012
    alternatives to 78
    break the glass 1011
    change interval 1010
    changing 68
    cracking 1017
    escrow 1010–1012
    expiration 251, 253
    hashes 246, 247–249
    management 1011
    obsolescence of 1008
    root 78
    strength 248, 1009
    vaults 1010–1012
PATH environment variable 199
path MTU discovery 385
pathnames 122
PAT (Port Address Translation) 395, 446
payload, packet 383
PCI DSS (Payment Card Industry Data Security Standard) 1050, 1148
PCIe (Peripheral Component Interconnect Express) interface 742
PCRE (Perl-Compatible Regular Expression) 671
Peek, Mark 283
penetration testing, application 1009–1010, 1016
perf command 1105
perf_events interface 1105
performance 1087–1108
    see also performance analysis tools

analysis methdology 1093
BIND 539
common issues 1089–1091
CPU 1091
disk 1101–1102
disk bandwidth 1091
kernel variables 1088
memory 1092, 1098–1099
network 1092
NFS 806, 814, 825–826
nice command 103–104
philosophy, tuning 1088–1089
resources that affect 1091
troubleshooting 1106–1107
tuning rules 1089
performance analysis tools
fio command 1102
iostat command 1101
mpstat command 1092, 1097
perf command 1105
ps command 1098, 1106
sar command 1103
top command 1092, 1106
uptime command 1106
vmstat command 1092, 1096, 1100
performance tests 974
periodic processes 109–119
Perl 186, 187
permission bits, file 132–133
permit_mynetworks option, Postfix 680
PGP (Pretty Good Privacy) 618, 1031
pgrep command 101
philosophy, IT management 1123, 1124
phishing 1001, 1150
PHP language 186, 187, 704
phpLDAPadmin 588
physical volumes 760
pickup daemon 672
pidof command 101
PID (Process ID) 91
/srv/pillar directory 884
ping6 command 430–432, 1072
ping command 430–432, 1072

Pingdom 1067
pip command 231
pipe daemon 673
pkg command 21, 22, 176
pkill command 97
PKI (Public Key Infrastructure) 1024
Platform as a Service (PaaS) 277, 707
Pluggable Authentication Modules *see* PAM
policy 1141–1144
    appropriate use 1151
    best practices 1142
    enforcement 1150
    standards 1141
POP3S protocol 619
portmap daemon 812, 1014
portmaster command 178
portsnap utility 177
postalias command 673
postcat command 673
postconf command 673, 675
Postel, Jon xxxiii
Postel's Law xxxiii
Postfix 622, 670–684
    see also email
    access control 680–682
    aliases file 672
    architecture 671
    configuration 673–681
    debugging 682–683
    encryption 682
    .forward file 677
    lookup tables 675–676
    null client configuration 674
    programs 671
    queues 672
    receiving mail 671
    security 673
    sending mail 672
    soft-bouncing 684
    utilities 673
    virtual domains 677–678
postfix command 673
postmap command 673

POSTROUTING chain, **iptables** 447
postsuper command 673
power factor 1113
poweroff.target target 49
Power over Ethernet (PoE) 471
power requirements
    blade servers 1112
    network equipment 1112
    storage equipment 1112
PowerShell Desired State Configuration 854
PPID (Parent PID) 91
Pratt, Ian 920
PREROUTING chain, **iptables** 445
preseed.cfg file 160
Pretty Good Privacy (PGP) encryption 618
preventative maintenance 1111
printenv command 193
PRINTER environment variable 367
printf command 201
printing 364–375
    see also CUPS
    architecture 364–365
    debugging 373–375
    Internet Printing Protocol (IPP) 365
    network printers 371
    service shutoff 372
privacy 1150
    and **sendmail** 646–647
PRIVACY_FLAGS option, **sendmail** 639
privacy legislation, E.U. 1148
PrivacyOption variable, **sendmail** 647
private addresses 394–396
private cloud 275
privileged ports 388, 1045
/proc directory 126
procedures 1141–1144
processes 90–119
    components of 90–93
    control terminal 93
    EGID 92
    EUID 92
    GID 92

init *see* **init** process
life cycle 93–98
monitoring of 98–101, 101–103
namespaces 91
niceness 93, 103–104
nice value 1106
open files 123
orphaned 94
ownership 67
periodic 109–119
PID 91
PPID 91
priorities 104
runaway 107–109, 1107
spontaneous 41
starting and stopping 93–98
states 97–109
tracing 106–107
UID 92
uninterruptible 98
zombie 98
/proc filesystem 104–105, 124, 335, 342, 782, 1094
procfs filesystem 105, 342
procmail command 609
production environment 970
/etc/profile.d directory 260
/etc/profile file 256, 260
.profile file 194, 259
Prometheus 1065
proxy cache, web server 700
proxy, HTTP 695–696
ps command 98–101, 124, 1098, 1106
pseudo-accounts 78–80
pseudo-devices 332
pseudo-groups 78–80
pseudo-random number generators 1028
PTR DNS records 525
public cloud 275
public key authentication 1036
Public Key Infrastructure (PKI) 1024
PUE (Power Usage Effectiveness) 1113
Puppet 853, 856, 862, 1127
and Docker 959

Purdue 1160
pvchange command 760
pvck command 760
pvcreate command 760, 761
pvdisplay command 760
PVH (ParaVirtualized Hardware) 917
PVHVM (ParaVirtualized HVM) 916
pw command 256, 258, 266, 1012
pwconv utility 253
PXELINUX 156
PXE (Preboot eXecution Environment) 154–155
Python language 6, 187, 215–223, 703
    best practices 231–232
    command-line arguments 220–221
    data types 219–220
    dictionaries 219–220
    files 219–220
    indentation 217–218
    input validation 220–221
    lists 219–220
    loops 222–223
    numbers 219–220
    package management 231
    strings 219–220
    tuples 219–220
    variables 219–220
    versions 215–216
    virtual environments 233
    vs. Ruby 223

Q

QCon conference 19
QEMU PC emulator 916
qmgr daemon 672
qshape command 683
quad A DNS records 525
Quagga 497
quay.io 952
QUEUE_LA option, **sendmail** 639, 648
QUIT signal 95, 96

R

RabbitMQ 323
rack density 1112
rack power 1112–1113
racks, equipment 1110
Rackspace 274
RAID (Redundant Array of Inexpensive Disks) 753, 765–775, 1090
  disk failure recovery 769
  levels 766–769
  RAID 5 drawbacks 770–771
  RAID 5 write hole 766, 771
  scrubbing 771
  software vs. hardware 766
  vs. LVM 759
RAID-Z *see* ZFS filesystem
Rails web development platform 223
RainerScript 314
Rake 976
RancherOS Linux 8
/dev/random device 1029
random number generation 1028
ransomware 998–999
Rapid7 1016
ratecontrol feature, **sendmail** 642
RBAC (Role-Based Access Control) 85, 261
/etc/defaults/rc.conf file 57
/etc/rc.conf file 428
/etc/rc.d directory 57
/etc/rc.d/rc.local script 55
/etc/rc script 57
real GID 67
realm command 596
realmd daemon 596–597
real UID 67
reboot command 59
reboot procedures 58
reboot.target target 49
recipient_delimiter option, Postfix 677
Red Flag Rule 1148
Red Hat Enterprise Linux *see* RHEL
Red Hat, Inc. 10
Red Hat Network (RHN) 167, 169

redirect driver, Exim 666
redirect feature, **sendmail** 634
Redis 282
Redundant Array of Inexpensive Disks *see* RAID (Redundant Array of Inexpensive Disks)
Reed, Darren 447
REFUSE_LA option, **sendmail** 639, 648
regexes *see* regular expressions
regions, cloud 279–280
regular expressions 209
  captures 213–214
  examples of 212–213
  failure modes 214
  literal characters 210
  matching process 210
re:Invent conference 19
reject command 373
reject_unauth_destination option, Postfix 680
RELAY_DOMAIN feature, **sendmail** 641
/etc/mail/relay-domains file 640
relay_entire_domain feature, **sendmail** 641
relay_hosts_only feature, **sendmail** 641
etc/postfix/relaying_access file 682
release 972
release candidate 972
Remedy 1133
removing accounts 265
rendezvous addresses, multicast 495
renice command 103–104, 1106
rescue mode 60
rescue.target target 49, 60
resize2fs command 764
resizing, filesystem 763–771
/etc/resolv.conf file 417, 504
resource records, DNS 503, 510–511, 518–530
  A 524
  AAAA 525
  CNAME 527
  DKIM 530
  DMARC 530
  DNSKEY 565
  DS 565
  MX 526

NS 524
NSEC 565
NSEC3 565
PTR 525
quad A 525
RRSIG 565
SOA 521
special characters in 519
SPF 530
SRV 528
TXT 529
types 520
REST (Representational State Transfer) 706
/etc/postfix/restricted_recipients file 682
reverse proxy, web server 700
reverse zone, DNS 506
revision control 236–241, 968
RFC1918 addresses 394–396, 547
RFC (Request for Comments) 17, 379
RHEL 7, 8, 10
RHN (Red Hat Network) 167, 169
Richards, Martin 1157
ripd daemon 497
RIPE DNSSEC tools 575
RIPE NCC 394
ripngd daemon 497
RIPng (Routing Information Protocol, next generation) 491, 492
RIP (Routing Information Protocol) 491, 492
risk assessment 1137, 1149
Ritchie, Dennis 1156
Rivest, Ron 1024
RJ-45 wiring standard 467
rm command 128
rmdir command 129
rmuser command 264, 266
rndc command 545–546, 556
/etc/rndc.conf file 546
rndc-confgen command 546
/etc/rndc.key file 546
Role-Based Access Control (RBAC) 85, 261
root account 249
see also RBAC
best practices 69

disabling 78
management of 69
user ID 67
root.cache file 545
/root directory 126
root filesystem 36, 38, 755
rootkits 1007
root server hints, DNS 544
root servers, DNS 510, 544
root shell 41
rotation, log file 321–323
round robin DNS 512–513
route command 415
routed daemon 497
routing, network 400–403, 485–500
adding 416
Cisco routers 498–500
configuration 415–417
cost metrics 491–492
daemons 489–492, 496–498
default 416, 422, 428
default routes 487, 495
deleting 416
distance-vector 490–491
link-state 491
multicast 494–495
next hop 486
protocols 489–492
redirects 403–405, 489
static 402, 489, 495
strategy 495–496
table 488
tables 401–402
RPC 808
rpc.idmapd daemon 824
rpm command 21, 164, 165
RRDtool 1075
RRSIG DNS records 565
RSA conference 19
RSA public key cryptosystem 1024
rsync command 604
/etc/rsyslog.conf file 306
rsyslog.conf file 316–318

rsyslogd daemon 304
architecture 305–306
configuration 306–316, 316–318
legacy configuration options 313
message properties 315
versions 306
RT: Request Tracker 1132
Ruby language 6, 187, 223–230, 703
as a filter 230
best practices 231–232
blocks 226–228
environment management 233–236
hashes 228
installation of 224
package management 231
regular expressions 228–230
symbols 228
vs. Python 223
runaway processes 107–109
/run directory 126
/var/run directory 126
run levels, **init** 42, 49
rvm environment manager 233–236

S

SaaS (Software as a Service) 277
Safari Books Online 17
Safe Harbor 1148
SAGE-AU 1153
SailPoint IdentityIQ 269
Salt 853, 856, 862–863, 883–906, 1127
comments on 862
comparison to Ansible 907–909
debugging 905
dependencies 893–895
and Docker 959
documentation 906–907
environments 901
formulas 900–901
functions 895–896
globbing 888

highstates 899–900
installation of 883–885
and Jinja 891–893
matching, minion 888–889
parameters 896–899
pillars 884
ports, network 885
pros and cons 909
security 885, 908
setup, minions 885
.sls files 887
state binding, minions 899
state IDs 893–895
states 884, 890–891
variables, minions 886–888
salt command 886
/srv/salt directory 884
salt-key command 886
salt-master daemon 884, 885, 886
salt-minion daemon 886
Salt Open 883
SaltStack 883
Samba 833–843
see also SMB (Server Message Block)
and AD 833, 836
browsing shares 839
character sets 843
configuring shares 836–841
debugging 841–843
group permissions 837–839
installation of 834
local authentication 835
logging 841
mounting shares 839
security 840
/var/log/samba/* file 299
SAML (Security Assertion Markup Language) 587
SANS (SysAdmin, Audit, Network, Security) Institute 1053, 1153, 1161
Sarbanes-Oxley Act (SOX) 261, 1050, 1146, 1149
sar command 1074, 1089, 1103–1104
SAS (Serial Attached SCSI) interface 743–744
formatting, disk 748
SATA (serial ATA) interface 742

formatting, disk 748
secure erase 749
Satellite Server 167
savecore command 363
saved GID 67
saved UID 67
/sbin directory 125, 126
/usr/sbin directory 126
SCALE conference 19
Schneier, Bruce 1010, 1023, 1052
Schroeder, Bianca 739
scientific method 1093
scp command 1033, 1044
scripting 182–243
see also **bash**
see also Perl
see also Python language
automation 184–185
choosing a language 186–187
error messages, useful 188
languages 6
microscripts 183–184
philosophy 183–189
style 188
SCSI (Small Computer System Interface) 743–744
SDN (Software-Defined Networking) 477
search path 20
second-level domain name 507
SecOps 1078
Secret Server 1011
/var/log/secure file 299
Secure Hash Algorithm (SHA-1, SHA-2, SHA-3) 1027
Secure Sockets Layer (SSL) 693, 1026
security
see also cryptography
access control 65–68
aging, password 1012
and Ansible 882, 908
architecture 1046
attack response 1021
attack surface 1004
auditing 1016
authentication, public key 1036–1037

and automation 1004
AWS 452–457
and backups 1006
basic measures 1004–1009
Blowfish hash 249
boot loader 1003
botnets 1002
broadcast ping 426
buffer overflows 1001
certifications 1048–1052
chain of trust, DNSSEC 572
CIA triad 1000
configuration errors 1003–1004
credit card data 1148
of credit cards *see* PCI DSS (Payment Card Industry Data Security Standard)
data loss prevention (DLP) 618
DDoS 1002
defense in depth 1004
deleting accounts 265
demoting data 971
DES hash 249
disk erasure 749–750
DMZ 1046
DNS 558–576
DNSSEC 564–576
and Docker 937, 955–958
elements of 1000
encryption 1026, 1036–1037
event logging 1006
of Exim 654
file integrity monitoring 1079
file transfer, secure 1044
firewall, Linux or UNIX as a 441–450
firewalls 1045–1047
group logins 1012
handling attacks 1054–1056
hash, cryptographic 1026–1028
home directory permissions 260
and HTTP 688
ICMP redirects 426
incident handling 1054–1056
incident hotline 1055
insider abuse 1003

intrusion detection 1017–1021, 1080–1081
IoT 1002, 1164
IP forwarding 426
and IP networking 408–411
IPsec 1048
Kerberos 81
least privilege 1004
and load balancing 698
locking accounts 78, 266–267
malware 1001
MD5 hash 247
monitoring 1078–1080
multifactor authentication 1008
NFS 806–807, 811–812, 823
of open source software 1029
open vs. closed operating systems 1002
packet filtering 1007–1008
passphrase 1009
password cracking 1017
password expiration 251, 253
password hashes 246, 247–249
passwords 1009–1013
passwords, obsolescence of 1008
password strength 248, 1009
patching schedule 1004
penetration testing 1009, 1016
phishing 1001, 1150
port scanning 1013–1015
of Postfix 673
power tools 1013–1021
privileged ports 388, 1045
and random numbers 1028
removing accounts 265
root account 249, 1013
root account, disabling 78
rootkits 1007
and Salt 885, 908
and Samba 840
self-assessments 1008
SELinux 83, 85
of **sendmail** 643–649
SHA-512 hash 247
shell, secure (ssh) 1033–1045

and SNMP 1083
social engineering 1000–1001
source routing 426
sources of compromise 1000–1003
sources of information on 1052–1054
spear phishing 1001
standards 1048–1052, 1141
sudo command 70
of syslog messages 318
system accounts, non-root 78–80
of TCP/IP 486, 489
TrustedBSD 83
unnecessary services 1005
updates, software 1004
vigilance 1008
viruses 1006–1007
VPN 411–412, 1047
vs. convenience 999
vulnerabilities, software 1001–1002
vulnerability scanning 1015–1016
of wireless networks 476
worms 1006–1007
Security Assertion Markup Language (SAML) 587
SecurityFocus 1052
security incidents 1140
SecuritySpace 623
Seeley, Donn 1161
segmentation violation 96
segment, TCP 384
SEGV signal 95
Selenium 974
/etc/selinux directory 87
SELinux (Security-Enhanced Linux) 83, 85
Sender ID 617
Sender Policy Framework (SPF) 612, 617
sendmail 622, 624–651
see also aliases, email
see also email
see also spam
blacklists 641
and **chroot** 647
command line flags 626
configuration 628–635

daemon mode 626
databases 631–632
directory locations 624
and LDAP 635
load average limit 639
logging 650–651
m4 and 624–625
masquerading 636–637
open relay 639, 640
permissions 645–646
privacy 639, 646–647
queue monitoring 649
queue processing 626
queues 627
rate and connection limits 639
sample configuration 630
security 643–649
and the service switch file 625
starting 625
testing and debugging 649–651
sendmail.cf file 626, 628
Sensu 1064
Server Fault 19
serverless, cloud 284
server mode 41
ServiceDesk 1133
Service Level Agreement (SLA) 1144–1146
ServiceNow 1133
/etc/services file 388, 1045
setfacl command 142–152
setgid bit 250
setgid execution 68
setrlimit system call 1107
setuid execution 68, 92
setuid/setgid bits 68–69, 133
sfdisk command 731
sftp command 1033, 1044
sftp-server command 1033
sg_format command 748
SHA-1, SHA-2, SHA-3 (Secure Hash Algorithm) 1027
SHA-512 hashing algorithm 247
/etc/shadow file 68, 246, 251–253, 1011
shadow passwords 252–253, 253–255

Shamir, Adi 1024
Shapiro, Greg 649
/usr/share directory 126
shares, NFS 809
sharing a filesystem *see* NFS
shell, root 41
shell scripting 189–198
Shibboleth 269
showmount command 821
shred command 749
sh shell 186
    see also **bash**
    arithmetic 209–210
    command-line arguments 203–205
    comparison operators 205
    control flow 205–206
    execution of 198–199
    file evaluation 206
    functions 203–205
    globbing 12, 209
    I/O 201–202
    loops 207–208
    scripting 198–209
shutdown command 59, 60
shutdown procedures 58
Shuttleworth, Mark 9
Siemon 483
SignalFx 1067
signals 94–97
    BUS 95
    caught, blocked, or ignored 95
    CONT 95, 96
    HUP 95, 96
    INT 95, 96
    KILL 95, 96
    list of important 95
    QUIT 95, 96
    SEGV 95
    sending 97
    STOP 95, 96
    TERM 95, 96
    tracing 106–107
    TSTP 95, 96

USR1 95
USR2 95
WINCH 95
Silicon Graphics, Inc. 777
Simple Mail Transport Protocol *see* SMTP
single-mode fiber 468
single-user mode 38, 41, 60, 60–62
cloud instances 62–63
FreeBSD 61
Linux 60–62
remounting the root filesystem 61
Site Reliability Engineer (SRE) 26
/etc/skel directory 260, 262
SLAAC (StateLess Address AutoConfiguration) 399
Slack 1126
Slackware Linux 8
/etc/openldap/slapd.conf file 592
slapd daemon 592
SLA (Service Level Agreement) 1144–1146
slices *see* partitions, disk
slurpd daemon 592
smartctl command 751
smartd daemon 751
SMART_HOST macro **sendmail** 637
SMART (self-monitoring, analysis, and reporting technology) 750–751
/usr/local/etc/smb4.conf file 834
smbclient command 839
/etc/samba/smb.conf file 834
smbd daemon 833–834
smbpasswd command 835
SMB (Server Message Block) 832–843
history of 832–833
vs. NFS 805, 834
smbstatus command 841
S/MIME email encryption 618
SmokePing 438–439
smrsh command 645
SMS notifications 1060
SMTP 613
authentication 615
commands 614
debugging 614
error codes 614

status messages 615
smtpd daemon 671
smtpd_recipient_restrictions option, Postfix 681, 682
smtpd_*_restrictions options, Postfix 680
smtpd_sasl_auth_enable option, Postfix 682
smtpd_tls_* options, Postfix 682
smtp transport, Exim 668
smurf attacks 409, 425
snapshots, volume 762–763
SNI (Server Name Indication) 694
snmpd.conf file 1083
snmpd daemon 1082, 1083–1086
snmpdelta command 1084
snmpdf command 1084
/etc/snmp directory 1083
snmpget command 1084
snmpgetnext command 1084
snmpset command 1084
SNMP (Simple Network Management Protocol) 1073, 1080–1085
agents 1082
community string 1083
graphing 440–442
MIB 1081
organization 1081–1082
protocol operations 1082–1083
traps 1082
snmptable command 1084
snmptranslate command 1084
snmptrap command 1084
snmpwalk command 1084
Snort network intrusion detection system 1018
Snowden, Edward 998
SOAP (Simple Object Access Protocol) 706
SOA (Start of Authority) DNS records 521
social coding 240–242
sockets, local domain 128, 130–131
socket system call 131
soft_bounce option, Postfix 684
software
see also software packages
installation from source code 23–24
installing from a web script 24
package management 21–23

Software as a Service (SaaS) 277
Software-Defined Networking (SDN) 477
software delivery 4
software packages
    see also software
    localization 178–181
    management 163–164
Solaris 10
SolarWinds 1067
Sony 1007
sort command 194
/etc/apt/sources.list file 171
SOX (Sarbanes-Oxley Act) 1149
spam
    see also email
    blacklists 641
    cloud-based services 616
    open relay 639, 640
    Sender ID 617
    and **sendmail** 639–643
    SPF (Sender Policy Framework) 612
spear phishing 1001
spectrum allocation, wireless 475
SPF (Sender Policy Framework) 612, 617
SPF (Sender Policy Framework) DNS records 530
splattercast 535
split DNS 547
/var/spool directory 126
SPOOL_DIRECTORY variable, Exim 652
Spotify 705
Squid caching server 701
/usr/src directory 126
SRE (Site Reliability Engineer) 26
/srv directory 126
SRV DNS records 528
ss command 419–420, 1005, 1090
SSD (Solid State Disk) 729, 733–734, 737–740, 1090, 1101
SSH 1033–1045
    agent 1037–1038
    aliases, host 1039
    client 1035–1036
    connection multiplexing 1040
    essentials 1033–1035

file transfer 1044
keys 1036–1037
password authentication, disabling 1039
port 1039
port forwarding 1040
server 1041–1043
SSHFP verification 1043–1044
ssh-add command 883, 1033, 1037
ssh-agent command 883, 1033
ssh-agent daemon 1037
ssh command 1033–1045, 1035–1036
ssh_config file 1034
sshd_config file 1013, 1027, 1034, 1042–1043
sshd daemon 856, 1033, 1041–1043
/etc/ssh directory 1034
~/.ssh directory 1035
SSHD (solid state hybrid drive) 740
SSHFP DNS record 1043
SSHFP host key verification 1043
ssh-keygen command 1033, 1036
ssh-keyscan command 1033
SSIDs, wireless 474
SSL (Secure Sockets Layer) 693, 1026
SSO (Single Sign-On) 587–605
see also LDAP
account management 268–270
for applications 268
concepts 587
elements of 588
and Kerberos 596–598
LDAP 589–595
PAM 588
and SaaS 587
SAML 587
sssd.conf file 599
sssd daemon 588, 596–598, 598, 836
Stack Overflow 19
staging environment 970
standard error 190
standard input 190
standard output 190
standards
see also IEEE standards

CJIS (Criminal Justice Information Systems) 1147
COBIT 1147
Common Criteria 1051
contingency planning 1149
Critical Infrastructure Protection (CIP) 1148
Family Educational Rights and Privacy Act (FERPA) 1147
Federal Information Security Management Act (FISMA) 1147
FISMA 1049
Gramm-Leach-Bliley Act (GLBA) 1148
Health Insurance Portability and Accountability Act (HIPAA) 1049, 1148
IEEE 802.1* 464, 470, 471, 473, 474
Information Technology Infrastructure Library (ITIL) 1149
ISO 27001:2013 1050, 1141, 1148
ISO 27002:2013 1148
NERC CIP 1049
NIST SP 800-34 1138, 1149
NIST SP 800-53 1149
NIST SP 800 series 1051
OWASP 1051
Payment Card Industry Data Security Standard (PCI DSS) 1148
PCI DSS 1050
Red Flag Rule 1148
RJ-45 wiring 467
Safe Harbor 1148
Sarbanes-Oxley Act (SOX) 1149
security 1048–1052
TIA/EIA-568A 467
TIA/EIA-606-B 479
wireless 473
wiring 479
standard services 388
Stanford Law School 1025
STARTTLS extension, **sendmail** 648
startup scripts 57
statd daemon 811
stateful inspection firewalls 1046
State University of New York (SUNY) Buffalo 1160
static code analysis 974
static routes 402, 495
StatsD 1059, 1069–1071, 1071
StatusCake 1068
STDERR file descriptor 190
STDIN file descriptor 190

STDOUT file descriptor 190
STD (Standard) 380
sticky bit 133–134
STOP signal 95, 96
storage
    block 282
    ephemeral 282
    layers of 752
    object 282
storage management *see* disks
/etc/carbon/storage-schemas.conf file 1069
strace command 106–107
Stuxnet worm 998
subdomains, DNS 507
submit.cf file 628
subnetting 390–391
Subversion 978
su command 70
sudo command xxxiii, 70–77, 1009
    configuration, example 71
    configuration, site-wide 76
    pros and cons 72
    using with Ansible 866
    using without password 75
    using with Salt 890
    vs. advanced access control 73
    without a control terminal 76
/etc/sudoers file 71–73
Sumo Logic 324
Sun Microsystems 925
superblock, filesystem 777
superuser *see* root account
Supervisor 1077
supervisord daemon 1077
SUSE Linux 8
swapctl command 784
swapon command 780, 784, 1099
/proc/sys/vm/swappiness parameter 1099
swap space 783–784, 1098
Swarm 963
Sweet, Michael 365
switches, Ethernet 469
symbolic links 128, 131–132

sync system call 777
/etc/sysconfig directory 55, 421–422
sysctl command 342, 347, 361, 429, 1073
/etc/sysctl.conf file 342, 347, 425, 429
Sysdig Cloud 1067, 1076
sysdig tool 1075, 1076
/sys directory 126, 334
/usr/src/sys directory 348
sysfs filesystem 334
syslog 304–320
    see also log files
    see also logging
    actions 312
    and DNS logging 576–582
    facility names 310
    messages 305
    security 318
    severity levels 311
    and **systemd** journal 303
/etc/syslog.conf file 309–312
syslogd daemon 304
/var/log/syslog* file 299
system administration
    adjacent disciplines 26–27
    conferences 19
    essential tasks 3–6
    GUI tools 6
    keeping current 18
    metrics 1146
    prioritization 1145–1146
    resources for reading about 18
    service descriptions 1144–1154
system administrator
    and CI/CD 966
    common tasks 1143
    distinguishing characteristics 1057
    history 1159–1160
    localization guidelines 179
    professional attributes of 1057
    responsibilities 1129
    role in DevOps 1129
    roles 1133
    tool box 1122

SYSTEM_ALIASES_FILE variable, Exim 652
system calls, tracing 106
system-config-kickstart tool 156
systemctl command 45–46, 60
systemd daemon 31–32, 43, 44–57, 94, 988
    and **init** scripts 54
    caveats 54
    dependencies 50–51
    and Docker 947
    execution order 51
    journal 296, 300–301, 301–304
    logging 55
    management of 45–46
    targets 48–50, 49
    timers 114–118
    unit files 44
    unit statuses 47–48
    vs. **init** 43
systemd-journald daemon 301–304, 320
systemd-journal-remote tool 303
System V UNIX 1159

T

tail command 197
tape, magnetic 802–803
targets, **systemd** 49
task management 1129
T-BERD line analyzer 478
tcpdump tool 435–438, 687
TCP/IP 380
    see also IP
    see also IPv6
    see also networking
    connection reuse 692
    Fast Open (TFO) 692
tcsh shell 189
technical debt 1123
tee command 196
Teleport 1044
telinit command 60
temperature

data center 1115
effect on hard disks 735
TERM signal 95, 96
Terraform 283, 454–457, 981, 989–992
terraform command 455
/bin/test command 205
testing
acceptance 974
infrastructure 975
integration 974
performance 974
software localization 180–181
static code analysis 974
unit 974, 983–984
testparm command 834
TFO (TCP Fast Open) 692
The Open Group 269
Thompson, Ken 1156
ThoughtWorks 965
threads 91, 97–109
threat categories, disaster 1137
Thycotic 1011
TIA/EIA-568A wiring 467
ticket-granting ticket, Kerberos 596
ticketing 1129
ticketing systems 1132
time-to-live field, IP 433
TLS (Transport Layer Security) 643, 648, 693, 1026
/tmp directory 125, 126
/usr/tmp directory 126
/var/tmp directory 126
/tmp filesystem 755
token ring 383
toolbox 1122
TO_* options, **sendmail** 639
top command 101–103, 1092, 1106
Torvalds, Linus 18, 236, 1162
Townsend, Jennine 67
traceroute command 432, 433–435
Track-It! 1133
Transport Layer Security (TLS) 411, 643, 648, 693, 1026
Tridgell, Andrew 833
Tripwire 1079

Troan, Erik 321
Troposphere 283
troubleshooting
see also performance
Amazon Web Services (AWS) instances 62
BIND 576–585
booting 59–60
cloud systems 62–63
DigitalOcean instances 63
DNS 576–585
Docker 958–960
Exim 670
Google Compute Engine (GCE) instances 63
HTTP connections 687, 691–692
kernel 360–363
network 429–438, 477–478
performance 1106–1107
Postfix 682
printing 373–375
Salt 905
Samba 841–843
sendmail 649
SMTP 614
syslog 320–321
TLS servers 1031
web caching 701
trunks, Ethernet 470
truss command 106–107
TrustedBSD 83
tshark command 438
Ts’o, Theodore 776
TSTP signal 95, 96
TTL (time-to-live), DNS 512
tugboat cli tool 290, 459
tune2fs command 779
tunefs command 782
Tweedie, Stephen 777
Twofish 1023
TXT DNS records 529
typographical conventions 11–12

U

UA (mail User Agent) 607
UBER (Uncorrectable Bit Error Rate) 740
Ubiquiti 475
Ubuntu Linux 8, 9
udevadm command 334, 335–336, 338
/etc/udev/udev.conf file 336
udevd daemon 333, 336–340
UDP (User Datagram Protocol) 380
UEFI (Unified Extensible Firmware Interface) 32, 33–35
    bootstrap path 34, 39
UFS filesystem 776–784
ufw command 442, 1008, 1045
UIDs *see* user IDs
ulimit command 1107
umask command 138
umask, default 254, 260
umount command 123–124, 780
uname command 329, 349, 939
Uncorrectable Bit Error Rate (UBER) 740
unicast, IP 388
unicast packets 465
unicast Reverse Path Forwarding (uRPF) 410
Uniform Resource Locators (URLs) 687–688
uninterruptible power supplies 1111–1112
uniq command 196
United Nations 378
units 12–13
unit tests 974, 983–984
Universal Plug and Play (UPnP) 396
Universal Serial Bus (USB) interface 744–745
University of California at Berkeley 962, 1158
University of Cambridge 622, 651, 920
University of Colorado at Boulder xxxii, 1160
University of Maryland 1160
University of Utah 1160
UNIX
    see also FreeBSD
    as a firewall 410, 441–450
    history of 1156–1158
    origin of name 1157
    philosophy 1158
    reasons to choose 1088
    security of 999

and viruses 1006
unlink system call 131
unshielded twisted pair *see* UTP cables
unsolicited commercial email *see* spam
updatedb command 21
updates, software 1004–1005
uptime command 1058, 1071, 1097, 1106
Uptime Institute, The 1119–1120
/dev/urandom device 332, 1029
URI (Uniform Resource Identifier) 687
URLs (Uniform Resource Locators) 687–688
URN (Uniform Resource Name) 687
uRPF (unicast Reverse Path Forwarding) 410
USB drive mounting 783
USB (Universal Serial Bus) interface 744–745
U.S. Department of Defense 378
use_cw_file feature, **sendmail** 633
USENIX Association 1153, 1161
user accounts 244–270
adding 257–261, 262–265
attributes 251–253
centralized management 268–270
defaults 254–255
deleting 262–265, 265
encrypted passwords 247
GECOS field 246, 250
GID 246, 250
home directory 251, 259–260, 260
identity management 269–270
idle timeout 255
locking 78, 266–267
login name 246–247
login shell 251
nobody 813
password algorithm 248
password expiration 252
password quality 248
passwords 1009–1013
password, setting 258
password strength 248
pseudo-accounts 79–80
RBAC 261
removing 262–265, 265

shadow passwords 252–253
startup files 259
UID 246, 249
umask 254, 260
useradd command 256, 262–263
/etc/default/useradd file 262
userdel command 266
userdel.local script 266
user IDs 92, 245, 246, 249
mapping to names 67
real, effective, and saved 67
usermod command 253, 267
usernames *see* user accounts
USR1 signal 95
USR2 signal 95
/usr directory 125, 126
UTP cables 464, 465, 478
UUID 783

V

Vagrant 928
van Rossum, Guido 187
/var directory 125, 126, 755
variables, environment 193–194
Varnish caching server 701
vault, password 1010–1012
VAX 1159
Velocity conference 19
vendors we like 483
Venema, Wietse 622, 670
VeriSign 1024
Veritas 760
Verizon Data Breach Investigations Report 1052
vgchange command 760
vgck command 760
vgcreate command 760, 761
vgdisplay command 760, 764
vgextend command 760
vgscan command 760
Viavi 478, 483
vi command 6

vigr command 258
.viminfo file 259
.vimrc file 259
vipw command 248, 253, 258
virsh command 923
virt-install command 921–923, 923–924
virt-manager package 921–923
virtual_alias_* options, Postfix 679
VirtualBox 925
virtualenv package 233
virtual hosts, web server 693–695
    in **httpd** 693–695
    in NGINX 718
virtualization 914–929
    see also KVM
    see also Xen
    containerization 918–920
    on FreeBSD 924
    full 915–916
    hardware-assisted 916
    HVM (Hardware Virtual Machine 916
    hypervisors 915–918
    images 918, 925–928
    on Linux 920–924
    live migration 918
    paravirtualization 916
    provisioning 928
    PVH (ParaVirtualized Hardware) 917
    PVHVM (ParaVirtualized HVM) 916
    QEMU 916
    type 1 vs. type 2 917–918
    vs. containers 920
virtual_mailbox_* options, Postfix 679
Virtual Private Network (VPN) 411–412, 1047
virtual private servers 281
virtusertable feature, **sendmail** 635
/etc/mail/virtuserable file 635
viruses 1006–1007
virus scanning
    see also email
Visa 1050
visudo command 72
Vixie, Paul 535

VLANs 470
trunking 470
wireless 475
vmstat command 1074, 1089, 1092, 1096–1097, 1106
VMware 924
VMware ESXi 918
VMware Identity Manager 269
VMware vCloud Air 274
VMware Workstation 918
VMWorld conference 19
volume groups 760
relations to other layers 752–754
VPN (Virtual Private Network) 411–412, 1047–1048
vtysh daemon 497
vulnerabilities, software 1001–1002
vulnerability scanning 1015–1016

W

Wall, Larry 187
Watson, Robert 83
wc command 196
web hosting 694–706
APIs 704–706
architecture 695, 697
build vs. buy 706
cache 699
in the cloud 706
components 695
proxy server 700
reverse proxy 700
serverless 708
server types 694
static content 708
TCP connection reuse 692
virtual hosts 693–695
Well-Known Service (WKS) ports 1045
wget command 24
wheel group 70, 250
whereis command 20
which command 20
Whisper 1066

Wi-Fi networks 473
Wi-Fi Protected Access (WPA) 476
WINCH signal 95
Windows Defender 1007
Wired Equivalent Privacy (WEP) 476
wireless networks 473–476
    access points (APs) 474
    channels 475
    frequency spectrum 475
    security 476
    SSIDs 474
    topology 474
    VLANs 475
Wireshark 435–438
wiring, building 478
wisdom, Evi's tenets of xxxiii
World Wide Web Consortium 269
worms 1006–1007
WPA *see* Wi-Fi Protected Access
wpa_supplicant command 474
write hole, RAID 5 766, 771
/var/log/wtmp file 299, 300

X

X.500 directory service 589
Xen 916, 920–921
    see also virtualization
    components 921
    dom0 921
    guest installation 921–923
    overhead 921
    virtual block devices (VBDs) 922
/etc/xen directory 921
/var/log/xen/* files 299
XenServer 918
XFS filesystem 776–784
xfs_growfs command 765
xl tool 922
XML (Extensible Markup Language) 705
/var/log/Xorg.n.log file 299
XORP (eXtensible Open Router Platform) 498

Y

YAML 857, 863–865, 891–893
Ylönen, Tatu 1033
yum command 22, 164, 167, 174–175
/var/log/yum.log file 299

Z

zebra daemon 497
Zenoss 1067, 1128
/dev/zero file 332
zero downtime deployment 977
zfs command 788, 789
ZFS filesystem 753, 769, 784–786, 786–796
    clones 792–793
    disk addition 788
    and Docker 946
    inheritance, property 790
    and Linux 787
    properties, filesystem 789–790
    RAID 788
    raw volumes 793–794
    snapshots 792–793
    spare disks 796
    storage pool 788, 794–796
    vs. Btrfs 796–797
Zimmermann, Phil 1031
Zix 618
zombie processes 98
zones, DNS 506
    forward 506
    forwarding 545
    localhost 549, 550
    master 542
    reverse 506, 525, 526
    signing 569
    slave 544
    transfers 555
zpool command 788
Zulip 1126