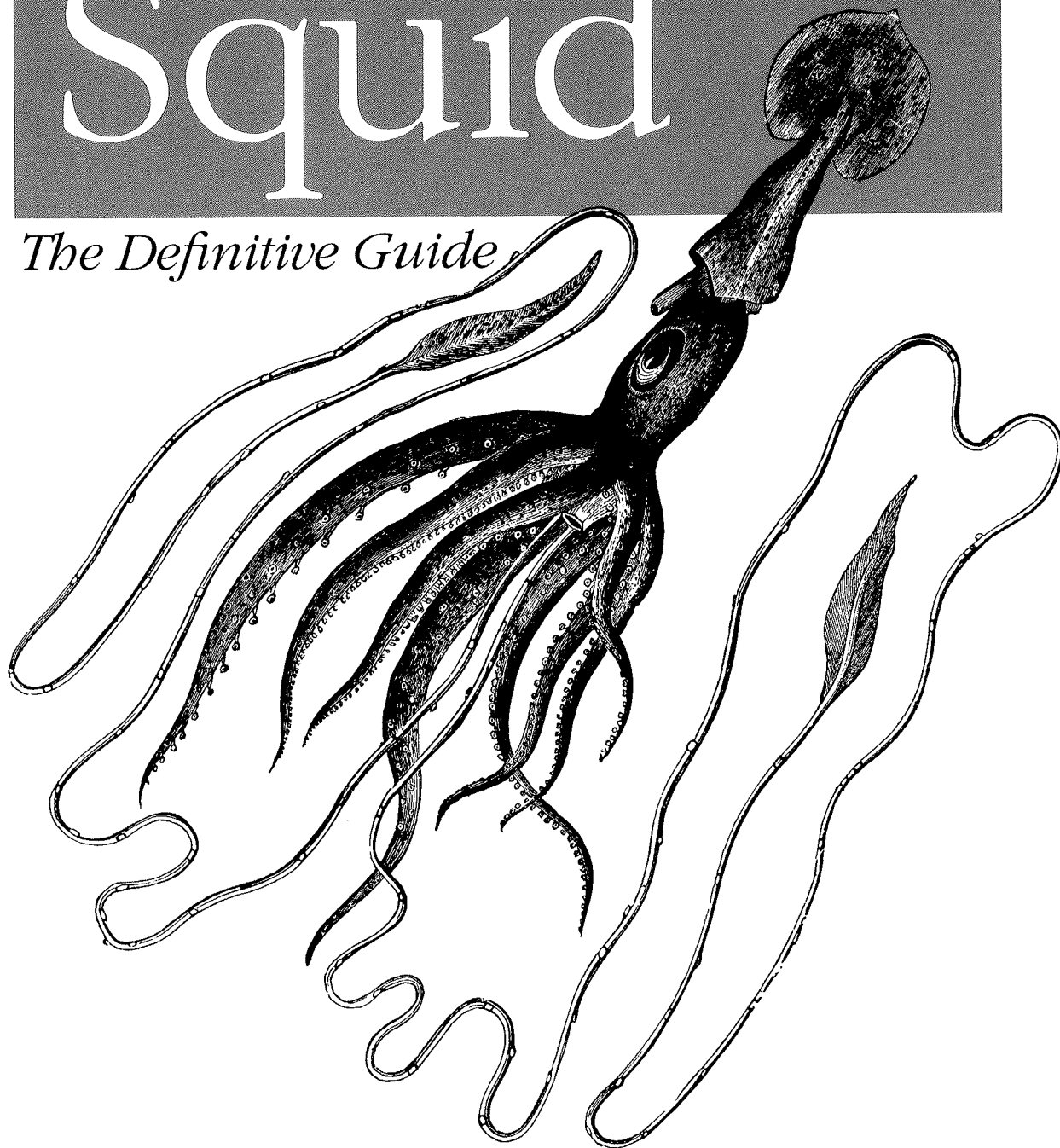


Making the Most of Your Internet Connection

Squid

The Definitive Guide



O'REILLY®

Duane Wessels
Bright Data Ltd. v. Oxylabs, UAB
IPR2023-00840, EX. 1009
1 of 468

Squid

The Definitive Guide

Other resources from O'Reilly

Related titles	Web Caching	Web Performance Tuning
	Apache: The Definitive Guide	Perl for Web Site Management
	Apache Cookbook	

oreilly.com *oreilly.com* is more than a complete catalog of O'Reilly books. You'll also find links to news, events, articles, weblogs, sample chapters, and code examples.



oreillynet.com is the essential portal for developers interested in open and emerging technologies, including new platforms, programming languages, and operating systems.

Conferences O'Reilly & Associates brings diverse innovators together to nurture the ideas that spark revolutionary industries. We specialize in documenting the latest tools and systems, translating the innovator's knowledge into useful skills for those in the trenches. Visit *conferences.oreilly.com* for our upcoming events.



Safari Bookshelf (*safari.oreilly.com*) is the premier online reference library for programmers and IT professionals. Conduct searches across more than 1,000 books. Subscribers can zero in on answers to time-critical questions in a matter of seconds. Read the books on your Bookshelf from cover to cover or simply flip to the page you need. Try it today with a free trial.

Squid

The Definitive Guide

Duane Wessels

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

Bright Data Ltd. v. Oxylabs, UAB
IPR2023-00840, EX. 1009
5 of 468

Squid: The Definitive Guide

by Duane Wessels

Copyright © 2004 O'Reilly & Associates, Inc. All rights reserved.
Printed in the United States of America.

Published by O'Reilly & Associates, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly & Associates books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (safari.oreilly.com). For more information, contact our corporate/institutional sales department: (800) 998-9938 or corporate@oreilly.com.

Editor: Tatiana Apandi Diaz
Production Editor: Mary Anne Weeks Mayo
Cover Designer: Ellie Volckhausen
Interior Designer: Melanie Wang

Printing History:

January 2004: First Edition.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly & Associates, Inc. *Squid: The Definitive Guide*, the image of a giant squid, and related trade dress are trademarks of O'Reilly & Associates, Inc.

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 O'Reilly & Associates, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 0-596-00162-2

[M]

*To my darling Anne.
You have no idea.*

Table of Contents

Preface	xiii
1. Introduction	1
Web Caching	2
A Brief History of Squid	3
Hardware and Operating System Requirements	4
Squid Is Open Source	5
Squid's Home on the Web	5
Getting Help	5
Getting Started with Squid	7
Exercises	8
2. Getting Squid	9
Versions and Releases	9
Use the Source, Luke	10
Precompiled Binaries	11
Anonymous CVS	11
devel.squid-cache.org	12
Exercises	12
3. Compiling and Installing	13
Before You Start	13
Unpacking the Source	14
Pretuning Your Kernel	14
The configure Script	18
make	27
make Install	29

Applying a Patch	32
Running configure Later	33
Exercises	34
4. Configuration Guide for the Eager	35
The squid.conf Syntax	35
User IDs	37
Port Numbers	38
Log File Pathnames	38
Access Controls	39
Visible Hostname	40
Administrative Contact Information	42
Next Steps	42
Exercises	42
5. Running Squid	43
Squid Command-Line Options	43
Check Your Configuration File for Errors	45
Initializing Cache Directories	46
Testing Squid in a Terminal Window	46
Running Squid as a Daemon Process	47
Boot Scripts	48
A chroot Environment	50
Stopping Squid	51
Reconfiguring a Running Squid Process	52
Rotating the Log Files	53
Exercises	53
6. All About Access Controls	54
Access Control Elements	54
Access Control Rules	78
Common Scenarios	85
Testing Access Controls	91
Exercises	92
7. Disk Cache Basics	93
The cache_dir Directive	94
Disk Space Watermarks	99
Object Size Limits	99

Allocating Objects to Cache Directories	100
Replacement Policies	101
Removing Cached Objects	102
refresh_pattern	104
Exercises	106
8. Advanced Disk Cache Topics	108
Do I Have a Disk I/O Bottleneck?	108
Filesystem Tuning Options	110
Alternative Filesystems	111
The aufs Storage Scheme	112
The diskd Storage Scheme	115
The coss Storage Scheme	119
The null Storage Scheme	122
Which Is Best for Me?	122
Exercises	123
9. Interception Caching	124
How It Works	124
Why (Not) Intercept?	127
The Network Device	130
Operating System Tweaks	141
Configure Squid	146
Debugging Problems	148
Exercises	151
10. Talking to Other Squids	152
Some Terminology	152
Why (Not) Use a Hierarchy?	153
Telling Squid About Your Neighbors	155
Restricting Requests to Neighbors	159
The Network Measurement Database	162
Internet Cache Protocol	164
Cache Digests	172
Hypertext Caching Protocol	174
Cache Array Routing Protocol	176
Putting It All Together	178
How Do I ...	182
Exercises	183

11. Redirectors	185
The Redirector Interface	186
Some Sample Redirectors	187
The Redirector Pool	189
Configuring Squid	190
Popular Redirectors	192
Exercises	193
12. Authentication Helpers	194
Configuring Squid	195
HTTP Basic Authentication	195
HTTP Digest Authentication	201
Microsoft NTLM Authentication	204
External ACLs	207
Exercises	210
13. Log Files	211
cache.log	212
access.log	215
store.log	227
referer.log	231
useragent.log	232
swap.state	233
Rotating the Log Files	235
Privacy and Security	236
Exercises	236
14. Monitoring Squid	237
cache.log Warnings	237
The Cache Manager	239
Using SNMP	290
Exercises	301
15. Server Accelerator Mode	302
Overview	303
Configuring Squid	304
Gee, That Was Confusing!	308
Access Controls	310
Content Negotiation	311

Gotchas	312
Exercises	314
16. Debugging and Troubleshooting	315
Some Common Problems	315
Debugging via cache.log	319
Core Dumps, Assertions, and Stack Traces	323
Replicating Problems	326
Reporting a Bug	329
Exercises	329
A. Config File Reference	331
B. The Memory Cache	399
C. Delay Pools	401
D. Filesystem Performance Benchmarks	407
E. Squid on Windows	418
F. Configuring Squid Clients	421
Index	433

Preface

About This Book

I started the Squid project eight years ago while working at the National Laboratory for Applied Network Research and the University of California. Back then I certainly enjoyed writing code and fixing bugs but always felt bad about the lack of decent documentation. This book is my attempt to rectify that situation. It's been a long time coming and almost didn't happen. Like they say, "better late than never!"

This book is written for those who are tasked with setting up and maintaining one or more Squid caches. If you're new to Squid, I'll show you how to download, compile, and install the code. Those of you who have been using Squid for a while will be more interested in the later chapters, where I talk about disk cache performance, modifying requests, surrogate mode, caching hierarchies, monitoring Squid, and more.

In order to use this book, you should have a basic knowledge of Unix systems. Many of the book's examples are based on free operating systems, such as Linux, FreeBSD, NetBSD, and OpenBSD. I also have some tips for Solaris users. If you're more comfortable with Windows systems, you can use Squid under a Unix emulator or give the native NT port a try.

Here's an overview of the book's contents:

Chapter 1, Introduction

This chapter introduces you to Squid and web caching. I give a brief history of the project, and a few notes on our future work. I explain how you can find additional support and information, including a FAQ, on the Squid web site.

Chapter 2, Getting Squid

In this chapter, I explain how and why you should download Squid's source code. You may prefer to install a precompiled binary or use a preconfigured package. I also talk about staying up to date with Squid using the anonymous CVS server.

Chapter 3, Compiling and Installing

Assuming you've downloaded the source code, this chapter explains how to configure and compile Squid. In some cases you may need to tune your system before compiling Squid. For example, your kernel may have relatively low file-descriptor limits that affect Squid's performance.

Chapter 4, Configuration Guide for the Eager

Here, I give a brief introduction to Squid's configuration file. If you are the impatient type and can't wait to start using Squid, this chapter will leave you with a minimal configuration file you can start playing with.

Chapter 5, Running Squid

In this chapter, I explain how to run Squid for the first time and how to test Squid in a terminal window. Following that, I suggest a number of ways to configure your system so that Squid starts each time it boots. I also explain how to reconfigure Squid while it is running and how to safely shut it down.

Chapter 6, All About Access Controls

I talk extensively about access controls in this chapter. Squid has a powerful collection of access control features and a number of different rule sets that determine how requests and responses are treated. This is an important chapter because a mistake in your access controls may leave your cache, or even internal systems, vulnerable to abuse from outsiders.

Chapter 7, Disk Cache Basics

This chapter is about Squid's primary function: storing cached responses on disk. I explain how to configure the disk cache, including replacement policies and freshness controls. I also show you how to manually remove unwanted objects from the cache.

Chapter 8, Advanced Disk Cache Topics

In this chapter, I explain how to improve the performance of Squid's disk cache. I'll talk about Squid's different storage schemes and a number of filesystem tuning options that may help. If your Squid cache handles a relatively light load, you probably don't need to worry about disk performance.

Chapter 9, Interception Caching

Here, I explain how to configure Squid for HTTP interception, sometimes also called transparent caching. Actually, configuring Squid is the easy part. The difficulty comes from setting up a router or switch on your network and the host from which Squid is running. I explain how to configure networking equipment from Cisco, Alteon, Foundry, and Extreme. I'll also show you how to configure your operating system (Linux, FreeBSD, NetBSD, OpenBSD, and Solaris) for HTTP interception. Finally, I talk about WCCP.

Chapter 10, Talking to Other Squids

In this chapter, I cover the ins and outs of cache cooperation, including meshes, arrays, and hierarchies. You may also find it useful if you simply need to forward

requests from Squid to another proxy or intermediary. I'll talk about the various intercache protocols supported by Squid (ICP, HTCP, Cache Digests, and CARP) and how Squid chooses the next-hop location for a given cache miss.

Chapter 11, Redirectors

Redirectors are the best way to make Squid rewrite HTTP requests before forwarding them. I describe the interface between Squid and a redirector program so that you can write your own. I also present a few of the more popular third-party redirectors available.

Chapter 12, Authentication Helpers

In this chapter, I explain how Squid interfaces with external authentication databases such as LDAP, NT domain controllers, and password files. Squid comes with a number of authentication helpers and understands Basic, Digest, and NTLM authentication credentials. I also document the API for each, in case you want to develop your own helper.

Chapter 13, Log Files

I cover Squid's various log files in this chapter, including *access.log*, *store.log*, *cache.log*, and others. I explain what each log file contains and how you should periodically maintain them.

Chapter 14, Monitoring Squid

This chapter provides a lot of information on monitoring Squid's operation. I cover both SNMP and Squid's own cache manager interface. You'll find it useful for both long-term monitoring and short-term problem diagnosis.

Chapter 15, Server Accelerator Mode

Squid's server accelerator mode is useful in a number of situations. You can use it to boost your origin server's poor performance, as a firewall to protect the server, or even to build your own content delivery network. I show how to set up Squid and make sure that outsiders can't abuse your service.

Chapter 16, Debugging and Troubleshooting

The book's final chapter explains how to debug and troubleshoot problems with Squid. You may find that some sites, or some user agents, don't work properly with Squid. I show how to isolate and reproduce the problem and how to present the information to Squid developers for assistance.

Appendix A, Config File Reference

This appendix is a reference guide for each of Squid's 200 configuration file directives. Each has a description, syntax, defaults, and examples.

Appendix B, The Memory Cache

This brief appendix explains a little about Squid's memory cache.

Appendix C, Delay Pools

You can use Squid's delay pools feature to limit bandwidth consumed by web surfers. I explain how the delay pools work and provide a number of example configurations.

Appendix D, Filesystem Performance Benchmarks

In this appendix, I present the results of numerous filesystem benchmarks. These may help you make informed decisions regarding particular operating systems, filesystem features, and Squid's storage techniques.

Appendix E, Squid on Windows

Have a look at this appendix if you'd like to run Squid on your Windows box. I talk about using Cygwin and about a native port of Squid, called SquidNT.

Appendix F, Configuring Squid Clients

This appendix contains information on how to configure various user agents to use Squid. I talk about manual configuration, environment variables, Proxy Auto-Configuration functions, and the Web Proxy Auto Discovery protocol.

As I'm finishing up this book, the latest stable version is Squid-2.5.STABLE4, and the development version is Squid-3.0. Perhaps the most important difference between the two is that Squid-3 is being rewritten in C++. You should find that most things are backward-compatible, although a few new configuration directives have been created. Please read the release notes carefully if you use Squid-3.0 or later.

I have created a web site for the book, located at <http://squidbook.org/>. There, you will find errata, supplemental information, and links to online resources.

Topics Not Covered

Due to a lack of time and space, there are some topics I was unable to cover in this book; they include:

Non-HTTP protocols

You'll find that I mostly talk about HTTP, even though Squid also supports FTP, Gopher, and some other relatively obscure protocols.

Customizing error messages

Squid's error messages can be customized and the source distribution includes versions of the error messages in a number of different languages. You can probably figure out how to customize the error messages by modifying the default pages or by reading Squid's source code.

Load balancing Squids

Load balancing is a popular way to increase the capacity of a caching service. Refer to one of the load balancing books mentioned in the following section if necessary.

What is cachable

HTTP has a number of somewhat complicated rules for determining what may, or may not be, cached, and for how long. Refer to *Web Caching*, or *HTTP: The Definitive Guide* (for more information, see the next section).

Copyright

A number of nontechnical issues surround web caching. These include copy-rights and privacy.

Modifying the source

I don't go into detail about Squid's source code in this book. The Squid project hosts a programmers' guide, which is generally incomplete and out of date. If you have questions about the source code, please join the *squid-dev* mailing list.

SOCKS

Squid doesn't support the SOCKS protocol at this time.

Recommended Reading

While reading this book, you may want to consult some of these other resources for more information (I'll refer to them throughout this book):

- *The Design and Implementation of the 4.4 BSD Operating System* by Marshall Kirk McKusick, Kieth Bostic, Michael J. Karels, and John S. Quarterman (Addison-Wesley Longman)
- *DNS and BIND* by Paul Albitz and Cricket Liu (O'Reilly & Associates)
- *HTTP: The Definitive Guide* by David Gourley and Brian Totty (O'Reilly)
- *Load Balancing Servers, Firewalls, and Caches* by Chandra Koopurapu (John Wiley & Sons)
- *Mastering Regular Expressions* by Jeffrey E. F. Friedl (O'Reilly)
- *Server Load Balancing* by Tony Bourke (O'Reilly)
- *Unix System Administration Handbook* and *Linux System Administration Handbook* by Evi Nemeth, Garth Snyder, Scott Seebass, and Trent R. Hein (Prentice Hall)
- My book, *Web Caching* (O'Reilly)
- RFC 1413: Identification Protocol
- RFC 1738: Uniform Resource Locators (URL)
- RFC 2186: Internet Cache Protocol (ICP), Version 2
- RFC 2187: Application of Internet Cache Protocol (ICP), Version 2
- RFC 2396: Uniform Resource Identifiers (URI): Generic Syntax
- RFC 2616: Hypertext Transfer Protocol—HTTP/1.1
- RFC 2617: HTTP Authentication: Basic and Digest Access Authentication
- RFC 2756: Hypertext Caching Protocol
- RFC 2817: Upgrading to TLS Within HTTP/1.1
- RFC 3040: Internet Web Replication and Caching Taxonomy

- RFC 3143: Known HTTP Proxy/Caching Problems
- Caching-related web sites, such as <http://www.caching.com/> and <http://www.web-cache.com/>

Conventions Used in This Book

I use the following typesetting conventions in this book:

Italic

Used for new terms where they are defined, buttons, pages, configuration file directives, filenames, modules, ACLs, directories, and URI/URLs

Constant width

Used for configuration file examples, program output, HTTP header names and directives, scripts, options, environment variables, functions, methods, rules, keywords, libraries, and command names

Constant width italic

Used for replaceable text within examples and code pieces

Constant width bold

Used to indicate commands to be typed verbatim

When displaying a Unix command, I'll include a shell prompt, like this:

```
% ls -l
```

If the command is specific to the Bourne shell (sh) or C shell (csh), the prompt will indicate which you should use:

```
sh$ ulimit -a
csh% limits
```

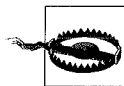
If the command requires super-user privileges, the shell prompt is a hash mark:

```
# make install
```

Occasionally, I provide configuration file examples with long lines. If the line is too wide to fit on the page, it's wrapped around and indented. Squid doesn't accept this sort of syntax, so you must make sure to place everything on one line.



This icon signifies a tip, suggestion, or general note.



This icon indicates a warning or caution.

Comments and Questions

Please address comments and questions concerning this book to the publisher:

O'Reilly & Associates, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
(800) 998-9938 (in the United States or Canada)
(707) 829-0515 (international or local)
(707) 829-0104 (fax)

There is a web page for this book, which lists errata, examples, and any additional information. You can access this page at:

<http://www.oreilly.com/catalog/squid>

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about books, conferences, Resource Centers, and the O'Reilly Network, check the O'Reilly web site at:

<http://www.oreilly.com>

You can contact the author at wessels@packet-pushers.com.

Acknowledgments

Looking back at the events and people that allowed me to write this book makes me feel extremely humble and grateful. I'm so happy to have been a part of the Harvest project with Mike Schwartz, Peter Danzig, and the others. That led directly to my work with kc claffy and Hans-Werner Braun at NLANR/UCSD. The Squid project would have never been at all without their support, and the grant from the National Science Foundation.

I'm also very thankful for all the hard work put in by the small crew of Squid developers: Henrik Nordström, Robert Collins, Adrian Chadd, and everyone else who has contributed time and code to the project. And I'm sorry that you ever had to read and/or fix any ugly code I wrote.

To all the reviewers who read the drafts—Joe Cooper, Scott Pepple, Robert Collins, and Adrian Chadd—thanks for finding my mistakes and suggesting ways to make the book better. I also owe so much to the people at O'Reilly for making the book possible, and for making it all come together. My editors Tatiana Diaz and Nat Torkington, the production editor Mary Anne Mayo, the graphic designer Melanie Wang, the illustrator, Rob Romano, the XML mungers Andrew Savikas and Joe Wizda, and the countless other folks working behind the scenes for me.

To my good friend, and business partner, Alex Rousskov: thanks for giving me the time and freedom to see this little project through. Finally, to the members of my new family, Annie and Blooey, thanks for putting up with the late nights. Can I make it up to you with extra back scratches?

Introduction

This long-overdue book is about Squid: a popular open source caching proxy for the Web. With Squid you can:

- Use less bandwidth on your Internet connection when surfing the Web
- Reduce the amount of time web pages take to load
- Protect the hosts on your internal network by proxying their web traffic
- Collect statistics about web traffic on your network
- Prevent users from visiting inappropriate web sites at work or school
- Ensure that only authorized users can surf the Internet
- Enhance your user's privacy by filtering sensitive information from web requests
- Reduce the load on your own web server(s)
- Convert encrypted (HTTPS) requests on one side, to unencrypted (HTTP) requests on the other

Squid's job is to be both a *proxy* and a *cache*. As a proxy, Squid is an intermediary in a web transaction. It accepts a request from a client, processes that request, and then forwards the request to the origin server. The request may be logged, rejected, and even modified before forwarding. As a cache, Squid stores recently retrieved web content for possible reuse later. Subsequent requests for the same content may be served from the cache, rather than contacting the origin server again. You can disable the caching part of Squid if you like, but the proxying part is essential.

As Figure 1-1 shows, Squid accepts HTTP (and HTTPS) requests from clients, and speaks a number of protocols to servers. In particular, Squid knows how to talk to HTTP, FTP, and Gopher servers.* Conceptually, Squid has two "sides." The *client-side* talks to web clients (e.g., browsers and user-agents); the *server-side* talks to

* Gopher servers are quite rare these days. Squid also knows about WAIS and whois, but these are even more obscure.

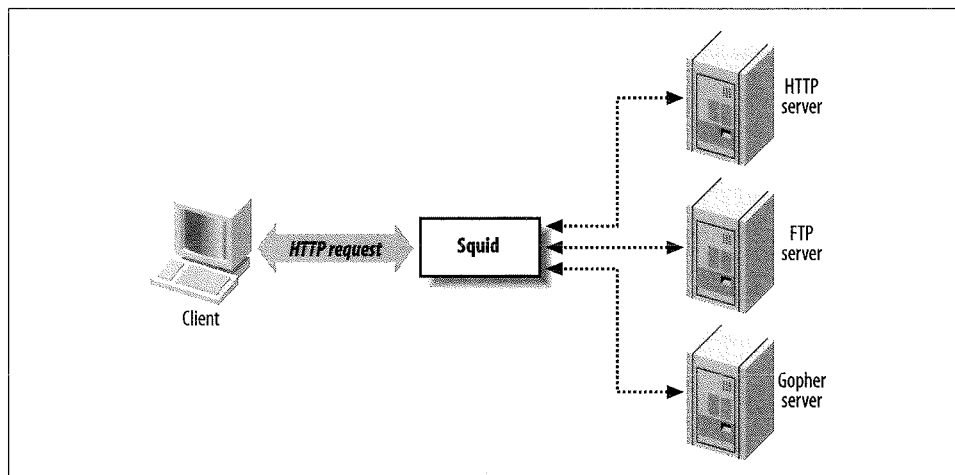


Figure 1-1. Squid sits between clients and servers

HTTP, FTP, and Gopher servers. These are called *origin servers*, because they are the origin location for the data they serve.

Note that Squid's client-side understands only HTTP (and HTTP encrypted with SSL/TLS). This means, for example, that you can't make an FTP client talk to Squid (unless the FTP client is also an HTTP client). Furthermore, Squid can't proxy protocols for email (SMTP), instant messaging, or Internet Relay Chat.

Web Caching

Web caching refers to the act of storing certain web resources (i.e., pages and other data files) for possible future reuse. For example, Matilda is the first person in the office each morning, and she likes to read the local newspaper online with her wake-up coffee. As she visits the various sections, the Squid cache on their office network stores the HTML pages and JPEG images. Harry comes in a short while later and also reads the newspaper online. For him, the site loads much faster because much of the content is served from Squid. Additionally, Harry's browsing doesn't waste the bandwidth of the company's DSL line by transferring the exact same data as when Matilda viewed the site.

A *cache hit* occurs each time Squid satisfies an HTTP request from its cache. The *cache hit ratio*, or *cache hit rate*, is the percentage of all requests satisfied as hits. Web caches typically achieve hit ratios between 30% and 60%. A similar metric, the *byte hit ratio*, represents the volume of data (i.e., number of bytes) served from the cache.

A *cache miss* occurs when Squid can't satisfy a request from the cache. A miss can happen for any number of reasons. Obviously, the first time Squid receives a request

for a particular resource, it is a cache miss. Similarly, Squid may have purged the cached copy to make room for new objects.

Another possibility is that the resource is uncachable. Origin servers can instruct caches on how to treat the response. For example, they can say that the data must never be cached, can be reused only within a certain amount of time, and so on. Squid also uses a few internal heuristics to determine what should, or should not, be saved for future use.

Cache validation is a process that ensures Squid doesn't serve stale data to the user. Before reusing a cached response, Squid often validates it with the origin server. If the server indicates that Squid's copy is still valid, the data is sent from Squid. Otherwise, Squid updates its cached copy as it relays the response to the client. Squid generally performs validation using timestamps. The origin server's response usually contains a *last-modified* timestamp. Squid sends the timestamp back to the origin server to find if the original resource has changed.

For a detailed treatment of web caching, have a look at my book *Web Caching*, also by O'Reilly.

A Brief History of Squid

In the beginning was the CERN HTTP server. In addition to functioning as an HTTP server, it was also the first caching proxy. The caching module was written by Ari Luotonen in 1994.

That same year, the Internet Research Task Force Group on Resource Discovery (IRTF-RD) started the Harvest project. It was "an integrated set of tools to gather, extract, organize, search, cache, and replicate" Internet information. I joined the Harvest project near the end of 1994. While most people used Harvest as a local (or distributed) search engine, the Object Cache component was quite popular as well. The Harvest cache boasted three major improvements over the CERN cache: faster use of the filesystem, a single process design, and caching hierarchies via the Internet Cache Protocol.

Towards the end of 1995, many Harvest team members made the move to the exciting world of Internet-based startup companies. The original authors of the Harvest cache code, Peter Danzig and Anawat Chankhunthod, turned it into a commercial product. Their company was later acquired by Network Appliance. In early 1996, I joined the National Laboratory for Applied Network Research (NLNR) to work on the Information Resource Caching (IRCache) project, funded by the National Science Foundation. Under this project, we took the Harvest cache code, renamed it Squid, and released it under the GNU General Public License.

Since that time Squid has grown in size and features. It now supports a number of cool things such as URL redirection, traffic shaping, sophisticated access controls,

numerous authentication modules, advanced disk storage options, HTTP interception, and surrogate mode (a.k.a. HTTP server acceleration).

Funding for the IRCache project ended in July 2000. Today, a number of volunteers continue to develop and support Squid. We occasionally receive financial or other types of support from companies that benefit from Squid.

Looking towards the future, we are rewriting Squid in C++ and, at the same time, fixing a number of design issues in the older code that are limiting to new features. We are adding support for protocols such as Edge Side Includes (ESI) and Internet Content Adaptation Protocol (ICAP). We also plan to make Squid support IPv6. A few developers are constantly making Squid run better on Microsoft Windows platforms. Finally, we will add more and more HTTP/1.1 features and work towards full compliance with the latest protocol specification.

Hardware and Operating System Requirements

Squid runs on all popular Unix systems, as well as Microsoft Windows. Although Squid's Windows support is improving all the time, you may have an easier time with Unix. If you have a favorite operating system, I'd suggest using that one. Otherwise, if you're looking for a recommendation, I really like FreeBSD.

Squid's hardware requirements are generally modest. Memory is often the most important resource. A memory shortage causes a drastic degradation in performance. Disk space is, naturally, another important factor. More disk space means more cached objects and higher hit ratios. Fast disks and interfaces are also beneficial. SCSI performs better than ATA, if you can justify the higher costs. While fast CPUs are nice, they aren't critical to good performance.

Because Squid uses a small amount of memory for every cached response, there is a relationship between disk space and memory requirements. As a rule of thumb, you need 32 MB of memory for each GB of disk space. Thus, a system with 512 MB of RAM can support a 16-GB disk cache. Your mileage may vary, of course. Memory requirements depend on factors such as the mean object size, CPU architecture (32- or 64-bit), the number of concurrent users, and particular features that you use.

People often ask such questions as, "I have a network with X users. What kind of hardware do I need for Squid?" These questions are difficult to answer for a number of reasons. In particular, it's hard to say how much traffic X users will generate. I usually find it easier to look at bandwidth usage, and go from there. I tell people to build a system with enough disk space to hold 3–7 days worth of web traffic. For example, if your users consume 1 Mbps (HTTP and FTP traffic only) for 8 hours per day, that's about 3.5 GB per day. So, I'd say you want between 10 and 25 GB of disk space for each Mbps of web traffic.

Squid Is Open Source

Squid is free software and a collaborative project. If you find Squid useful, please consider contributing back to the project in one or more of the following ways:

- Participate on the *squid-users* discussion list. Answer questions and help out new users.
- Try out new versions and report bugs or other problems.
- Contribute to the online documentation and Frequently Asked Questions (FAQ). If you notice an inconsistency, report it to the maintainers.
- Submit your local modifications back to the developers for inclusion into the code base.
- Provide financial support to one or more developers through small development contracts.
- Tell the developers about features you would like to have.
- Tell your friends and colleagues that Squid is cool.

Squid is released as free software under the GNU General Public License. This means, for example, that anyone who distributes Squid must make the source code available to you. See <http://www.gnu.org/licenses/gpl-faq.html> for more information about the GPL.

Squid's Home on the Web

The main source for up-to-date information about Squid is <http://www.squid-cache.org>. There you can:

- Download the source code.
- Read the FAQ and other documentation.
- Subscribe to the mailing list, or read the archives.
- Contact the developers.
- Find links to third-party applications.
- And more!

Getting Help

Given that Squid is free software, you may need to rely on the kindness of strangers for occasional assistance. The best place to do this is the *squid-users* mailing list. Before posting a message to the mailing list, however, you should check Squid's FAQ document to see if your question has already been asked and answered. If neither

resource provides the help you need, you can contact one of the many services offering professional support for Squid.

Frequently Asked Questions

Squid's FAQ document, located at <http://www.squid-cache.org/Doc/FAQ/FAQ.html>, is a good source of information for new users. The FAQ evolves over time, so it will contain entries written after this book. The FAQ also contains some historical information that may be irrelevant today.

Even so, the FAQ is one of the first places you should look for answers to your questions. This is especially true if you are a new user. While it is certainly less effort for you to simply write to the mailing list for help, veteran mailing list members grow tired of reading and answering the same questions. If your question is frequently asked, it may simply be ignored.

The FAQ is quite large. The HTML version exists as approximately 25 different chapters, each in a separate file. These can be difficult to search for keywords and awkward to print. You can also download PostScript, PDF, and text versions by following links at the top of the HTML version.

Mailing Lists

Squid has three mailing lists you might find useful. I explain how to become a subscriber below, but you may want to check Squid's mailing list page, <http://www.squid-cache.org/mailling-lists.html>, for possibly more up-to-date information.

squid-users

The *squid-users* mailing list is an excellent place to find answers for such questions as:

- How do I ... ?
- Is this a bug ... ?
- Does this feature/program work on my platform?
- What does this error message mean?

Note that you must subscribe before you can post a message. To subscribe to the *squid-users* list, send a message to squid-users-subscribe@squid-cache.org.

If you prefer, you can receive the digest version of the list. In this case, you'll receive multiple postings in a single email message. To sign up this way, send a message to squid-users-digest-subscribe@squid-cache.org.

Once you subscribe, you can post a message to the list by writing to squid-users@squid-cache.org. If you have a question, consider checking the FAQ and/or mailing list archives first. You can browse the list archive by visiting <http://www.squid-cache.org/mail-archive/squid-users/>. However, if you are looking for something

specific, you'll probably have more luck with the search interface at <http://www.squid-cache.org/search/>.

squid-announce

The moderated *squid-announce* list is used to announce new Squid versions and important security updates. The volume is quite low, usually less than one message per month. Write to squid-announce-subscribe@squid-cache.org if you'd like to subscribe.

squid-dev

The *squid-dev* list is a place where Squid hackers and developers can exchange ideas and information. Anyone can post a message to *squid-dev*, but subscriptions are moderated. If you'd like to join the discussion, please send a message about yourself and your interests in Squid. One of the list members should subscribe you within a few days.

The *squid-dev* messages are archived at <http://www.squid-cache.org/mail-archive/squid-dev/>, where anyone may browse them.

Professional Support

A number of companies now offer professional assistance for Squid. They may be able to help you get started with Squid for the first time, recommend a configuration for your network environment, and even fix some bugs.

Some of the consulting companies are associated with core Squid developers. By giving them your business, you ensure that fixes and features will be committed to future Squid software releases. If necessary, you can also arrange for development of private features.

Visit <http://www.squid-cache.org/Support/services.html> for the list of professional support services.

Getting Started with Squid

If you are new to Squid, the next few chapters will help you get started. First, I'll show you how to get the code, either the original source or precompiled binaries. In Chapter 3, I go through the steps necessary to compile and install Squid on your Unix system; this chapter is important because you'll probably need to tune your system before compiling the source code. Chapter 4 provides a very brief introduction to Squid's configuration file. Finally, Chapter 5 explains how to run Squid.

If you've already had a little experience installing and running Squid, you may want to skip ahead to Chapter 6.

Exercises

- Visit the Squid site and locate the *squid-users* mailing list archive. Browse the messages for the past few weeks.
- Search the Squid FAQ for information about file descriptors.
- Check one of the Squid mirror sites. Is it up to date with the primary site?

Getting Squid

Squid is normally distributed as source code. This means you'll probably need to compile it, as described in Chapter 3. The installation process should be relatively painless. The developers put a lot of effort into making sure Squid compiles easily on all the popular operating systems.

You can also find precompiled binaries for some operating systems. Linux users can get Squid in one of the various package formats (e.g., RPM, Debian, etc.). The FreeBSD, NetBSD, and OpenBSD projects offer Squid *ports*. The BSD ports aren't binary distributions but rather a small set of files that know how to download, compile, and install the Squid source. While these precompiled or preconfigured packages may be easier to install, I recommend that you download and compile the source yourself.

Anonymous CVS is a great way for developers and users to stay current with the official source tree. Instead of downloading entire new releases, you run a command to retrieve only the parts that have changed since your last update.

Versions and Releases

The Squid developers make periodic releases of the source code. Each release has a version number, such as 2.5.STABLE4. The third component starts either with STABLE or DEVEL (short for *development*).

As you can probably guess, the DEVEL releases tend to have newer, experimental features. They are also more likely to have bugs. Inexperienced users should not run DEVEL releases. If you choose to try a DEVEL release, and you encounter problems, please report them to the Squid maintainers.

After spending some time in the development state, the version number changes to STABLE. These releases are suitable for all users. Of course, even the stable releases may have some bugs. The higher-numbered stable versions (e.g., STABLE3,

STABLE4) are likely to have fewer bugs. If you are really concerned about stability, you may want to wait for one of these later releases.

Use the Source, Luke

So why can't you just copy a precompiled binary to your system and expect it to work perfectly? The primary reason is that the code needs to know about certain operating system parameters. In particular, the most important parameter is the maximum number of open file descriptors. Squid's `./configure` script (see Chapter 3) probes for these values before compiling. If you take a Squid binary built for one value and run it on a system with a different value, you may encounter problems.

Another reason is that many of Squid's features must be enabled at compile time. If you take a binary that somebody else compiled, and it doesn't include the code for the features that you want, you'll need to compile your own version anyway.

Finally, note that shared libraries sometimes make it difficult to share executable files between systems. Shared libraries are loaded at runtime. This is also known as *dynamic linking*. Squid's `./configure` script probes your system to find out certain things about your C library functions (if they are present, if they work, etc.). Although library functions don't usually change, it is possible that two different systems have slightly different shared C libraries. This may become a problem for Squid if the two systems are different enough.

Getting the Squid source code is really quite easy. To get it, visit the Squid home page, <http://www.squid-cache.org/>. The home page has links to the current stable and development releases. If you aren't located in the United States, you can select one of the many mirror sites. The mirror sites are usually named "wwwN.CC.squid-cache.org," where N is a number and CC is a two-letter country code. For example, www1.au.squid-cache.org is an Australian mirror site. The home page has links to the current mirror sites.

Each Squid release branch (e.g., Squid-2.5) has its own HTML page. This page has links to the source code releases and "diffs" between releases. If you are upgrading from one release to the next, you may want to download the diff file and apply the patch as described in Chapter 3. The release pages describe the new features and important changes in each version, and also have links to bugs that have been fixed.

When web access isn't an option, you can get the source release from the [ftp.squid-cache.org](ftp://ftp.squid-cache.org) FTP server or one of the FTP mirror sites. For the current versions, look in the `pub/squid-2/DEVEL` or `pub/squid-2/STABLE` directories. The Squid FTP site is mirrored at many locations as well. You can use the same country-code trick to guess some mirror sites, such as [ftp1.uk.squid-cache.org](ftp://ftp1.uk.squid-cache.org).

The current Squid release distributions are about 1 MB in size. After downloading the compressed tar file, you can proceed to Chapter 3.

Precompiled Binaries

Some Unix distributions include, or make available, precompiled Squid packages. For Linux, you can easily find Squid RPMs. Often the Squid RPM is included on Linux CD-ROMs you can buy. The FreeBSD/NetBSD/OpenBSD distributions also contain Squid in their ports and/or packages collections.

While RPMs and precompiled packages may initially save you some time, they also have some drawbacks. As I already mentioned, certain features must be enabled or disabled *before* you start compiling Squid. The precompiled package that you install may not have the particular feature you want. Furthermore, Squid's `./configure` script probes your operating system for certain parameters. These parameters may be configured differently on your machine on which Squid was compiled. Finally, if you want to apply a patch to Squid, you'll either have to wait for someone to build a new RPM/package or get the source and do it yourself.

I strongly encourage you to compile Squid from the source, but the decision is yours to make.

Anonymous CVS

The Concurrent Versioning System (CVS) is a nifty package that allows you to simultaneously edit and manage source code and other files. Almost every open source software project uses CVS.

You can anonymously access Squid's CVS files (read-only) to keep your source code up to date. The nice thing about CVS is that you can easily retrieve only the changes (diffs) of your current version. Thus, it is easy to see what has changed recently. Applying the changes to your current files efficiently synchronizes your source code with the official version.

CVS uses a tree-like indexing system. The trunk of the tree is called the *head branch*. For Squid's repository, this is where all new changes and features are placed. The head branch usually contains experimental and, possibly unstable, code. The stable code is typically found on other branches.

To effectively use Squid's anonymous CVS server, you first need to understand how different versions and branches are *tagged*. For example, the Version 2.5 branch is named `SQUID_2_5`. Particular releases, which represent a snapshot in time, have longer names, such as `SQUID_2_5_STABLE4`. To get exactly Squid Version 2.5. STABLE4, use the `SQUID_2_5_STABLE4` tag; to get the latest code on the 2.5 branch, use `SQUID_2_5`.

To use the Squid anonymous CVS server, you first need to set the CVSR00T environment variable:

```
csh% setenv CVSR00T :pserver:anoncvs@cvs.squid-cache.org:/squid
```

Or, for Bourne shell users:

```
sh$ CVSR00T=:pserver:anoncvs@cvs.squid-cache.org:/squid
sh$ export CVSR00T
```

You then log in to the server:

```
% cvs login
(logging in to anoncvs@cvs.squid-cache.org)
CVS password:
```

At the prompt, enter **anoncvs** for the password. Now you can check out the source tree with this command:

```
% cvs checkout -r SQUID_2_5 -d squid-2.5 squid
```

The `-r` option specifies the revision tag to retrieve. Omitting the `-r` option gets you the head branch. The `-d` option changes the top-level directory name in which files are placed. If you omit the `-d` option, the top-level directory is the same as the module name. The final command-line argument (`squid`) is the name of the module to check out.

Once you have the Squid source tree checked out, you can run the `cvs update` command to update your files and synchronize with the master repository. Additional interesting commands are `cvs diff`, `cvs log`, and `cvs annotate`.

To learn more about CVS, visit <http://www.cvshome.org/>.

devel.squid-cache.org

The Squid developers maintain a separate site, currently hosted at SourceForge, for experimental Squid features. Check it out at <http://devel.squid-cache.org/>. There you'll find a number of cutting-edge development projects that haven't yet been integrated into the official Squid code base. You can access these projects through SourceForge's anonymous CVS server or download diff files based on the standard releases.

Exercises

- Visit the Squid web site or FTP server and look at the recent stable and development releases. How often are new releases made?
- Download the most recent stable code.
- Use Squid's anonymous CVS server to check out the recent stable branch. Change one of the source files by inserting a blank line, then run **cvs diff**.

Compiling and Installing

Squid is designed to be portable and should compile on all major Unix systems, including Linux, BSD/OS, FreeBSD, NetBSD, OpenBSD, Solaris, HP-UX, OSF/DUNIX/TRU-64, Mac OS/X, IRIX, and AIX. Squid also runs on Microsoft Windows. Please see Appendix E for instructions on compiling and running Squid on Windows.

Compiling Squid is relatively straightforward. If you've installed more than a few open source packages, you're probably already familiar with the procedure. You first use a program called `./configure` to probe your system and then a program called `make` to do the actual compiling.

Before getting to that step, however, let's talk about tuning your system in preparation for Squid. Your operating system may have default resource limits that are too low for Squid to run correctly. Most importantly, you need to worry about the number of available file descriptors.

Before You Start

If you've been using Unix for a while, chances are that you've already compiled a number of other software packages. If so, you can probably quickly scan this chapter. The procedure for compiling and installing Squid is similar to many other software distributions.

To compile Squid, you need an ANSI C compiler. Don't be too alarmed by the "ANSI" part. Chances are that if you already have a C compiler, it is compliant with the ANSI specification. The GNU C compiler (`gcc`) is an excellent choice and widely available. Most operating systems come with a C compiler as a part of the standard installation. The common exceptions are Solaris and HP-UX. If you're using one of those operating systems, you might not have a compiler installed.

Ideally you should compile Squid on the same system on which it will run. Part of the installation process probes your system for certain parameters, such as the

number of available file descriptors. However, if your system doesn't have a C compiler, you may be able to compile Squid elsewhere and then copy the binaries back. If the operating systems are different, Squid may encounter some problems. Also, Squid may become confused if the two systems have different kernel configurations.

In addition to a C compiler, you'll also need Perl and awk. awk is a standard program on all Unix systems, so you shouldn't need to worry about it. Perl is quite common, but it may not be installed on your system by default. You may need the gzip program to uncompress the source distribution file.

Solaris users, make sure that `/usr/ccs/bin` is in your PATH, even if you're using gcc. To compile Squid, you may need the make and ar programs found in that directory.

Unpacking the Source

After downloading the source distribution, you need to unpack it somewhere. The particular location doesn't really matter. You can unpack Squid in your home directory or anywhere; you'll need about 20 MB of free disk space. Personally, I like to use `/tmp`. Use the tar command to extract the source directory:

```
% cd /tmp
% tar xzvf /some/where/squid-2.5.STABLE4-src.tar.gz
squid-2.5.STABLE4/
squid-2.5.STABLE4/CONTRIBUTORS
squid-2.5.STABLE4/COPYING
squid-2.5.STABLE4/COPYRIGHT
squid-2.5.STABLE4/CREDITS
squid-2.5.STABLE4/ChangeLog
squid-2.5.STABLE4/INSTALL
squid-2.5.STABLE4/QUICKSTART
squid-2.5.STABLE4/README
...
```

Some tar programs don't have the z option, which automatically uncompresses gzip files. In that case, you'll need to use this command:

```
% gzip -dc /some/where/squid-2.5.STABLE4-src.tar.gz | tar xvf -
```

Once the source code has been unpacked, the next step is usually to configure the source tree. However, if this is the first time you're compiling Squid, you should make sure certain kernel resource limits are high enough; to find out how, read on.

Pretuning Your Kernel

Squid requires a fair amount of kernel resources under moderate and high loads. In particular, you may need to configure your system with a higher-than-normal number of file descriptors and mbuf clusters. The file-descriptor limit can be especially annoying. You'd be better off to increase the limit before compiling Squid.

At this point, you might be tempted to get the precompiled binaries to avoid the hassle of building a new kernel.* Unfortunately, you need to make a new kernel, regardless. Squid and the kernel exchange information through data structures that must not exceed the set file-descriptor limits. Squid checks these limits at runtime and uses the safest (smallest) value. Thus, even if a precompiled binary has higher file descriptors than the kernel, the kernel value takes precedence.

To change some settings, you must build and install a new kernel. This procedure varies among different operating systems. Consult *Unix System Administration Handbook* (Prentice Hall) or your operating-system documentation if necessary. If you're using Linux, you probably don't need to recompile your kernel.

File Descriptors

File descriptors are simply integers that identify each file and socket that a process has opened. The first opened file is 0, the second is 1, and so on. Unix operating systems usually impose a limit on the number of file descriptors that each process can open. Furthermore, Unix also normally has a systemwide limit.

Because of the way Squid works, the file-descriptor limits may adversely affect performance. When Squid uses up all the available file descriptors, it is unable to accept new connections from users. In other words, running out of file descriptors causes denial of service. Squid can't accept new requests until some of the current requests complete, and the corresponding files and sockets are closed. Squid issues a warning when it detects a file-descriptor shortage.

You can save yourself some trouble by making sure the file descriptor limits are appropriate before running `./configure`. In most cases, 1024 file descriptors will be sufficient. Very busy caches may require 4096 or more. When configuring file descriptor limits, I recommend setting the systemwide limit to twice the per-process limit.

You can usually discover your system's file-descriptor limit from your Unix shell. All C shells and similar have the built-in `limit` command. Newer Bourne shells and similar have a command called `ulimit`. To find your file-descriptor limits, try running these commands:

```
csh% limit descriptors unlimited
csh% limit descriptors
descriptors      4096
```

or:

```
sh$ ulimit -n unlimited
sh$ ulimit -n
4096
```

* Not all operating systems require building a new kernel. Some may be tunable at runtime.

On FreeBSD, you can also use the `sysctl` command:

```
% sysctl -a | grep maxfiles
kern.maxfiles: 8192
kern.maxfilesperproc: 4096
```

If you can't figure out the file-descriptor limit, Squid's `./configure` script can do it for you. When you run `./configure`, as described in the section “The configure Script,” watch for output like this near the end:

```
checking Maximum number of file descriptors we can open... 4096
```

If either limit, `ulimit`, or `./configure` report a value less than 1024, you should invest the time to increase the limit before compiling Squid. Otherwise, Squid's performance will be poor under a moderate load.

Increasing the file descriptor limit varies from system to system. The following sections offer some tips to help get you started.

FreeBSD, NetBSD, OpenBSD

Edit your kernel configuration file, and add a line like this:

```
options      MAXFILES=8192
```

On OpenBSD, use `option` instead of `options`. Then, configure, compile, and install the new kernel. Reboot your system so the change takes effect.

Linux

Configuring file descriptors on Linux is a little complicated. You must edit one of the system include files, and execute some shell commands before compiling and running Squid. Start off by editing the file `/usr/include/bits/types.h`. Change the value for `__FD_SETSIZE` as follows:

```
#define __FD_SETSIZE 8192
```

Next, increase the kernel file descriptor limit with this command:

```
# echo 8192 > /proc/sys/fs/file-max
```

Finally, increase the process file-descriptor limit in the same shell in which you will configure and compile Squid:

```
sh# ulimit -Hn 8192
```

This command must be executed as *root* and only works from the bash shell. There is no need to reboot on Linux.



With this technique, you must execute the `echo` and `ulimit` commands each time your system boots, or at least before starting Squid. If you use an *rc.d* script to start Squid (see Chapter 5), that is a good place to stick these commands.

Solaris

Add this line to your `/etc/system` file:

```
set rlim_fd_max = 4096
```

Then, reboot the system for the change to take effect.

Mbuf Clusters

The BSD-based networking code uses a data structure known as an *mbuf* (see W.R. Stevens' book, *TCP/IP Illustrated, Vol 2*). Mbufs are typically small (e.g., 128 octets) chunks of memory. The data for larger network packets are stored in mbuf *clusters*. The kernel may enforce an upper limit on the total number of mbuf clusters available in the system. You can find this limit with the `netstat` command:

```
% netstat -m
196/6368/32768 mbufs in use (current/peak/max):
    146 mbufs allocated to data
    50 mbufs allocated to packet headers
103/6182/8192 mbuf clusters in use (current/peak/max)
13956 Kbytes allocated to network (56% of mb_map in use)
0 requests for memory denied
0 requests for memory delayed
0 calls to protocol drain routines
```

In this example, there are 8,192 mbuf clusters available, but there are never more than 6,182 used at once. When the system runs out of mbuf clusters, I/O routines such as `read()` and `write()` return the “No buffer space available” error message.

NetBSD and OpenBSD don't display mbuf usage in `netstat -m` output. Instead, they report “WARNING: mclpool limit reached” via `syslog`.

To increase the number of mbuf clusters, you need to add an option to your kernel configuration file:

```
options          NMBCLUSTERS=16384
```

Ephemeral Port Range

Ephemeral ports are the local port numbers the TCP/IP stack assigns to outgoing connections. In other words, when Squid makes a connection to an origin server, the kernel assigns a port number to the local socket. These local port numbers fall within a certain range. On FreeBSD, for example, the default ephemeral port range is 1024–5000.

A shortage of ephemeral ports may adversely affect performance for very busy proxies (i.e., hundreds of requests per second). This is because some TCP connections enter a `TIME_WAIT` state when they are closed. An ephemeral port number can't be reused while the connection is in the `TIME_WAIT` state.

You can see how many connections are in this state with the `netstat` command:

```
% netstat -n | grep TIME_WAIT
Proto Recv-Q Send-Q Local Address      Foreign Address    (state)
tcp4      0      0 192.43.244.42.19583 212.67.202.80.80   TIME_WAIT
tcp4      0      0 192.43.244.42.19597 202.158.66.190.80   TIME_WAIT
tcp4      0      0 192.43.244.42.19600 207.99.19.230.80    TIME_WAIT
tcp4      0      0 192.43.244.42.19601 216.131.72.121.80    TIME_WAIT
tcp4      0      0 192.43.244.42.19602 209.61.183.115.80    TIME_WAIT
tcp4      0      0 192.43.244.42.3128  128.109.131.47.25666 TIME_WAIT
tcp4      0      0 192.43.244.42.3128  128.109.131.47.25795 TIME_WAIT
tcp4      0      0 192.43.244.42.3128  128.182.72.190.1488  TIME_WAIT
tcp4      0      0 192.43.244.42.3128  128.182.72.190.2194  TIME_WAIT
```

Note that this example has both client- and server-side connections. Client-side connections have 3128 as the local port number; server-side connections have 80 as the remote (foreign) port number. The ephemeral port numbers appear under the Local Address heading. In this example, they are in the 19,000s.

Unless you see thousands of ephemeral ports in the `TIME_WAIT` state, you probably don't need to increase the range. On FreeBSD, you can increase the range with this command:

```
# sysctl -w net.inet.ip.portrange.last=30000
```

On OpenBSD, the command is almost the same, but the `sysctl` variable has a different name:

```
# sysctl -w net.inet.ip.portlast=49151
```

On NetBSD, things work a little differently. The default range is 49,152–65,535. To increase the range, change the lower limit:

```
# sysctl -w net.inet.ip.anonportmin=10000
```

On Linux, simply write a pair of numbers to the following special file:

```
# echo "1024 40000" > /proc/sys/net/ipv4/ip_local_port_range
```

Don't forget to add these commands to your system startup scripts so that they take effect each time your machine reboots.

The configure Script

Like many other Unix software packages, Squid uses a `./configure` script to learn about an operating system before compiling. The `./configure` script is generated by the popular GNU `autoconf` program. When the script runs, it probes the system in various ways to find out about libraries, functions, types, parameters, and features that may or may not be present. One of the first things that `./configure` does is look for a working C compiler. If the compiler can't be found or fails to compile a simple test program, the `./configure` script can't proceed.

The `./configure` script has a number of different options. The most important is the installation prefix. Before running `./configure`, you need to decide where Squid should live. The installation prefix determines the default locations for the Squid logs, binaries, and configuration files. You can change the location for those files after installing, but it's easier if you decide now.

The default installation prefix is `/usr/local/squid`. Squid puts files in seven different subdirectories under the prefix:

```
% ls -l /usr/local/squid
total 5
drwxr-x--- 2 wessels wheel 512 Apr 28 20:42 bin
drwxr-x--- 2 wessels wheel 512 Apr 28 20:42 etc
drwxr-x--- 2 wessels wheel 512 Apr 28 20:42 libexec
drwxr-x--- 3 wessels wheel 512 Apr 28 20:43 man
drwxr-x--- 2 wessels wheel 512 Apr 28 20:42 sbin
drwxr-x--- 4 wessels wheel 512 Apr 28 20:42 share
drwxr-x--- 4 wessels wheel 512 Apr 28 20:43 var
```

Squid uses the `bin`, `etc`, `libexec`, `man`, `sbin`, and `share` directories for a few, relatively small files (or other directories) that don't change very often. The files under the `var` directory, however, are a different story. This is where you'll find Squid's log files, which may grow quite large (tens or hundreds of megabytes). `var` is also the default location for the actual disk cache. You may want to put `var` on a different partition with plenty of space. One easy way to do this is with the `--localstatedir` option:

```
% ./configure --localstatedir=/bigdisk/var
```

You don't need to worry too much about pathnames when configuring Squid. You can always change the pathnames later, in the `squid.conf` file.

configure Options

The `./configure` script has a number of different options that all start with `--`. You can see the full list of options by typing `./configure --help`. Some of these options are common to all configure scripts, and some are unique to Squid. Here are the standard options that you might find useful:

`--prefix=PREFIX`

This sets the installation prefix directory, as described earlier. The installation prefix is the default directory for all executables, logs, and configuration files. Throughout this book, `$prefix` refers to your choice for the installation prefix.

`--localstatedir=DIR`

This option allows you to change the location for the `var` directory. The default is `$prefix/var`, but you might want to change it so that Squid's disk cache and log files are stored elsewhere.

`--sysconfdir=DIR`

This option allows you to change the location for the *etc* directory. The default is `$prefix/etc`. If you like to use `/usr` as the installation prefix, you might want to set `--sysconfdir` to `/etc`.

Here are the Squid-specific `./configure` options:

`--enable-dlmalloc[=LIB]`

On some systems, the built-in memory allocation (`malloc`) functions have poor performance characteristics when used with Squid. Using the `--enable-dlmalloc` option builds and links with the `dlmalloc` package included in the Squid source code. If you already have `dlmalloc` built on your system, you can specify the library's pathname as the `=LIB` argument. See <http://g.oswego.edu/dl/html/malloc.html> for more information on `dlmalloc`.

`--enable-gnuregex`

Squid uses *regular expressions* for pattern matching in access control lists and other configuration directives. The GNU regular expression library comes with the Squid source code; it can be used on operating systems that don't have built-in regular expression functions. The `./configure` script probes your system for a regular expression library and enables the use of GNU `regex` if necessary. If, for some reason, you want to force the usage of GNU `regex`, you can add this option to the `./configure` command.

`--enable-carp`

The Cache Array Routing Protocol (CARP) is useful for forwarding cache misses to an array, or cluster, of parent caches. There's more about CARP in Chapter 10.

`--enable-async-io[=N_THREADS]`

Async I/O refers to one of Squid's techniques for improved storage performance. The *aufs* storage module uses a number of thread processes to perform disk I/O operations. This code works only on Linux and Solaris systems. The `=N_THREADS` argument changes the number of thread processes Squid uses. *aufs* and Async I/O are discussed in the section called Chapter 8.

Note that the `--enable-async-io` option is a shortcut that turns on three other `./configure` options. It is equivalent to specifying:

```
--with-aufs-threads=N_THREADS
--with-pthreads
--enable-storeio=ufs,aufs
```

`--with-pthreads`

The `--with-pthreads` option causes the compilation procedure to link with your system's Pthreads library. The *aufs* storage module is the only part of Squid that uses threads. Normally, you don't specify this option on the `./configure` command line because it's enabled automatically when you use `--enable-async-io`.

`--enable-storeio=LIST`

Squid supports a number of different storage modules. With this option, you tell `./configure` which modules to compile. The *ufs*, *aufs*, *diskd*, *coss*, and *null* modules are supported in Squid-2.5. You can also get a list by looking at the directories under *src/fs*.

LIST is a comma-separated list of module names. For example:

```
% ./configure --enable-storeio=afus,diskd,ufs
```

The *ufs* module is the default and least likely to cause problems. Unfortunately, it also has limited performance characteristics. The other modules may not necessarily compile on your particular operating system. For a complete description of Squid's storage modules, see Chapter 8.

`--with-aufs-threads=N_THREADS`

Specifies the number of threads to use for the *aufs* storage scheme (see Chapter 8). By default, Squid automatically calculates how many threads to use, based on the number of cache directories.

`--enable-heap-replacement`

This option has been deprecated but remains for backward compatibility. You should always use the `--enable-removal-policies` option instead.

`--enable-removal-policies=LIST`

Removal policies are the algorithms Squid uses to eject cached objects when making room for new ones. Squid-2.5 supports three removal policies: least recently used (LRU), greed dual size (GDS), and least frequently used (LFU).

However, for some reason, the `./configure` options blur the distinction between a particular replacement policy and the underlying data structures required to implement them. LRU, which is the default, is implemented with a doubly linked list. The GDS and LFU implementations use a data structure known as a *heap*.

To use the GDS or LFU policies, you specify:

```
% ./configure --enable-removal-policies=heap
```

You then select between GDS and LFU in the Squid configuration file. If you want to retain the option of using LRU, specify:

```
% ./configure --enable-removal-policies=heap,lru
```

There's more about replacement policies in Chapter 7.

`--enable-icmp`

As you'll see in Chapter 10, Squid can make round-trip time measurements with ICMP messages, much like the *ping* program. You can use this option to enable these features.

`--enable-delay-pools`

Delay pools are Squid's technique for traffic shaping or bandwidth limiting. The pools consist of groups of client IP addresses. When requests from these clients

are cache misses, their responses may be artificially delayed. See more about delay pools in Appendix C.

--enable-useragent-log

This option enables logging of the HTTP User-Agent header from client requests. See more about this in Chapter 13.

--enable-referer-log

This option enables logging of the HTTP referer header from client requests. See more about this in Chapter 13.

--disable-wccp

The Web Cache Coordination Protocol (WCCP) is Cisco's once-proprietary protocol for intercepting and distributing HTTP requests to one or more caches. WCCP is enabled by default, but you can use this option to prevent compilation of the WCCP code if you like.

--enable-snmp

The Simple Network Management Protocol (SNMP) is a popular way to monitor network devices and servers. This option causes the build procedure to compile all of the SNMP-related code, including a cut-down version of the CMU SNMP library.

--enable-cachemgr-hostname[=*hostname*]

cachemgr is a CGI program you can use to administratively query Squid. By default, *cachemgr*'s hostname field is blank, but you can create a default value with this option. For example:

```
% ./configure --enable-cachemgr-hostname=mycache.myorg.net
```

--enable-arp-acl

Squid supports ARP, or Ethernet address, access control lists on some operating systems. The code to implement ARP ACLs uses nonstandard function interfaces, so it is disabled by default. If you run Squid on Linux or Solaris, you may be able to use this feature.

--enable-htcp

HTCP is the Hypertext Caching Protocol—an intercache protocol similar to ICP. See Chapter 10 for more information.

--enable-ssl

Use this option to give Squid the ability to terminate SSL/TLS connections. Note this only works for accelerated requests in surrogate mode. See Chapter 15 for more information.

--with-openssl[=*DIR*]

This option exists so that you can tell the compiler where to find the OpenSSL libraries and header files, if necessary. If they aren't in the default location, enter the parent directory after this option. For example:

```
% ./configure --enable-ssl --with-ssl=/opt/foo/openssl
```

Given this example, your compiler looks for the OpenSSL header files in */opt/foopenssl/include*, and for libraries in */opt/foopenssl/lib*.

--enable-cache-digests

Cache Digests are another alternative to ICP, but with significantly different characteristics. See Chapter 10.

--enable-err-languages="lang1 lang2 ..."

Squid supports customizable error messages and comes with error messages in many different languages. This option determines the languages that are copied to the installation directory (*\$prefix/share/errors*). If you don't use this option, all available languages are installed. To see which languages are available, look at a directory listing of the *errors* directory in the source distribution. Here's how to enable more than one language:

```
% ./configure --enable-err-languages="Dutch German French" ...
```

--enable-default-err-language=lang

This option sets the default value for the *error_directory* directive. For example, if you want to use Dutch error messages, you can use this *./configure* option:

```
% ./configure --enable-default-err-language=Dutch
```

You can also set the *error_directory* directive in *squid.conf*, as described in Appendix A. English is the default error language if you omit this option.

--with-coss-membuf-size=N

The Cyclic Object Storage System (*coss*) is an experimental storage scheme for Squid. This option sets the memory buffer size for *coss* cache directories. Note that in order to use *coss*, you must specify it as a storage type in the *--enable-storeio* option.

The argument is given in bytes. The default is 1,048,576 bytes or 1 MB. You can specify a 2-MB buffer like this:

```
% ./configure --with-coss-membuf-size=2097152
```

--enable-poll

Unix provides two similar functions that scan open file descriptors for I/O events: *select()* and *poll()*. The *./configure* script usually does a very good job of figuring out when to use *poll()* over *select()*. Use this option if you want to override the *./configure* script and force it to use *poll()*.

--disable-poll

Similarly, Unix gurus may want to force *./configure* to not use *poll()*.

--disable-http-violations

By default, Squid can be configured to violate the HTTP protocol specifications. You can use this option to remove the code completely that would violate HTTP.

`--enable-ipf-transparent`

In Chapter 9, I'll describe how to configure Squid for interception caching. Some operating systems use the IP Filter package to assist with the interception. In these cases you should use this `./configure` option. If you enable this option and get compiler errors on the `src/client_side.c` file, chances are that the IP Filter package isn't actually (or correctly) installed on your system.

`--enable-pf-transparent`

You may need this option to use HTTP interception on systems that use the PF packet filter. PF is the standard packet filter for OpenBSD and may have been ported to other systems as well. If you enable this option and get compiler errors on the `src/client_side.c` file, chances are that PF isn't actually installed on your system.

`--enable-linux-netfilter`

Netfilter is the name of the Linux packet filter for the 2.4 kernel series. Enable this option if you want to use HTTP interception with Linux 2.4 or later.

`--disable-ident-lookups`

ident is a simple protocol that allows a server to find the username associated with a client's particular TCP connection. If you use this option, the compiler excludes completely the code that performs such lookups. Even if you leave the code enabled at compile time, Squid doesn't make *ident* lookups unless you configure them in *squid.conf*.

`--disable-internal-dns`

The Squid source code includes two different DNS resolution implementations, called *internal* and *external*. Internal lookups are the default, but some people prefer the external technique. This option disables the internal functionality and reverts to the older method.

Internal lookups use Squid's own implementation of the DNS protocol. That is, Squid generates raw DNS queries and sends them to a resolver. It retransmits queries that time out, and you can specify any number of resolvers. One of the benefits to this implementation is that Squid gets accurate TTLs for DNS replies.

External lookups use the C library's `gethostbyname()` and `gethostbyaddr()` functions. Since these routines block the process until the answer comes back, they must be called from external, helper processes. Squid uses a pool of external processes to make queries in parallel. The primary drawback to external DNS resolution is that you need more helper processes as Squid's load increases. Another annoyance is that the C library functions don't convey TTLs with the answers, in which case Squid uses a constant value supplied by the *positive_dns_ttl* directive.

`--enable-truncate`

The `truncate()` system call is an alternative to using `unlink()`. While `unlink()` removes a cache file altogether, `truncate()` sets the file size to zero. This frees the disk space associated with the file but leaves the directory entry in place.

This option exists because some people believed (or hoped) that `truncate()` would produce better performance than `unlink()`. However, benchmarks have shown little or no real difference.

`--disable-hostname-checks`

By default, Squid requires that URL hostnames conform to the somewhat archaic specifications in RFC 1034:

The labels must follow the rules for ARPANET host names. They must start with a letter, end with a letter or digit, and have as interior characters only letters, digits, and hyphen.

Here, “letter” means the ASCII characters A through Z. Since internationalized domain names are becoming increasingly popular, you may want to use this option to remove the restriction.

`--enable-underscores`

This option controls Squid’s behavior regarding underscore characters in hostnames. General consensus is that hostnames must not include underscore characters, although some people disagree. Squid, by default, generates an error message for requests that have an underscore in a URL hostname. You can use this option to make Squid treat them as valid. However, your DNS resolver may also enforce the no-underscore requirement and fail to resolve such hostnames.

`--enable-auth[=LIST]`

This option controls which HTTP authentication schemes to support in the Squid binary. You can select any combination of the following schemes: *basic*, *digest*, and *ntlm*. If you omit the option, Squid supports only basic authentication. If you give the `--enable-auth` option without any arguments, the build process adds support for all schemes. Otherwise, you can give a comma-separated list of schemes to support:

```
% ./configure --enable-auth=digest,ntlm
```

I talk more about authentication in Chapters 6 and 12.

`--enable-auth-helpers=LIST`

This old option is now deprecated, but still remains for backward compatibility. You should use `--enable-basic-auth-helpers=LIST` instead.

`--enable-basic-auth-helpers=LIST`

With this option, you can build one or more of the HTTP Basic authentication helper programs found in *helpers/basic_auth*. See Chapter 12 for their names and descriptions.

`--enable-ntlm-auth-helpers=LIST`

With this option, you can build one or more of the HTTP NTLM authentication helper programs found in *helpers/ntlm_auth*. See Chapter 12 for their names and descriptions.

- `--enable-ntlm-fail-open`
When you enable this option, Squid's NTLM authentication module defaults to allow access in the event of an error or problem.
- `--enable-digest-auth-modules=LIST`
With this option, you can build one or more of the HTTP Digest authentication helper programs found in *helpers/digest_auth*. See Chapter 12 for their names and descriptions.
- `--enable-external-acl-helpers=LIST`
With this option, you can build one or more of the external ACL helper programs that I discuss in Chapter 12. For example:

```
% ./configure --enable-external-acl-helpers=ip_user,ldap_group
```
- `--disable-unlinkd`
Unlinkd is another one of Squid's external helper processes. Its sole job is to execute the `unlink()` (or `truncate()`) system call on cache files. Squid realizes a significant performance gain by implementing file deletion in an external process. Use this option to disable the external unlink daemon feature.
- `--enable-stacktrace`
Some operating systems support automatic generation of stack trace data in the event of a program crash. When you enable this feature and Squid crashes, the stack trace information is written to the *cache.log* file. This information is often helpful to developers in tracking down programming bugs.
- `--enable-x-accelerator-vary`
This advanced feature may be used when Squid is configured as a surrogate. It instructs Squid to look for X-Accelerator-Vary headers in responses from back-end origin servers. See Chapter 15.

Running configure

Now we're ready to run the `./configure` script. Go to the top-level source directory and type `./configure`, followed by any of the options mentioned previously. For example:

```
% cd squid-2.5.STABLE4
% ./configure --enable-icmp --enable-htcp
```

`./configure`'s job is to probe your operating system and find out which things are available, and which are not. One of the first things it does is make sure your C compiler is working. If `./configure` detects a problem with your C compiler, the script exits with this error message:

```
configure: error: installation or configuration problem: C compiler
cannot create executables.
```

Most likely, you'll never see that message. If you do, it means either your system doesn't have a C compiler at all or that the compiler isn't installed correctly. Look at

the *config.log* file for hints as to the exact problem. If your system has more than one C compiler, you can tell `./configure` which to use by setting the `CC` environment variable before running `./configure`:

```
% setenv CC /usr/local/bin/gcc
% ./configure ...
```

After `./configure` checks out the compiler, it looks for a long list of header files, libraries, and functions. Normally you won't have to worry about this part. In some cases, `./configure` pauses to get your attention about something that may be a problem (such as not enough file descriptors). It may also stop if you specify incompatible or unreasonable command-line options. If something does go wrong, check the *config.log* output. `./configure`'s final task is to create *Makefiles* and other files based on the things it learned about your system. At this point, you're ready to begin compiling.

make

Once `./configure` has done its job, you can simply type `make` to begin compiling the source code:

```
% make
```

Normally, this part goes smoothly. You'll see a lot of lines that look like this:

```
source='cbdata.c' object='cbdata.o' libtool=no depfile='.deps/cbdata.Po'
tmpdepfile='.deps/cbdata.TPo' depmode=gcc /bin/sh ../cfgaux/depcomp gcc -DHAVE_
CONFIG_H -DDEFAULT_CONFIG_FILE=\"/usr/local/squid/etc/squid.conf\" -I. -I. -I../
include -I. -I. -I../include -I../include -g -O2 -Wall -c 'test -f cbdata.c ||
echo './'cbdata.c
source='client_db.c' object='client_db.o' libtool=no depfile='.deps/client_db.Po'
tmpdepfile='.deps/client_db.TPo' depmode=gcc /bin/sh ../cfgaux/depcomp gcc -DHAVE_
CONFIG_H -DDEFAULT_CONFIG_FILE=\"/usr/local/squid/etc/squid.conf\" -I. -I. -I../
include -I. -I. -I../include -I../include -g -O2 -Wall -c 'test -f client_db.c ||
echo './'client_db.c
source='client_side.c' object='client_side.o' libtool=no depfile='.deps/client_side.
Po' tmpdepfile='.deps/client_side.TPo' depmode=gcc /bin/sh ../cfgaux/depcomp gcc -
DHAVE_CONFIG_H -DDEFAULT_CONFIG_FILE=\"/usr/local/squid/etc/squid.conf\" -I. -I. -I..
/include -I. -I. -I../include -I../include -g -O2 -Wall -c 'test -f client_side.c
|| echo './'client_side.c
source='comm.c' object='comm.o' libtool=no depfile='.deps/comm.Po' tmpdepfile='.
deps/comm.TPo' depmode=gcc /bin/sh ../cfgaux/depcomp gcc -DHAVE_CONFIG_H -DDEFAULT_
CONFIG_FILE=\"/usr/local/squid/etc/squid.conf\" -I. -I. -I../include -I. -I. -I../
include -I../include -g -O2 -Wall -c 'test -f comm.c || echo './'comm.c
```

You may see some compiler warnings. In most cases, it is safe to ignore these. If you see a lot of them or something that looks really serious, report it to the developers as described in Chapter 16.

* The `make` output used to be much prettier, but such is the price we pay for advanced compiling tools such as `automake`.

If the compilation gets all the way to the end without any errors, you can move to the next section, which describes how to install the programs you just built.

To verify that compilation was successful, you can run `make` again. You should see this output:*

```
% make
Making all in lib...
Making all in scripts...
Making all in src...
Making all in fs...
Making all in repl...
'squid' is up to date.
'client' is up to date.
'unlinkd' is up to date.
'cachemgr.cgi' is up to date.
Making all in icons...
Making all in errors...
Making all in auth_modules...
```

The compilation step may fail for a number of reasons, including:

Source code bugs

Usually the Squid source code is thoroughly debugged. However, you may encounter some bugs or problems that prevent Squid from compiling. You're more likely to find these sorts of bugs in the newer development versions. Report these to the developers.

Compiler installation problems

An improperly installed C compiler probably won't be able to compile Squid or any other moderately sized software package. Usually, compilers come pre-installed with the operating system, so you don't have to worry about that. However, if you attempt to upgrade your compiler after installing the operating system, you might make a mistake. Never copy a compiler installation from one machine to another, unless you are absolutely sure about what you are doing. I feel it is always better to install the compiler on each machine separately.

Always make sure that your compiler's header files are synchronized with the library files. The header files normally reside in `/usr/include`, while libraries are found in `/usr/lib`. Linux's popular RPM system makes it possible to upgrade one, but not the other. If the libraries are based on different header files, Squid may not compile.

If you want to upgrade the compiler on one of the open-source BSD variants, be sure to run `make world` from the `/usr/src` directory, rather than from the `/usr/src/lib` or `/usr/src/include` directories.

* If `make` recompiles the source every time you run it, and there are no errors, your system clock may be set wrong.

Here are some common compilation problems and error messages:

Solaris: make[1]: *** [libmiscutil.a] Error 255

This means that `./configure` didn't find the `ar` program. Make sure `/usr/ccs/bin` is listed in your `PATH` environment variable. If you don't have the Sun compiler installed, you'll need the GNU binutils (<http://www.gnu.org/directory/binutils.html>).

Linux: storage size of 'rl' isn't known

This happens when the header and library files don't match, as described earlier. Be sure to upgrade both packages at the same time.

Digital Unix: Don't know how to make EXTRA_libmiscutil_a_SOURCES. Stop.

Digital Unix's make program isn't tolerant of the *Makefile* produced by the automake package. For example, *lib/Makefile.in* contains these lines:

```
noinst_LIBRARIES = \  
  @LIBDLMALLOC@ \  
  libmiscutil.a \  
  libntlmauth.a \  
  @LIBREGEX@
```

After substitution, when *lib/Makefile* is created, it looks like this:

```
noinst_LIBRARIES = \  
  \  
  libmiscutil.a \  
  libntlmauth.a \  
  <TAB>
```

As shown above, the last line contains an (invisible) TAB character, which confuses make. You can get past this problem by installing and using GNU make, or by manually editing *lib/Makefile* (and any others exhibiting this problem) to make it look like this:

```
noinst_LIBRARIES = \  
  \  
  libmiscutil.a \  
  libntlmauth.a
```

If you have problems compiling Squid, check the FAQ first. You may also want to search the Squid web site (use the search box on the home page). Finally, if you're still stuck, send email to the squid-users@squid-cache.org list.

make Install

After compiling, you need to install the programs into their permanent directories. This might require superuser privileges, to put files in the installation directories. If so, become *root* first:

```
% su  
Password:  
# make install
```

If you enable Squid's ICMP measurement features with the `--enable-icmp` option, you must install the pinger program. The pinger program must be installed with superuser privileges because only *root* is allowed to send and receive ICMP messages. The following command installs pinger with the appropriate permissions:

```
# make install-pinger
```

After installing Squid, you should see the following directories and files listed under the installation prefix directory (*/usr/local/squid* by default):

sbin

The *sbin* directory contains programs normally started by *root*.

sbin/squid

This is the main Squid program.

bin

The *bin* directory contains programs for all users.

bin/RunCache

RunCache is a shell script you can use to start Squid. If Squid dies, this script automatically starts it again, unless it detects frequent restarts. The *RunCache* script is a relic from the time when Squid was not a daemon process. With the current versions, *RunCache* is less useful because Squid automatically restarts itself when you don't use the `-N` option.

bin/RunAccel

The *RunAccel* script is nearly identical to *RunCache*, except that it adds a command-line argument that tells Squid where to listen for HTTP requests.

bin/squidclient

squidclient is a simple HTTP client you can use to test Squid. It also has some special features for making management requests to a running Squid process.

libexec

The *libexec* directory traditionally contains helper programs. These are commands that you wouldn't normally run yourself. Rather, these programs are normally started by other programs.

libexec/unlinkd

unlinkd is a helper program that removes files from the cache directories. As you'll see later, file deletion can be a significant bottleneck. By implementing the delete operation in an external process, Squid achieves some performance gain.

libexec/cachemgr.cgi

cachemgr.cgi is a CGI interface to Squid's management functions. To use it, you'll probably need to copy this program to your HTTP server's *cgi-bin* directory. You'll see more about this in Chapter 14.

libexec/diskd (optional)

You get this only if you specify `--enable-storeio=diskd`.

libexec/pinger (optional)

You get this only if you specify `--enable-icmp`.

etc

The *etc* directory contains Squid's configuration files.

etc/squid.conf

This is the primary configuration file for Squid. Initially, this file contains a lot of comments to explain what each option does. After you understand the configuration directives, it's a good idea to remove the comments to make the configuration file smaller and easier to read. Note that the installation procedure doesn't overwrite this file if it already exists.

etc/squid.conf.default

This is a copy of the default configuration file from the source distribution. You may find it useful to have a copy of the current default configuration file after upgrading your Squid installation. New configuration directives may be added, and some of the existing directives may have changed.

etc/mime.conf

The *mime.conf* file tells Squid which MIME types to use for data retrieved from FTP and Gopher servers. The file is a table that correlates filename extensions to MIME types. Normally, you won't need to edit this file. However, you may need to add entries for special file types used within your organization.

etc/mime.conf.default

This is the default *mime.conf* file from the source distribution.

share

The *share* directory normally contains read-only data files used by Squid.

share/mib.txt

This is the SNMP Management Information Base (MIB) file for Squid. Squid doesn't use this file itself. Rather, your SNMP agent software (such as *snmpget* and Multi-Router Traffic Grapher (MRTG)) needs this file to understand the SNMP objects available from Squid.

share/icons

The *share/icons* directory contains a number of small icon files Squid uses in FTP and Gopher directory listings. Normally, you won't need to worry about these files, but you can change them if you want.

share/errors

The *share/errors* directory contains templates for the error messages Squid shows to users. These files are copied from the source directory when you install Squid. You can edit them if you like. However, the installation procedure always overwrites these files every time you run `make install`. So if you want to have customized error messages, it's a good idea to put them in a different directory.

var

The *var* directory contains files that aren't critical and that change frequently. These are the sort of files you don't normally back up.

var/logs

The *var/logs* directory is the default location for Squid's various log files. It is empty when you first install Squid. Once Squid gets running, you can expect to find files here named *access.log*, *cache.log*, and *store.log*.

var/cache

This is the default cache directory (*cache_dir*) if you don't specify one in *squid.conf*. See Chapter 7 for all the details about cache directories.

Applying a Patch

After you've been running Squid for a while, you may find that you need to patch the source code to fix a bug or add an experimental feature. Patches are posted for important bug fixes on the *squid-cache.org* web site. If you don't want to wait for the next official release, you can download and apply the patch to your source code. You will then need to recompile Squid.

To apply a patch—also sometimes called a *diff*—you need a program called *patch*. Chances are that your operating system already has the *patch* program. If not, you can download it from the GNU collection (<http://www.gnu.org/directory/patch.html>). Note that if you're using anonymous CVS (see Chapter 2), you don't need to worry about patching files. The CVS system does it for you automatically when you update your tree.

To apply a patch, you need to save the patch file somewhere on your system. Then *cd* to the Squid source directory and run the command like this:

```
% cd squid-2.5.STABLE4
% patch < /tmp/patch_file
```

By default, the *patch* program tells you what it's doing as it runs. Usually this output scrolls by very quickly, unless there is a problem. You can safely ignore the warnings that say offset NNN lines. If you don't want to see all this output, use the *-s* option to make *patch* silent.

When *patch* updates the source files, it creates a backup copy of the original file. For example, if you're applying a patch to *src/http.c*, *patch* names the backup file *src/http.c.orig*. Thus, if you want to undo the patch after applying it, you can simply rename all the *.orig* files back to their former names. To use this technique successfully, it's a good idea to remove all *.orig* files before applying a patch.

If *patch* encounters a problem, it stops and prompts you for advice. Common problems are as follows:

- Running patch from the wrong directory. To fix this problem, you may need to `cd` to a different directory or use patch's `-p` option.
- Patch is already applied. patch can usually tell if the patch file has already been applied. In this case, it asks if you want to unpatch the file.
- The patch program doesn't understand the file you are giving it. Patch files come in three flavors: normal, context, and unified. Old versions of patch may not understand context or unified diff output. Getting the latest version from the GNU FTP site will solve this problem.
- Corrupted patch file. If you aren't careful when downloading and saving the patch file, it may become corrupted. Sometimes people send patch files in email messages, and it is tempting to simply cut-and-paste them into a new window. On some systems, cut-and-paste can change Tab characters into spaces, or incorrectly wrap long lines. Both changes confuse patch. The `-l` option may be helpful, but it's best to make sure you copy and save the patch file correctly.

Sometimes patch can't apply part or all of the diff. In these cases, you'll see such messages as `Hunk 3 of 4 failed`. The failed sections are saved to files named `.rej`. For example, if a failure occurs while processing `src/http.c`, patch saves that piece of the diff to `src/http.c.rej`. In some cases, you may be able to fix these by hand, but it's usually not worth the trouble. If you have a lot of "failed hunks" or `.rej` files, it's a good idea to download a whole new copy of the latest source code.

After you apply a patch, you need to recompile Squid. One of the great things about make is that it only recompiles the files that have changed. But sometimes make doesn't comprehend all the intricate dependencies, and it doesn't rebuild enough of the files. To be safe, it's usually a good idea to recompile everything. The best way to do this is to clean the source tree before recompiling:

```
% make clean
% make
```

Running configure Later

Sometimes you may find it necessary to rerun `./configure`. For example, if you tune your kernel parameters, you must run `./configure` again so it picks up the new settings. As you read this book, you may also find that you want to use features that must be enabled with `./configure` options.

To rerun `./configure` with the same options, use this command:

```
% ./config.status --recheck
```

Another technique is to "touch" the `config.status` file, which updates its timestamp. This causes make to re-run the `./configure` script before compiling the source code:

```
% touch config.status
% make
```

To add or remove `./configure` options, you need to type in the whole command again. If you can't remember the previous options, just look at the top of the `config.status` file. For example:

```
% head config.status
#! /bin/sh
# Generated automatically by configure.
# Run this file to recreate the current configuration.
# This directory was configured as follows,
# on host foo.life-gone-hazy.com:
#
# ./configure --enable-storeio=ufs,diskd --enable-carp \
# --enable-auth-modules=NCSA
# Compiler output produced by configure, useful for debugging
# configure, is in ./config.log if it exists.
```

After rerunning `./configure`, you must compile and install Squid again. To be safe, it's a good idea to run `make clean` first:

```
% make clean
% make
```

Recall that `./configure` caches the things it discovers about your system. In some situations, you'll want to clear this cache and start the compilation process from the very beginning. You can simply remove the `config.cache` file if you like. Then, the next time `./configure` runs, it won't use the previous values. You can also restore the Squid source tree to its preconfigure state with the following command:

```
% make distclean
```

This removes all object files and other files created by the `./configure` and `make` commands.

Exercises

- After compiling Squid, remove one or more of the `.o` files and run `make` again.
- Use the `ulimit` or `limits` command to change the file descriptor limit to some small value before compiling Squid. Does `./configure` obey or ignore your new limit?
- Compile Squid with a high file-descriptor limit, then try to run it on a system with a lower limit. Does Squid use the lower or higher limit?
- What happens if you mistype one of the `--enable` options? What if you specify an invalid storage scheme with the `--enable-store-io` option?
- After compiling Squid, remove `src/Makefile` and try to compile it again. What's the easiest way to restore the file?

Configuration Guide for the Eager

After compiling and installing Squid, your next task is to delve into the configuration file. If you're new to Squid, you're likely to find it a bit overwhelming. The most recent version has approximately 200 configuration file directives and 2700 lines of comments. I certainly don't expect you to read about, and configure, every directive before starting Squid. This chapter can help you get Squid running quickly.

All the *squid.conf* directives have default values. You might be able to get Squid going without even touching the configuration file. However, I don't recommend trying that. You'll be much happier if you read the following sections first.

If you are really turned off by Squid's configuration file syntax, you might want to try the Webmin graphical user interface. It allows you to configure Squid (and numerous other programs) from your web browser. See <http://www.webmin.com> and *The Book of Webmin* by Joe Cooper (No Starch Press) for more information.

The squid.conf Syntax

Squid's configuration file is relatively straightforward. It is similar in style to many other Unix programs. Each line begins with a configuration directive, followed by some number of values and/or keywords. Squid ignores empty lines and comment lines (beginning with #) when reading the configuration file. Here are some sample configuration lines:

```
cache_log /squid/var/cache.log

# define the localhost ACL
acl localhost src 127.0.0.1/32

connect_timeout 2 minutes

log_fqdn on
```

Some directives take a single value. For these, repeating the directive with a different value overwrites the previous value. For example, there is only one *connect_timeout*

value. The first line in the following example has no effect because the second line overwrites it:

```
connect_timeout 2 minutes
connect_timeout 1 hour
```

On the other hand, some directives are actually lists of values. For these, each occurrence of the directive adds a new value to the list. The *extension_methods* directive works this way:

```
extension_methods UNGET
extension_methods UNPUT
extension_methods UNPOST
```

For these list-based directives, you can also usually put multiple values on the same line:

```
extension_methods UNGET UNPUT UNPOST
```

Many of the directives have common types. For example, *connect_timeout* is a time specification that has a number followed by a unit of time. For example:

```
connect_timeout 3 hours
client_lifetime 4 days
negative_ttl 27 minutes
```

Similarly, a number of directives refer to the size of a file or chunk of memory. For these, you can write a size specification as a decimal number, followed by bytes, KB, MB, or GB. For example:

```
minimum_object_size 12 bytes
request_header_max_size 10 KB
maximum_object_size 187 MB
```

Another type worth mentioning is the toggle, which can be either on or off. Many directives use this type. For example:

```
server_persistent_connections on
strip_query_terms off
prefer_direct on
```

In general, the configuration file directives may appear in any order. However, the order is important when one directive makes reference to something defined by another. Access controls are a good example. An *acl* must be defined before it can be used in an *http_access* rule:

```
acl Foo src 1.2.3.4
http_access deny Foo
```

Many things in *squid.conf* are case-sensitive, such as directive names. You can't write *HTTP_port* instead of *http_port*.

The default *squid.conf* file contains comments describing each directive, as well as the default values. For example:

```
# TAG: persistent_request_timeout
#     How long to wait for the next HTTP request on a persistent
#     connection after the previous request completes.
#
#Default:
# persistent_request_timeout 1 minute
```

Each time you install Squid, the current default configuration file is saved as *squid.conf.default* in the *\$prefix/etc* directory. Since directives change from time to time, you can refer to this file for the most up-to-date documentation on *squid.conf*.

The rest of this chapter is about the handful of directives you need to know before running Squid for the very first time.

User IDs

As you probably know, Unix processes and files have *user* and *group* ownership attributes. You need to select a user and group for Squid. This user and group combination must have read and write access to most of the Squid-related files and directories.

I highly recommend creating a dedicated *squid* user and group. This minimizes the chance that someone can exploit Squid to read other files on the system. If more than one person has administrative authority over Squid, you can add them to the *squid* group.

Unix processes inherit their parent process' ownership attributes. That is, if you start Squid as user *joe*, Squid also runs as user *joe*. If you don't want Squid to run as *joe*, you need to change your user ID beforehand. This is typically accomplished with the *su* command. For example:

```
joe% su - squid
squid% /usr/local/squid/sbin/squid
```

Unfortunately, running Squid isn't always so simple. In some cases, you may need to start Squid as *root*, depending on your configuration. For example, only *root* can bind a TCP socket to privileged ports like port 80. If you need to start Squid as *root*, you must set the *cache_effective_user* directive. It tells Squid which user to become after performing the tasks that require special privileges. For example:

```
cache_effective_user squid
```

The name that you provide must be a valid user (i.e., in the */etc/passwd* file). Furthermore, note that this directive is used only when you start Squid as *root*. Only *root* has the ability to become another user. If you start Squid as *joe*, it can't switch to user *squid*.

You might be tempted to just run Squid as *root* without setting *cache_effective_user*. If you try, you'll find that Squid refuses to run. This, again, is due to security concerns. If an outsider were somehow able to compromise or exploit Squid, he could

gain full access to your system. Although we strive to make Squid secure and bug-free, this requirement provides some extra insurance, just in case.

If you start Squid as *root* without setting *cache_effective_user*, Squid uses nobody as the default value. Whatever user ID you choose for Squid, make sure it has read access to the files installed in *\$prefix/etc*, *\$prefix/libexec*, and *\$prefix/share*. The user ID must also have write access to the log files and cache directory.

Squid also has a *cache_effective_group* directive, but you probably don't need to set it. By default, Squid uses the *cache_effective_user*'s default group (from the password file).

Port Numbers

The *http_port* directive tells Squid which port number to listen on for HTTP requests. The default is port 3128:

```
http_port 3128
```

If you are running Squid as a surrogate (see Chapter 15), you should probably set this to 80.

You can instruct Squid to listen on multiple ports with additional *http_port* lines. This is often useful if you must support groups of clients that have been configured differently. For example, the browsers from one department may be sending requests to port 3128, while another department uses port 8080. Simply list both port numbers as follows:

```
http_port 3128
http_port 8080
```

You can also use the *http_port* directive to make Squid listen on specific interface addresses. When Squid is used on a firewall, it should have two network interfaces: one internal and one external. You probably don't want to accept HTTP requests coming from the external side. To make Squid listen on only the internal interface, simply put the IP address in front of the port number:

```
http_port 192.168.1.1:3128
```

Log File Pathnames

I'll discuss all the details of Squid's log files in Chapter 13. For now the only thing you may need to worry about is where you want Squid to put its log files. The default location is a directory named *logs* under the installation prefix. For example, if you don't use the *--prefix=* option with *./configure*, the default log file directory is */usr/local/squid/var/logs*.

You need to make sure that log files are stored on a disk partition with enough space. When Squid receives a write error for a log file, it exits and restarts. The primary rea-

son for this behavior is to grab your attention. Squid wants to make sure you don't miss any important logging information, especially if your system is being abused or attacked.

Squid has three main log files: *cache.log*, *access.log*, and *store.log*. The first of these, *cache.log*, contains informational and debugging messages. When you start Squid the first few times, you should closely watch this file. If Squid refuses to run, the reason is probably at the end of *cache.log*. Under normal conditions, this log file doesn't become large enough to warrant any special attention. Also note that if you start Squid with the *-s* option, the important *cache.log* messages are also sent to your *syslog* daemon. You can change the location for this log file with the *cache_log* directive:

```
cache_log /squid/logs/cache.log
```

The *access.log* file contains a single line for each client request made to Squid. On average, each line is about 150 bytes. In other words, it takes about 150 MB to log one million client requests. Use the *cache_access_log* directive to change the location of this log file:

```
cache_access_log /squid/logs/access.log
```

If, for some reason, you don't want Squid to log client requests, you can specify the log file pathname as */dev/null*.

The *store.log* file is probably not very useful to most cache administrators. It contains a record for each object that enters and leaves the cache. The average record size is typically 175–200 bytes. However, Squid doesn't create an entry in *store.log* for cache hits, so it contains fewer records than *access.log*. Use the *cache_store_log* directive to change the location:

```
cache_store_log /squid/logs/store.log
```

You can easily disable *store.log* altogether by specifying the location as *none*:

```
cache_store_log none
```

If you're not careful, Squid's log files increase in size without limit. Some operating systems enforce a 2-GB file size limit, even if you have plenty of free disk space. Exceeding this limit results in a write error, which then causes Squid to exit. To keep log file sizes reasonable, you should create a *cron* job that regularly renames and archives the log files. Squid has a built-in feature to make this easy. See Chapter 13 for an explanation of log file rotation.

Access Controls

I'll have a lot to say about access controls in Chapter 6. For now, I'll cover a few controls so that more enthusiastic readers can quickly start using Squid.

Squid's default configuration file denies every client request. You must place additional access control rules in *squid.conf* before anyone can use the proxy. The

simplest approach is to define an ACL that corresponds to your user's IP addresses and an access rule that tells Squid to allow HTTP requests from those addresses. Squid has many different ACL types. The *src* type matches client IP addresses, and the *http_access* rules are checked for client HTTP requests. Thus, you need to add only two lines:

```
acl MyNetwork src 192.168.0.0/16
http_access allow MyNetwork
```

The tricky part is putting these lines in the right place. The order of *http_access* lines is very important, but the order of *acl* lines doesn't matter. You should also be aware that the default configuration file contains some important access controls. You shouldn't change or disrupt these until you fully comprehend their significance. When you edit *squid.conf* for the first time, look for this comment:

```
#
# INSERT YOUR OWN RULE(S) HERE TO ALLOW ACCESS FROM YOUR CLIENTS
#
```

Insert your new rules below this comment, and before the *http_access deny All* line.

For the sake of completeness, here is a suitable initial access control configuration, including the recommended default controls and the example earlier:

```
acl All src 0/0
acl Manager proto cache_object
acl Localhost src 127.0.0.1/32
acl Safe_ports port 80 21 443 563 70 210 280 488 591 777 1025-65535
acl SSL_ports 443 563
acl CONNECT method CONNECT
acl MyNetwork src 192.168.0.0/16

http_access allow Manager Localhost
http_access deny Manager
http_access deny !Safe_ports
http_access deny CONNECT !SSL_ports
http_access allow MyNetwork
http_access deny All
```

Visible Hostname

Hopefully, you won't need to worry about the *visible_hostname* directive. However, you'll need to set it if Squid can't figure out the hostname of the machine on which it is running. When this happens, Squid complains and refuses to run:

```
% squid -Nd1
FATAL: Could not determine fully qualified hostname. Please set 'visible_hostname'
```

Squid wants to be sure about its hostname for a number of reasons:

- The hostname appears in Squid's error messages. This helps users identify the source of potential problems.
- The hostname appears in the HTTP Via header of cache misses that Squid forwards. When the request arrives at the origin server, the Via header contains a list of all proxies involved in the transaction. Squid also uses the Via header to detect forwarding loops. I'll talk about forwarding loops in Chapter 10.
- Squid uses internal URLs for certain things, such as the icons for FTP directory listings. When Squid generates an HTML page for an FTP directory, it inserts embedded images for little icons that indicate the type of each file in the directory. The icon URLs contain the cache's hostname so that web browsers request them directly from Squid.
- Each HTTP reply from Squid includes an X-Cache header. This isn't an official HTTP header. Rather, it is an extension header that indicates if the response was a cache hit or a cache miss. Since requests and responses may flow through more than one cache, each X-Cache header includes the name of the cache reporting hit or miss. Here's a sample response that passed through two caches:

```
HTTP/1.0 200 OK
Date: Mon, 29 Sep 2003 22:57:23 GMT
Content-type: text/html
Content-length: 733
X-Cache: HIT from bo2.us.ircache.net
X-Cache: MISS from bo1.us.ircache.net
```

Squid tries to figure out the hostname automatically at startup. First it calls the `gethostname()` function, which usually returns the correct hostname. Next, Squid attempts a DNS lookup on the hostname with `gethostbyname()`. This function typically returns both IP addresses and the *canonical* name for the system. If `gethostbyname()` succeeds, Squid uses the canonical name in error messages, Via headers, etc.

Squid may be unable to determine its fully qualified hostname for a number of reasons, including:

- The hostname may not be set.
- The hostname may be missing from the DNS zone or */etc/hosts* files.
- The Squid system's DNS client configuration may be incorrect or missing. On Unix, you should check the */etc/resolv.conf* and */etc/host.conf* files.

If you see the fatal message mentioned previously, you need either to fix the hostname and DNS information or explicitly configure the hostname for Squid. In most cases, it is sufficient to ensure the `hostname` command returns a fully qualified hostname and add an entry to */etc/hosts*. If that doesn't work, just set the visible hostname in *squid.conf*:

```
visible_hostname squid.packet-pushers.net
```

Administrative Contact Information

You should set the *cache_mgr* directive as a favor to your users. The value is an email address users can write to in case a problem surfaces. The *cache_mgr* address appears in Squid's error messages by default. For example:

```
cache_mgr squid@web-cache.net
```

Next Steps

After creating the minimal configuration file, you're more or less ready to run Squid for the first time. To do that, just follow the instructions in the next chapter.

When you've mastered starting and stopping Squid, you can spend some time beefing up the configuration file. You may want to add more sophisticated access controls, which you'll find documented in Chapter 6. Since I didn't say anything about the disk cache yet, you should also spend a fair amount of time in Chapters 7 and 8.

Exercises

- Parse Squid's configuration file with **squid -k parse** and check the process exit status.
- Intentionally introduce a some errors into the configuration file and run **squid -k parse** again. Notice how Squid reports different errors.
- Insert comments into the configuration file. Can you start a comment anywhere, even after a valid directive?
- Why do you think some configuration file errors are fatal, but others are not?

Running Squid

Now that you have Squid installed, and maybe even configured, you need to learn the ins and outs of running Squid. Although most of the configuration occurs in *squid.conf*, you may find some of Squid's command-line options useful. For example, one of the first things you must do is use the `-z` option to initialize the cache directories. You may also find the `-d` option useful for debugging.

Squid normally runs as a daemon process. If you are new to Squid, however, I recommend running Squid in the foreground from a terminal window until you are confident that it is working properly. Following that, you can run Squid as a daemon, in the background. Most likely, you'll want to start Squid each time your system boots. Different operating systems have different approaches to startup scripts. I'll show you how to make it happen in three different ways.

You can send signals to the running Squid process to execute various tasks, such as halting and reconfiguring Squid, and rotating the log files. Although you can use the `kill` command to send signals, it is easier to use the `squid -k` commands.

Squid Command-Line Options

Before getting too far into other things, let's look at Squid's command-line options. Many of these you will never use and some are useful only when debugging problems:

`-a port`

Specifies a new *http_port* value. This option always overrides the value from *squid.conf*. Note, however, that you can specify multiple values in *squid.conf*. The `-a` option overrides only the first value from the config file. (This option uses the letter "a" because in the Harvest cache, the HTTP port was called the ASCII port.)

- d *level*
Makes Squid write its debugging messages to *stderr* (as well as *cache.log* and *syslog*, if configured). The *level* argument specifies the maximum level for messages that should be shown on *stderr*. In most cases *-d1* works well. See Chapter 16 for a description of debugging levels.
- f *file*
Specifies an alternate configuration file.
- h Displays the usage information.
- k *function*
Signals Squid to perform various administrative functions. The *function* argument may be one of the following: *reconfigure*, *rotate*, *shutdown*, *interrupt*, *kill*, *debug*, *check*, or *parse*. *reconfigure* causes the running Squid process to reread its configuration file. *rotate* causes Squid to rotate its log files, which involves closing them, possibly renaming them, and opening them again. *shutdown* sends the signal to shut down the Squid process. *interrupt* also shuts down Squid but does so immediately, without waiting for active transactions to finish. *kill* sends the unstoppable KILL signal to Squid, which should only be used as a last resort. *debug* puts Squid into full debugging mode. It can quickly fill up your disk space if your cache is busy. *check* simply checks for a running Squid process. The process return value indicates whether Squid is running or not. Finally, *parse* simply parses the *squid.conf* file. The process return value is non-zero if the configuration file contains errors.
- s Enables logging to the *syslog* daemon. Squid uses the *LOCAL4* *syslog* facility. Level 0 debug messages are logged with priority *LOG_WARNING*, and level 1 messages are logged with *LOG_NOTICE*. Higher level debugging messages aren't sent to *syslogd*. You might use an entry like this in */etc/syslogd.conf*:

local4.warning */var/log/squid.log*
- u *port*
Specifies an alternate ICP port number, overriding *icp_port* in *squid.conf*.
- v Prints the version string.
- z Initializes cache, or swap, directories. You must use this option when running Squid for the first time or whenever you add a new cache directory.
- C Prevents the installation of signal handlers that trap certain fatal signals such as *SIGBUS* and *SIGSEGV*. Normally, the signals are trapped by Squid so that it can attempt a clean shutdown. However, trapping the signal may make it harder to debug the problem afterwards. With this option, the fatal signals cause their default actions, which is usually to dump core.
- D Disables initial DNS tests. Normally, Squid won't start until it verifies that its DNS server is working. This option prevents that check. You can also alter or remove the *dns_testnames* option in *squid.conf*.

- F Makes Squid refuse all requests until it rebuilds the storage metadata. If your cache is busy, this option may shorten the time required to rebuild the metadata. If your cache is large, however, the rebuild procedure may take a long time anyway.
- N Prevents Squid from becoming a background daemon process.
- R Prevents Squid from using the `SO_REUSEADDR` option before binding to the HTTP port.
- V Enables virtual host surrogate mode. Similar to entering `httpd_accel_host virtual` in `squid.conf`.
- X Forces full debugging, as though you had specified `debug_options ALL,9` in `squid.conf`.
- Y Returns `ICP_MISS_NOFETCH` instead of `ICP_MISS` when rebuilding store metadata. For busy parent caches, this option may result in less load while the cache is rebuilding. See Chapter 10.

Check Your Configuration File for Errors

Before trying to start Squid, you should verify that your `squid.conf` file makes sense. This is easy to do. Just run the following command:

```
% squid -k parse
```

If you see no output, the configuration file is valid, and you can proceed to the next step.

However, if your configuration file contains an error, Squid tells you about it:

```
squid.conf line 62: http_access allow okay2
aclParseAccessLine: ACL name 'okay2' not found.
```

Here you can see that the `http_access` directive on line 62 references an ACL that doesn't exist. Sometimes the error messages are less informative:

```
FATAL: Bungled squid.conf line 76: memory_pools
```

In this case, we forgot to put either on or off after the `memory_pools` directive on line 76.

It's a good idea to develop the habit of using `squid -k parse` every time you modify your configuration file. If you don't bother, and your file has some errors, Squid tells you about them and refuses to start anyway. If you end up managing a number of caches, it is likely that you'll develop some scripts to automate starting, stopping, and reconfiguring Squid. You can use this feature in your scripts to ensure that the configuration files are always valid.

Initializing Cache Directories

Before running Squid for the first time, and whenever you add a new *cache_dir*, you must initialize the cache directories. The command is simply:

```
% squid -z
```

For the UFS-related storage schemes (*ufs*, *aufs*, and *diskd*; see Chapter 8), this command creates the subdirectories needed under each *cache_dir*. You don't need to worry that Squid will wipe out your current cache directories (if any).

Ownership and permissions are a common problem at this stage. Squid runs under a certain user ID, specified with *cache_effective_user* in *squid.conf*. This user ID must have read and write permission under each *cache_dir* directory. If not, you'll see a message like this:

```
Creating Swap Directories
FATAL: Failed to make swap directory /usr/local/squid/var/cache/00:
(13) Permission denied
```

In this case, you should make sure that all components of */usr/local/squid/var/cache* are accessible to the user ID given in *squid.conf*. The final component—the *cache* directory—must be writable by this user ID as well.

Cache directory initialization may take a couple of minutes, depending on the size and number of cache directories, and the speed of your disk drives. If you want to watch the progress, use the *-X* option:

```
% squid -zX
```

Testing Squid in a Terminal Window

Once you've initialized the cache directories, you should run Squid in a terminal window with logging to *stderr*. This way, you can easily spot any errors or problems and make sure that Squid successfully starts. Use the *-N* option to keep Squid in the foreground and the *-d1* option to display level 1 debugging on *stderr*:

```
% squid -N -d1
```

You should see output like this:

```
2003/09/29 12:57:52| Starting Squid Cache version 2.5.STABLE4 for i386-unknown-
freebsd4.8...
2003/09/29 12:57:52| Process ID 294
2003/09/29 12:57:52| With 1064 file descriptors available
2003/09/29 12:57:52| DNS Socket created on FD 4
2003/09/29 12:57:52| Adding nameserver 206.107.176.2 from /etc/resolv.conf
2003/09/29 12:57:52| Adding nameserver 205.162.184.2 from /etc/resolv.conf
2003/09/29 12:57:52| Unlinkd pipe opened on FD 9
2003/09/29 12:57:52| Swap maxSize 102400 KB, estimated 7876 objects
2003/09/29 12:57:52| Target number of buckets: 393
2003/09/29 12:57:52| Using 8192 Store buckets
```

```

2003/09/29 12:57:52| Max Mem size: 8192 KB
2003/09/29 12:57:52| Max Swap size: 102400 KB
2003/09/29 12:57:52| Rebuilding storage in /usr/local/squid/var/cache (DIRTY)
2003/09/29 12:57:52| Using Least Load store dir selection
2003/09/29 12:57:52| Set Current Directory to /usr/local/squid/var/cache
2003/09/29 12:57:52| Loaded Icons.
2003/09/29 12:57:52| Accepting HTTP connections at 0.0.0.0, port 3128, FD 11.
2003/09/29 12:57:52| Accepting ICP messages at 0.0.0.0, port 3130, FD 12.
2003/09/29 12:57:52| WCCP Disabled.
2003/09/29 12:57:52| Ready to serve requests.

```

If you see an error message, you need to fix it before proceeding. Be sure to check the first few lines of output for warning messages. The most common errors are file/directory permissions and configuration file syntax errors. If you see an error message that doesn't make sense, have a look at Chapter 16 for advice and information on troubleshooting Squid. If that doesn't help, check the Squid FAQ, or search the mailing list archives for an explanation.

Once you see the Ready to serve requests message, test Squid with a few HTTP requests. You can do this by configuring your browser to use Squid as a proxy and then open a web page. If Squid is working correctly, the page should load as quickly as it would without using Squid. Alternatively, you can use the *squidclient* program that comes with Squid:

```
% squidclient http://www.squid-cache.org/
```

If this works, Squid's home page HTML file will scroll across your terminal window. Once you're confident that Squid works okay, you can interrupt the Squid process (i.e., with Ctrl-C) and run Squid as a daemon.

Running Squid as a Daemon Process

Normally you'll want to run Squid as a daemon process (i.e., not attached to your terminal window). The easiest way to do this is simply execute Squid as follows:

```
% squid -s
```

The *-s* option causes Squid to write important status and warning messages to syslogd. Squid uses the LOCAL4 facility and the LOG_WARNING and LOG_NOTICE priorities. Your syslog daemon may or may not actually log Squid's messages, depending on how it is configured. These same messages are written to the *cache.log* file, so it is safe to omit the *-s* option if you prefer.

When you start Squid without the *-N* option (as shown earlier), Squid automatically backgrounds itself and creates a parent/child process pair. The child process is the one that does all the real work. The parent process makes sure that a child process is always running. Thus, if the child process dies unexpectedly, the parent starts

another so that Squid remains in operation. You can see this parent/child process interaction by looking at your syslog messages:

```
Jul 31 14:58:35 zapp squid[294]: Squid Parent: child process 296 started
```

Here you can see that the parent is process ID 294, and the child is 296. When you look at ps output, you'll see that the child process is listed as (squid):

```
% ps ax | grep squid
294 ?? Is    0:00.01 squid -sD
296 ?? S     0:00.27 (squid) -sD (squid)
```

If the child Squid process dies unexpectedly, the parent starts another. For example:

```
Jul 31 15:02:53 zapp squid[294]:SquidParent:child process 296 exited due to signal 6
Jul 31 15:02:56 zapp squid[294]: Squid Parent: child process 359 started
```

In some situations, the child Squid process may die immediately. Rather than constantly spawning new Squid processes, the parent process gives up if the child processes won't stay running for at least 10 seconds five times in a row:

```
Jul 31 15:13:48 zapp squid[455]: Squid Parent: child process 474 exited with status 1
Jul 31 15:13:48 zapp squid[455]: Exiting due to repeated, frequent failures
```

If this happens to you, check syslog and Squid's *cache.log* for error messages.

The squid_start Script

When Squid runs as a daemon process, it looks for a file named *squid_start* in the same directory as the *squid* binary. If found, this program is executed before the parent process forks to run the child process. You can use this script for certain administrative tasks, such as notifying someone that Squid is starting, managing log files, etc. Squid doesn't start the child process until the *squid_start* program exits.



The *squid_start* script only works when you start Squid by its absolute or relative pathname. In other words, Squid doesn't use the *PATH* environment variable to locate *squid_start*. Thus, you may want to develop the habit of starting Squid like this:

```
% /usr/local/squid/sbin/squid -sD
```

rather than starting Squid like this:

```
% squid -sD
```

Boot Scripts

Most likely, you'll want Squid to start automatically every time your computer boots. Different operating systems vary widely in how their boot-up scripts work. I'll describe some common environments here, but you may need to refer to your particular operating system for specific information.

/etc/rc.local

One of the easiest schemes is the */etc/rc.local* script. This is simply a shell script that runs as *root* each time the system boots. Using this script to start Squid is as easy as adding the following line:

```
/usr/local/squid/sbin/squid -s
```

Of course your installation prefix may be different, and you may like to use some other command-line options. Don't use the *-N* option here.

If, for some reason, you're not using the *cache_effective_user* directive, you can try using *su* to start Squid as a non-*root* user:

```
/usr/bin/su nobody -c '/usr/local/squid/sbin/squid -s'
```

init.d and rc.d

The *init.d* and *rc.d* schemes use a separate shell script to start different services. These scripts are often located in one of the following directories: */sbin/init.d*, */etc/init.d*, and */usr/local/etc/rc.d*. The scripts usually take a single command-line argument, which is either *start* or *stop*. Some systems only use the start argument. Here's a basic script for starting Squid:

```
#!/bin/sh
#
# this script starts and stops Squid

case "$1" in
start)
    /usr/local/squid/sbin/squid -s
    echo -n ' Squid'
    ;;
stop)
    /usr/local/squid/sbin/squid -k shutdown
    ;;
esac
```



Linux users may want to add commands that set the file-descriptor limits before running Squid. For example:

```
echo 8192 > /proc/sys/fs/file-max
limit -HSn 8192
```

To use this script, find the appropriate directory in which such scripts are stored. Give it a meaningful name, similar to the others. Perhaps *S98squid* or simply *squid.sh*. Be sure to test the script by rebooting your computer rather than assuming it will work.

/etc/inittab

Another scheme supported on some operating systems is the */etc/inittab* file. On these systems, the *init* process starts and stops services based on the *run level*. A typical *inittab* entry looks like this:

```
sq:2345:once:/usr/local/squid/sbin/squid -s
```

With this entry, the *init* process starts Squid just once and then forgets about it. Squid makes sure it stays running as described previously. Alternatively, you can do it like this:

```
sq:2345:respawn:/usr/local/squid/sbin/squid -Ns
```

Here, since we use the *respawn* option, *init* restarts Squid if the process exits. If you use *respawn*, be sure to use the *-N* option.

After editing the *inittab* file, use this command to make *init* reread its configuration file and start Squid:

```
# init q
```

A chroot Environment

Some people like to run Squid in a *chroot* environment. This is a Unix feature that gives a process a new root filesystem directory. It provides an extra level of security in the event that Squid is compromised. If an attacker somehow gains access to the operating system through Squid, she can only access files under the *chroot* filesystem. The other system files, outside of the *chroot* tree, remain inaccessible.

The easiest way to run Squid in a *chroot* environment is by specifying the new root directory in the *squid.conf* file with this directive:

```
chroot /new/root/directory
```



The `chroot()` system call requires superuser privileges, so you must start Squid as *root* to use this feature.

The *chroot* environment isn't for first-time Unix users. It is a little tricky because you must replicate a number of files underneath the new root directory. For example, if the default configuration file is normally */usr/local/squid/etc/squid.conf*, and you use the *chroot* directive, the file must be located at */new/root/directory/usr/local/squid/etc/squid.conf*. You must copy all of the files under *\$prefix/etc*, *\$prefix/share*, and *\$prefix/libexec* to the *chroot* directory. Make sure that *\$prefix/var* and the cache directories exist and are writable under the *chroot* directory as well.

Chances are that your operating system requires a number of files in the *chroot* directory, such as */etc/resolv.conf* and */dev/null*. If you use an external helper program,

such as a redirector (see Chapter 11) or an authenticator (see Chapter 12), you'll also need some shared libraries from */usr/lib*. You can use the *ldd* utility to find out which shared libraries are required for a given program:

```
% ldd /usr/local/squid/libexec/ncsa_auth
/usr/local/squid/libexec/ncsa_auth:
    libcrypt.so.2 => /usr/lib/libcrypt.so.2 (0x28067000)
    libm.so.2 => /usr/lib/libm.so.2 (0x28080000)
    libc.so.4 => /usr/lib/libc.so.4 (0x28098000)
```

You can also use the *chroot* command to test helpers:

```
# chroot /new/root/directory /usr/local/squid/libexec/ncsa_auth
/usr/libexec/ld-elf.so.1: Shared object "libcrypt.so.2" not found
```

For more information on *chroot*, see the *chroot()* manpage on your system.

Stopping Squid

The safest way to shut down Squid is with the *squid -k shutdown* command:

```
% squid -k shutdown
```

This command sends the TERM signal to the running Squid process. Upon receipt of the TERM signal, Squid closes its incoming sockets so that new requests aren't accepted. It then waits some amount of time for outstanding requests to complete. The default is 30 seconds, which you can change with the *shutdown_lifetime* directive.

If, for some reason, the *squid.pid* file is missing or unreadable, the *squid -k* commands don't work. In this case, you can manually kill Squid by finding the process ID with *ps*. For example:

```
% ps ax | grep squid
```

If you see more than one Squid process, be sure to kill the one that shows up as (squid). For example:

```
% ps ax | grep squid
294 ?? Is    0:00.01 squid -sD
296 ?? S     0:00.27 (squid) -sD (squid)
% kill -TERM 296
```

After sending the TERM signal, you may want to watch the log file to double-check that Squid is shutting down:

```
% tail -f logs/cache.log
2003/09/29 21:49:30| Preparing for shutdown after 9316 requests
2003/09/29 21:49:30| Waiting 10 seconds for active connections to finish
2003/09/29 21:49:30| FD 11 Closing HTTP connection
2003/09/29 21:49:31| Shutting down...
2003/09/29 21:49:31| FD 12 Closing ICP connection
2003/09/29 21:49:31| Closing unlinked pipe on FD 9
2003/09/29 21:49:31| storeDirWriteCleanLogs: Starting...
2003/09/29 21:49:32| Finished. Wrote 253 entries.
```

```
2003/09/29 21:49:32| Took 0.1 seconds (1957.6 entries/sec).
2003/09/29 21:49:32| Squid Cache (Version 2.5.STABLE4): Exiting normally.
```

If you use `squid -k interrupt`, Squid shuts down immediately, without waiting for active requests to complete. This is equivalent to sending the `INT` signal with `kill`.

Reconfiguring a Running Squid Process

As you learn more about Squid, you'll probably find yourself making many changes to the *squid.conf* file. To have the new settings take effect, you can either shut down and restart Squid, or you can reconfigure Squid while it is running.

The best way to reconfigure a running Squid process is with the `squid -k reconfigure` command:

```
% squid -k reconfigure
```

When you run this command, a `HUP` signal is sent to the running Squid process. Squid then reads and parses the *squid.conf* file. If the operation is successful, you'll see this in *cache.log*:

```
2003/09/29 22:02:25| Restarting Squid Cache (version 2.5.STABLE4)...
2003/09/29 22:02:25| FD 12 Closing HTTP connection
2003/09/29 22:02:25| FD 13 Closing ICP connection
2003/09/29 22:02:25| Cache dir '/usr/local/squid/var/cache' size remains unchanged
                    at 102400 KB
2003/09/29 22:02:25| DNS Socket created on FD 5
2003/09/29 22:02:25| Adding nameserver 10.0.0.1 from /etc/resolv.conf
2003/09/29 22:02:25| Accepting HTTP connections at 0.0.0.0, port 3128, FD 9.
2003/09/29 22:02:25| Accepting ICP messages at 0.0.0.0, port 3130, FD 11.
2003/09/29 22:02:25| WCCP Disabled.
2003/09/29 22:02:25| Loaded Icons.
2003/09/29 22:02:25| Ready to serve requests.
```

You need to be a little careful with the `reconfigure` option because it's possible to make changes that cause a fatal error. For example, note that Squid closes and reopens the incoming HTTP and ICP sockets. If you change the *http_port* to a port number that Squid can't open, it exits with a fatal error message.

Certain options and directives can't be changed while Squid is running. This includes:

- Removal of cache directories (*cache_dir* directive).
- Changes to the *store_log* directive.
- Changing the block-size value for *compress cache_dirs*. In fact, whenever you change this value, you must reinitialize the *compress cache_dir*.
- The *coredump_dir* directive isn't examined during the reconfigure procedure. Thus, you can't make Squid change its current directory after it has started.

Solaris users may experience a subtle problem when reconfiguring Squid. The `fopen()` call in the Solaris *stdio* implementation requires an unused file descriptor less than 256. The *FILE* structure stores the file descriptor as an 8-bit value. Normally this isn't a problem because Squid uses raw I/O (e.g., `open()`) to open cache files. However, certain tasks that occur during the reconfigure procedure use `fopen()`. These may fail if the first 256 file descriptors are already allocated.

Rotating the Log Files

Squid writes to a number of log files unless you disable them in *squid.conf*. You must periodically *rotate* the log files to prevent them from consuming too much disk space. Squid places a lot of importance on log files and exits with an error message when it can't write to them. To keep disk space consumption under control, use the following command in a *cron* job:

```
% squid -k rotate
```

For example, this *crontab* entry rotates the logs every 24 hours, at 4 A.M.:

```
0 4 * * * /usr/local/squid/sbin/squid -k rotate
```

This command does two things. First, it closes the currently open log files. Then, it renames the *cache.log*, *store.log*, and *access.log* files by appending a numeric extension. For example, *cache.log* becomes *cache.log.0*, *cache.log.0* becomes *cache.log.1*, and so on, up to the value of the *logfile_rotate* option.

Squid keeps only the last *logfile_rotate* versions of each log file. The older versions are simply removed during the renaming process. If you want to keep more copies, you need to increase the *logfile_rotate* limit or write some custom scripts that move the log files to a different location.

See Chapter 13 for additional information about rotating log files.

Exercises

- Use Squid's `-s` option and verify that its messages are saved by your syslog daemon.
- Run **squid -X -d9**, and examine some of the debugging messages.
- Write a shell script that stops Squid but doesn't exit until all Squid processes exit.
- Play with **squid -k rotate**. What happens if you have **tail -f cache.log** running when you rotate the log files?

CHAPTER 6

All About Access Controls

Access controls are the most important part of your Squid configuration file. You'll use them to grant access to your authorized users and to keep out the bad guys. You can use them to restrict, or prevent access to, certain material; to control request rewriting; to route requests through a hierarchy; and to support different qualities of service.

Access controls are built from two different components. First, you define a number of access control list (ACL) elements. These elements refer to specific aspects of client requests, such as IP addresses, URL hostnames, request methods, and origin server port numbers. After defining the necessary elements, you combine them into a number of access list rules. The rules apply to particular services or operations within Squid. For example, the *http_access* rules are applied to incoming HTTP requests. I cover the access control elements first, and then the rules later in this chapter.

Access Control Elements

ACL elements are the building blocks of Squid's access control implementation. These are how you specify things such as IP addresses, port numbers, hostnames, and URL patterns. Each ACL element has a name, which you refer to when writing the access list rules. The basic syntax of an ACL element is as follows:

```
acl name type value1 value2 ...
```

For example:

```
acl Workstations src 10.0.0.0/16
```

In most cases, you can list multiple values for one ACL element. You can also have multiple *acl* lines with the same name. For example, the following two configurations are equivalent:

```
acl Http_ports port 80 8000 8080
```

```
acl Http_ports port 80  
acl Http_ports port 8000  
acl Http_ports port 8080
```

A Few Base ACL Types

Squid has approximately 25 different ACL types, some of which have a common base type. For example, both *src* and *dst* ACLs use IP addresses as their base type. To avoid being redundant, I'll cover the base types first and then describe each type of ACL in the following sections.

IP addresses

Used by: *src*, *dst*, *myip*

Squid has a powerful syntax for specifying IP addresses in ACLs. You can write addresses as subnets, address ranges, and domain names. Squid supports both “dotted quad” and CIDR prefix^{*} subnet specifications. In addition, if you omit a netmask, Squid calculates the appropriate netmask for you. For example, each group in the next example are equivalent:

```
acl Foo src 172.16.44.21/255.255.255.255  
acl Foo src 172.16.44.21/32  
acl Foo src 172.16.44.21  
  
acl Xyz src 172.16.55.32/255.255.255.248  
acl Xyz src 172.16.55.32/28  
  
acl Bar src 172.16.66.0/255.255.255.0  
acl Bar src 172.16.66.0/24  
acl Bar src 172.16.66.0
```

When you specify a netmask, Squid checks your work. If your netmask masks out non-zero bits of the IP address, Squid issues a warning. For example, the following lines results in the subsequent warning:

```
acl Foo src 127.0.0.1/8
```

```
aclParseIpData: WARNING: Netmask masks away part of the specified IP in 'Foo'
```

The problem here is that the /8 netmask (255.0.0.0) has all zeros in the last three octets, but the IP address 127.0.0.1 doesn't. Squid warns you about the problem so you can eliminate the ambiguity. To be correct, you should write:

```
acl Foo src 127.0.0.1/32
```

* CIDR stands for Classless Inter-Domain Routing. It is from an Internet-wide effort to support routing by any prefix length, instead of the old class A, B, and C subnet lengths.

or:

```
acl Foo src 127.0.0.0/8
```

Sometimes you may need to list multiple, contiguous subnets. In these cases, it may be easier to specify an address range. For example:

```
acl Bar src 172.16.10.0-172.16.19.0/24
```

This is equivalent to, and more efficient than, this approach:

```
acl Foo src 172.16.10.0/24
acl Foo src 172.16.11.0/24
acl Foo src 172.16.12.0/24
acl Foo src 172.16.13.0/24
acl Foo src 172.16.14.0/24
acl Foo src 172.16.15.0/24
acl Foo src 172.16.16.0/24
acl Foo src 172.16.18.0/24
acl Foo src 172.16.19.0/24
```

Note that with IP address ranges, the netmask goes only at the very end. You can't specify different netmasks for the beginning and ending range values.

You can also specify hostnames in IP ACLs. For example:

```
acl Squid dst www.squid-cache.org
```



Squid converts hostnames to IP addresses at startup. Once started, Squid never makes another DNS lookup for the hostname's address. Thus, Squid never notices if the address changes while it's running.

If the hostname resolves to multiple addresses, Squid adds each to the ACL. Also note that you can't use netmasks with hostnames.

Using hostnames in address-based ACLs is usually a bad idea. Squid parses the configuration file before initializing other components, so these DNS lookups don't use Squid's nonblocking IP cache interface. Instead, they use the blocking `gethostbyname()` function. Thus, the need to convert ACL hostnames to addresses can delay Squid's startup procedure. Avoid using hostnames in *src*, *dst*, and *myip* ACLs unless absolutely necessary.

Squid stores IP address ACLs in memory with a data structure known as a *splay tree* (see <http://www.link.cs.cmu.edu/splay/>). The splay tree has some interesting self-organizing properties, one of which being that the list automatically adjusts itself as lookups occur. When a matching element is found in the list, that element becomes the new root of the tree. In this way frequently referenced items migrate to the top of the tree, which reduces the time for future lookups.

All subnets and ranges belonging to a single ACL element must not overlap. Squid warns you if you make a mistake. For example, this isn't allowed:

```
acl Foo src 1.2.3.0/24
acl Foo src 1.2.3.4/32
```

It causes Squid to print a warning in *cache.log*:

```
WARNING: '1.2.3.4' is a subnetwork of '1.2.3.0/255.255.255.0'
WARNING: because of this '1.2.3.4' is ignored to keep splay tree searching
predictable
WARNING: You should probably remove '1.2.3.4' from the ACL named 'Foo'
```

In this case, you need to fix the problem, either by removing one of the ACL values or by placing them into different ACL lists.

Domain names

Used by: *srcdomain*, *dstdomain*, and the *cache_host_domain* directive

A domain name is simply a DNS name or zone. For example, the following are all valid domain names:

```
www.squid-cache.org
squid-cache.org
org
```

Domain name ACLs are tricky because of a subtle difference relating to matching domain names and subdomains. When the ACL domain name begins with a period, Squid treats it as a wildcard, and it matches any hostname in that domain, even the domain name itself. If, on the other hand, the ACL domain name doesn't begin with a period, Squid uses exact string comparison, and the hostname must be exactly the same for a match.

Table 6-1 shows Squid's rules for matching domain and hostnames. The first column shows hostnames taken from requested URLs (or client hostnames for *srcdomain* ACLs). The second column indicates whether or not the hostname matches *lrrr.org*. The third column shows whether the hostname matches an *.lrrr.org* ACL. As you can see, the only difference is in the second case.

Table 6-1. Domain name matching

URL hostname	Matches ACL <i>lrrr.org</i> ?	Matches ACL <i>.lrrr.org</i> ?
<i>lrrr.org</i>	Yes	Yes
<i>i.am.lrrr.org</i>	No	Yes
<i>iamlrrr.org</i>	No	No

Domain name matching can be confusing, so let's look at another example so that you really understand it. Here are two slightly different ACLs:

```
acl A dstdomain foo.com
acl B dstdomain .foo.com
```

A user's request to get *http://www.foo.com/* matches ACL B, but not A. ACL A requires an exact string match, but the leading dot in ACL B is like a wildcard.

On the other hand, a user's request to get *http://foo.com/* matches both ACLs A and B. Even though there is no word before *foo.com* in the URL hostname, the leading dot in ACL B still causes a match.

Squid uses splay trees to store domain name ACLs, just as it does for IP addresses. However, Squid's domain name matching algorithm presents an interesting problem for splay trees. The splay tree technique requires that only one key can match any particular search term. For example, let's say the search term (from a URL) is *i.am.lrrr.org*. This hostname would be a match for both *.lrrr.org* and *.am.lrrr.org*. The fact that two ACL values match one hostname confuses the splay algorithm. In other words, it is a mistake to put something like this in your configuration file:

```
acl Foo dstdomain .lrrr.org .am.lrrr.org
```

If you do, Squid generates the following warning message:

```
WARNING: '.am.lrrr.org' is a subdomain of '.lrrr.org'
WARNING: because of this '.am.lrrr.org' is ignored to keep splay tree searching
predictable
WARNING: You should probably remove '.am.lrrr.org' from the ACL named 'Foo'
```

You should follow Squid's advice in this case. Remove one of the related domains so that Squid does exactly what you intend. Note that you can use both domain names as long as you put them in different ACLs:

```
acl Foo dstdomain .lrrr.org
acl Bar dstdomain .am.lrrr.org
```

This is allowed because each named ACL uses its own splay tree.

Username

Used by: *ident*, *proxy_auth*

ACLs of this type are designed to match usernames. Squid may learn a username through the RFC 1413 ident protocol or via HTTP authentication headers. Usernames must be matched exactly. For example, bob doesn't match bobby. Squid also has related ACLs (*ident_regex* and *proxy_auth_regex*) that use regular-expression pattern matching on usernames.

You can use the word **REQUIRED** as a special value to match any username. If Squid can't determine the username, the ACL isn't matched. This is how Squid is usually configured when using username-based access controls.

Regular expressions

Used by: *srcdom_regex*, *dstdom_regex*, *url_regex*, *urlpath_regex*, *browser*, *referer_regex*, *ident_regex*, *proxy_auth_regex*, *req_mime_type*, *rep_mime_type*

A number of ACLs use *regular expressions* (regex) to match character strings. (For a complete regular-expression reference, see O'Reilly's *Mastering Regular Expressions*.) For Squid, the most commonly used regex features match the beginning and/or end of a string. For example, the `^` character is special because it matches the beginning of a line or string:

```
^http://
```

This regex matches any URL that begins with `http://`. The `$` character is also special because it matches the end of a line or string:

```
.jpg$
```

Actually, the previous example is slightly wrong because the `.` character is special too. It is a wildcard that matches any character. What we really want is this:

```
\.jpg$
```

The backslash escapes the `.` so that its specialness is taken away. This regex matches any string that ends with `.jpg`. If you don't use the `^` or `$` characters, regular expressions behave like standard substring searches. They match an occurrence of the word (or words) anywhere in the string.

With all of Squid's regex types, you have the option to use case-insensitive comparison. Matching is case-sensitive by default. To make it case-insensitive, use the `-i` option after the ACL type. For example:

```
acl Foo url_regex -i ^http://www
```

TCP port numbers

Used by: *port, myport*

This type is relatively straightforward. The values are individual port numbers or port number ranges. Recall that TCP port numbers are 16-bit values and, therefore, must be greater than 0 and less than 65,536. Here are some examples:

```
acl Foo port 123
acl Bar port 1-1024
```

Autonomous system numbers

Used by: *src_as, dst_as*

Internet routers use Autonomous System (AS) numbers to construct routing tables. Essentially, an AS number refers to a collection of IP networks managed by a single organization. For example, my ISP has been assigned the following network blocks: 134.116.0.0/16, 137.41.0.0/16, 206.168.0.0/16, and many more. In the Internet routing tables, these networks are advertised as belonging to AS 3404. When routers forward packets, they typically select the path that traverses the fewest autonomous systems. If none of this makes sense to you, don't worry. AS-based ACLs should only be used by networking gurus.

Here's how the AS-based types work: when Squid first starts up, it sends a special query to a *whois* server. The query essentially says, "Tell me which IP networks belong to this AS number." This information is collected and managed by the Routing Arbiter Database (RADB). Once Squid receives the list of IP networks, it treats them similarly to the IP address-based ACLs.

AS-based types only work well when ISPs keep their RADB information up to date. Some ISPs are better than others about updating their RADB entries; many don't bother with it at all. Also note that Squid converts AS numbers to networks only at startup or when you signal it to reconfigure. If the ISP updates its RADB entry, your cache won't know about the changes until you restart or reconfigure Squid.

Another problem is that the RADB server may be unreachable when your Squid process starts. If Squid can't contact the RADB server, it removes the AS entries from the access control configuration. The default server, *whois.ra.net*, may be too far away from many users to be reliable.

ACL Types

Now we can focus on the ACL types themselves. I present them here roughly in order of decreasing importance.

src

IP addresses are the most commonly used access control elements. Most sites use IP address controls to specify clients that are allowed to access Squid and those that aren't. The *src* type refers to client (source) IP addresses. That is, when an *src* ACL appears in an access list, Squid compares it to the IP address of the client issuing the request.

Normally you want to allow requests from hosts inside your network and block all others. For example, if your organization is using the 192.168.0.0 subnet, you can use an ACL like this:

```
acl MyNetwork src 192.168.0.0
```

If you have many subnets, you can list them all on the same *acl* line:

```
acl MyNetwork src 192.168.0.0 10.0.1.0/24 10.0.5.0/24 172.16.0.0/12
```

Squid has a number of other ACL types that check the client's address. The *srcdomain* type compares the client's fully qualified domain name. It requires a reverse DNS lookup, which may add some delay to processing the request. The *srcdomain_regex* ACL is similar, but it allows you to use a regular expression to compare domain names. Finally, the *src_as* type compares the client's AS number.

dst

The *dst* type refers to origin server (destination) IP addresses. Among other things, you can use this to prevent some or all of your users from visiting certain web sites. However, you need to be a little careful with the *dst* ACL. Most of the requests received by Squid have origin server hostnames. For example:

```
GET http://www.web-cache.com/ HTTP/1.0
```

Here, *www.web-cache.com* is the hostname. When an access list rule includes a *dst* element, Squid must find the IP addresses for the hostname. If Squid's IP cache contains a valid entry for the hostname, the ACL is checked immediately. Otherwise, Squid postpones request processing while the DNS lookup is in progress. This can add significant delay to some requests. To avoid those delays, you should use the *dst-domain* ACL type (instead of *dst*) whenever possible.*

Here is a simple *dst* ACL example:

```
acl AdServers dst 1.2.3.0/24
```

Note that one problem with *dst* ACLs is that the origin server you are trying to allow or deny may change its IP address. If you don't notice the change, you won't bother to update *squid.conf*. You can put a hostname on the *acl* line, but that adds some delay at startup. If you need many hostnames in ACLs, you may want to preprocess the configuration file and turn the hostnames into IP addresses.

myip

The *myip* type refers to the IP address where clients connect to Squid. This is what you see under the *Local Address* column when you run `netstat -n` on the Squid box. Most Squid installations don't use this type. Usually, all clients connect to the same IP address, so this ACL element is useful only on systems that have more than one IP address.

To understand how *myip* may be useful, consider a simple company local area network with two subnets. All users on subnet-1 are programmers and engineers. Subnet-2 consists of accounting, marketing, and other administrative departments. The system on which Squid runs has three network interfaces: one on subnet-1, one on subnet-2, and the third connecting to the outbound Internet connection (see Figure 6-1).

When properly configured, all users on subnet-1 connect to Squid's IP address on that subnet, and similarly, all subnet-2 users connect to Squid's second IP address.

* Apart from access controls, Squid only needs an origin server's IP address when establishing a connection to that server. DNS lookups normally occur much later in request processing. If the HTTP request results in a cache hit, Squid doesn't need to know the server's address. Additionally, Squid doesn't need IP addresses for cache misses that are forwarded to a neighbor cache.

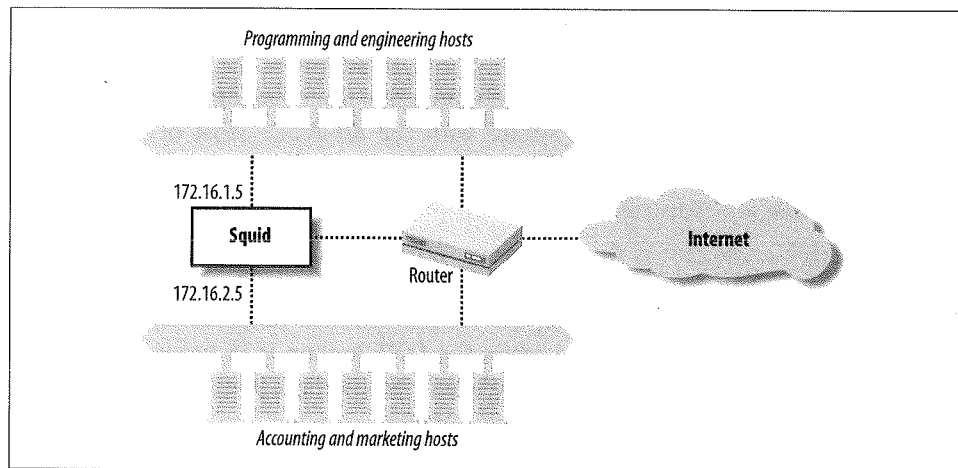


Figure 6-1. An application of the myip ACL

You can use this to give the technical staff on subnet-1 full access, while limiting the administrative staff to only work-related web sites.

The ACLs might look like this:

```

acl Eng myip 172.16.1.5
acl Admin myip 172.16.2.5

```

Note, however, that with this scheme you must take special measures to prevent users on one subnet from connecting to Squid's address on the other subnet. Otherwise, clever users on the accounting and marketing subnet can connect through the programming and engineering subnet and bypass your restrictions.

dstdomain

In some cases, you're likely to find that name-based access controls make a lot of sense. You can use them to block access to certain sites, to control how Squid forwards requests and to make some responses uncacheable. The *dstdomain* type is very useful because it checks the hostname in requested URLs.

First, however, I want to clarify the difference between the following two lines:

```

acl A dst www.squid-cache.org
acl B dstdomain www.squid-cache.org

```

A is really an IP address ACL. When Squid parses the configuration file, it looks up the IP address for *www.squid-cache.org* and stores the address in memory. It doesn't store the name. If the IP address for *www.squid-cache.org* changes while Squid is running, Squid continues using the old address.

The *dstdomain* ACL, on the other hand, is stored as a domain name (i.e., a string), not as an IP address. When Squid checks ACL B, it uses string comparison functions

on the hostname part of the URL. In this case, it doesn't really matter if the *www.squid-cache.org* IP changes while Squid is running.

The primary problem with *dstdomain* ACLs is that some URLs have IP addresses instead of hostnames. If your goal is to block access to certain sites with *dstdomain* ACLs, savvy users can simply look up the site's IP address manually and insert it into the URL. For example, these two URLs bring up the same page:

```
http://www.squid-cache.org/docs/FAQ/  
http://206.168.0.9/docs/FAQ/
```

The first can be easily matched with *dstdomain* ACLs, but the second can't. Thus, if you elect to rely on *dstdomain* ACLs, you may want to also block all requests that use an IP address instead of a hostname. See the later section "Denying Requests with IP Addresses" for an example.

srcdomain

The *srcdomain* ACL is somewhat tricky as well. It requires a so-called reverse DNS lookup on each client's IP address. Technically, Squid requests a DNS PTR record for the address. The answer—a fully qualified domain name (FQDN)—is what Squid compares to the ACL value. (Refer to O'Reilly's *DNS and BIND* for more information about DNS PTR records.)

As with *dst* ACLs, FQDN lookups are a potential source of significant delay. The request is postponed until the FQDN answer comes back. FQDN answers are cached, so the *srcdomain* lookup delay usually occurs only for the client's first request.

Unfortunately, *srcdomain* lookups sometimes don't work. Many organizations fail to keep their reverse lookup databases current. If an address doesn't have a PTR record, the ACL check fails. In some cases, requests may be postponed for a very long time (e.g., two minutes) until the DNS lookup times out. If you choose to use the *srcdomain* ACL, make sure that your own DNS *in-addr.arpa* zones are properly configured and working. Assuming that they are, you can use an ACL like this:

```
acl LocalHosts srcdomain .users.example.com
```

port

Most likely, you'll want to use the *port* ACL to limit access to certain origin server port numbers. As I'll explain shortly, Squid really shouldn't connect to certain services, such as email and IRC servers. The *port* ACL allows you to define individual ports, and port ranges. Here is an example:

```
acl HTTPports port 80 8000-8010 8080
```

HTTP is similar in design to other protocols, such as SMTP. This means that clever users can trick Squid into relaying email messages to an SMTP server. Email relays are one of the primary reasons we must deal with a daily deluge of spam.

Historically, spam relays have been actual mail servers. Recently, however, more and more spammers are using open HTTP proxies to hide their tracks. You definitely don't want your Squid cache to be used as a spam relay. If it is, your IP address is likely to end up on one of the many mail-relay blacklists (MAPS, ORDB, spamhaus, etc.). In addition to email, there are a number of other TCP/IP services that Squid shouldn't normally communicate with. These include IRC, Telnet, DNS, POP, and NNTP. Your policy regarding port numbers should be either to deny the known-to-be-dangerous ports and allow the rest, or to allow the known-to-be-safe ports and deny the rest.

My preference is to be conservative and allow only the safe ports. The default *squid.conf* includes the following `Safe_ports` ACL:

```
acl Safe_ports port 80      # http
acl Safe_ports port 21      # ftp
acl Safe_ports port 443 563 # https, snews
acl Safe_ports port 70      # gopher
acl Safe_ports port 210     # wais
acl Safe_ports port 1025-65535 # unregistered ports
acl Safe_ports port 280     # http-mgmt
acl Safe_ports port 488     # gss-http
acl Safe_ports port 591     # filemaker
acl Safe_ports port 777     # multiling http
```

```
http_access deny !Safe_ports
```

This is a sensible approach. It allows users to connect to any nonprivileged port (1025–65535), but only specific ports in the privileged range. If one of your users tries to request a URL, such as <http://www.lrrr.org:123/>, Squid returns an access denied error message. In some cases, you may need to add additional port numbers to the `Safe_ports` ACL to keep your users happy.

A more liberal approach is to deny access to certain ports that are known to be particularly dangerous. The Squid FAQ includes an example of this:

```
acl Dangerous_ports 7 9 19 22 23 25 53 109 110 119

http_access deny Dangerous_ports
```

One drawback to the *Dangerous_ports* approach is that Squid ends up searching the entire list for almost every request. This places a little extra burden on your CPU. Most likely, 99% of the requests reaching Squid are for port 80, which doesn't appear in the *Dangerous_ports* list. The list is searched for all of these requests without resulting in a match. However, integer comparison is a fast operation and should not significantly impact performance.

myport

Squid also has a *myport* ACL. Whereas the *port* ACL refers to the origin server port number, *myport* refers to the port where Squid receives client requests. Squid listens on different port numbers if you specify more than one with the *http_port* directive.

The *myport* ACL is particularly useful if you use Squid as an HTTP accelerator for your web site and as a proxy for your users. You can accept the accelerator requests on port 80 and the proxy requests on port 3128. You probably want the world to access the accelerator, but only your users should access Squid as a proxy. Your ACLs may look something like this:

```
acl AccelPort myport 80
acl ProxyPort myport 3128
acl MyNet src 172.16.0.0/22

http_access allow AccelPort      # anyone
http_access allow ProxyPort MyNet # only my users
http_access deny ProxyPort       # deny others
```

method

The *method* ACL refers to the HTTP request method. GET is typically the most common method, followed by POST, PUT, and others. This example demonstrates how to use the *method* ACL:

```
acl Uploads method PUT POST
```

Squid knows about the following standard HTTP methods: GET, POST, PUT, HEAD, CONNECT, TRACE, OPTIONS, and DELETE. In addition, Squid knows about the following methods from the WEBDAV specification, RFC 2518: PROPFIND, PROPPATCH, MKCOL, COPY, MOVE, LOCK, UNLOCK.* Certain Microsoft products use nonstandard WEBDAV methods, so Squid knows about them as well: BMOVE, BDELETE, BPROPFIND. Finally, you can configure Squid to understand additional request methods with the *extension_methods* directive. See Appendix A.

Note that the CONNECT method is special in a number of ways. It is the method used for tunneling certain requests through HTTP proxies (see also RFC 2817: Upgrading to TLS Within HTTP/1.1). Be especially careful with the CONNECT method and remote server port numbers. As I talked about in the previous section, you don't want Squid to connect to certain remote services. You should limit the CONNECT method to only the HTTPS/SSL and perhaps NNTPS ports (443 and 563, respectively). The default *squid.conf* does this:

```
acl CONNECT method CONNECT
acl SSL_ports 443 563
```

* For the RFC database, visit <http://www.rfc-editor.org/rfc.html>.

```
http_access allow CONNECT SSL_ports
http_access deny CONNECT
```

With this configuration, Squid only allows tunneled requests to ports 443 (HTTPS/SSL) and 563 (NNTPS). CONNECT method requests to all other ports are denied.

PURGE is another special request method. It is specific to Squid and not defined in any of the RFCs. It provides a way for the administrator to forcibly remove cached objects. Since this method is somewhat dangerous, Squid denies PURGE requests by default, unless you define an ACL that references the method. Otherwise, anyone with access to the cache may be able to remove any cached object. I recommend allowing PURGE from *localhost* only:

```
acl Purge method PURGE
acl Localhost src 127.0.0.1
http_access allow Purge Localhost
http_access deny Purge
```

See Chapter 7 for more information on removing objects from Squid's cache.

proto

This type refers to a URI's access (or transfer) protocol. Valid values are the following: http, https (same as HTTP/TLS), ftp, gopher, urn, whois, and cache_object. In other words, these are the URL scheme names (RFC 1738 terminology) supported by Squid. For example, suppose that you want to deny all FTP requests. You can use the following directives:

```
acl FTP proto FTP
http_access deny FTP
```

The *cache_object* scheme is a feature specific to Squid. It is used to access Squid's cache management interface, which I'll talk about in Chapter 14. Unfortunately, it's not a very good name, and it should probably be changed.

The default *squid.conf* file has a couple of lines that restrict cache manager access:

```
acl Manager proto cache_object
acl Localhost src 127.0.0.1

http_access allow Manager Localhost
http_access deny Manager
```

These configuration lines allow cache-manager requests only when they come from the localhost address. All other cache-manager requests are denied. This means that any user with an account on the Squid machine can access the potentially sensitive cache-manager information. You may want to modify the cache-manager access controls or protect certain pages with passwords. I'll talk about that in Chapter 14.

time

The *time* ACL allows you to control access based on the time of day and the day of the week. The syntax is somewhat cryptic:

```
acl name [days] [h1:m1-h2:m2]
```

You can specify days of the week, starting and stopping times, or both. Days are specified by the single-letter codes shown in Table 6-2. Times are specified in 24-hour format. The starting time must be less than the ending time, which makes it awkward to write *time* ACLs that span “midnights.”

Table 6-2. Day codes for the time ACL

Code	Day
S	Sunday
M	Monday
T	Tuesday
W	Wednesday
H	Thursday
F	Friday
A	Saturday
D	All weekdays (M-F)



Days and times are interpreted with the `localtime()` function, which takes into account your local time zone and daylight savings time settings. Make sure that your computer knows what time zone it is in! You'll also want to make sure that your clock is synchronized to the correct time.

To specify a *time* ACL that matches your weekday working hours, you can write:

```
acl Working_hours MTWHF 08:00-17:00
```

or:

```
acl Working_hours D 08:00-17:00
```

Let's look at a trickier example. Perhaps you're an ISP that relaxes access during off-peak hours, say 8 P.M. to 4 A.M. Since this time spans midnight, you can't write “20:00-04:00.” Instead you'll need either to split this into two ACLs or define the peak hours and use negation. For example:

```
acl Offpeak1 20:00-23:59
acl Offpeak2 00:00-04:00
http_access allow Offpeak1 ...
http_access allow Offpeak2 ...
```

Alternatively, you can do it like this:

```
acl Peak 04:00-20:00
http_access allow !Peak ...
```

Although Squid allows it, you probably shouldn't put more than one day list and time range on a single *time* ACL line. The parser isn't always smart enough to figure out what you want. For example, if you enter this:

```
acl Blah time M 08:00-10:00 W 09:00-11:00
```

what you really end up with is this:

```
acl Blah time MW 09:00-11:00
```

The parser ORs weekdays together and uses only the last time range. It does work, however, if you write it like this, on two separate lines:

```
acl Blah time M 08:00-10:00
acl Blah time W 09:00-11:00
```

ident

The *ident* ACL matches usernames returned by the ident protocol. This is a simple protocol, that's documented in RFC 1413. It works something like this:

1. A user-agent (client) establishes a TCP connection to Squid.
2. Squid connects to the ident port (113) on the client's system.
3. Squid writes a line containing the two TCP port numbers of the client's first connection. The Squid-side port number is probably 3128 (or whatever you configured in *squid.conf*). The client-side port is more or less random.
4. The client's ident server writes back the username belonging to the process that opened the first connection.
5. Squid records the username for access control purposes and for logging in *access.log*.

When Squid encounters an *ident* ACL for a particular request, that request is postponed until the ident lookup is complete. Thus, the *ident* ACL may add some significant delays to your users' requests.

We recommend using the *ident* ACL only on local area networks and only if all or most of the client workstations run the ident server. If Squid and the client workstations are connected to a LAN with low latency, the *ident* ACL can work well. Using *ident* for clients connecting over WAN links is likely to frustrate both you and your users.

The ident protocol isn't very secure. Savvy users will be able to replace their normal ident server with a fake server that returns any username they select. For example, if I know that connections from the user administrator are always allowed, I can write a simple program that answers every ident request with that username.



You can't use *ident* ACLs with interception caching (see Chapter 9). When Squid is configured for interception caching, the operating system pretends that it is the origin server. This means that the local socket address for intercepted TCP connections has the origin server's IP address. If you run `netstat -n` on Squid, you'll see a lot of foreign IP addresses in the *Local Address* column. When Squid makes an *ident* query, it creates a new TCP socket and binds the local endpoint to the same IP address as the local end of the client's TCP connection. Since the local address isn't really local (it's some far away origin server's IP address), the `bind()` system call fails. Squid handles this as a failed *ident* query.

Note that Squid also has a feature to perform “lazy” *ident* lookups on clients. In this case, requests aren't delayed while waiting for the *ident* query. Squid logs the *ident* information if it is available by the time the HTTP request is complete. You can enable this feature with the *ident_lookup_access* directive, which I'll discuss later in this chapter.

proxy_auth

Squid has a powerful, and somewhat confusing, set of features to support HTTP proxy authentication. With proxy authentication, the client's HTTP request includes a header containing authentication credentials. Usually, this is simply a username and password. Squid decodes the credential information and then queries an external authentication process to find out if the credentials are valid.

Squid currently supports three techniques for receiving user credentials: the HTTP Basic protocol, Digest authentication protocol, and NTLM. Basic authentication has been around for a long time. By today's standards, it is a very insecure technique. Usernames and passwords are sent together, essentially in cleartext. Digest authentication is more secure, but also more complicated. Both Basic and Digest authentication are documented in RFC 2617. NTLM also has better security than Basic authentication. However, it is a proprietary protocol developed by Microsoft. A handful of Squid developers have essentially reverse-engineered it.

In order to use proxy authentication, you must also configure Squid to spawn a number of external helper processes. The Squid source code includes some programs that authenticate against a number of standard databases, including LDAP, NTLM, NCSA-style password files, and the standard Unix password database. The *auth_param* directive controls the configuration of all helper programs. I'll go through it in detail in Chapter 12.

The *auth_param* directive and *proxy_auth* ACL is one of the few cases where their order in the configuration file is important. You must define at least one authentication helper (with *auth_param*) before any *proxy_auth* ACLs. If you don't, Squid

prints an error message and ignores the *proxy_auth* ACLs. This isn't a fatal error, so Squid may start anyway, and all your users' requests may be denied.

The *proxy_auth* ACL takes usernames as values. However, most installations simply use the special value *REQUIRED*:

```
auth_param ...
acl Auth1 proxy_auth REQUIRED
```

In this case, any request with valid credentials matches the ACL. If you need fine-grained control, you can specify individual usernames:

```
auth_param ...
acl Auth1 proxy_auth allan bob charlie
acl Auth2 proxy_auth dave eric frank
```



Proxy authentication doesn't work with HTTP interception because the user-agent doesn't realize it's talking to a proxy rather than the origin server. The user-agent doesn't know that it should send a Proxy-Authorization header in its requests. See Chapter 9 for additional details.

src_as

This type checks that the client (source) IP address belongs to a specific AS number. (See the earlier section "Autonomous system numbers" for information on how Squid maps AS numbers to IP addresses.) As an example, consider the fictitious ISP that uses AS 64222 and advertises the 10.0.0.0/8, 172.16.0.0/12, and 192.168.0.0/16 networks. You can write an ACL like this, which allows requests from any host in the ISP's address space:

```
acl TheISP src 10.0.0.0/8
acl TheISP src 172.16.0.0/12
acl TheISP src 192.168.0.0/16
http_access allow TheISP
```

Alternatively, you can write it like this:

```
acl TheISP src_as 64222
http_access allow TheISP
```

Not only is the second form shorter, it also means that if the ISP adds more networks, you won't have to update your ACL configuration.

dst_as

The *dst_as* ACL is often used with the *cache_peer_access* directive. In this way, Squid can forward cache misses in a manner consistent with IP routing. Consider an ISP that exchanges routes with a few other ISPs. Each ISP operates their own caching proxy, and these proxies can forward requests to each other. Ideally, ISP A forwards

cache misses for servers on ISP B's network to ISP B's caching proxy. An easy way to do this is with AS ACLs and the *cache_peer_access* directive:

```
acl ISP-B-AS dst_as 64222
acl ISP-C-AS dst_as 64333
cache_peer proxy.isp-b.net parent 3128 3130
cache_peer proxy.isp-c.net parent 3128 3130
cache_peer_access proxy.isb-b.net allow ISP-B-AS
cache_peer_access proxy.isb-c.net allow ISP-C-AS
```

These access controls make sure that the only requests sent to the two ISPs are for their own origin servers. I'll talk further about cache cooperation in Chapter 10.

snmp_community

The *snmp_community* ACL is meaningful only for SNMP queries, which are controlled by the *snmp_access* directive. For example, you might write:

```
acl OurCommunityName snmp_community hIgHsEcUrItY
acl All src 0/0
snmp_access allow OurCommunityName
snmp_access deny All
```

In this case, an SNMP query is allowed only if the community name is set to hIgHsEcUrItY.

maxconn

The *maxconn* ACL refers to the number of simultaneous connections from a client's IP address. Some Squid administrators find this a useful way to prevent users from abusing the proxy or consuming too many resources.

The *maxconn* ACL matches a request when that request exceeds the number you specify. For this reason, you should use *maxconn* ACLs only in *deny* rules. Consider this example:

```
acl OverConnLimit maxconn 4
http_access deny OverConnLimit
```

In this case, Squid allows up to four connections at once from each IP address. When a client makes the fifth connection, the *OverConnLimit* ACL is matched, and the *http_access* rule denies the request.

The *maxconn* ACL feature relies on Squid's client database. This database keeps a small data structure in memory for each client IP address. If you have a lot of clients, this database may consume a significant amount of memory. You can disable the client database in the configuration file with the *client_db* directive. However, if you disable the client database, the *maxconn* ACL will no longer work.

arp

The *arp* ACL is used to check the Media Access Control (MAC) address (typically Ethernet) of cache clients. The Address Resolution Protocol (ARP) is the way that hosts find the MAC address corresponding to an IP address. This feature came about when some university students discovered that, under Microsoft Windows, they could set a system's IP address to any value. Thus, they were able to circumvent Squid's address-based controls. To escalate this arms race, a savvy system administrator gave Squid the ability to check the client's Ethernet addresses.

Unfortunately, this feature uses nonportable code. If you use Solaris or Linux, you should be able to use *arp* ACLs. If not, you're out of luck. The best way to find out is to add the `--enable-arp-acl` option when you run `./configure`.

The *arp* ACL feature contains another important limitation. ARP is a datalink layer protocol. It works only for hosts on the same subnet as Squid. You can't easily discover the MAC address of a host on a different subnet. If you have routers between Squid and your users, you probably can't use *arp* ACLs.

Now that you know when not to use them, let's see how *arp* ACLs actually look. The values are Ethernet addresses, as you would see in `ifconfig` and `arp` output. For example:

```
acl WinBoxes arp 00:00:21:55:ed:22
acl WinBoxes arp 00:00:21:ff:55:38
```

srcdom_regex

The *srcdom_regex* ACL allows you to use regular expression matching on client domain names. This is similar to the *srcdomain* ACL, which uses modified substring matching. The same caveats apply here: some client addresses don't resolve back to domain names. As an example, the following ACL matches hostnames that begin with `dhcp`:

```
acl DHCPUser srcdom_regex -i ^dhcp
```

Because of the leading `^` symbol, this ACL matches the hostname `dhcp12.example.com`, but not `host12.dhcp.example.com`.

dstdom_regex

The *dstdom_regex* ACL is obviously similar, except that it applies to origin server names. The issues with *dstdomain* are relevant here, too. The following example matches hostnames that begin with `www`:

```
acl WebSite dstdom_regex -i ^www\.
```

Here is another useful regular expression that matches IP addresses given in URL hostnames:

```
acl IPAddr dstdom_regex [0-9]$
```

This works because Squid requires URL hostnames to be fully qualified. Since none of the global top-level domains end with a digit, this ACL matches only IP addresses, which do end with a number.

url_regex

You can use the *url_regex* ACL to match any part of a requested URL, including the transfer protocol and origin server hostname. For example, this ACL matches MP3 files requested from FTP servers:

```
acl FTPMP3 url_regex -i ^ftp://.*\.mp3$
```

urlpath_regex

The *urlpath_regex* ACL is very similar to *url_regex*, except that the transfer protocol and hostname aren't included in the comparison. This makes certain types of checks much easier. For example, let's say you need to deny requests with *sex* in the URL, but still possibly allow requests that have *sex* in their hostname:

```
acl Sex urlpath_regex sex
```

As another example, let's say you want to provide special treatment for *cgi-bin* requests. You can catch some of them with this ACL:

```
acl CGI1 urlpath_regex ^/cgi-bin
```

Of course, CGI programs aren't necessarily kept under */cgi-bin/*, so you'd probably want to write additional ACLs to catch the others.

browser

Most HTTP requests include a User-Agent header. The value of this header is typically something strange like:

```
Mozilla/4.51 [en] (X11; I; Linux 2.2.5-15 i686)
```

The *browser* ACL performs regular expression matching on the value of the User-Agent header. For example, to deny requests that don't come from a Mozilla browser, you can use:

```
acl Mozilla browser Mozilla
http_access deny !Mozilla
```

Before using the *browser* ACL, be sure that you fully understand the User-Agent strings your cache receives. Some user-agents lie about their identity. Even Squid has a feature to rewrite User-agent headers in requests that it forwards. With browsers such as Opera and KDE's Konqueror, users can send different user-agent strings to different origin servers or omit them altogether.

req_mime_type

The *req_mime_type* ACL refers to the Content-Type header of the client's HTTP request. Content-Type headers usually appear only in requests with message bodies. POST and PUT requests might include the header, but GET requests don't. You might be able to use the *req_mime_type* ACL to detect certain file uploads and some types of HTTP tunneling requests.

The *req_mime_type* ACL values are regular expressions. To catch audio file types, you can use an ACL like this:

```
acl AudioFileUploads req_mime_type -i ^audio/
```

rep_mime_type

The *rep_mime_type* ACL refers to the Content-Type header of the origin server's HTTP response. It is really only meaningful when used in an *http_reply_access* rule. All other access control forms are based on aspects of the client's request. This one is based on the response.

If you want to try blocking Java code with Squid, you might use some access rules like this:

```
acl JavaDownload rep_mime_type application/x-java
http_reply_access deny JavaDownload
```

ident_regex

You saw the *ident* ACL earlier in this section. The *ident_regex* simply allows you to use regular expressions, instead of exact string matching on usernames returned by the ident protocol. For example, this ACL matches usernames that contain a digit:

```
acl NumberInName ident_regex [0-9]
```

proxy_auth_regex

As with *ident*, the *proxy_auth_regex* ACL allows you to use regular expressions on proxy authentication usernames. For example, this ACL matches *admin*, *administrator*, and *administrators*:

```
acl Admins proxy_auth_regex -i ^admin
```

External ACLs

Squid Version 2.5 introduces a new feature: *external ACLs*. You instruct Squid to send certain pieces of information to an external process. This helper process then tells Squid whether the given data is a match or not.

Squid comes with a number of external ACL helper programs; most determine whether or not the named user is a member of a particular group. See Chapter 12 for

descriptions of those programs and for information on how to write your own. For now, I'll explain how to define and utilize an external ACL type.

The *external_acl_type* directive defines a new external ACL type. Here's the general syntax:

```
external_acl_type type-name [options] format helper-command
```

type-name is a user-defined string. You'll also use it in an *acl* line to reference this particular helper.

Squid currently supports the following options:

ttl=n

The amount of time, in seconds, to cache the result for values that are a match. The default is 3600 seconds, or 1 hour.

negative_ttl=n

The amount of time, in seconds, to cache the result for values that aren't a match. The default is 3600 seconds, or 1 hour.

concurrency=n

The number of helper processes to spawn. The default is 5.

cache=n

The maximum number of results to cache. The default is 0, which doesn't limit the cache size.

format is one or more keywords that begin with the % character. Squid currently supports the following format tokens:

%LOGIN

The username, taken from proxy authentication credentials.

%IDENT

The username, taken from an RFC 1413 ident query.

%SRC

The IP address of the client.

%DST

The IP address of the origin server.

%PROTO

The transfer protocol (e.g., HTTP, FTP, etc.).

%PORT

The origin server TCP port number.

%METHOD

The HTTP request method.

`%{Header}`

The value of an HTTP request header; for example, `%{User-Agent}` causes Squid to send strings like this to the authenticator:

```
"Mozilla/4.0 (compatible; MSIE 6.0; Win32)"
```

`%{Hdr:member}`

Selects certain members of list-based HTTP headers, such as `Cache-Control`; for example, given this HTTP header:

```
X-Some-Header: foo=xyzyz, bar=plugh, foo=zoinks
```

and the token `%{X-Some-Header:foo}`, Squid sends this string to the external ACL process:

```
foo=xyzyz, foo=zoinks
```

`%{Hdr;;member}`

The same as `%{Hdr:member}`, except that the `;` character is the list separator. You can use any nonalphanumeric character as the separator.

`helper-command` is the command that Squid spawns for the helper. You may include command arguments here as well. For example, the entire command may be something like:

```
/usr/local/squid/libexec/my-acl-prog.pl -X -5 /usr/local/squid/etc/datafile
```

Putting all these together results in a long line. Squid's configuration file doesn't support the backslash line-continuation technique shown here, so remember that all these must go on a single line:

```
external_acl_type MyAcIType cache=100 %LOGIN %{User-Agent} \  
/usr/local/squid/libexec/my-acl-prog.pl -X -5 \  
/usr/local/squid/share/usernames \  
/usr/local/squid/share/useragents
```

Now that you know how to define an external ACL, the next step is to write an *acl* line that references it. This is relatively straightforward. The syntax is as follows:

```
acl acl-name external type-name [args ...]
```

Here is a simple example:

```
acl MyAcI external MyAcIType
```

Squid accepts any number of optional arguments following the *type-name*. These are sent to the helper program for each request, after the expanded tokens. See my description of the *unix_group* helper in Chapter 12 for an example of this feature.

Dealing with Long ACL Lists

ACL lists can sometimes be very long. Such lists are awkward to maintain inside the *squid.conf* file. Also, you may need to generate Squid ACL lists automatically from other sources. In these cases, you'll be happy to know that you can include ACL lists from external files. The syntax is as follows:

```
acl name "filename"
```

The double quotes here instruct Squid to open *filename* and assign its contents to the ACL. For example, instead of this:

```
acl Foo BadClients 1.2.3.4 1.2.3.5 1.2.3.6 1.2.3.7 1.2.3.9 ...
```

you can do this:

```
acl Foo BadClients "/usr/local/squid/etc/BadClients"
```

and put the IP addresses into the *BadClients* file:

```
1.2.3.4
1.2.3.5
1.2.3.6
1.2.3.7
1.2.3.9
...
```

Your file may include comments that begin with a # character. Note that each entry in the file must be on a separate line. Whereas a space character delimits values on an *acl* line, newlines are the delimiter for files containing ACL values.

How Squid Matches Access Control Elements

It is important to understand how Squid searches ACL elements for a match. When an ACL element has more than one value, any single value can cause a match. In other words, Squid uses OR logic when checking ACL element values. Squid stops searching when it finds the first value that causes a match. This means that you can reduce delays by placing likely matches at the beginning of a list.

Let's look at a specific example. Consider this ACL definition:

```
acl Simpsons ident Maggie Lisa Bart Marge Homer
```

When Squid encounters the *Simpsons* ACL in an access list, it performs the *ident* lookup. Let's see what happens when the user's *ident* server returns *Marge*. Squid's ACL code compares this value to *Maggie*, *Lisa*, and *Bart* before finding a match with *Marge*. At this point, the search terminates, and we say that the *Simpsons* ACL matches the request.

Actually, that's a bit of a lie. The *ident* ACL values aren't stored as an unordered list. Rather, they are stored as a splay tree. This means that Squid doesn't end up searching all the names in the event of a nonmatch. Searching a splay tree with N items requires $\log(N)$ comparisons. Many other ACL types use splay trees as well. The regular expression-based types, however, don't.

Since regular expressions can't be sorted, they are stored as linked lists. This makes them inefficient for large lists, especially for requests that don't match any of the regular expressions in the list. In an attempt to improve this situation, Squid moves a regular expression to the top of the list when a match occurs. In fact, due to the nature of the ACL matching code, Squid moves matched entries to the second

position in the list. Thus, commonly matched values naturally migrate to the top of the ACL list, which should reduce the number of comparisons.

Let's look at another simple example:

```
acl Schmever port 80-90 101 103 107 1 2 3 9999
```

This ACL is a match for a request to an origin server port between 80 and 90, and all the other individual listed port numbers. For a request to port 80, Squid matches the ACL by looking at the first value. For port 9999, all the other values are checked first. For a port number not listed, Squid checks every value before declaring the ACL isn't a match. As I've said before, you can optimize the ACL matching by placing the more common values first.

Access Control Rules

As I mentioned earlier, ACL elements are the first step in building access controls. The second step is the access control rules, where you combine elements to allow or deny certain actions. You've already seen some *http_access* rules in the preceding examples. Squid has a number of other access control lists:

http_access

This is your most important access list. It determines which client HTTP requests are allowed, and which are denied. If you get the *http_access* configuration wrong, your Squid cache may be vulnerable to attacks and abuse from people who shouldn't have access to it.

http_reply_access

The *http_reply_access* list is similar to *http_access*. The difference is that the former list is checked when Squid receives a *reply* from an origin server or upstream proxy. Most access controls are based on aspects of the client's request, in which case the *http_access* list is sufficient. However, some people prefer also to allow or deny requests based on the reply content type. Because Squid doesn't know the content type value until it receives the server's reply, this additional access list is necessary. See the later section "An *http_reply_access* Example" for more information.

icp_access

If your Squid cache is configured to serve ICP replies (see Chapter 10), you should use the *icp_access* list. In most cases, you'll want to allow ICP requests only from your neighbor caches.

no_cache

You can use the *no_cache* access list to tell Squid it must never store certain responses (on disk or in memory). This list is typically used in conjunction with *dst*, *dstdomain*, and *url_regex* ACLs.

The “no” in *no_cache* causes some confusion because of double negatives. A request that is denied by the *no_cache* list isn’t cached. In other words *no_cache deny ...* is the way to make something uncachable. See the later section “Preventing Cache Hits for Local Sites” for an example.

miss_access

The *miss_access* list is primarily useful for a Squid cache with sibling neighbors. It determines how Squid handles requests that are cache misses. This feature is necessary for Squid to enforce sibling relationships with its neighbors. See the later section “Preventing Abuse from Siblings” for an example.

redirector_access

This access list determines which requests are sent to one of the redirector processes (see Chapter 11). By default, all requests go through a redirector if you are using one. You can use the *redirector_access* list to prevent certain requests from being rewritten. This is particularly useful because a redirector receives less information about a particular request than does the access control system.

ident_lookup_access

The *ident_lookup_access* list is similar to *redirector_access*. It enables you to make “lazy” ident lookups for certain requests. Squid doesn’t issue ident queries by default. It does so only for requests that are allowed by the *ident_lookup_access* rules (or by an *ident* ACL).

always_direct

This access list affects how a Squid cache with neighbors forwards cache misses. Usually Squid tries to forward cache misses to a parent cache, and/or Squid uses ICP to locate cached responses in neighbors. However, when a request matches an *always_direct* rule, Squid forwards the request directly to the origin server.

With this list, matching an allow rule causes Squid to forward the request directly. See Chapter 10 for more information and an example.

never_direct

Not surprisingly, *never_direct* is the opposite of *always_direct*. Cache miss requests that match this list must be sent to a neighbor cache. This is particularly useful for proxies behind firewalls.

With this list, matching an allow rule causes Squid to forward the request to a neighbor. See Chapter 10 for more information and an example.

snmp_access

This access list applies to queries sent to Squid’s SNMP port. The ACLs that you can use with this list are *snmp_community* and *src*. You can also use *srcdomain*, *srcdom_regex*, and *src_as* if you really want to. See Chapter 14 for an example.

broken_posts

This access list affects the way that Squid handles certain POST requests. Some older user-agents are known to send an extra CRLF (carriage return and line-feed) at the end of the request body. That is, the message body is two bytes

longer than indicated by the Content-Length header. Even worse, some older HTTP servers actually rely on this incorrect behavior. When a request matches this access list, Squid emulates the buggy client and sends the extra CRLF characters.

Squid has a number of additional configuration directives that use ACL elements. Some of these used to be global settings that were modified to use ACLs to provide more flexibility.

cache_peer_access

This access list controls the HTTP requests and ICP/HTCP queries that are sent to a neighbor cache. See Chapter 10 for more information and examples.

reply_body_max_size

This access list restricts the maximum acceptable size of an HTTP reply body. See Appendix A for more information.

delay_access

This access rule list controls whether or not the delay pools are applied to the (cache miss) response for this request. See Appendix C.

tcp_outgoing_address

This access list binds server-side TCP connections to specific local IP addresses. See Appendix A.

tcp_outgoing_tos

This access list can set different TOS/Diffserv values in TCP connections to origin servers and neighbors. See Appendix A.

header_access

With this directive, you can configure Squid to remove certain HTTP headers from the requests that it forwards. For example, you might want to automatically filter out Cookie headers in requests sent to certain origin servers, such as *doubleclick.net*. See Appendix A.

header_replace

This directive allows you to replace, rather than just remove, the contents of HTTP headers. For example, you can set the User-Agent header to a bogus value to keep certain origin servers happy while still protecting your privacy. See Appendix A.

Access Rule Syntax

The syntax for an access control rule is as follows:

```
access_list allow|deny [!]ACLname ...
```

For example:

```
http_access allow MyClients
http_access deny !Safe_Ports
http_access allow GameSites AfterHours
```

When reading the configuration file, Squid makes only one pass through the access control lines. Thus, you must define the ACL elements (with an `acl` line) before referencing them in an access list. Furthermore, the order of the access list rules is very important. Incoming requests are checked in the same order that you write them. Placing the most common ACLs early in the list may reduce Squid's CPU usage.



For most of the access lists, the meaning of deny and allow are obvious. Some of them, however, aren't so intuitive. In particular, pay close attention when writing *always_direct*, *never_direct*, and *no_cache* rules. In the case of *always_direct*, an allow rule means that matching requests are forwarded directly to origin servers. An *always_direct* deny rule means that matching requests aren't forced to go directly to origin servers, but may still do so if, for example, all neighbor caches are unreachable. The *no_cache* rules are tricky as well. Here, you must use deny for requests that must not be cached.

How Squid Matches Access Rules

Recall that Squid uses OR logic when searching ACL elements. Any single value in an *acl* can cause a match.

It's the opposite for access rules, however. For *http_access* and the other rule sets, Squid uses AND logic. Consider this generic example:

```
access_list allow ACL1 ACL2 ACL3
```

For this rule to be a match, the request must match each of *ACL1*, *ACL2*, and *ACL3*. If any of those ACLs don't match the request, Squid stops searching this rule and proceeds to the next. Within a single rule, you can optimize rule searching by putting least-likely-to-match ACLs first. Consider this simple example:

```
acl A method http
acl B port 8080
http_access deny A B
```

This *http_access* rule is somewhat inefficient because the A ACL is more likely to be matched than B. It is better to reverse the order so that, in most cases, Squid only makes one ACL check, instead of two:

```
http_access deny B A
```

One mistake people commonly make is to write a rule that can never be true. For example:

```
acl A src 1.2.3.4
acl B src 5.6.7.8
http_access allow A B
```

This rule is never going to be true because a source IP address can't be equal to both 1.2.3.4 and 5.6.7.8 at the same time. Most likely, someone who writes a rule like that really means this:

```
acl A src 1.2.3.4 5.6.7.8
http_access allow A
```

As with the algorithm for matching the values of an ACL, when Squid finds a matching rule in an access list, the search terminates. If none of the access rules result in a match, the default action is the *opposite* of the last rule in the list. For example, consider this simple access configuration:

```
acl Bob ident bob
http_access allow Bob
```

Now if the user Mary makes a request, she is denied. The last (and only) rule in the list is an allow rule, and it doesn't match the username Mary. Thus, the default action is the opposite of allow, so the request is denied. Similarly, if the last entry is a deny rule, the default action is to allow the request. It is good practice always to end your access lists with explicit rules that either allow or deny all requests. To be perfectly clear, the previous example should be written this way:

```
acl All src 0/0
acl Bob ident bob
http_access allow Bob
http_access deny All
```

The src 0/0 ACL is an easy way to match each and every type of request.

Access List Style

Squid's access control syntax is very powerful. In most cases, you can probably think of two or more ways to accomplish the same thing. In general, you should put the more specific and restrictive access controls first. For example, rather than:

```
acl All src 0/0
acl Net1 src 1.2.3.0/24
acl Net2 src 1.2.4.0/24
acl Net3 src 1.2.5.0/24
acl Net4 src 1.2.6.0/24
acl WorkingHours time 08:00-17:00

http_access allow Net1 WorkingHours
http_access allow Net2 WorkingHours
http_access allow Net3 WorkingHours
http_access allow Net4
http_access deny All
```

you might find it easier to maintain and understand the access control configuration if you write it like this:

```
http_access allow Net4
http_access deny !WorkingHours
```

```
http_access allow Net1
http_access allow Net2
http_access allow Net3
http_access deny All
```

Whenever you have a rule with two or more ACL elements, it's always a good idea to follow it up with an opposite, more general rule. For example, the default Squid configuration denies cache manager requests that don't come from the localhost IP address. You might be tempted to write it like this:

```
acl CacheManager proto cache_object
acl Localhost src 127.0.0.1
http_access deny CacheManager !Localhost
```

However, the problem here is that you haven't yet allowed the cache manager requests that do come from localhost. Subsequent rules may cause the request to be denied anyway. These rules have this undesirable behavior:

```
acl CacheManager proto cache_object
acl Localhost src 127.0.0.1
acl MyNet 10.0.0.0/24
acl All src 0/0
http_access deny CacheManager !Localhost
http_access allow MyNet
http_access deny All
```

Since a request from localhost doesn't match MyNet, it gets denied. A better way to write the rules is like this:

```
http_access allow CacheManager localhost
http_access deny CacheManager
http_access allow MyNet
http_access deny All
```

Delayed Checks

Some ACLs can't be checked in one pass because the necessary information is unavailable. The *ident*, *dst*, *srcdomain*, and *proxy_auth* types fall into this category. When Squid encounters an ACL that can't be checked, it postpones the decision and issues a query for the necessary information (IP address, domain name, username, etc.). When the information is available, Squid checks the rules all over again, starting at the beginning of the list. It doesn't continue where the previous check left off. If possible, you may want to move these likely-to-be-delayed ACLs near the top of your rules to avoid unnecessary, repeated checks.

Because these delays are costly (in terms of time), Squid caches the information whenever possible. Ident lookups occur for each *connection*, rather than each request. This means that persistent HTTP connections can really benefit you in situations where you use ident queries. Hostnames and IP addresses are cached as specified by the DNS replies, unless you're using the older external *dnsserver* processes.

Proxy Authentication information is cached as I described previously in the “proxy_auth” section.

Slow and Fast Rule Checks

Internally, Squid considers some access rule checks *fast*, and others *slow*. The difference is whether or not Squid postpones its decision to wait for additional information. In other words, a slow check may be deferred while Squid asks for additional data, such as:

- A reverse DNS lookup: the hostname for a client’s IP address
- An RFC 1413 ident query: the username associated with a client’s TCP connection
- An authenticator: validating the user’s credentials
- A forward DNS lookup: the origin server’s IP address
- An external, user-defined ACL

Some access rules use fast checks out of necessity. For example, the *icp_access* rule is a fast check. It must be fast, to serve ICP queries quickly. Furthermore, certain ACL types, such as *proxy_auth*, are meaningless for ICP queries. The following access rules are fast checks:

- *header_access*
- *reply_body_max_size*
- *reply_access*
- *ident_lookup*
- *delay_access*
- *miss_access*
- *broken_posts*
- *icp_access*
- *cache_peer_access*
- *redirector_access*
- *snmp_access*

The following ACL types may require information from external sources (DNS, authenticators, etc.) and are thus incompatible with fast access rules:

- *srcdomain*, *dstdomain*, *srcdom_regex*, *dstdom_regex*
- *dst*, *dst_as*
- *proxy_auth*
- *ident*
- *external_acl_type*

This means, for example, that you can't reliably use an *ident* ACL in a *header_access* rule.

Common Scenarios

Because access controls can be complicated, this section contains a few examples. They demonstrate some of the common uses for access controls. You should be able to adapt them to your particular needs.

Allowing Local Clients Only

Almost every Squid installation should restrict access based on client IP addresses. This is one of the best ways to protect your system from abuses. The easiest way to do this is write an ACL that contains your IP address space and then allow HTTP requests for that ACL and deny all others:

```
acl All src 0/0
acl MyNetwork src 172.16.5.0/24 172.16.6.0/24

http_access allow MyNetwork
http_access deny All
```

Most likely, this access control configuration will be too simple, so you'll need to add more lines. Remember that the order of the *http_access* lines is important. Don't add anything after *deny All*. Instead, add the new rules before or after *allow MyNetwork* as necessary.

Blocking a Few Misbehaving Clients

For one reason or another, you may find it necessary to deny access for a particular client IP address. This can happen, for example, if an employee or student launches an aggressive web crawling agent that consumes too much bandwidth or other resources. Until you can stop the problem at the source, you can block the requests coming to Squid with this configuration:

```
acl All src 0/0
acl MyNetwork src 172.16.5.0/24 172.16.6.0/24
acl ProblemHost src 172.16.5.9

http_access deny ProblemHost
http_access allow MyNetwork
http_access deny All
```

Denying Pornography

Blocking access to certain content is a touchy subject. Often, the hardest part about using Squid to deny pornography is coming up with the list of sites that should be

blocked. You may want to maintain such a list yourself, or get one from somewhere else. The “Access Controls” section of the Squid FAQ has links to freely available lists.

The ACL syntax for using such a list depends on its contents. If the list contains regular expressions, you probably want something like this:

```
acl PornSites url_regex "/usr/local/squid/etc/pornlist"
http_access deny PornSites
```

On the other hand, if the list contains origin server hostnames, simply change *url_regex* to *dstdomain* in this example.

Restricting Usage During Working Hours

Some corporations like to restrict web usage during working hours, either to save bandwidth, or because policy forbids employees from doing certain things while working. The hardest part about this is differentiating between appropriate and inappropriate use of the Internet during these times. Unfortunately, I can’t help you with that. For this example, I’m assuming that you’ve somehow collected or acquired a list of web site domain names that are known to be inappropriate. The easy part is configuring Squid:

```
acl NotWorkRelated dstdomain "/usr/local/squid/etc/not-work-related-sites"
acl WorkingHours time D 08:00-17:30

http_access deny !WorkingHours NotWorkRelated
```

Notice that I’ve placed the *!WorkingHours* ACL first in the rule. The *dstdomain* ACL is expensive (comparing strings and traversing lists), but the *time* ACL is a simple inequality check.

Let’s take this a step further and understand how to combine something like this with the source address controls described previously. Here’s one way to do it:

```
acl All src 0/0
acl MyNetwork src 172.16.5.0/24 172.16.6.0/24
acl NotWorkRelated dstdomain "/usr/local/squid/etc/not-work-related-sites"
acl WorkingHours time D 08:00-17:30

http_access deny !WorkingHours NotWorkRelated
http_access allow MyNetwork
http_access deny All
```

This scheme works because it accomplishes our goal of denying certain requests during working hours and allowing requests only from your own network. However, it might be somewhat inefficient. Note that the *NotWorkRelated* ACL is searched for all requests, regardless of the source IP address. If that list is long, you’ll waste CPU resources by searching it for requests from outside your network. Thus, you may want to change the rules around somewhat:

```
http_access deny !MyNetwork
http_access deny !WorkingHours NotWorkRelated
http_access Allow All
```

Here we've delayed the most expensive check until the very end. Outsiders that may be trying to abuse Squid will not be wasting your CPU cycles.

Preventing Squid from Talking to Non-HTTP Servers

You need to minimize the chance that Squid can communicate with certain types of TCP/IP servers. For example, people should never be able to use your Squid cache to relay SMTP (email) traffic. I covered this previously when introducing the *port* ACL. However, it is such an important part of your access controls that I'm presenting it here as well.

First of all, you have to worry about the `CONNECT` request method. User agents use this method to tunnel TCP connections through an HTTP proxy. It was invented for HTTP/TLS (a.k.a. SSL) requests, and this remains the primary use for the `CONNECT` method. Some user-agents may also tunnel NNTP/TLS traffic through firewall proxies. All other uses should be rejected. Thus, you'll need an access list that allows `CONNECT` requests to HTTP/TLS and NNTP/TLS ports only.

Secondly, you should prevent Squid from connecting to certain services such as SMTP. You can either allow safe ports or deny dangerous ports. I'll give examples for both techniques.

Let's start with the rules present in the default *squid.conf* file:

```
acl Safe_ports port 80          # http
acl Safe_ports port 21          # ftp
acl Safe_ports port 443 563     # https, snews
acl Safe_ports port 70          # gopher
acl Safe_ports port 210         # wais
acl Safe_ports port 280         # http-mgmt
acl Safe_ports port 488         # gss-http
acl Safe_ports port 591         # filemaker
acl Safe_ports port 777         # multiling http
acl Safe_ports port 1025-65535  # unregistered ports

acl SSL_ports port 443 563
acl CONNECT method CONNECT

http_access deny !Safe_ports
http_access deny CONNECT !SSL_ports
<additional http_access lines as necessary...>
```

Our `Safe_ports` ACL lists all privileged ports (less than 1024) to which Squid may have valid reasons for connecting. It also lists the entire nonprivileged port range. Notice that the `Safe_ports` ACL includes the secure HTTP and NNTP ports (443 and 563) even though they also appear in the `SSL_ports` ACL. This is because the `Safe_ports` ACL is checked first in the rules. If you swap the order of the first two `http_`

access lines, you could probably remove 443 and 563 from the `Safe_ports` list, but it's hardly worth the trouble.

The other way to approach this is to list the privileged ports that are known to be unsafe:

```
acl Dangerous_ports 7 9 19 22 23 25 53 109 110 119
acl SSL_ports port 443 563
acl CONNECT method CONNECT

http_access deny Dangerous_ports
http_access deny CONNECT !SSL_ports
<additional http_access lines as necessary...>
```

Don't worry if you're not familiar with all these strange port numbers. You can find out what each one is for by reading the `/etc/services` file on a Unix system or by reading IANA's list of registered TCP/UDP port numbers at <http://www.iana.org/assignments/port-numbers>.

Giving Certain Users Special Access

Organizations that employ username-based access controls often need to give certain users special privileges. In this simple example, there are three elements: all authenticated users, the usernames of the administrators, and a list of pornographic web sites. Normal users aren't allowed to view pornography, but the admins have the dubious job of maintaining the list. They need to connect to all servers to verify whether or not a particular site should be placed in the pornography list. Here's how to accomplish the task:

```
auth_param basic program /usr/local/squid/libexec/ncsa_auth
                /usr/local/squid/etc/passwd

acl Authenticated proxy_auth REQUIRED
acl Admins proxy_auth Pat Jean Chris
acl Porn dstdomain "/usr/local/squid/etc/porn.domains"
acl All src 0/0

http_access allow Admins
http_access deny Porn
http_access allow Authenticated
http_access deny All
```

Let's examine how this all works. First, there are three ACL definitions. The *Authenticated* ACL matches any valid proxy authentication credentials. The *Admins* ACL matches valid credentials from users Pat, Jean, and Chris. The *Porn* ACL matches certain origin server hostnames found in the *porn.domains* file.

This example has four access control rules. The first checks only the *Admins* ACL and allows all requests from Pat, Jean, and Chris. For other users, Squid moves on to the next rule. According to the second rule, a request is denied if its origin server

hostname is in the *porn.domains* file. For requests that don't match the *Porn* ACL, Squid moves on to the third rule. Here, the request is allowed if it contains valid authentication credentials. The external authenticator (*ncsa_auth* in this case) is responsible for deciding whether or not the credentials are valid. If they aren't, the final rule applies, and the request is denied.

Note that the *ncsa_auth* authenticator isn't a requirement. You can use any of the numerous authentication helpers described in Chapter 12.

Preventing Abuse from Siblings

If you open up your cache to peer with other caches, you need to take additional precautions. Caches often use ICP to discover which objects are stored in their neighbors. You should accept ICP queries only from known and approved neighbors.

Furthermore, you can configure Squid to enforce a sibling relationship by using the *miss_access* rule list. Squid checks these rules only when forwarding cache misses, never cache hits. Thus, all requests must first pass the *http_access* rules before the *miss_access* list comes into play.

In this example, there are three separate ACLs. One is for the local users that connect directly to this cache. Another is for a child cache, which is allowed to forward requests that are cache misses. The third is a sibling cache, which must never forward a request that results in a cache miss. Here's how it all works:

```
alc All src 0/0
acl OurUsers src 172.16.5.0/24
acl ChildCache src 192.168.1.1
acl SiblingCache src 192.168.3.3

http_access allow OurUsers
http_access allow ChildCache
http_access allow SiblingCache
http_access deny All

miss_access deny SiblingCache

icp_access allow ChildCache
icp_access allow SiblingCache
icp_access deny All
```

Denying Requests with IP Addresses

As I mentioned in the “*dstdomain*” section, the *dstdomain* type is good for blocking access to specific origin servers. However, clever users might be able to get around the rule by replacing URL hostnames with their IP addresses. If you are desperate to stop such requests, you may want to block all requests that contain an IP address.

You can do so with a redirector (see Chapter 11) or with a semicomplicated *dstdom_regex* ACL like this:

```
acl IPForHostname dstdom_regex ^[0-9]+\.[0-9]+\.[0-9]+\.[0-9]+$
http_access deny IPForHostname
```

An *http_reply_access* Example

Recall that the response's content type is the only new information available when Squid checks the *http_reply_access* rules. Thus, you can keep the *http_reply_access* rules very simple. You need only check the *rep_mime_type* ACLs. For example, here's how you can deny responses with certain content types:

```
acl All src 0/0
acl Movies rep_mime_type video/mpeg
acl MP3s rep_mime_type audio/mpeg
http_reply_access deny Movies
http_reply_access deny MP3s
http_reply_access allow All
```



You don't need to repeat your *http_access* rules in the *http_reply_access* list. The *allow All* rule shown here doesn't mean that all requests to Squid are allowed. Any request that is denied by *http_access* never makes it to the stage where Squid checks the *http_reply_access* rules.

Preventing Cache Hits for Local Sites

If you have a number of origin servers on your network, you may want to configure Squid so that their responses are never cached. Because the servers are nearby, they don't benefit too much from cache hits. Additionally, it frees up storage space for other (far away) origin servers.

The first step is to define an ACL for the local servers. You might want to use an address-based ACL, such as *dst*:

```
acl LocalServers dst 172.17.1.0/24
```

If the servers don't live on a single subnet, you might find it easier to create a *dstdomain* ACL:

```
acl LocalServers dstdomain .example.com
```

Next, you simply deny caching of those servers with a *no_cache* access rule:

```
no_cache deny LocalServers
```



The *no_cache* rules don't prevent your clients from sending these requests to Squid. There is nothing you can configure in Squid to stop such requests from coming. Instead, you must configure the user-agents themselves.

If you add a *no_cache* rule after Squid has been running for a while, the cache may contain some objects that match the new rule. Prior to Squid Version 2.5, these previously cached objects might be returned as cache hits. Now, however, Squid purges any cached response for a request that matches a *no_cache* rule.

Testing Access Controls

As your access control configuration becomes longer, it also becomes more complicated. I strongly encourage you to test your access controls before turning them loose on a production server. Of course, the first thing you should do is make sure that Squid can correctly parse your configuration file. Use the `-k parse` feature for this:

```
% squid -k parse
```

To further test your access controls, you may need to set up a fake Squid installation. One easy way to do that is compile another copy of the Squid source code with a different *\$prefix* location. For example:

```
% tar xzvf squid-2.5.STABLE4.tar.gz
% cd squid-2.5.STABLE4
% ./configure --prefix=/tmp/squid ...
% make && make install
```

After installing, you need to edit the new *squid.conf* file and change a few directives. Change *http_port* if Squid is already running on the default port. For simple testing, create a single, small cache directory like this:

```
cache_dir ufs /tmp/squid/cache 100 4 4
```

If you don't want to recompile Squid again, you can also just create a new configuration file. The drawback to this approach is that you'll need to set all the log-file pathnames to the temporary location so that you don't overwrite the real files.

You can easily test some access controls with the *squidclient* program. For example, if you have a rule that depends on the origin server hostname (*dstdomain* ACL), or some part of the URL (*url_regex* or *urlpath_regex*), simply enter a URI that you would expect to be allowed or denied:

```
% squidclient -p 4128 http://blocked.host.name/blah/blah
```

or:

```
% squidclient -p 4128 http://some.host.name/blocked.ext
```

Certain aspects of the request are harder to control. If you have *src* ACLs that block requests from outside your network, you may need to actually test them from an external host. Testing *time* ACLs may be difficult unless you can change the clock on your system or stay awake long enough.

You can use *squidclient*'s `-H` option to set arbitrary request headers. For example, use the following if you need to test a *browser* ACL.

```
% squidclient -p 4128 http://www.host.name/blah \
-H 'User-Agent: Mozilla/5.0 (compatible; Konqueror/3)\r\n'
```

For more complicated request, with many headers, you may want to use the technique described in Chapter 16.

You might also consider developing a routine *cron* job that checks your ACLs for expected behavior and reports any anomalies. Here is a sample shell script to get you started:

```
#!/bin/sh
set -e

TESTHOST="www.squid-cache.org"

# make sure Squid is not proxying dangerous ports
#
ST=`squidclient 'http://$TESTHOST:25/' | head -1 | awk '{print $2}'`
if test "$ST" != 403 ; then
    echo "Squid did not block HTTP request to port 25"
fi

# make sure Squid requires user authentication
#
ST=`squidclient 'http://$TESTHOST/' | head -1 | awk '{print $2}'`
if test "$ST" != 407 ; then
    echo "Squid allowed request without proxy authentication"
fi

# make sure Squid denies requests from foreign IP addresses
# elsewhere we already created an alias 192.168.1.1 on one of
# the system interfaces
#
EXT_ADDR=192.168.1.1
ST=`squidclient -l $EXT_ADDR 'http://$TESTHOST/' | head -1 | awk '{print $2}'`
if test "$ST" != 403 ; then
    echo "Squid allowed request from external address $EXT_ADDR"
fi

exit 0
```

Exercises

- Define an ACL for each known type (*src*, *dst*, *ident*, etc.) and write a rule that uses all of them.
- Intentionally mistype the name of an ACL in one of your rules. Does **squid -k parse** catch the error? Does Squid start anyway?
- Write an *http_access* that uses slow ACLs, like *srcdomain* or *ident*. Time how long Squid takes to serve a request with and without the slow ACL checks.

Disk Cache Basics

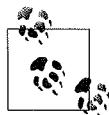
I'm going to talk a lot about disk storage and filesystems in this chapter. It is important to make sure you understand the difference between two related things: disk filesystems and Squid's storage schemes.

Filesystems are features of particular operating systems. Almost every Unix variant has an implementation of the Unix File System (UFS). It is also sometimes known as the Berkeley Fast File System (FFS). Linux's default filesystem is called *ext2fs*. Many operating systems also support newer filesystem technologies. These include names and acronyms such as *advfs*, *xfs*, and *reiserfs*.

Programs (such as Squid) interact with filesystems via a handful of system calls. These are functions such as `open()`, `close()`, `read()`, `write()`, `stat()`, and `unlink()`. The arguments to these system calls are either pathnames (strings) or file descriptors (integers). Filesystem implementation details are hidden from programs. They typically use internal data structures such as inodes, but Squid doesn't know about that.

Squid has a number of different storage schemes. The schemes have different properties and techniques for organizing and accessing cache data on the disk. Most of them use the filesystem interface system calls (e.g., `open()`, `write()`, etc.).

Squid has five different storage schemes: *ufs*, *aufs*, *diskd*, *coss*, and *null*. The first three use the same directory layout, and they are thus interchangeable. *coss* is an attempt to implement a new filesystem specifically optimized for Squid. *null* is a minimal implementation of the API: it doesn't actually read or write data to/from the disk.



Due to a poor choice of names, "UFS" might refer to either the Unix filesystem or the Squid storage scheme. To be clear here, I'll write the filesystem as UFS and the storage scheme as *ufs*.

The remainder of this chapter focuses on the *squid.conf* directives that control the disk cache. This includes replacement policies, object removal, and freshness

controls. For the most part, I'll only talk about the default storage scheme: *ufs*. We'll get to the alternative schemes and other tricks in the next chapter.

The `cache_dir` Directive

The `cache_dir` directive is one of the most important in *squid.conf*. It tells Squid where and how to store cache files on disk. The `cache_dir` directive takes the following arguments:

```
cache_dir scheme directory size L1 L2 [options]
```

Scheme

Squid supports a number of different storage schemes. The default (and original) is *ufs*. Depending on your operating system, you may be able to select other schemes. You must use the `--enable-storeio=LIST` option with `./configure` to compile the optional code for other storage schemes. I'll discuss *aufs*, *diskd*, *coss*, and *null* in Chapter 8. For now, I'll only talk about the *ufs* scheme, which is compatible with *aufs* and *diskd*.

Directory

The *directory* argument is a filesystem directory, under which Squid stores cached objects. Normally, a `cache_dir` corresponds to a whole filesystem or disk partition. It usually doesn't make sense to put more than one cache directory on a single filesystem partition. Furthermore, I also recommend putting only one cache directory on each physical disk drive. For example, if you have two unused hard drives, you might do something like this:

```
# newfs /dev/da1d
# newfs /dev/da2d
# mount /dev/da1d /cache0
# mount /dev/da2d /cache1
```

And then add these lines to *squid.conf*:

```
cache_dir ufs /cache0 7000 16 256
cache_dir ufs /cache1 7000 16 256
```

If you don't have any spare hard drives, you can, of course, use an existing filesystem partition. Select one with plenty of free space, perhaps */usr* or */var*, and create a new directory there. For example:

```
# mkdir /var/squidcache
```

Then add a line like this to *squid.conf*:

```
cache_dir ufs /var/squidcache 7000 16 256
```

Size

The third *cache_dir* argument specifies the size of the cache directory. This is an upper limit on the amount of disk space that Squid can use for the *cache_dir*. Calculating an appropriate value can be tricky. You lose some space to filesystem overheads, and you must leave enough free space for temporary files and *swap.state* logs (see Chapter 13). I recommend mounting the empty filesystem and running *df*:

```
% df -k
Filesystem 1K-blocks    Used Avail Capacity  Mounted on
/dev/da1d   3037766         8 2794737    0%  /cache0
/dev/da2d   3037766         8 2794737    0%  /cache1
```

Here you can see that the filesystem has about 2790 MB of available space. Remember that UFS reserves some “minfree” space, 8% in this case, which is why Squid can’t use the full 3040 MB in the filesystem.

You might be tempted just to put 2790 on the *cache_dir* line. You might even to get away with it if your cache isn’t very busy and if you rotate the log files often. To be safe, however, I recommend taking off another 10% or so. This extra space will be used by Squid’s *swap.state* file and temporary files.

Note that the *cache_swap_low* directive also affects how much space Squid uses. I’ll talk about the low and high watermarks in the “Disk Space Watermarks” section.

The bottom line is that you should initially be conservative about the size of your *cache_dir*. Start off with a low estimate and allow the cache to fill up. After Squid runs for a week or so with full cache directories, you’ll be in a good position to re-evaluate the size settings. If you have plenty of free space, feel free to increase the cache directory size in increments of a few percent.

Inodes

Inodes are fundamental building blocks of Unix filesystems. They contain information about disk files, such as permissions, ownership, size, and timestamps. If your filesystem runs out of inodes, you can’t create new files, even if it has space available. Running out of inodes is bad, so you may want to make sure you have enough before running Squid.

The programs that create new filesystems (e.g., *newfs* or *mkfs*) reserve some number of inodes based on the total size. These programs usually allow you to set the ratio of inodes to disk space. For example, see the *-i* option in the *newfs* and *mkfs* manpages. The ratio of disk space to inodes determines the mean file size the filesystem can support. Most Unix systems create one inode for each 4 KB, which is usually sufficient for Squid. Research shows that, for most caching proxies, the mean file size is about 10 KB. You may be able to get away with 8 KB per inode, but it is risky.

You can monitor your system's inode usage with `df -i`. For example:

```
% df -ik
Filesystem 1K-blocks    Used   Avail Capacity  iused   ifree %iused Mounted on
/dev/ad0s1a  197951    57114  125001    31%   1413  52345    3%    /
/dev/ad0s1f  5004533  2352120  2252051    51%  129175 1084263   11%   /usr
/dev/ad0s1e   396895    6786  358358     2%    205  99633    0%   /var
/dev/da0d    8533292  7222148  628481    92%  430894  539184   44%  /cache1
/dev/da1d    8533292  7181645  668984    91%  430272  539806   44%  /cache2
/dev/da2d    8533292  7198600  652029    92%  434726  535352   45%  /cache3
/dev/da3d    8533292  7208948  641681    92%  427866  542212   44%  /cache4
```

As long as the inode usage (%iused) is less than the space usage (Capacity), you're in good shape. Unfortunately, you can't add more inodes to an existing filesystem. If you find that you are running out of inodes, you need to stop Squid and recreate your filesystems. If you're not willing to do that, decrease the `cache_dir` size instead.

The relationship between disk space and process size

Squid's disk space usage directly affects its memory usage as well. Every object that exists on disk requires a small amount of memory. Squid uses the memory as an index to the on-disk data. If you add a new cache directory or otherwise increase the disk cache size, make sure that you also have enough free memory. Squid's performance degrades very quickly if its process size reaches or exceeds your system's physical memory capacity.

Every object in Squid's cache directories takes either 76 or 112 bytes of memory, depending on your system. The memory is allocated as *StoreEntry*, *MD5 Digest*, and *LRU policy node* structures. Small-pointer (i.e., 32-bit) systems, like those based on the Intel Pentium, take 76 bytes. On systems with CPUs that support 64-bit pointers, each object takes 112 bytes. You can find out how much memory these structures use on your system by viewing the Memory Utilization page of the cache manager (see Chapter 14).

Unfortunately, it is difficult to predict precisely how much additional memory is required for a given amount of disk space. It depends on the mean reply size, which typically fluctuates over time. Additionally, Squid uses memory for many other data structures and purposes. Don't assume that your estimates are, or will remain, correct. You should constantly monitor Squid's process size and consider shrinking the cache size if necessary.

L1 and L2

For the *ufs*, *aufs*, and *diskd* schemes, Squid creates a two-level directory tree underneath the cache directory. The L1 and L2 arguments specify the number of first- and second-level directories. The defaults are 16 and 256, respectively. Figure 7-1 shows the filesystem structure.

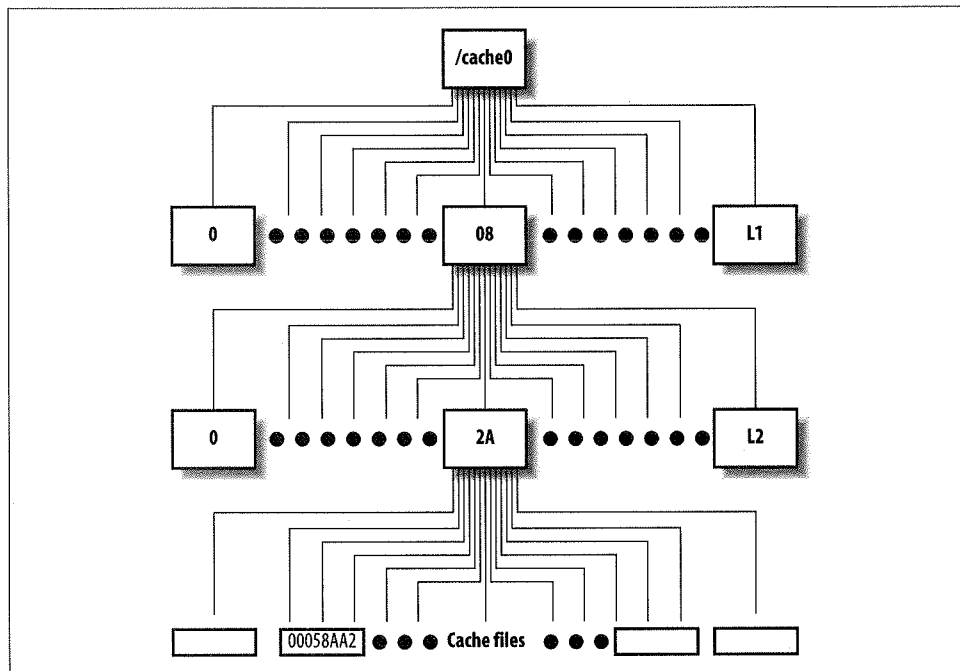


Figure 7-1. The cache directory structure for ufs-based storage schemes

Some people think that Squid performs better, or worse, depending on the particular values for L1 and L2. It seems to make sense, intuitively, that small directories can be searched faster than large ones. Thus, L1 and L2 should probably be large enough so that each L2 directory has no more than a few hundred files.

For example, let's say you have a cache directory that stores about 7000 MB. Given a mean file size of 10 KB, you can store about 700,000 files in this *cache_dir*. With 16 L1 and 256 L2 directories, there are 4096 total second-level directories. $700,000 \div 4096$ leaves about 170 files in each second-level directory.

The process of creating swap directories with `squid -z`, goes faster for smaller values of L1 and L2. Thus, if your cache size is really small, you may want to reduce the number of L1 and L2 directories.

Squid assigns each cache object a unique *file number*. This is a 32-bit integer that uniquely identifies files on disk. Squid uses a relatively simple algorithm for turning file numbers into pathnames. The algorithm uses L1 and L2 as parameters. Thus, if you change L1 and L2, you change the mapping from file number to pathname. Changing these parameters for a nonempty *cache_dir* makes the existing files inaccessible. You should never change L1 and L2 after the cache directory has become active.

Squid allocates file numbers within a cache directory sequentially. The file number-to-pathname algorithm (e.g., `storeUfsDirFullPath()`) is written so that each group of L2 files go into the same second-level directory. Squid does this to take advantage of locality of reference. This algorithm increases the probability that an HTML file and its embedded images are stored in the same second-level directory. Some people expect Squid to spread cache files evenly among the second-level directories. However, when the cache is initially filling, you'll find that only the first few directories contain any files. For example:

```
% cd /cache0; du -k
2164  ./00/00
2146  ./00/01
2689  ./00/02
1974  ./00/03
2201  ./00/04
2463  ./00/05
2724  ./00/06
3174  ./00/07
1144  ./00/08
1     ./00/09
1     ./00/0A
1     ./00/0B
...
```

This is perfectly normal and nothing to worry about.

Options

Squid has two scheme-independent *cache_dir* options: a read-only flag and a max-size value.

read-only

The read-only option instructs Squid to continue reading from the *cache_dir*, but to stop storing new objects there. It looks like this in *squid.conf*:

```
cache_dir ufs /cache0 7000 16 256 read-only
```

You might use this option if you want to migrate your cache storage from one disk to another. If you simply add one *cache_dir* and remove another, Squid's hit ratio decreases sharply. You can still get cache hits from the old location when it is read-only. After some time, you can remove the read-only cache directory from the configuration.

max-size

With this option, you can specify the maximum object size to be stored in the cache directory. For example:

```
cache_dir ufs /cache0 7000 16 256 max-size=1048576
```

Note that the value is in bytes. In most situations, you shouldn't need to add this option. If you do, try to put the *cache_dir* lines in order of increasing max-size.

Disk Space Watermarks

The *cache_swap_low* and *cache_swap_high* directives control the replacement of objects stored on disk. Their values are a percentage of the maximum cache size, which comes from the sum of all *cache_dir* sizes. For example:

```
cache_swap_low 90
cache_swap_high 95
```

As long as the total disk usage is below *cache_swap_low*, Squid doesn't remove cached objects. As the cache size increases, Squid becomes more aggressive about removing objects. Under steady-state conditions, you should find that disk usage stays relatively close to the *cache_swap_low* value. You can see the current disk usage by requesting the *storedir* page from the cache manager (see Chapter 14).

Note that changing *cache_swap_high* probably won't have a big impact on Squid's disk usage. In earlier versions of Squid, this parameter played a more important role; now, however, it doesn't.

Object Size Limits

You can control both the maximum and minimum size of cached objects. Responses larger than *maximum_object_size* aren't stored on disk. They are still proxied, however. The logic behind this directive is that you don't want a really big response to take up space better utilized by many small responses. The syntax is as follows:

```
maximum_object_size size-specification
```

Here are some examples:

```
maximum_object_size 100 KB
maximum_object_size 1 MB
maximum_object_size 12382 bytes
maximum_object_size 2 GB
```

Squid checks the response size in two different ways. If the reply includes a Content-Length header, Squid compares its value to the *maximum_object_size* value. If the content length is the larger of the two numbers, the object becomes immediately uncachable and never consumes any disk space.

Unfortunately, not every response has a Content-Length header. In this case, Squid writes the response to disk as data comes in from the origin server. Squid checks the object size again only when the response is complete. Thus, if the object's size reaches the *maximum_object_size* limit, it continues consuming disk space. Squid increments the total cache size only when it is done reading a response.

In other words, the active, or in-transit, objects don't contribute to the cache size value Squid maintains internally. This is good because it means Squid won't remove other objects in the cache, unless the object remains cachable and then contributes to the total cache size. However, it is also bad because Squid may run out of free disk space if the reply is very large. To reduce the chance of this happening, you should also use the *reply_body_max_size* directive. A response that reaches the *reply_body_max_size* limit is cut off immediately.

Squid also has a *minimum_object_size* directive. It allows you to place a lower limit on the size of cached objects. Responses smaller than this size aren't stored on disk or in memory. Note that this size is compared to the response's content length (i.e., the size of the reply body), which excludes the HTTP headers.

Allocating Objects to Cache Directories

When Squid wants to store a cachable response on disk, it calls a function that selects one of the cache directories. It then opens a disk file for writing on the selected directory. If, for some reason, the `open()` call fails, the response isn't stored. In this case, Squid doesn't try opening a disk file on one of the other cache directories.

Squid has two of these *cache_dir* selection algorithms. The default algorithm is called least-load; the alternative is round-robin.

The least-load algorithm, as the name implies, selects that cache directory that currently has the smallest workload. The notion of load depends on the underlying storage scheme. For the *aufs*, *cos*, and *diskd* schemes, the load is related to the number of pending operations. For *ufs*, the load is constant. For cases in which all *cache_dirs* have equal load, the algorithm uses free space and maximum object sizes as tie-breakers.

The selection algorithm also takes into account the *max-size* and *read-only* options. Squid skips a cache directory if it knows the object size is larger than the limit. It also always skips any read-only directories.

The round-robin algorithm also uses load measurements. It always selects the next cache directory in the list (subject to *max-size* and *read-only*), as long as its load is less than 100%.

Under some circumstances, Squid may fail to select a cache directory. This can happen if all *cache_dirs* are overloaded or if all have *max-size* limits less than the size of the object. In this case, Squid simply doesn't write the object to disk. You can use the cache manager to track the number of times Squid fails to select a cache directory. View the *store_io* page (see Chapter 14), and find the `create.select_fail` line.

Replacement Policies

The `cache_replacement_policy` directive controls the replacement policy for Squid's disk cache. Version 2.5 offers three different replacement policies: least recently used (LRU), greedy dual-size frequency (GDSF), and least frequently used with dynamic aging (LFUDA).

LRU is the default policy, not only for Squid, but for most other caching products as well. LRU is a popular choice because it is almost trivial to implement and provides very good performance. On 32-bit systems, LRU uses slightly less memory than the others (12 versus 16 bytes per object). On 64-bit systems, all policies use 24 bytes per object.

Over the years, many researchers have proposed alternatives to LRU. These other policies are typically designed to optimize a specific characteristic of the cache, such as response time, hit ratio, or byte hit ratio. While the research almost always shows an improvement, the results can be misleading. Some of the studies use unrealistically small cache sizes. Other studies show that as cache size increases, the choice of replacement policy becomes less important.

If you want to use the GDSF or LFUDA policies, you must pass the `--enable-removal-policies` option to the `./configure` script (see Chapter 3). Martin Arlitt and John Dille of HP Labs wrote the GDSF and LFUDA implementation for Squid. You can read their paper online at <http://www.hpl.hp.com/techreports/1999/HPL-1999-69.html>. My O'Reilly book, *Web Caching*, also talks about these algorithms.

The `cache_replacement_policy` directive is unique in an important way. Unlike most of the other `squid.conf` directives, the location of this one is significant. The `cache_replacement_policy` value is actually used when Squid parses a `cache_dir` directive. You can change the replacement policy for a `cache_dir` by setting the replacement policy beforehand. For example:

```
cache_replacement_policy lru
cache_dir ufs /cache0 2000 16 32
cache_dir ufs /cache1 2000 16 32
cache_replacement_policy heap GDSF
cache_dir ufs /cache2 2000 16 32
cache_dir ufs /cache3 2000 16 32
```

In this case, the first two cache directories use LRU replacement, and the second two use GDSF. This characteristic of the `replacement_policy` directive is important to keep in mind if you ever decide to use the `config` option of the cache manager (see Chapter 14). The cache manager outputs only one (the last) replacement policy value, and places it before all of the cache directories. For example, you may have these lines in `squid.conf`:

```
cache_replacement_policy heap GDSF
cache_dir ufs /tmp/cache1 10 4 4
```

```
cache_replacement_policy lru
cache_dir ufs /tmp/cache2 10 4 4
```

but when you select *config* from the cache manager, you get:

```
cache_replacement_policy lru
cache_dir ufs /tmp/cache1 10 4 4
cache_dir ufs /tmp/cache2 10 4 4
```

As you can see, the heap GDSF setting for the first cache directory has been lost.

Removing Cached Objects

At some point you may find it necessary to manually remove one or more objects from Squid's cache. This might happen if:

- One of your users complains about always receiving stale data.
- Your cache becomes “poisoned” with a forged response.
- Squid's cache index becomes corrupted after experiencing disk I/O errors or frequent crashes and restarts.
- You want to remove some large objects to free up room for new data.
- Squid was caching responses from local servers, and now you don't want it to.

Some of these problems can be solved by forcing a reload in a web browser. However, this doesn't always work. For example, some browsers display certain content types externally by launching another program; that program probably doesn't have a reload button or even know about caches.

You can always use the `squidclient` program to reload a cached object if necessary. Simply insert the `-r` option before the URI:

```
% squidclient -r http://www.lrrr.org/junk >/tmp/foo
```

If you happen to have a *refresh_pattern* directive with the `ignore-reload` option set, you and your users may be unable to force a validation of the cached response. In that case, you'll be better off purging the offending object or objects.

Removing Individual Objects

Squid accepts a custom request method for removing cached objects. The `PURGE` method isn't one of the official HTTP request methods. It is different from `DELETE`, which Squid forwards to an origin server. A `PURGE` request asks Squid to remove the object given in the URI. Squid returns either 200 (Ok) or 404 (Not Found).

The `PURGE` method is somewhat dangerous because it removes cached objects. Squid disables the `PURGE` method unless you define an ACL for it. Normally you should allow `PURGE` requests only from localhost and perhaps a small number of trusted hosts. The configuration may look like this:

```
acl AdminBoxes src 127.0.0.1 172.16.0.1 192.168.0.1
acl Purge method PURGE
http_access allow AdminBoxes Purge
http_access deny Purge
```

The `squidclient` program provides an easy way to generate PURGE requests. For example:

```
% squidclient -m PURGE http://www.lrrr.org/junk
```

Alternatively, you could use something else (such as a Perl script) to generate your own HTTP request. It can be very simple:

```
PURGE http://www.lrrr.org/junk HTTP/1.0
Accept: */*
```

Note that a URI alone doesn't uniquely identify a cached response. Squid also uses the original request method in the cache key. It may also use other request headers if the response contains a Vary header. When you issue a PURGE request, Squid looks for cached objects originally requested with the GET and HEAD methods. Furthermore, Squid also removes all variants of a response, unless you remove a specific variant by including the appropriate headers in the PURGE request. Squid removes only variants for GET and HEAD requests.

Removing a Group of Objects

Unfortunately, Squid doesn't provide a good mechanism for removing a bunch of objects at once. This often comes up when someone wants to remove all objects belonging to a certain origin server.

Squid lacks this feature for a couple of reasons. First, Squid would have to perform a linear search through all cached objects. This is CPU-intensive and takes a long time. While Squid is searching, your users can experience a performance degradation. Second, Squid keeps MD5s, rather than URIs, in memory. MD5s are one-way hashes, which means, for example, that you can't tell if a given MD5 hash was generated from a URI that contains the string "www.example.com." The only way to know is to recalculate the MD5 from the original URI and see if they match. Because Squid doesn't have the URI, it can't perform the calculation.

So what can you do?

You can use the data in *access.log* to get a list of URIs that might be in the cache. Then, feed them to `squidclient` or another utility to generate PURGE requests. For example:

```
% awk '{print $7}' /usr/local/squid/var/logs/access.log \
| grep www.example.com \
| xargs -n 1 squidclient -m PURGE
```

Removing All Objects

In extreme circumstances you may need to wipe out the entire cache, or at least one of the cache directories. First, you must make sure that Squid isn't running.

One of the easiest ways to make Squid forget about all cached objects is to overwrite the *swap.state* files. Note that you can't simply remove the *swap.state* files because Squid then scans the cache directories and opens all the object files. You also can't simply truncate *swap.state* to a zero-sized file. Instead, you should put a single byte there, like this:

```
# echo '' > /usr/local/squid/var/cache/swap.state
```

When Squid reads the *swap.state* file, it gets an error because the record that should be there is too short. The next read results in an end-of-file condition, and Squid completes the rebuild procedure without loading any object metadata.

Note that this technique doesn't remove the cache files from your disk. You've only tricked Squid into thinking that the cache is empty. As Squid runs, it adds new files to the cache and may overwrite the old files. In some cases, this might cause your disk to run out of free space. If that happens to you, you need to remove the old files before restarting Squid again.

One way to remove cache files is with `rm`. However, it often takes a very long time to remove all the files that Squid has created. To get Squid running faster, you can rename the cache directory, create a new one, start Squid, and remove the old one at the same time. For example:

```
# squid -k shutdown
# cd /usr/local/squid/var
# mv cache oldcache
# mkdir cache
# chown nobody:nobody cache
# squid -z
# squid -s
# rm -rf oldcache &
```

Another technique is to simply run `newfs` (or `mkfs`) on the cache filesystem. This works only if you have the *cache_dir* on its own disk partition.

refresh_pattern

The *refresh_pattern* directive controls the disk cache only indirectly. It helps Squid decide whether or not a given request can be a cache hit or must be treated as a miss. Liberal settings increase your cache hit ratio but also increase the chance that users receive a stale response. Conservative settings, on the other hand, decrease hit ratios and stale responses.



The *refresh_pattern* rules apply only to responses without an explicit expiration time. Origin servers can specify an expiration time with either the Expires header, or the Cache-Control: max-age directive.

You can put any number of *refresh_pattern* lines in the configuration file. Squid searches them in order for a regular expression match. When Squid finds a match, it uses the corresponding values to determine whether a cached response is fresh or stale. The *refresh_pattern* syntax is as follows:

```
refresh_pattern [-i] regexp min percent max [options]
```

For example:

```
refresh_pattern -i \.jpg$ 30 50% 4320 reload-into-ims
refresh_pattern -i \.png$ 30 50% 4320 reload-into-ims
refresh_pattern -i \.htm$ 0 20% 1440
refresh_pattern -i \.html$ 0 20% 1440
refresh_pattern -i . 5 25% 2880
```

The *regexp* parameter is a regular expression that is normally case-sensitive. You can make them case-insensitive with the *-i* option. Squid checks the *refresh_pattern* lines in order; it stops searching when one of the regular expression patterns matches the URI.

The *min* parameter is some number of minutes. It is, essentially, a lower bound on stale responses. A response can't be stale unless its time in the cache exceeds the minimum value. Similarly, *max* is an upper limit on fresh responses. A response can't be fresh unless its time in the cache is less than the maximum time.

Responses that fall between the minimum and maximum are subject to Squid's *last-modified factor* (LM-factor) algorithm. For such responses, Squid calculates the response age and the LM-factor and compares it to the *percent* value. The response age is simply the amount of time passed since the origin server generated, or last validated, the response. The resource age is the difference between the Last-Modified and Date headers. The LM-factor is the ratio of the response age to the resource age.

Figure 7-2 demonstrates the LM-factor algorithm. Squid caches an object that is 3 hours old (based on the Date and Last-Modified headers). With an LM-factor value of 50%, the response will be fresh for the next 1.5 hours, after which the object expires and is considered stale. If a user requests the cached object during the fresh period, Squid returns an unvalidated cache hit. For a request that occurs during the stale period, Squid forwards a validation request to the origin server.

It's important to understand the order that Squid checks the various values. Here is a simplified description of Squid's *refresh_pattern* algorithm:

- The response is stale if the response age is greater than the *refresh_pattern max* value.

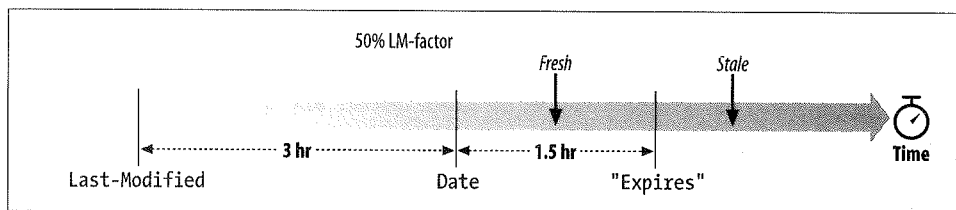


Figure 7-2. Calculating expiration times based on LM-factor

- The response is fresh if the LM-factor is less than the *refresh_pattern percent* value.
- The response is fresh if the response age is less than the *refresh_pattern min* value.
- Otherwise, the response is stale.

The *refresh_pattern* directive also has a handful of options that cause Squid to disobey the HTTP protocol specification. They are as follows:

override-expire

When set, this option causes Squid to check the *min* value before checking the Expires header. Thus, a non-zero *min* time makes Squid return an unvalidated cache hit even if the response is preexpired.

override-lastmod

When set, this option causes Squid to check the *min* value before the LM-factor percentage.

reload-into-ims

When set, this option makes Squid transform a request with a no-cache directive into a validation (If-Modified-Since) request. In other words, Squid adds an If-Modified-Since header to the request before forwarding it on. Note that this only works for objects that have a Last-Modified timestamp. The outbound request retains the no-cache directive, so that it reaches the origin server.

ignore-reload

When set, this option causes Squid to ignore the no-cache directive, if any, in the request.

Exercises

- Run **df** on your existing filesystems and calculate the ratio of inodes to disk space. If any of those partitions are used for Squid's disk cache, do you think you'll run out of space, or inodes first?
- Try to intentionally make Squid run out of disk space on a cache directory. How does Squid deal with this situation?

- Write a shell script to search the cache for given URIs and optionally remove them.
- Examine Squid's *store.log* and estimate the percentage of requests that are subject to the *refresh_pattern* rules.
- Can you think of any negative side effects of the *ignore-reload*, *override-expire*, and related options?

CHAPTER 8

Advanced Disk Cache Topics

Performance is one of the biggest concerns for Squid administrators. As the load placed on Squid increases, disk I/O is typically the primary bottleneck. The reason for this performance limitation is due to the importance that Unix filesystems place on consistency after a system crash.

By default, Squid uses a relatively simple storage scheme (*ufs*). All disk I/O is performed by the main Squid process. With traditional Unix filesystems, certain operations always block the calling process. For example, calling `open( )` on the Unix Fast Filesystem (UFS) causes the operating system to allocate and initialize certain on-disk data structures. The system call doesn't return until these I/O operations complete, which may take longer than you'd like if the disks are already busy with other tasks.

Under heavy load, these filesystem operations can block the Squid process for small, but significant, amounts of time. The point at which the filesystem becomes a bottleneck depends on many different factors, including:

- The number of disk drives
- The rotational speed and seek time of your hard drives
- The type of disk drive interface (ATA, SCSI)
- Filesystem tuning options
- The number of files and percentage of free space

Do I Have a Disk I/O Bottleneck?

Web caches such as Squid don't usually come right out and tell you when disk I/O is becoming a bottleneck. Instead, response time and/or hit ratio degrade as load increases. The tricky thing is that response time and hit ratio may be changing for other reasons, such as increased network latency and changes in client request patterns.

Perhaps the best way to explore the performance limits of your cache is with a benchmark, such as Web Polygraph. The good thing about a benchmark is that you can fully control the environment and eliminate many unknowns. You can also repeat the same experiment with different cache configurations. Unfortunately, benchmarking often takes a lot of time and requires spare systems that aren't already being used.

If you have the resources to benchmark Squid, begin with a standard caching workload. As you increase the load, at some point you should see a significant increase in response time and/or a decrease in hit ratio. Once you observe this performance degradation, run the experiment again but with disk caching disabled. You can configure Squid never to cache any response (with the *null* storage scheme, see the later section "The null Storage Scheme"). Alternatively, you can configure the workload to have 100% uncacheable responses. If the average response time is significantly better without caching, you can be relatively certain that disk I/O is a bottleneck at that level of throughput.

If you're like most people, you have neither the time nor resources to benchmark Squid. In this case, you can examine Squid's runtime statistics to look for disk I/O bottlenecks. The cache manager General Runtime Information page (see Chapter 14) gives you median response times for both cache hits and misses:

Median Service Times (seconds)	5 min	60 min:
HTTP Requests (All):	0.39928	0.35832
Cache Misses:	0.42149	0.39928
Cache Hits:	0.12783	0.11465
Near Hits:	0.37825	0.39928
Not-Modified Replies:	0.07825	0.07409

For a healthy Squid cache, hits are significantly faster than misses. Your median hit response time should usually be 0.5 seconds or less. I strongly recommend that you use SNMP or another network monitoring tool to collect periodic measurements from your Squid caches (see Chapter 14). A significant (factor of two) increase in median hit response time is a good indication that you have a disk I/O bottleneck.

If you believe your production cache is suffering in this manner, you can test your theory with the same technique mentioned previously. Configure Squid not to cache any responses, thus avoiding all disk I/O. Then closely observe the *cache miss* response time. If it goes down, your theory is probably correct.

Once you've convinced yourself that disk throughput is limiting Squid's performance, you can try a number of things to improve it. Some of these require recompiling Squid, while others are relatively simple steps you can take to tune the Unix filesystems.

Filesystem Tuning Options

First of all, you should never use RAID for Squid cache directories. In my experience, RAID always degrades filesystem performance for Squid. It is much better to have a number of separate filesystems, each dedicated to a single disk drive.

I have found four simple ways to improve UFS performance for Squid. Some of these are specific to certain operating systems, such as BSD and Linux, and may not be available on your platform:

- Some UFS implementations support a *noatime* mount option. Filesystems mounted with *noatime* don't update the inode access time value for reads. The easiest way to use this option is to add it to the */etc/fstab* like this:

# Device	Mountpoint	FStype	Options	Dump	Pass#
/dev/ad1s1c	/cache0	ufs	rw,noatime	0	0

- Check your `mount(8)` manpage for the *async* option. With this option set, certain I/O operations (such as directory updates) may be performed asynchronously. The documentation for some systems notes that it is a dangerous flag. Should your system crash, you may lose the entire filesystem. For many installations, the performance improvement is worth the risk. You should use this option only if you don't mind losing the contents of your entire cache. If the cached data is very valuable, the *async* option is probably not for you.
- BSD has a feature called *soft updates*. Soft updates are BSD's alternative to journaling filesystems.* On FreeBSD, you can enable this option on an unmounted filesystem with the `tunefs` command:

```
# umount /cache0
# tunefs -n enable /cache0
# mount /cache0
```

You only have to run the `tunefs` once for each filesystem. Soft updates are automatically enabled on the filesystem again when your system reboots.

On OpenBSD and NetBSD, you can use the *softdep* mount option:

# Device	Mountpoint	FStype	Options	Dump	Pass#
/dev/sd0f	/usr	ffs	rw,softdep	1	2

If you're like me, you're probably wondering what the difference is between the *async* option and *soft updates*. One important difference is that *soft update* code has been designed to maintain filesystem consistency in the event of a system crash, while the *async* option has not. This might lead you to conclude that *async* performs better than *soft updates*. However, as I show in Appendix D, the opposite is true.

* For further information, please see "Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast File System" by Marshall Kirk McKusik and Gregory R. Ganger. *Proceedings of the 1999 USENIX Annual Technical Conference*, June 6–11, 1999, Monterey, California.

Previously, I mentioned that UFS performance, especially writing, depends on the amount of free space. Disk writes for empty filesystems are much faster than for full ones. This is one reason behind UFS's *minfree* parameter and space/time optimization tradeoffs. If your cache disks are full and Squid's performance seems bad, try reducing the *cache_dir* capacity values so that more free space is available. Of course, this reduction in cache size also decreases your hit ratio, but the response time improvement may be worth it. If you're buying the components for a new Squid cache, consider getting much larger disks than you need and using only half the space.

Alternative Filesystems

Some operating systems support filesystems other than UFS (or ext2fs). Journaling filesystems are a common alternative. The primary difference between UFS and journaling filesystems is in the way that they handle updates. With UFS, updates are made in-place. For example, when you change a file and save it to disk, the new data replaces the old data. When you remove a file, UFS updates the directory directly.

A journaling filesystem, on the other hand, writes updates to a separate journal, or log file. You can typically select whether to journal file changes, metadata changes, or both. A background process reads the journal during idle moments and applies the actual changes. Journaling filesystems typically recover much faster from crashes than UFS. After a crash, the filesystem simply reads the journal and commits all the outstanding changes.

The primary drawback of journaling filesystems is that they require additional disk writes. Changes are first written to the log and later to the actual files and/or directories. This is particularly relevant for web caches because they tend to have more disk writes than reads in the first place.

Journaling filesystems are available for a number of operating systems. On Linux, you can choose from ext3fs, reiserfs, XFS, and others. XFS is also available for SGI/IRIX, where it was originally developed. Solaris users can use the Veritas filesystem product. The TRU64 (formerly Digital Unix) Advanced Filesystem (advfs) supports journaling.

You can use a journaling filesystem without making any changes to Squid's configuration. Simply create and mount the filesystem as described in your operating system documentation. You don't need to change the *cache_dir* line in *squid.conf*. Use a command like this to make a reiserfs filesystem on Linux:

```
# /sbin/mkreiserfs /dev/sda2
```

For XFS, use:

```
# mkfs -t xfs -f /dev/sda2
```

Note that ext3fs is simply ext2fs with journaling enabled. Use the -j option to mke2fs when creating the filesystem:

```
# /sbin/mke2fs -j /dev/sda2
```

Refer to your documentation (e.g., manpages) for other operating systems.

The aufs Storage Scheme

The *aufs* storage scheme has evolved out of the very first attempt to improve Squid's disk I/O response time. The "a" stands for asynchronous I/O. The only difference between the default *ufs* scheme and *aufs* is that I/Os aren't executed by the main Squid process. The data layout and format is the same, so you can easily switch between the two schemes without losing any cache data.

aufs uses a number of thread processes for disk I/O operations. Each time Squid needs to read, write, open, close, or remove a cache file, the I/O request is dispatched to one of the thread processes. When the thread completes the I/O, it signals the main Squid process and returns a status code. Actually, in Squid 2.5, certain file operations aren't executed asynchronously by default. Most notably, disk writes are always performed synchronously. You can change this by setting `ASYNC_WRITE` to 1 in `src/lfs/aufs/store_asyncufs.h` and recompiling.

The *aufs* code requires a *pthread*s library. This is the standard threads interface, defined by POSIX. Even though *pthread*s is available on many Unix systems, I often encounter compatibility problems and differences. The *aufs* storage system seems to run well only on Linux and Solaris. Even though the code compiles, you may encounter serious problem on other operating systems.

To use *aufs*, you must add a special `./configure` option:

```
% ./configure --enable-storeio=aufs,ufs
```

Strictly speaking, you don't really need to specify *ufs* in the list of storeio modules. However, you might as well because if you try *aufs* and don't like it, you'll be able to fall back to the plain *ufs* storage scheme.

You can also use the `--with-aio-threads=N` option if you like. If you omit it, Squid automatically calculates the number of threads to use based on the number of *aufs* `cache_dirs`. Table 8-1 shows the default number of threads for up to six cache directories.

Table 8-1. Default number of threads for up to six cache directories

cache_dirs	Threads
1	16
2	26
3	32

Table 8-1. Default number of threads for up to six cache directories (continued)

cache_dirs	Threads
4	36
5	40
6	44

After you compile *aufs* support into Squid, you can specify it on a *cache_dir* line in *squid.conf*:

```
cache_dir aufs /cache0 4096 16 256
```

After starting Squid with *aufs* enabled, make sure everything still works correctly. You may want to run `tail -f store.log` for a while to make sure that objects are being swapped out to disk. You should also run `tail -f cache.log` and look for any new errors or warnings.

How aufs Works

Squid creates a number of thread processes by calling `pthread_create()`. All threads are created upon the first disk activity. Thus, you'll see all the thread processes even if Squid is idle.

Whenever Squid wants to perform some disk I/O operation (e.g., to open a file for reading), it allocates a couple of data structures and places the I/O request into a queue. The thread processes have a loop that take I/O requests from the queue and executes them. Because the request queue is shared by all threads, Squid uses mutex locks to ensure that only one thread updates the queue at a given time.

The I/O operations block the thread process until they are complete. Then, the status of the operation is placed on a *done queue*. The main Squid process periodically checks the done queue for completed operations. The module that requested the disk I/O is notified that the operation is complete, and the request or response processing proceeds.

As you may have guessed, *aufs* can take advantage of systems with multiple CPUs. The only locking that occurs is on the request and result queues. Otherwise, all other functions execute independently. While the main process executes on one CPU, another CPU handles the actual I/O system calls.

aufs Issues

An interesting property of threads is that all processes share the same resources, including memory and file descriptors. For example, when a thread process opens a file as descriptor 27, all other threads can then access that file with the same descriptor number. As you probably know, file-descriptor shortage is a common problem with first-time Squid administrators. Unix kernels typically have two file-descriptor

limits: per process and systemwide. While you might think that 256 file descriptors per process is plenty (because of all the thread processes), it doesn't work that way. In this case, all threads share that small number of descriptors. Be sure to increase your system's per-process file descriptor limit to 4096 or higher, especially when using *aufs*.

Tuning the number of threads can be tricky. In some cases, you might see this warning in *cache.log*:

```
2003/09/29 13:42:47| squidaido_queue_request: WARNING - Disk I/O overloading
```

It means that Squid has a large number of I/O operations queued up, waiting for an available thread. Your first instinct may be to increase the number of threads. I would suggest, however, that you decrease the number instead.

Increasing the number of threads also increases the queue size. Past a certain point, it doesn't increase *aufs*'s load capacity. It only means that more operations become queued. Longer queues result in higher response times, which is probably something you'd like to avoid.

Decreasing the number of threads, and the queue size, means that Squid can detect the overload condition faster. When a *cache_dir* is overloaded, it is removed from the selection algorithm (see Chapter 7). Then, Squid either chooses a different *cache_dir* or simply doesn't store the response on disk. This may be a better situation for your users. Even though the hit ratio goes down, response time remains relatively low.

Monitoring aufs Operation

The Async IO Counters option in the cache manager menu displays a few statistics relating to *aufs*. It shows counters for the number of open, close, read, write, stat, and unlink requests received. For example:

```
% squidclient mgr:squidaido_counts
...
ASYNC IO Counters:
Operation      # Requests
open           15318822
close          15318813
cancel         15318813
write          0
read           19237139
stat           0
unlink         2484325
check_callback 311678364
queue          0
```

The cancel counter is normally equal to the close counter. This is because the close function always calls the cancel function to ensure that any pending I/O operations are ignored.

The write counter is zero because this version of Squid performs writes synchronously, even for *aufs*.

The `check_callback` counter shows how many times the main Squid process has checked the done queue for completed operations.

The queue value indicates the current length of the request queue. Normally, the queue length should be less than the number of threads $\times$ 5. If you repeatedly observe a queue length larger than this, you may be pushing Squid too hard. Adding more threads may help but only to a certain point.

The diskd Storage Scheme

diskd (short for disk daemons) is similar to *aufs* in that disk I/Os are executed by external processes. Unlike *aufs*, however, *diskd* doesn't use threads. Instead, inter-process communication occurs via message queues and shared memory.

Message queues are a standard feature of modern Unix operating systems. They were invented many years ago in AT&T's Unix System V, Release 1. The messages passed between processes on these queues are relatively small: 32–40 bytes. Each *diskd* process uses one queue for receiving requests from Squid and another queue for transmitting results back.

How diskd Works

Squid creates one *diskd* process for each *cache_dir*. This is different from *aufs*, which uses a large pool of threads for all *cache_dirs*. Squid sends a message to the corresponding *diskd* process for each I/O operation. When that operation is complete, the *diskd* process sends a status message back to Squid. Squid and the *diskd* processes preserve the order of messages in the queues. Thus, there is no concern that I/Os might be executed out of sequence.

For reads and writes, Squid and the *diskd* processes use a shared memory area. Both processes can read from, and write to, this area of memory. For example, when Squid issues a read request, it tells the *diskd* process where to place the data in memory. *diskd* passes this memory location to the `read()` system call and notifies Squid that the read is complete by sending a message on the return queue. Squid then accesses the recently read data from the shared memory area.

diskd (as with *aufs*) essentially gives Squid nonblocking disk I/Os. While the *diskd* processes are blocked on I/O operations, Squid is free to work on other tasks. This works really well as long as the *diskd* processes can keep up with the load. Because the main Squid process is now able to do more work, it's possible that it may overload the *diskd* helpers. The *diskd* implementation has two features to help out in this situation.

First, Squid waits for the *diskd* processes to catch up if one of the queues exceeds a certain threshold. The default value is 64 outstanding messages. If a *diskd* process gets this far behind, Squid “sleeps” a small amount of time and waits for it to complete some of the pending operations. This essentially puts Squid into a blocking I/O mode. It also makes more CPU time available to the *diskd* processes. You can configure this threshold by specifying a value for the Q2 parameter on a *cache_dir* line:

```
cache_dir diskd /cache0 7000 16 256 Q2=50
```

Second, Squid stops asking the *diskd* process to open files if the number of outstanding operations reaches another threshold. Here, the default value is 72 messages. If Squid would like to open a disk file for reading or writing, but the selected *cache_dir* has too many pending operations, the open request fails internally. When trying to open a file for reading, this causes a cache miss instead of a cache hit. When opening files for writing, it prevents Squid from storing a cachable response. In both cases the user still receives a valid response. The only real effect is that Squid’s hit ratio decreases. This threshold is configurable with the Q1 parameter:

```
cache_dir diskd /cache0 7000 16 256 Q1=60 Q2=50
```

Note that in some versions of Squid, the Q1 and Q2 parameters are mixed-up in the default configuration file. For optimal performance, Q1 should be greater than Q2.

Compiling and Configuring diskd

To use *diskd*, you must add it to the `--enable-storeio` list when running `./configure`:

```
% ./configure --enable-storeio=ufs,diskd
```

diskd seems to be portable since shared memory and message queues are widely supported on modern Unix systems. However, you’ll probably need to adjust a few kernel limits relating to both. Kernels typically have the following variables or parameters:

MSGMNB

This is the maximum characters (octets) per message queue. With *diskd*, the practical limit is about 100 outstanding messages per queue. The messages that Squid passes are 32–40 octets, depending on your CPU architecture. Thus, MSGMNB should be 4000 or more. To be safe, I recommend setting this to 8192.

MSGMNI

This is the maximum number of message queues for the whole system. Squid uses two queues for each *diskd cache_dir*. If you have 10 disks, that’s 20 queues. You should probably add even more in case other applications also use message queues. I recommend a value of 40.

MSGSSZ

This is the size of a message segment, in octets. Messages larger than this size are split into multiple segments. I usually set this to 64 so that the *diskd* message isn't split into multiple segments.

MSGSEG

This is the maximum number of message segments that can exist in a single queue. Squid normally limits the queues to 100 outstanding messages. Remember that if you don't increase MSGSSZ to 64 on 64-bit architectures, each message requires more than one segment. To be safe, I recommend setting this to 512.

MSGTQL

This is the maximum number of messages that can exist in the whole system. It should be at least 100 multiplied by the number of *cache_dirs*. I recommend setting it to 2048, which should be more than enough for as many as 10 cache directories.

MSGMAX

This is the maximum size of a single message. For Squid, 64 bytes should be sufficient. However, your system may have other applications that use larger messages. On some operating systems such as BSD, you don't need to set this. BSD automatically sets it to $\text{MSGSSZ} \times \text{MSGSEG}$. On other systems you may need to increase the value from its default. In this case, you can set it to the same as MSGMNB.

SHMSEG

This is the maximum number of shared memory segments allowed per process. Squid uses one shared memory identifier for each *cache_dir*. I recommend a setting of 16 or higher.

SHMMNI

This is the systemwide limit on the number of shared memory segments. A value of 40 is probably enough in most cases.

SHMMAX

This is the maximum size of a single shared memory segment. By default, Squid uses about 409,600 bytes for each segment. Just to be safe, I recommend setting this to 2 MB, or 2,097,152.

SHMALL

This is the systemwide limit on the amount of shared memory that can be allocated. On some systems, SHMALL may be expressed as a number of pages, rather than bytes. Setting this to 16 MB (4096 pages) is enough for 10 *cache_dirs* with plenty remaining for other applications.

To configure message queues on BSD, add these options to your kernel configuration file:*

```
# System V message queues and tunable parameters
options      SYSVMSG      # include support for message queues
options      MSGMNB=8192  # max characters per message queue
options      MSGMNI=40    # max number of message queue identifiers
options      MSGSEG=512   # max number of message segments per queue
options      MSGSSZ=64    # size of a message segment MUST be power of 2
options      MSGTQL=2048  # max number of messages in the system
options      SYSVSHM
options      SHMSEG=16    # max shared mem segments per process
options      SHMMNI=32    # max shared mem segments in the system
options      SHMMAX=2097152 # max size of a shared mem segment
options      SHMALL=4096  # max size of all shared memory (pages)
```

To configure message queues on Linux, add these lines to */etc/sysctl.conf*:

```
kernel.msgmnb=8192
kernel.msgmni=40
kernel.msgmax=8192
kernel.shmall=2097152
kernel.shmmni=32
kernel.shmmax=16777216
```

Alternatively, or if you find that you need more control, you can manually edit *include/linux/msg.h* and *include/linux/shm.h* in your kernel sources.

For Solaris, add these lines to */etc/system* and then reboot:

```
set msgsys:msginfo_msgmax=8192
set msgsys:msginfo_msgmnb=8192
set msgsys:msginfo_msgmni=40
set msgsys:msginfo_msgssz=64
set msgsys:msginfo_msgtql=2048
set shmsys:shminfo_shmmax=2097152
set shmsys:shminfo_shmmni=32
set shmsys:shminfo_shmseg=16
```

For Digital Unix (TRU64), you can probably add lines to the kernel configuration in the style of BSD, seen previously. Alternatively, you can use the *sysconfig* command. First, create a file called *ipc.stanza* like this:

```
ipc:
    msg-max = 2048
    msg-mni = 40
    msg-tql = 2048
    msg-mnb = 8192
    shm-seg = 16
    shm-mni = 32
    shm-max = 2097152
    shm-max = 4096
```

* OpenBSD is a little different. Use *option* instead of *options*, and specify the *SHMMAX* value in pages, rather than bytes.

Now, run this command and reboot:

```
# sysconfigdb -a -f ipc.stanza
```

After you have message queues and shared memory configured in your operating system, you can add the *cache_dir* lines to *squid.conf*:

```
cache_dir diskd /cache0 7000 16 256 Q1=72 Q2=64
cache_dir diskd /cache1 7000 16 256 Q1=72 Q2=64
...
```

If you forget to increase the message queue limits, or if you don't set them high enough, you'll see messages like this in *cache.log*:

```
2003/09/29 01:30:11| storeDiskdSend: msgsnd: (35) Resource temporarily unavailable
```

Monitoring diskd

The best way to monitor *diskd* performance is with the cache manager. Request the *diskd* page; for example:

```
% squidclient mgr:diskd
...
sent_count: 755627
recv_count: 755627
max_away: 14
max_shmuse: 14
open_fail_queue_len: 0
block_queue_len: 0
```

	OPS	SUCCESS	FAIL
open	51534	51530	4
create	67232	67232	0
close	118762	118762	0
unlink	56527	56526	1
read	98157	98153	0
write	363415	363415	0

See Chapter 14 for a description of this output.

The coss Storage Scheme

The Cyclic Object Storage Scheme (*coss*) is an attempt to develop a custom filesystem for Squid. With the *ufs*-based schemes, the primary performance bottleneck comes from the need to execute so many *open()* and *unlink()* system calls. Because each cached response is stored in a separate disk file, Squid is always opening, closing, and removing files.

coss, on the other hand, uses one big file to store all responses. In this sense, it is a small, custom filesystem specifically for Squid. *coss* implements many of the functions normally handled by the underlying filesystem, such as allocating space for new data and remembering where there is free space.

Unfortunately, *css* is still a little rough around the edges. Development of *css* has been proceeding slowly over the last couple of years. Nonetheless, I'll describe it here in case you feel adventurous.

How *css* Works

On the disk, each *css cache_dir* is just one big file. The file grows in size until it reaches its maximum size. At this point, Squid starts over at the beginning of the file, overwriting any data already stored there. Thus, new objects are always stored at the “end” of this cyclic file.*

Squid actually doesn't write new object data to disk immediately. Instead, the data is copied into a 1-MB memory buffer, called a *stripe*. A stripe is written to disk when it becomes full. *css* uses asynchronous writes so that the main Squid process doesn't become blocked on disk I/O.

As with other filesystems, *css* also uses the *blocksize* concept. Back in Chapter 7, I talked about file numbers. Each cached object has a file number that Squid uses to locate the data on disk. For *css*, the file number is the same as the block number. For example, a cached object with a swap file number equal to 112 starts at the 112th block in a *css* filesystem. File numbers aren't allocated sequentially with *css*. Some file numbers are unavailable because cached objects generally occupy more than one block in the *css* file.

The *css* block size is configurable with a *cache_dir* option. Because Squid's file numbers are only 24 bits, the block size determines the maximum size of a *css* cache directory: $\text{size} = \text{block_size} \times 2^{24}$. For example, with a 512-byte block size, you can store up to 8 GB in a *css cache_dir*.

css doesn't implement any of Squid's normal cache replacement algorithms (see Chapter 7). Instead, cache hits are “moved” to the end of the cyclic file. This is, essentially, the LRU algorithm. It does, unfortunately, mean that cache hits cause disk writes, albeit indirectly.

With *css*, there is no need to unlink or remove cached objects. Squid simply forgets about the space allocated to objects that are removed. The space will be reused eventually when the end of the cyclic file reaches that place again.

Compiling and Configuring *css*

To use *css*, you must add it to the `--enable-storeio` list when running `./configure`:

```
% ./configure --enable-storeio=ufs,css ...
```

* The beginning is the location where data was first written; the end is the location where data was most recently written.

*cos*s cache directories require a *max-size* option. Its value must be less than the stripe size (1 MB by default, but configurable with the *--enable-coss-membuf-size* option). Also note that you must omit the L1 and L2 values that are normally present for *ufs*-based schemes. Here is an example:

```
cache_dir coss /cache0/coss 7000 max-size=1000000
cache_dir coss /cache1/coss 7000 max-size=1000000
cache_dir coss /cache2/coss 7000 max-size=1000000
cache_dir coss /cache3/coss 7000 max-size=1000000
cache_dir coss /cache4/coss 7000 max-size=1000000
```

Furthermore, you can change the default *cos*s block size with the *block-size* option:

```
cache_dir coss /cache0/coss 30000 max-size=1000000 block-size=2048
```

One tricky thing about *cos*s is that the *cache_dir* directory argument (e.g., */cache0/coss*) isn't actually a directory. Instead, it is a regular file that Squid opens, and creates if necessary. This is so you can use raw partitions as *cos*s files. If you mistakenly create the *cos*s file as a directory, you'll see an error like this when starting Squid:

```
2003/09/29 18:51:42| /usr/local/squid/var/cache: (21) Is a directory
FATAL: storeCossDirInit: Failed to open a coss file.
```

Because the *cache_dir* argument isn't a directory, you must use the *cache_swap_log* directive (see Chapter 13). Otherwise Squid attempts to create a *swap.state* file in the *cache_dir* directory. In that case, you'll see an error like this:

```
2003/09/29 18:53:38| /usr/local/squid/var/cache/coss/swap.state:
(2) No such file or directory
FATAL: storeCossDirOpenSwapLog: Failed to open swap log.
```

*cos*s uses asynchronous I/Os for better performance. In particular, it uses the *aio_read()* and *aio_write()* system calls. These may not be available on all operating systems. At this time, they are available on FreeBSD, Solaris, and Linux. If the *cos*s code seems to compile okay, but you get a "Function not implemented" error message, you need to enable these system calls in your kernel. On FreeBSD, your kernel must have this option:

```
options      VFS_AIO
```

cos Issues

*cos*s is still an experimental feature. The code has not yet proven stable enough for everyday use. If you want to play with and help improve it, be prepared to lose any data stored in a *cos cache_dir*. On the plus side, *cos*s's preliminary performance tests are very good. For an example, see Appendix D.

*cos*s doesn't support rebuilding cached data from disk very well. When you restart Squid, you might find that it fails to read the *cos swap.state* files, thus losing any cached data. Furthermore, Squid doesn't remember its place in the cyclic file after a restart. It always starts back at the beginning.

coss takes a nonstandard approach to object replacement. This may cause a lower hit ratio than you might get with one of the other storage schemes.

Some operating systems have problems with files larger than 2 GB. If this happens to you, you can always create more, smaller *coss* areas. For example:

```
cache_dir coss /cache0/coss0 1900 max-size=1000000 block-size=128
cache_dir coss /cache0/coss1 1900 max-size=1000000 block-size=128
cache_dir coss /cache0/coss2 1900 max-size=1000000 block-size=128
cache_dir coss /cache0/coss3 1900 max-size=1000000 block-size=128
```

Using a raw disk device (e.g., */dev/da0s1c*) doesn't work very well yet. One reason is that disk devices usually require that I/Os take place on 512-byte block boundaries. Another concern is that direct disk access bypasses the systems buffer cache and may degrade performance. Many disk drives, however, have built-in caches these days.

The null Storage Scheme

Squid has a fifth storage scheme called *null*. As the name implies, this is more of a nonstorage scheme. Files that are “written” to a *null cache_dir* aren't actually written to disk.

Most people won't have any reason to use the *null* storage system. It's primarily useful if you want to entirely disable Squid's disk cache.* You can't simply remove all *cache_dir* lines from *squid.conf* because then Squid adds a default *ufs cache_dir*. The *null* storage system is also sometimes useful for testing and benchmarking Squid. Since the filesystem is typically the performance bottleneck, using the *null* storage scheme gives you an upper limit of Squid's performance on your hardware.

To use this scheme you must first specify it on the *--enable-storeio* list when running *./configure*:

```
% ./configure --enable-storeio=ufs,null ...
```

You can then create a *cache_dir* of type *null* in *squid.conf*:

```
cache_dir /tmp null
```

It may seem odd that you need to specify a directory for the *null* storage scheme. However, Squid uses the directory name as a *cache_dir* identifier. For example, you'll see it in the cache manager output (see Chapter 14).

Which Is Best for Me?

Squid's storage scheme choices may seem a little overwhelming and confusing. Is *aufs* better than *diskd*? Does my system support *aufs* or *coss*? Will I lose my data if I use one of these fancy schemes? Is it okay to mix-and-match storage schemes?

* Some responses may still be cached in memory, however.

First of all, if your Squid is lightly used (say, less than five requests per second), the default *ufs* storage scheme should be sufficient. You probably won't see a noticeable performance improvement from the other schemes at this low request rate.

If you are trying to decide which scheme to try, your operating system may be a determining factor. For example, *aufs* runs well on Linux and Solaris but seems to have problems on other systems. The *css* code uses functions that aren't available on certain operating systems (e.g., NetBSD) at this time.

It seems to me that higher-performing storage schemes are also more susceptible to data loss in the event of a system crash. This is the tradeoff for better performance. For many people, however, cached data is of relatively low value. If Squid's cache becomes corrupted due to a crash, you may find it easier to simply *newfs* the disk partition and let the cache fill back up from scratch. If you find it difficult or expensive to replace the contents of Squid's cache, you probably want to use one of the slow, but reliable, filesystems and storage schemes.

Squid certainly allows you to use different filesystems and storage schemes for each *cache_dir*. In practice, however, this is uncommon. You'll probably have fewer hassles if all cache directories are approximately the same size and use the same storage scheme.

Exercises

- Try to compile all possible storage schemes on your system.
- Run Squid with a separate *cache_dir* for each storage scheme you can get to compile.
- Run Squid with one or more *diskd cache_dirs*. Then run the **ipcs -o** command.

CHAPTER 9

Interception Caching

Interception caching is a popular technique for getting traffic to Squid without configuring any clients. Instead, you configure a router or switch to divert HTTP connections to the machine on which Squid is running. Squid's operating system is configured to accept the foreign packets and deliver them to the Squid process. To make HTTP interception work, you need to configure three separate components: a network device, Squid's operating system, and Squid itself.

This chapter begins with an overview of HTTP interception. I'll explain how it all works and define some terms so that the remaining sections make sense. I also explain the tradeoffs involved with HTTP interception.

Following that, I'll discuss your options for devices and configurations that can intercept client traffic. In particular, I cover Cisco policy routing, Cisco's WCCP, layer four switches, and running Squid on a host that also functions as a router or bridge.

Next, I'll show how to configure the operating system to handle the intercepted connections. This functionality is a feature of the IP packet filtering software, which varies from system to system. It is called *iptables* (Netfilter) on Linux; *ipfw* on FreeBSD; *pf* on OpenBSD; and *IPFilter* on NetBSD, Solaris, and other BSD variants.

Squid is the final component you need to configure. Fortunately, this is relatively straightforward because it doesn't depend on your operating system or network device.

I finish the chapter with a little checklist that may help you debug HTTP interception problems.

How It Works

Interception caching involves some network trickery, so it is helpful for you to understand what happens between the client and Squid. I'll use Figure 9-1 and the following sample *tcpdump* output to explain how the packets are intercepted as they flow through your network.

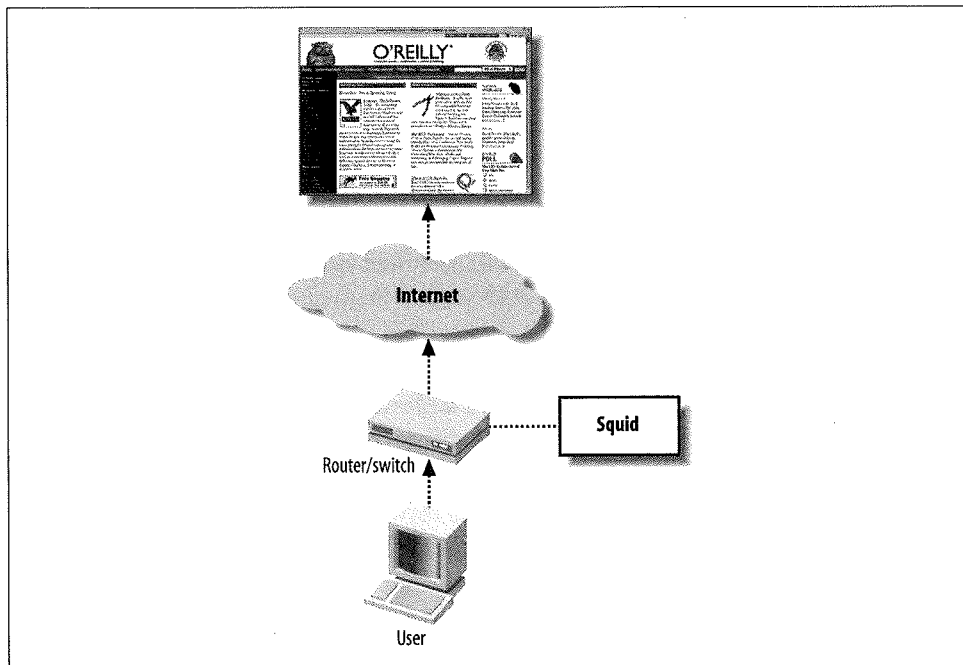


Figure 9-1. How HTTP interception works

1. The user-agent wants to request a resource, say `/index.html` from an origin server, say `www.oreilly.com`. It needs the origin server's IP address, so it makes a DNS request:

```

Packet 1
TIME: 19:54:41.317310
UDP: 206.168.0.3.2459 -> 206.168.0.2.53
DATA: .d.....www.oreilly.com.....
-----
Packet 2
TIME: 19:54:41.317707 (0.000397)
UDP: 206.168.0.2.53 -> 206.168.0.3.2459
DATA: .d.....www.oreilly.com.....PR....%.....PR.
     .$......PR...ns1.sonic.net.....PR...ns2.Q.....PR
     ...ns...M.....h.....!z.....b.....
  
```

2. Now that it has the IP address, the user-agent initiates a TCP connection to the origin server on port 80:

```

Packet 3
TIME: 19:54:41.320652 (0.002945)
TCP: 206.168.0.3.3897 -> 208.201.239.37.80 Syn
DATA: <No data>
  
```

3. The switch/router notices a TCP SYN packet with destination port 80. What happens next depends on the particular interception technology. In the case of layer four switches and policy routing, the device simply forwards the TCP

packet to Squid's datalink layer (Ethernet) address. This works only when Squid is directly attached to the network device. For WCCP, the router encapsulates the TCP packet into a GRE packet. Because the GRE packet has its own IP address, it can be routed through multiple subnets. In other words, WCCP doesn't require Squid to be directly attached to the router.

4. The Squid host's operating system receives the intercepted packet. For layer four switches, the TCP/IP packet is unchanged from the earlier explanation.

If the packet is encapsulated with GRE, the host removes the outer IP and GRE headers and places the original TCP/IP packet on the input queue.

Note that the Squid host receives an IP packet for a foreign address (the origin server's). Normally this packet is dropped because its destination address doesn't match any of the local interface addresses. To make the host accept the foreign packet, you must enable IP forwarding on most operating systems.

5. The client's TCP/IP packet is processed by the packet filtering code. The packet matches a rule that instructs the kernel to forward or divert this packet to Squid. Without this rule, the kernel simply forwards this packet on its way to the origin server, which isn't what you want.

Note that the SYN packet's destination port is 80, but Squid may be listening on a different port, such as 3128. The packet filtering rules allow you to change the port number. You don't need to make Squid listen on port 80. You can't see this step with `tcpdump` because the diverted packet doesn't flow through the network interface code again.

The packet filter's redirection rule is still necessary even if you have Squid listen on port 80. Simply making the port numbers match doesn't allow Squid to receive the intercepted packets. The redirection rule is the magic that delivers foreign packets to Squid.

6. Squid receives notification of the new connection, which it accepts. The kernel sends a SYN/ACK packet back to the client:

```
Packet 4
TIME: 19:54:41.320735 (0.000083)
TCP: 208.201.239.37.80 -> 206.168.0.3.3897 SynAck
DATA: <No data>
```

As you can see, the source address is the origin server's, even though this packet didn't reach the origin. The operating system simply copies and swaps the source and destination IP addresses from the SYN packet into the reply.

7. The user-agent receives the SYN/ACK packet, fully establishing the TCP connection. The user-agent now believes it is connected to the origin server, so it writes the HTTP request:

```
Packet 5
TIME: 19:54:41.323080 (0.002345)
TCP: 206.168.0.3.3897 -> 208.201.239.37.80 Ack
DATA: <No data>
```

```
-----
Packet 6
TIME: 19:54:41.323482 (0.000402)
TCP: 206.168.0.3.3897 -> 208.201.239.37.80 AckPsh
DATA: GET / HTTP/1.0
      User-Agent: Wget/1.8.2
      Host: www.oreilly.com
      Accept: */*
      Connection: Keep-Alive
```

8. Squid receives the HTTP request. It uses the HTTP Host header to convert the partial URL into a full URL. In this case, you'll see *http://www.oreilly.com/* in the *access.log* file.
9. From this point on, Squid treats the request normally. As usual, cache hits are returned immediately. Cache misses are forwarded to the origin server.
10. Lastly, here is the response that Squid receives from the origin server:

```
Packet 8
TIME: 19:54:41.448391 (0.030030)
TCP: 208.201.239.37.80 -> 206.168.0.3.3897 AckPsh
DATA: HTTP/1.0 200 OK
      Date: Mon, 29 Sep 2003 01:54:41 GMT
      Server: Apache/1.3.26 (Unix) PHP/4.2.1 mod_gzip/1.3.19.1a mod_perl/1.27
      P3P: policyref="http://www.oreillynet.com/w3c/p3p.xml",CP="CAO DSP COR CURa ADMa DEVa TAIa PSAa PSDa IVAa IVDa CONo OUR DELa PUBi OTRa IND PHY ONL UNI PUR COM NAV INT DEM CNT STA P RE"
      Last-Modified: Sun, 28 Sep 2003 23:54:44 GMT
      ETag: "1b76bf-b910-3ede86c4"
      Accept-Ranges: bytes
      Content-Length: 47376
      Content-Type: text/html
      X-Cache: MISS from www.oreilly.com
      X-Cache: MISS from 10.0.0.1
      Connection: keep-alive
```

You don't want your switch/router to intercept the connections that Squid makes to origin servers. If that happens, Squid ends up talking to itself and can't satisfy any cache misses. The best way to avoid forwarding loops like this is to make sure that your users and Squid connect to separate interfaces on the switch/router. Whenever feasible, you should apply the interception rules to specific interfaces. Obviously, you should not enable interception on the interface that Squid uses.

Why (Not) Intercept?

Many organizations find interception caching attractive because they can't, or would rather not, configure all their user's web browsers. It's probably easier to perform a little network trickery on a single switch or router than it is to configure hundreds or thousands of workstations. As with many choices we face, interception caching is

really a tradeoff. It brings both benefits and drawbacks. It may make your life easier, or more difficult.

The obvious benefit of interception caching is that all HTTP requests leaving your network automatically go through Squid. You don't need to worry about configuring any browsers or that users might disable their proxy settings. Interception caching puts you, the network administrator, in control of the HTTP traffic. You can change, add, or remove Squid caches from service without significantly interrupting your users' web surfing.

Most of the disadvantages surrounding HTTP interception are because this technique violates the TCP/IP standards. These protocols mandate that routers (and switches) forward TCP/IP packets to the host specified by the destination IP address. Diverting the packets to a caching proxy breaks the rules. The proxy accepts diverted connections under false pretense. User agents are tricked into believing they have established a TCP connection with the origin server.

This confusion causes a serious problem with older versions of Microsoft's Internet Explorer. The browser's Reload button is the easiest way to refresh an HTML page. When Explorer is configured to use a caching proxy, a reload request includes a Cache-Control: no-cache header to force a cache miss (or validation) and ensure that the response is up to date. Explorer omits this header when not explicitly configured for proxying. With interception caching, Explorer thinks it is connecting to the origin server anyway, and there is no need to send this header. Squid can't tell that the user pressed the Reload button in this case and may not validate the cached response. Squid's *ie_refresh* provides a partial workaround for this bug (see Appendix A). According to Microsoft, this problem has been corrected in Explorer Version 5.5, Service Pack 1.*

For similar reasons, you can't use HTTP proxy authentication in combination with interception caching. Because the client is unaware of the proxy, it doesn't send the necessary Proxy-Authorization header. Additionally, the 407 (Proxy Authorization Required) response code is inappropriate because the response should look like it came from the origin server, which would never send such a reply.

You also can't use RFC 1413 ident lookups (see Chapter 6) with interception. Squid can't bind a new TCP socket to the necessary IP address. The operating system cheats when forwarding the intercepted connection to Squid. However, it can't cheat when Squid wants to bind a new TCP socket to the foreign IP address. The address that it wants to bind to isn't really local, so the bind system call fails.

Interception caching is also incompatible with IP filtering designed to prevent address spoofing (See also RFC 2267: Network Ingress Filtering: Defeating Denial of

* See Microsoft support knowledge base article Q266121 for more (or less) information: <http://support.microsoft.com/support/kb/articles/Q266/1/21.ASP>.

Service Attacks Which Employ IP Source Address Spoofing). Consider the network shown in Figure 9-2. The router has two LAN interfaces: lan0 and lan1. The network administrator uses packet filters on the router to make sure that the internal hosts don't transmit packets with spoofed source addresses. The router forwards only packets with source addresses corresponding to the connected networks. The packet filter rules might look something like this:

```
# lan0
allow ip from 172.16.1.0/24 to any via lan0
deny ip from any to any via lan0
# lan1
allow ip from 10.0.0.0/16 to any via lan1
deny ip from any to any via lan1
```

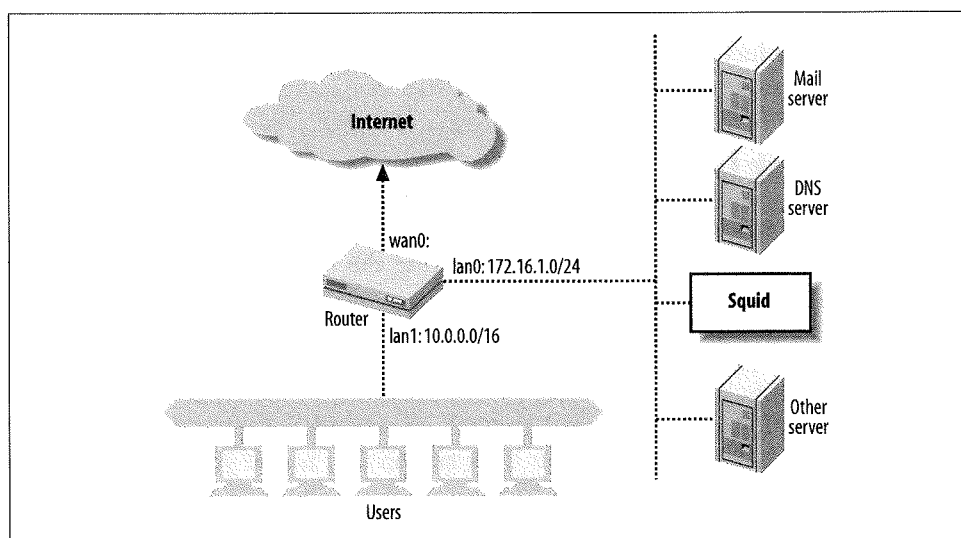


Figure 9-2. Interception caching breaks address spoofing filters

Now consider what happens when the router and Squid box on lan1 are configured to intercept HTTP connections coming from lan0. Squid pretends to be the origin server, which means that the TCP packets carrying response data from Squid back to the users have spoofed source addresses. These lan0 filter rules cause the router to deny these packets. To make interception caching work, the network administrator must remove the lan0 rules. This, in turn, leaves the network vulnerable to being the source of denial-of-service attacks.

As I explained in the previous section, clients must make DNS queries before opening a connection. This may be undesirable or difficult in certain firewall environments. A host whose HTTP traffic you want to intercept must be able to query the DNS. Clients that know they are using a proxy (due to manual configuration or proxy auto-configuration, for example) don't usually try to resolve hostnames.

Instead, they simply forward full URLs to Squid, and it becomes Squid's job to look up origin server IP addresses.

Another little problem is that Squid accepts connections for any destination IP address. Consider, for example, a web site that still has a DNS entry even though the site and server have been taken down. Squid accepts the TCP connection for this bogus site. The client believes the site is up and running, because its connection is established. When Squid fails to connect to the origin server, it is forced to return an error message.

In case it's not clear, HTTP interception can be tricky and difficult to get working the first time. A number of different components must all work together and be correctly configured. Furthermore, it can be difficult to recreate the entire configuration from memory. I strongly encourage you to set up a test environment before attempting this on a production system. Once you get it all working, be sure to document every little step.

The Network Device

Now that you know all the ins and outs of interception caching, let's see how to actually make it work. We'll start by configuring the network devices that will be intercepting your HTTP connections.

Inline Squid

In this configuration, you don't need a switch or network router to intercept HTTP connections. Instead, Squid runs on a Unix system that is also your router (or perhaps bridge), as shown in Figure 9-3.

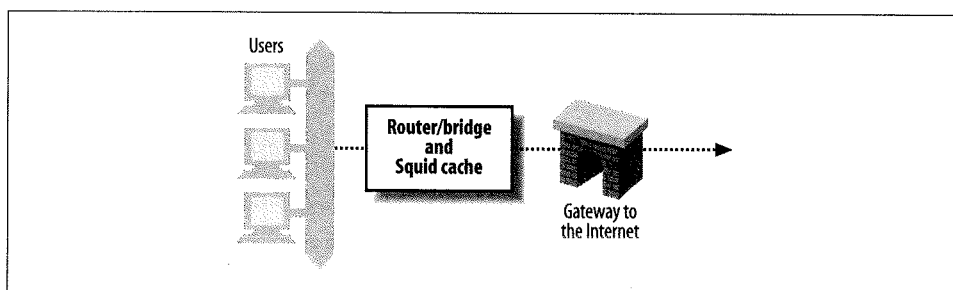


Figure 9-3. A system that combines routing and caching can easily intercept HTTP traffic

This configuration essentially skips the first three steps shown in the section "How It Works." The Squid host already receives the HTTP connection packets because it is the router for your network. If you are taking this approach, feel free to skip ahead to the "Operating System Tweaks" section.

Layer Four Switches

Many organizations use layer four switches specifically for their HTTP interception support. These products offer additional features as well, such as health checks and load balancing. I'll only cover interception here. For information on health checks and load balancing, see O'Reilly's *Server Load Balancing and Load Balancing Servers, Firewalls, and Caches* (John Wiley & Sons). The following subsections contain working-example configurations for a number of products and techniques.

Alteon/Nortel

The following configuration is from an ACEswitch 180 and Alteon's WebOS 8.0.21. The network setup is shown in Figure 9-4.

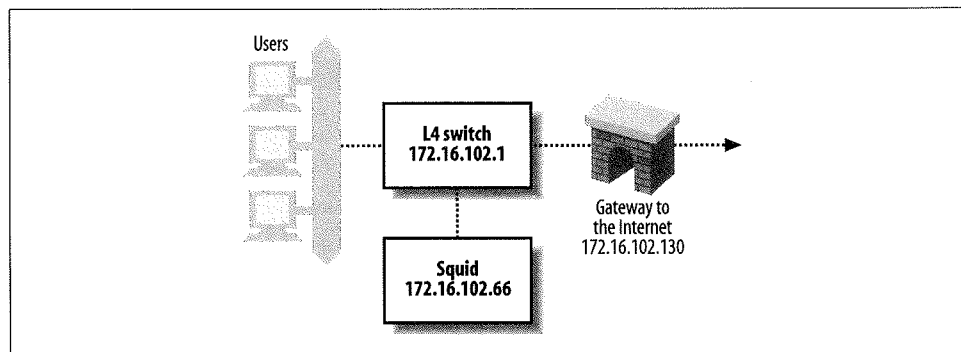


Figure 9-4. Sample network for layer four switch interception, for Alteon and Foundry examples

Clients are connected to port 1, the connection to the Internet is via port 2, and Squid is on port 3. The following lines are the relevant output of a `/cfg/dump` command on the switch. You don't necessarily need to type all of these lines. Furthermore, some of the commands may have changed for newer versions of Alteon's software. Note that Alteon calls this feature Web Cache Redirection (WCR). Here's the process, step by step:

1. First, you must give the Alteon switch an IP address. This seems necessary so that the switch can perform health checks with Squid:

```
/cfg/ip/if 1
ena
addr 172.16.102.1
mask 255.255.255.0
broad 172.16.102.255
```

2. Alteon's WCR is a feature of its Server Load Balancing (SLB) configuration. Thus, you need to enable SLB features on the switch with this command:

```
/cfg/slb
on
```

3. Next, you define a real server with Squid's IP address:

```
/cfg/slb/real 1
ena
rip 172.16.102.66
```

4. You must also define a group and make the real server a member:

```
/cfg/slb/group 1
health tcp
add 1
```

5. The next step is to define two filters. The first filter matches HTTP connections—TCP packets with destination port 80—and redirects them to a server in group 1. The second filter matches all other packets and forwards them normally:

```
/cfg/slb/filt 1
ena
action redir
sip any
smask 0.0.0.0
dip any
dmask 0.0.0.0
proto tcp
sport any
dport http
group 1
rport 0
/cfg/slb/filt 224
ena
action allow
sip any
smask 0.0.0.0
dip any
dmask 0.0.0.0
proto any
```

6. The final step is to configure specific switch ports for SLB. On port 1, you enable client processing (this is where the clients connect), and add the two filters. On the second port you need only configure it for servers (i.e., the upstream Internet connection):

```
/cfg/slb/port 1
client ena
filt ena
add 1
add 224
/cfg/slb/port 2
server ena
```

To verify that HTTP interception is configured and working correctly, you can use the commands under the `/stats/slb` and `/info/slb` menus. The `/info/slb/dump` command is a quick and easy way to see the entire SLB configuration:

```

>> Main# /info/slb/dump
Real server state:
  1: 172.16.102.66, 00:c0:4f:23:d7:05, vlan 1, port 3, health 3, up

Virtual server state:

Redirect filter state:
  1: dport http, rport 0, group 1, health tcp, backup none
     real servers:
       1: 172.16.102.66, backup none, up

Port state:
  1: 0.0.0.0, client
     filt enabled, filters: 1 224
  2: 0.0.0.0, server
     filt disabled, filters: empty
  3: 0.0.0.0
     filt disabled, filters: empty

```

In this output, notice that the switch says Squid is reachable via port 3 and that the health checks show Squid is up. You can also see that filter 1 has been applied to port 1, where the clients connect. In the Port state section, port 1 is designated as a place where clients connect, and port 2 is similarly marked as a server port.

The `/stats/slb/real` command shows a handful of statistics for the real server (i.e., Squid):

```

>> Main# /stats/slb/real 1
-----
Real server 1 stats:
Health check failures:      0
Current sessions:          41
Total sessions:            760
Highest sessions:          55
Octets:                    0

```

Most of the statistics relate to the number of sessions (i.e., TCP connections). The Total sessions counter should increase if you execute the command again.

Lastly, the `/stats/slb/group` command shows almost the same information:

```

>> Main# /stats/slb/group 1
-----
Real server group 1 stats:

```

Real IP address	Current Sessions	Total Sessions	Highest Sessions	Octets
1 172.16.102.66	65	2004	90	0
	65	2004	90	0

This output would be more interesting if there was more than one real server in the group.

Foundry

The configuration in the following example comes from a ServerIron XL, running software version 07.0.07T12. As before, clients are on port 1, the Internet link is on port 2, and Squid is on port 3. However, that matters less for this particular configuration because you can enable HTTP interception globally. Foundry's name for interception caching is Transparent Cache Switching (TCS). Refer back to Figure 9-4 for this example.

The first step is to give the switch an IP address so it can perform health checks:

```
ip address 172.16.102.1 255.255.255.0
```

Foundry allows you to enable or disable TCS on particular ports. However, for the sake of simplicity, let's enable it globally:

```
ip policy 1 cache tcp http global
```

In this line, `cache` is a keyword that corresponds to the TCS feature. The next line defines a web cache. I've given it the name *squid1* and told the switch its IP address:

```
server cache-name squid1 172.16.102.66
```

The final step is to add the web cache to a cache group:

```
server cache-group 1  
cache-name squid1
```

If you're having problems getting the Foundry switch to divert connections, have a look at the `show cache-group` output:

```
ServerIron#show cache-group
```

```
Cache-group 1 has 1 members Admin-status = Enabled Active = 0  
Hash_info: Dest_mask = 255.255.255.0 Src_mask = 0.0.0.0
```

Cache Server Name	Admin-status	Hash-distribution
squid1	6	3

```
HTTP Traffic From <-> to Web-Caches
```

```
Name: squid1          IP: 172.16.102.66   State: 6   Groups = 1
```

	State	CurConn	TotConn	Host->Web-cache Packets	Octets	Web-cache->Host Packets	Octets
Client	active	441	12390	188871	15976623	156962	154750098
Web-Server	active	193	11664	150722	151828731	175796	15853612
Total		634	24054	339593	167805354	332758	170603710

Some of this output is cryptic, but you can tell interception is working by repeating the command and watching the counters increase.

The `show server real` command provides almost the same information:

```
ServerIron#show server real squid1
Real Servers Info
```

```
Name : squid1                               Mac-addr: 00c0.4f23.d705
IP:172.16.102.66  Range:1  State:Active      Wt:1      Max-conn:1000000
Src-nat (cfg:op):(off:off)  Dest-nat (cfg:op):(off:off)
squid1 is a TRANSPARENT CACHE in groups 1
Remote server : No          Dynamic : No      Server-resets:0
Mem:server: 02009eae Mem:mac: 045a3714
```

Port	State	Ms	CurConn	TotConn	Rx-pkts	Tx-pkts	Rx-octet	Tx-octet	Reas
http	active	0	855	29557	379793	471713	373508204	39425322	0
default	active	0	627	28335	425106	366016	38408994	368496301	0
Server	Total		1482	57892	804899	837729	411917198	407921623	0

Finally, you can use the show logging command to see if the switch believes Squid is up or down:

```
ServerIron#show logging
...
00d00h11m51s:N:L4 server 172.16.102.66 squid1 port 80 is up
00d00h11m49s:N:L4 server 172.16.102.66 squid1 port 80 is down
00d00h10m21s:N:L4 server 172.16.102.66 squid1 port 80 is up
00d00h10m21s:N:L4 server 172.16.102.66 squid1 is up
```

Note that the ServerIron thinks the server is running on port 80. As you'll see later, my examples have Squid running on port 3128. The packet filtering rules actually change the packet's destination port from 80 to 3128. This has some interesting consequences for health checks, which I address later in the section "A comment on HTTP servers and health checks."

Extreme Networks

In this example, the hardware is a Summit1i, and the software is Version 6.1.3b11. Once again, the clients are on port 1, the Internet link is on port 2, and Squid is on port 3. The network configuration is shown in Figure 9-5.

The Extreme switch can intercept HTTP connections only for packets that it routes between subnets. In other words, if you use the Extreme switch in layer two mode (with a single VLAN), you can't divert traffic to Squid. To make HTTP interception work, you must configure separate VLANs for users, Squid, and the Internet:

```
configure Default delete port 1-8

create vlan Users
configure Users ip 172.16.102.1 255.255.255.192
configure Users add port 1

create vlan Internet
configure Internet ip 172.16.102.129 255.255.255.192
configure Internet add port 2
```

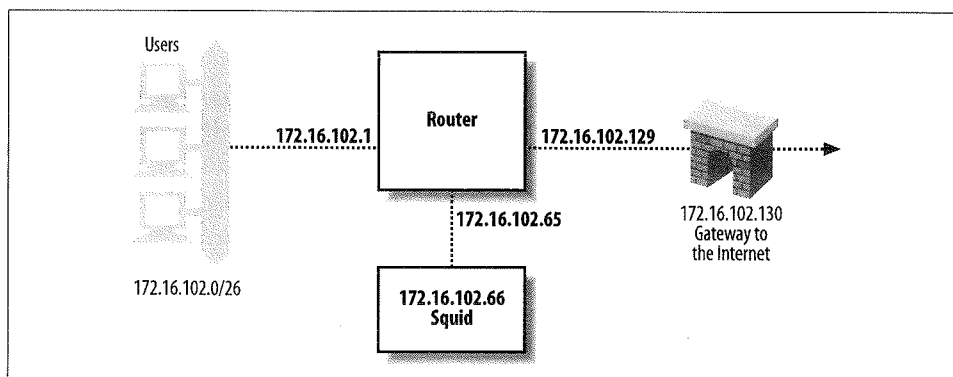


Figure 9-5. Sample network for intercepting with a router, for the Extreme and Cisco policy routing examples

```
create vlan Squid
configure Squid ip 172.16.102.65 255.255.255.192
configure Squid add port 3
```

The next step is to enable and configure routing in the switch:

```
enable ipforwarding
configure iproute add default 172.16.102.130
```

Lastly, you configure the switch to redirect HTTP connections to Squid:

```
create flow-redirect http tcp destination any ip-port 80 source any
configure http add next-hop 172.16.102.66
```

Cisco Arrowpoint

The following configuration is based on notes from an old test I ran. However, I don't have access to an arrowpoint switch now and can't verify that these lines are correct.

```
circuit VLAN1
  ip address 172.16.102.1 255.255.255.0

service pxy1
  type transparent-cache
  ip address 172.16.102.66
  port 80
  protocol tcp
  active

owner foo
  content bar
  add service pxy1
  protocol tcp
  port 80
  active
```

A comment on HTTP servers and health checks

I've set up these examples so that the router/switch forwards packets without changing the destination TCP port. The packet filtering rules that I'll cover in the section "Operating System Tweaks" change the destination port. An interesting problem arises when you also run an HTTP server on the Squid box.

To run an HTTP server on port 80 while running Squid on port 3128, your packet filter configuration must have a special rule that accepts TCP connections for the HTTP server. Otherwise, the connection gets diverted to Squid. The special rule is simple to construct. If the destination port is 80, and the destination address is the server's, accept the packet normally. All the intercepted packets have foreign destination addresses, so they won't match the special rule.

However, when the router/switch makes an HTTP health check, it connects to the server's IP address. Thus, the health-check packet matches the special rule and isn't diverted to Squid. The router/switch is checking the health of the wrong server. If the HTTP server is down, but Squid is up (or vice versa), the health check will be wrong.

If you find yourself in this situation, you have a few options:

- Don't run an HTTP server on the Squid host.
- Add a specific packet filtering rule that diverts TCP health check connections from the router/switch to Squid.
- Configure your router/switch to change the destination port to 3128.
- Disable layer four health checks.

Cisco Policy Routing

Policy routing isn't that different from what I've talked about with layer four switches. It is implemented in routing products made by Cisco and others. The primary difference is that policy routing doesn't include any health checking. Thus, if Squid becomes overloaded or fails entirely, the router continues to forward packets to Squid, rather than route them directly to origin servers. Policy routing requires that Squid be on one of the router's directly connected subnets.

In this example, I'm using a Cisco 7204 router running IOS Version 12.0(5)T. The network configuration is the same as the previous example, shown in Figure 9-5.

The first configuration step is to define an *access list* that matches port 80 packets coming from clients. You must make sure that port 80 packets coming from Squid aren't reintercepted. One way to do this is with a specific rule that denies packets coming from Squid, followed by a rule that allows all others:

```
access-list 110 deny tcp host 172.16.102.66 any eq www
access-list 110 permit tcp any any eq www
```

Alternatively, if Squid and your users are on different subnets, you can permit only those packets that originate from the client network:

```
access-list 110 permit tcp 10.102.0.0 0.0.255.255 any eq www
```

The next step is to define a *route map*. This is where you tell the router where to forward the intercepted packets:

```
route-map proxy-redirect permit 10
match ip address 110
set ip next-hop 172.16.102.66
```

Those commands say, “If the IP address matches access-list 110, forward the packet to 172.16.102.66.” The 10 on the route-map line is a sequence number in case you have multiple route maps. The final step is to apply the route map to interfaces where your clients connect:

```
interface Ethernet0/0
ip policy route-map proxy-redirect
```

IOS doesn’t provide much in the way of debugging for policy routing. However, the `show route-map` command may be sufficient:

```
router#show route-map proxy-redirect
route-map proxy-redirect, permit, sequence 10
Match clauses:
  ip address (access-lists): 110
Set clauses:
  ip next-hop 172.16.102.66
Policy routing matches: 730 packets, 64649 bytes
```

Web Cache Coordination Protocol

Cisco’s answer to layer four switching technology (before they acquired Arrowpoint) is the Web Cache Coordination Protocol (WCCP).^{*} WCCP is different from the typical layer four interception in a couple of ways.

First, intercepted packets are encapsulated with GRE (Generic Routing Encapsulation). This simply allows them to traverse subnets, which means Squid doesn’t need to be directly connected to the router. Because they are encapsulated, the Squid host must unencapsulate them. Not all Unix systems have the code for unwrapping GRE packets.

The second difference is in how the router decides to spread the load among multiple caches. In fact, the router doesn’t make this decision, the cache does. When a router has a group of WCCP-enabled caches, one nominates itself to be the leader. The leader decides how to spread the load and informs the router. This is an extra step that must occur before the router can redirect any connections.

^{*} At various times it has also been called Web Cache Control Protocol.

Because WCCP uses GRE, the router may be forced to fragment large TCP packets from HTTP requests. Fortunately, this shouldn't occur very often because most HTTP requests are smaller than the Ethernet MTU size (1500 octets). The default TCP and IP packet headers are 20 octets each, which means an Ethernet frame can carry 1460 octets of actual data. GRE encapsulation adds 20 octets for the GRE header, plus another 20 for the second IP header. Thus a normal 1500-octet TCP/IP packet from the client becomes 1540 octets after encapsulation. This is too large to transmit in a single Ethernet frame, so the router fragments the original packet into two packets.

WCCPv1

The configuration examples in this section were tested on a Cisco 7204 running IOS Version 12.0(5)T. The network configuration is the same as shown in Figure 9-5.

First, enter these two lines in the IOS configuration to enable WCCP for the router:

```
ip wccp version 1
ip wccp web-cache
```

Second, you must enable WCCP on individual router interfaces. You should do this only on interfaces where HTTP packets leave the router. In other words, select interfaces that connect to origin servers or your Internet gateway:

```
interface Ethernet0/1
ip address 172.16.102.129 255.255.255.192
ip wccp web-cache redirect out
```

Be sure to save your configuration changes.

You may need to use an access list to prevent interception for certain web sites. You can also use the access list to prevent forwarding loops. For example:

```
! don't re-intercept connections coming from Squid:
access-list 112 deny tcp host 172.16.102.66 any eq www

! don't intercept this broken web site
access-list 112 deny tcp any 192.16.8.7 255.255.255.255 eq www

! allow other HTTP traffic
access-list 110 permit tcp any any eq www

ip wccp web-cache redirect-list 112
```

The router doesn't send any traffic to Squid until Squid announces itself to the router. I explain how to configure Squid for WCCP in the section "Configuring WCCPv1."

WCCPv2

The standard Squid distribution currently only supports WCCPv1. However, you can find a patch for WCCPv2 on the <http://devel.squid-cache.org/> site. This code is still experimental.

Note that the GRE packets sent from the router to Squid contain an additional four octets. WCCPv2 inserts a redirect header between the GRE header, and the encapsulated IP packet. You may need to modify your kernel code to account for this additional header.

Debugging

IOS provides a couple of commands to monitor and debug WCCP. The `show ip wccp web-cache` command provides some basic information:

```
router#show ip wccp web-cache
Global WCCP information:
  Router information:
    Router Identifier:      172.16.102.129
    Protocol Version:      1.0

  Service Identifier: web-cache
    Number of Cache Engines: 1
    Number of routers:      1
    Total Packets Redirected: 1424
    Redirect access-list:    -none-
    Total Packets Denied Redirect: 0
    Total Packets Unassigned: 0
    Group access-list:      -none-
    Total Messages Denied to Group: 0
    Total Authentication failures: 0
```

For a few more details, add the word `detail` to the end of the previous command:

```
router#show ip wccp web-cache detail
WCCP Cache-Engine information:
  IP Address:      172.16.102.66
  Protocol Version: 0.4
  State:           Usable
  Initial Hash Info: 00000000000000000000000000000000
                    00000000000000000000000000000000
  Assigned Hash Info: FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
                    FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
  Hash Allotment:   256 (100.00%)
  Packets Redirected: 1424
  Connect Time:     00:17:40
```

Here you can see Squid's IP address and state. If more than one cache speaks WCCP to the router, the hash assignment information should look different. Most likely, each cache receives an equal proportion of the hash buckets.

Note that the detailed output has a Protocol Version line with a different value than the first command. Unfortunately, the word “version” is overloaded. The `show ip wccp web-cache` command appears to report the WCCP protocol major version number (i.e., 1 or 2), while the `detaill version` seems to be a different (perhaps internal, or minor version) number that matches the value of Squid’s `wccp_version` directive.

Operating System Tweaks

You must enable certain networking features in your operating system to make interception caching work. First, you need to enable IP packet forwarding. This allows the operating system to receive packets with foreign destination addresses. Second, you must enable and configure optional code in the kernel that redirects the foreign packets to Squid.

Linux

The instructions in this section should work for the 2.4 series of Linux kernels. I used RedHat Linux 7.2 (kernel 2.4.7-10). If you are using an older or newer version, these may not work. I recommend searching the Squid FAQ and other places for updated or historical information.

In my tests with iptables, it wasn’t necessary to enable IP forwarding. However, you may want to enable it initially and see if you can disable it after everything else is working. The best way to enable packet forwarding is to add this line to `/etc/sysctl.conf`:

```
net.ipv4.ip_forward = 1
```

Most likely you’ll need to make a new kernel before HTTP interception will work. See O’Reilly’s *Running Linux* by Matt Welsh, Matthias Kalle Dalheimer, and Lar Kaufman, if you don’t know how to configure and create a Linux kernel. When you configure the kernel, make sure these options are enabled:

- o General setup
 - Networking support (CONFIG_NET=y)
 - Sysctl support (CONFIG_SYSCTL=y)
- o Networking options
 - Network packet filtering (CONFIG_NETFILTER=y)
 - TCP/IP networking (CONFIG_INET=y)
 - Netfilter Configuration
 - Connection tracking (CONFIG_IP_NF_CONNTRACK=y)
 - IP tables support (CONFIG_IP_NF_IPTABLES=y)
 - Full NAT (CONFIG_IP_NF_NAT=y)
 - REDIRECT target support (CONFIG_IP_NF_TARGET_REDIRECT=y)
- o File systems
 - /proc filesystem support (CONFIG_PROC_FS=y)

Additionally, make sure this option isn't enabled:

- o Networking options
 - Fast switching (CONFIG_NET_FASTROUTE=n)

The code that redirects foreign packets to Squid is part of the Netfilter software. Here is a rule that sends the intercepted HTTP connections to Squid:

```
iptables -t nat -A PREROUTING -i eth0 -p tcp --dport 80 -j REDIRECT --to-port 3128
```

The Linux kernel maintains a number of different *tables*. The `-t nat` option indicates that we are modifying the Network Address Translation (NAT) table. In essence, we're using iptables to translate origin server TCP/IP addresses to Squid's local TCP/IP address.

Each iptables table has a number of *chains*. The `-A PREROUTING` option indicates that we are appending a rule to the built-in chain named PREROUTING. The PREROUTING chain applies only to packets entering the system from the outside network.

The next three options determine which packets match this rule. The `-i eth0` option restricts the rule to packets received on the eth0 interface. The `-p tcp` option specifies TCP packets, and `--dport 80` specifies packets with destination port equal to 80. If all three conditions are true, the packet matches the rule.

The `-j REDIRECT` option indicates the target, or action to take, for packets that match the rule. REDIRECT is a built-in target name that causes iptables to change the packet's destination address to 127.0.0.1. The `--to-port 3128` option instructs iptables also to change the destination TCP port number to 3128.

If you are also running an HTTP server (such as Apache) on the Squid host, you must add another iptables rule. The additional rule is necessary to allow connections to your HTTP server. Otherwise, the REDIRECT rule causes iptables to send those connections to Squid on port 3128. You can use the `-I` option to insert a new rule at the top of the list:

```
iptables -t nat -I PREROUTING -i eth0 -p tcp -d 172.16.102.66 --dport 80 -j ACCEPT
```

Once you have all your iptables rules working correctly, be sure to save them with this command:

```
/sbin/service iptables save
```

This saves the current rules to `/etc/sysconfig/iptables` so they get automatically loaded when you reboot.

Linux and WCCP

Version 2.4 of the Linux kernel comes with a GRE pseudo-interface. However, it doesn't work for decoding GRE-encapsulated packets from a WCCP session. The problem seems to be that the router sets the Protocol Type field to 0x883E for WCCP/GRE packets. Linux's GRE driver doesn't know what to do with these packets because it doesn't know about protocol type 0x883E.

You can try patching Linux's GRE module so that it works with WCCP. The Squid FAQ contains a link to such a patch. However, you'll probably find it easier to use the WCCP-specific module for Linux. You can find it at http://www.squid-cache.org/WCCP-support/Linux/ip_wccp.c.

You need to compile the *ip_wccp.c* file as a loadable kernel module. This can be a little tricky because the specific compiler options may change depending on your kernel version. One thing you can do is go to your kernel source directory, type `make modules` and watch the compiler commands scroll by. Then copy one of those commands and change the last argument to *ip_wccp.c*. Here are the commands that I used with the 2.4.7-10 Linux kernel:

```
% gcc -Wall -D__KERNEL__ -I/usr/src/linux-2.4.7-10/include \
-DMODULE -DMODVERSIONS -DEXPORT_SYMBAB \
-include /usr/src/linux-2.4.7-10/include/linux/modversions.h \
-O2 -c ip_wccp.c
```

The gcc command should leave you with an *ip_wccp.o* file in the current directory. The next step is to load that file into the kernel with the `insmod` command:

```
# insmod ip_wccp.o
```

Note that the *ip_wccp* module accepts GRE/WCCP packets from any source address. In other words, a malicious person might be able to send traffic to your Squid cache. If you use this module, you should also install an iptables rule to deny foreign GRE packets. For example:

```
# iptables -A INPUT -p gre -s 172.16.102.65 -j ACCEPT
# iptables -A INPUT -p gre -j DROP
```

Again, don't forget to save your working rules with the `/sbin/service iptables save` command.

FreeBSD

The examples in this section are based on FreeBSD-4.8 and should work for any later version of FreeBSD-4 and FreeBSD-5.

To enable IP packet forwarding, add this line to */etc/sysctl.conf*:

```
net.inet.ip.forwarding=1
```

You'll need a kernel with two special options enabled. If you don't know how to make a kernel, refer to Section 9 of the FreeBSD Handbook (<http://www.freebsd.org/handbook/index.html>). Edit your kernel config file and make sure these lines are present:

```
options      IPFIREWALL
options      IPFIREWALL_FORWARD
```

If the Squid box is in an unattended machine room, I also recommend using the `IPFWALL_DEFAULT_TO_ACCEPT` option. In case you mess up the firewall rules, you'll still be able to log in.

These `ipfw` commands tell the kernel to redirect intercepted connections to Squid:

```
/sbin/ipfw add allow tcp from 172.16.102.66 to any out
/sbin/ipfw add allow tcp from any 80 to any out
/sbin/ipfw add fwd 127.0.0.1,3128 tcp from any to any 80 in
/sbin/ipfw add allow tcp from any 80 to 172.16.102.66 in
```

The first rule matches packets originating from the Squid host. It ensures that outgoing TCP connections won't be redirected back to Squid.* The second rule matches TCP packets sent from Squid back to the clients. I've added it here in case you have additional `ipfw` rules later that would deny these packets. The third rule is the one that actually redirects incoming connections to Squid. The fourth rule matches packets coming back from origin servers to Squid. Again, this is in case you have subsequent deny rules.

If you're also running an HTTP server on the Squid host, you must add another rule that passes, rather than redirects, TCP packets destined for the origin server. The following rule goes before the `fwd` rule:

```
/sbin/ipfw add allow tcp from any to 172.16.102.66 80 in
```

FreeBSD typically stores `ipfw` rules in `/etc/rc.firewall`. Once you get your rule set working properly, be sure to save them. Add this line to `/etc/rc.conf` to make FreeBSD automatically run the `/etc/rc.firewall` script when it boots:

```
firewall_enable="YES"
```

FreeBSD and WCCP

FreeBSD Version 4.8 and later have built-in support for GRE and WCCP. Earlier versions require patches, which you can still find at <http://www.squid-cache.org/WCCP-support/FreeBSD/>. The built-in implementation is much better, however, as it is written by real kernel gurus. You'll probably need to make a new kernel that supports GRE. Add this line to your kernel configuration:

```
pseudo-device  gre
```

For FreeBSD-5, use `device` instead of `pseudo-device`. Of course, you also need the `FIREWALL` options mentioned in the preceding section.

After installing and booting from the new kernel, you must configure a GRE tunnel to accept GRE packets from the router. For example:

```
# ifconfig gre0 create
# ifconfig gre0 172.16.102.66 172.16.102.65 netmask 255.255.255.255 up
```

* Although a misconfiguration on the switch/router may still reintercept these packets.

```
# ifconfig gre0 tunnel 172.16.102.66 172.16.102.65
# route delete 172.16.102.65
```

The `ifconfig` command adds a routing table entry for the router (172.16.102.65) over the `gre0` interface. I found it necessary to delete that route so that Squid can talk to the router.

You may want or need to add an `ipfw` rule for the GRE packets coming from the router:

```
/sbin/ipfw add allow gre from 172.16.102.65 to 172.16.102.66
```

OpenBSD

The examples in this section are based on OpenBSD 3.3.

To enable packet forwarding, uncomment or add this line in `/etc/sysctl.conf`:

```
net.inet.ip.forwarding=1
```

Now, configure the packet filter rules for interception by adding lines like these to `/etc/pf.conf`:

```
 rdr inet proto tcp from any to any port = www -> 127.0.0.1 port 3128
  pass out proto tcp from 172.16.102.66 to any
  pass out proto tcp from any port = 80 to any
  pass in proto tcp from any port = 80 to 172.16.102.66
```

If you aren't already using OpenBSD's packet filter, you need to enable it with this line in `/etc/rc.conf.local`:

```
pf=YES
```

OpenBSD and WCCP

First, tell the system to accept and process GRE and WCCP packets by adding these lines to `/etc/sysctl.conf`:

```
net.inet.gre.allow=1
net.inet.gre.wccp=1
```

Then, configure a GRE interface with commands like these:

```
# ifconfig gre0 172.16.102.66 172.16.102.65 netmask 255.255.255.255 up
# ifconfig gre0 tunnel 172.16.102.66 172.16.102.65
# route delete 172.16.102.65
```

As with FreeBSD, I found it necessary to delete the route that is automatically added by `ifconfig`. Finally, depending on your packet filter configuration, you may need to add a rule that allows the GRE packets:

```
pass in proto gre from 172.16.102.65 to 172.16.102.66
```

IPFilter on NetBSD and Others

The examples in this section are based on NetBSD 1.6.1. They might also work on Solaris, HP-UX, IRIX, and Tru64 since IPFilter runs on those systems as well.

To enable packet forwarding (on NetBSD), add this line to */etc/sysctl.conf*:

```
net.inet.ip.forwarding=1
```

Then, insert a line like this into the NAT (network address translation) configuration file, */etc/ipnat.conf*:

```
rdr fxp0 0/0 port 80 -> 172.16.102.66 port 3128 tcp
```

Your interface name may be different from *fxp0* in this example.

NetBSD and WCCP

I was not able to make WCCP work with NetBSD, even after patching the GRE code to accept WCCP packets. The problem seems to arise because the IPFilter *rdr* rule is bound to a specific interface. Packets coming from the router go through NetBSD's *gre0* interface (where they are unencapsulated). However, packets going the other way, back to the router, aren't encapsulated and don't go through the same network interface. Therefore, the IPFilter code doesn't translate Squid's local IP address back to the origin server's address.

Configure Squid

If you are using Linux 2.4 and *iptables*, you should probably use the *--enable-linux-netfilter* option when you run (or re-run) *./configure*. It enables some Linux-specific code so that Squid can find the IP address of the origin server from where the request was originally sent. Squid normally gets the origin server name (and/or address) from the *Host* header. The *--enable-linux-netfilter* feature is necessary only for requests that don't have a *Host* header. Statistics show that almost all requests have the *Host* header, so you may actually be able to get by without the *--enable-linux-netfilter* option.

If you are using the IPFilter package (with NetBSD, Solaris, and others), you should use the *--enable-ipf-transparent* option for the same reason. On OpenBSD, you should use the *--enable-pf-transparent* option. Each time you run *./configure* you must recompile Squid, as described in Chapter 3.

After you get the *./configure* options figured out, and Squid recompiled, you can edit *squid.conf*. As a starting point, make sure the following directives are defined with the given values:

```
httpd_accel_host virtual
httpd_accel_port 80
httpd_accel_uses_host_header on
```

```
httpd_accel_with_proxy on
httpd_accel_single_host off
```

The `httpd_accel_host` directive is the key. It instructs Squid to accept HTTP requests with partial URIs. The `httpd_accel_uses_host_header` directive is enabled so that Squid uses the Host header to reconstruct full URIs. The virtual keyword instructs Squid to put the origin server's IP address in the URL when the Host header is absent.

The `httpd_accel_with_proxy` directive controls whether or not Squid accepts both HTTP server (partial URI) requests, and proxy (full URI) requests. It should probably be enabled for interception caching. Squid may still work if `httpd_accel_with_proxy` is disabled as long as none of your clients are explicitly configured for Squid as a proxy.

The `httpd_accel_single_host` directive is normally disabled, but it was enabled by default in some earlier versions of Squid. I've listed it here to make sure that it is disabled for interception caching.

If you are intercepting more than just port 80, you may want to set `httpd_accel_port` to 0. See Appendix A for more information.

If you're not using WCCP, you should be ready to start sending intercepted traffic to Squid. Give it a try by surfing the Web with your browser or by making some test requests with `squidclient`. If you are using WCCP, there is just one more step that you must complete.

Configuring WCCPv1

The router doesn't send any traffic to Squid until Squid announces itself to the router. To make Squid do that, add these lines to your `squid.conf`:

```
wccp_router 172.16.102.65
wccp_version 4
```

Your router has many interfaces. Be sure to use the IP address of the interface closest to Squid. This is necessary because the WCCP messages coming from the router have the source IP address set to the address of the outgoing interface. Squid rejects WCCP messages if the source address doesn't match the `wccp_router` value.

The WCCPv1 document specifies 4 as the protocol version number. However, some users report that Cisco IOS 11.2 supports only Version 3. If you are using this version of IOS, change the version in `squid.conf`:

```
wccp_version 3
```

Debugging Problems

HTTP interception is complicated because many different devices must all work correctly together. To help you track down problems, here's a trouble-shooting check list:

Are client packets going through the router/switch?

This should be obvious for simple networks. You can trace the cables and watch the activity lights blink. In a large, complex network, however, packets may be taking an alternate path. If your organization is large enough to have a network sniffer, you may want to observe the traffic on the link that should carry requests from web clients. A low-tech approach is to disconnect the link in question and see if it affects the client's web browsing.

Is the router/switch configured properly?

You may want to double-check your router/switch configuration. If you configured specific interfaces, did you get the right ones?

Is your new configuration actually *running* on the device? Perhaps the router/switch was rebooted before you could save the configuration. You may need to reboot before the changes take effect.

Can the switch/router talk to the Squid host?

Can you ping Squid from the router/switch? Most layer four interception configurations require that the device and Squid be on the same subnet. Log into the router/switch, and make sure you can ping Squid's IP address.

Does the switch/router believe that Squid is up?

Many traffic interception devices don't send traffic to Squid unless they know it's healthy. Use the debugging commands to view Squid's health status. You may find that a layer three health check (e.g., ICMP ping) is simpler than a layer four check (e.g., HTTP), and more likely to make the network device mark Squid as up.

Is Squid actually running?

Double-check that Squid is really running, especially if the system has recently been rebooted.

Are packets arriving at the Squid host?

You should be able to see intercepted TCP connections with `tcpdump`. Here's an example:

```
# tcpdump -n -i eth0 port 80
```

If you use WCCP, check for GRE packets coming from the router:

```
# tcpdump -n -i eth0 ip proto gre
```

If you don't see any output from `tcpdump`, the router/switch is probably not sending anything. In that case, return to the previous suggestions.

Note, if the device is using layer four health checks, you should see those in the `tcpdump` output. Health checks come from the router/switch IP address, so they should be easy to spot. If you see health checks, but no other traffic, it probably means the router/switch is interpreting Squid's reply as unhealthy. For example, the device may want to see a 200 (OK) response, but Squid returns an error, such as 401 (Unauthorized) or 404 (Not Found). You may want to run `tail -f` on the `access.log`.

Did you enable IP forwarding?

Double-check that Squid's operating system is configured to forward IP packets. If not, the host may drop intercepted packets because the destination IP address isn't local.

Did you configure the packet filter?

Make sure that the packet filter (i.e., `ipfw`, `iptables`, `pf`, etc.) is configured correctly. When everything is working well, you should be able to run the command periodically that displays the filtering rules and see the counters increase. For example:

```
# ipfw show 300 ; sleep 3; ipfw show 300
00300  86216 8480458 fwd 127.0.0.1,3128 tcp from any to any 80 in
00300  86241 8482240 fwd 127.0.0.1,3128 tcp from any to any 80 in
```

Note that in this example on FreeBSD, the packet and byte counters (second and third columns) are being incremented.

Is the loopback interface up and configured?

If you have a rule to forward/redirect packets to 127.0.0.1, make sure that the loopback (e.g., `lo0`, `lo`) interface is up and configured. If not, the kernel may simply skip the forward/redirect rule.

Are WCCP/GRE packets being unencapsulated correctly?

If you use WCCP, make sure that the GRE packets are being unencapsulated. If, for some reason, your system doesn't know what to do with GRE packets, it probably increments the "unknown/unsupported protocol" counter in `netstat -s` output:

```
# netstat -s | grep unknown
46 packets for unknown/unsupported protocol
```

If your OS has a GRE interface, run `netstat -i` every so often and look for increasing packet counts:

```
# netstat -in | grep ^gre0
Name      Mtu Network      Address      IpKts Ierrs   Opkts Oerrs  Coll
gre0      1476 <Link#4>      304452      0        0      4      0
```

Also, try running `tcpdump` on the GRE interface:

```
# tcpdump -n -i gre0
```

Can Squid talk back to the clients?

You may have a situation in which the router/switch is able to send packets to Squid, but Squid can't send packets back to the clients. This can happen if your

firewall filter rules reject those outgoing packets or if Squid just doesn't have a route to the client addresses. To check for this condition, run `netstat -n` and look for a lot of sockets in the `SYN_RCVD` state:

```
% netstat -n
Active Internet connections
Proto Recv-Q Send-Q Local Address      Foreign Address    (state)
tcp4      0      0 10.102.129.246.80  10.102.0.1.36260  SYN_RCVD
tcp4      0      0 10.102.129.226.80  10.102.0.1.36259  SYN_RCVD
tcp4      0      0 10.102.128.147.80  10.102.0.1.36258  SYN_RCVD
tcp4      0      0 10.102.129.26.80   10.102.0.2.36257  SYN_RCVD
tcp4      0      0 10.102.129.29.80   10.102.0.2.36255  SYN_RCVD
tcp4      0      0 10.102.129.226.80  10.102.0.1.36254  SYN_RCVD
tcp4      0      0 10.102.128.117.80  10.102.0.1.36253  SYN_RCVD
tcp4      0      0 10.102.128.149.80  10.102.0.1.36252  SYN_RCVD
```

If you see this, use `ping` and `traceroute` to make sure that Squid has bidirectional communication with the clients.

Can Squid talk to origin servers?

Intercepted HTTP connections get stuck if Squid can't connect to origin servers. When this happens, `netstat` should show you a lot of connections in the `SYN_SENT` state:

```
% netstat -n
Active Internet connections
Proto Recv-Q Send-Q Local Address      Foreign Address    (state)
tcp4      0      0 172.16.102.66.5217 10.102.129.145.80  SYN_SENT
tcp4      0      0 172.16.102.66.5216 10.102.129.224.80  SYN_SENT
tcp4      0      0 172.16.102.66.5215 10.102.128.71.80   SYN_SENT
tcp4      0      0 172.16.102.66.5214 10.102.129.209.80  SYN_SENT
tcp4      0      0 172.16.102.66.5213 10.102.129.62.80   SYN_SENT
tcp4      0      0 172.16.102.66.5212 10.102.129.160.80  SYN_SENT
tcp4      0      0 172.16.102.66.5211 10.102.128.129.80  SYN_SENT
tcp4      0      0 172.16.102.66.5210 10.102.129.44.80   SYN_SENT
tcp4      0      0 172.16.102.66.5209 10.102.128.73.80   SYN_SENT
tcp4      0      0 172.16.102.66.5208 10.102.128.43.80   SYN_SENT
```

Again, use `ping` and `traceroute` to make sure that Squid can talk to origin servers.

Are outgoing connections being intercepted?

If Squid can ping origin servers, and you still see a lot of connections in the `SYN_SENT` state, the router/switch may be intercepting Squid's outgoing TCP connections. In some cases, Squid can detect such forwarding loops, and it writes a warning message to `cache.log`. Such a forwarding loop can quickly exhaust all of Squid's file descriptors, which also generates a warning in `cache.log`.

If you suspect this problem, use the `squidclient` program to make some simple HTTP requests. For example, this command makes an HTTP request directly to the origin server:

```
% /usr/local/squid/bin/squidclient -p 80 -h slashdot.org /
```

If this command succeeds, you should see a bunch of ugly HTML from the Slashdot site on your screen. You can then try the same request through Squid:

```
% /usr/local/squid/bin/squidclient -r -p 3128 -h 127.0.0.1 http://slashdot.org/
```

Again, you should see some HTML on your screen. If not check for error messages in *cache.log*. If you see forwarding loop errors, you need to reconfigure your router/switch so that it allows Squid's outgoing connections to pass without being intercepted.

Exercises

- Try running Squid with a bogus *httpd_accel_host* value. For example:
`httpd_accel_host blah.blah.blah`
Does it still work, or do you get error messages?
- Disconnect Squid's network connection while your router/switch is diverting traffic to it. Does the network device bypass Squid? How long does it take to notice the problem?
- Repeat the same experiment, but this time kill the Squid process instead of unplugging the network cable.
- Enable Squid's user-agent log and see if you are intercepting any nonbrowser web traffic.

CHAPTER 10

Talking to Other Squids

For one reason or another, you may find that you want Squid to forward its cache misses to another cache or HTTP proxy. This is necessary, for example, if you are using Squid inside a large corporate network that has one or more firewalls protecting you from the outside world. If your caching service is actually a cluster of Squid caches, you probably want them to cooperate with each other to minimize duplication of cached responses. You can also use Squid as a *content router*—routing web traffic in different directions based on some aspect of the request. Or, perhaps you'd like to participate in an informal collection of caches to further improve response time and reduce wide-area network traffic.

Intercache communication is a complex undertaking, and Squid has numerous features and protocols to accomplish the task. After explaining some of the terminology and discussing the issues, I'll introduce the configuration file directives that control request routing. Following that I describe the nifty network measurement database.

Most likely, you'll use one or more of Squid's intercache protocols to assist in communicating with the other caches or proxies. The Internet Cache Protocol (ICP) is the oldest but not necessarily the best. It is widely implemented in non-Squid products, so you may need to use it for that reason alone. The newer protocols are Cache Digests, the Hypertext Caching Protocol (HTCP), and the Cache Array Routing Protocol (CARP).

There are many choices here, so I'll spend a bit of time explaining how everything works inside Squid.

Some Terminology

Caching hierarchy is the name generally given to a collection of caches (or proxies) that forward requests to one another. We say that the members of the hierarchy are *neighbors* or *peers*.

Neighbor caches have either a *parent* or *sibling* relationship. Topologically, parent caches are one level up in the hierarchy, while siblings are on the same level. The real difference is that parents can forward cache misses for their children. Siblings, on the other hand, aren't allowed to forward cache misses. This means that, before sending a request to a sibling, the originator should know that it will be a cache hit. Inter-cache protocols like ICP, HTCP, and Cache Digests can predict cache hits in neighbors. CARP, however, can't.

Sometimes, cache hierarchies aren't really hierarchical. Consider, for example, a group of five sibling caches. Because there are no parents or children, there is no sense of up or down. In this case, you could call it a *cache mesh*, or even an *array*, instead of a hierarchy.

Why (Not) Use a Hierarchy?

A neighbor cache improves performance by providing some extra fraction of requests as cache hits. In other words, some of the requests that are misses in your cache may be hits in the neighbor cache. If your cache can download these neighbor hits faster than from the origin server, the hierarchy should improve performance overall. The downside is that neighbor caches usually provide only a small percentage of requests as hits. About 5%, or maybe 10% if you're lucky, of your requests that are cache misses will be hits in a neighbor. In some cases, this small benefit doesn't justify the hassle of joining a hierarchy. In other cases, such as networks with poor or overutilized connectivity, hierarchies definitely improve performance for end users.

If you use Squid inside a firewalled network, you may need to configure the firewall proxy as a parent. In this case, Squid forwards every request to the firewall because it can't connect directly to outside origin servers. If you have some origin servers inside the firewall, you can instruct Squid to connect to them directly.

You can also use a hierarchy to send web traffic in different directions. This is sometimes called *application-layer routing*, or more recently, *content routing*. Consider, for example, a large organization with two Internet connections. Perhaps the second connection costs less, or has higher latency, than the other. This organization may want to use the second connection for low-priority traffic, such as downloading binaries, audio and video files, or other kinds of large transfers. Or, perhaps they want to send all HTTP traffic over one link, and non-HTTP traffic over the other. Or, perhaps certain users' traffic should go through the low-priority connection, while premium customers get to use the more expensive link. You can accomplish any of these scenarios with a hierarchy of caching proxies.

Trust is one of the most important issues for the members of a cache hierarchy. You must trust your neighbors to serve correct, unmodified responses. You must trust them with sensitive information, such as the URIs requested by your users. You must

trust that they maintain secure and up-to-date systems to minimize the chances of unauthorized access and denials of service.

Another problem with hierarchies is the way that they normally propagate errors. When a neighbor cache experiences an error, such as an unreachable server, it generates an HTML page that explains the error and its origin. Your users may become confused if they get errors from neighbor caches outside the immediate organization. If the problem persists, they'll have a hard time finding an administrator who can help them.

Sibling relationships are subject to special problem, known as *false hits*. This occurs when Squid sends a request to a sibling, believing it will be a cache hit, but the sibling is unable to satisfy the request without contacting the origin server. False hits happen in a number of circumstances, but usually with a low probability. Furthermore, Squid and other HTTP proxies have features for automatically retrying such requests so that the user isn't even aware of the problem.

A *forwarding loop* is another problem sometimes seen in cache hierarchies. It occurs when Squid forwards a request somewhere, but that request comes back to Squid again, as shown in Figure 10-1.

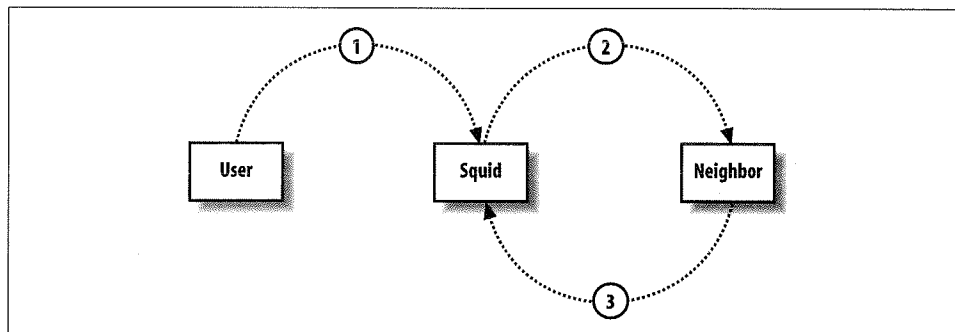


Figure 10-1. A forwarding loop

Forwarding loops typically happen when two caches consider each other parents. If you have such an arrangement, make sure that you use the `cache_peer_access` directive to prevent loops. For example, if the neighbor's IP address is 192.168.1.1, the following lines ensure Squid won't cause a forwarding loop:

```
acl FromNeighbor src 192.168.1.1
cache_peer_access the.neighbor.name deny FromNeighbor
```

Forwarding loops can also occur with HTTP interception, especially if the interception device is on the path between Squid and an origin server.

Squid detects forwarding loops by looking for its own hostname in the `Via` header. You may actually get false forwarding loops if two cooperating caches have the same hostname. The `unique_hostname` directive is useful in this situation. Note that if the

Via header is filtered out (e.g., with *headers_access*), Squid can't detect forwarding loops.

Telling Squid About Your Neighbors

The *cache_peer* directive defines your neighbor caches and tells Squid how to communicate with them:

```
cache_peer hostname type http-port icp-port [options]
```

The first argument is the neighbor's hostname, or IP address. You can safely use hostnames here because Squid doesn't block while resolving them. In fact, Squid periodically resolves the hostname in case the IP address changes while Squid is running. Neighbor hostnames must be unique: you can't have two neighbors with the same name, even if they have different ports.

The second argument specifies the type of neighbor cache. The choices are: *parent*, *sibling*, or *multicast*. Parent and sibling are straightforward. I'll talk about multicast in the later section "Multicast ICP."

The third argument is the neighbor's HTTP port number. It should correspond to the neighbor's *http_port* (or equivalent) setting. You must always specify a nonzero HTTP port number.

The fourth argument specifies either the ICP or HTCP port number. By default, Squid uses ICP to query other caches. That is, Squid sends ICP queries to the neighbor on the port given here. If you add the *htcp* option, Squid sends HTCP queries to this port instead. The default ICP port is 3130, and the default HTCP port is 4827. Make sure that you change the port number if you add the *htcp* option. Setting this port number to zero disables both ICP and HTCP. However, you should instead (or also) use the *no-query* option to disable these protocols.

cache_peer Options

The *cache_peer* directive has quite a few options. I'll describe some of them here, and the others in the sections relating to specific protocols.

proxy-only

This option instructs Squid not to store any responses it receives from the neighbor. This is often useful when you have a cluster and don't want a resource to be stored on more than one cache.

weight=n

This option is specific to ICP/HTCP. See the section "cache_peer options for ICP clients."

ttl=n

This option is specific to multicast ICP. See the section "Multicast ICP."

no-query

This option is specific to ICP/HTCP. See the section “cache_peer options for ICP clients.”

default

This option specifies the neighbor as a suitable choice in the absence of other hints. Squid normally prefers to forward a cache miss to a parent that is likely to have a cached copy of the particular resource. Sometimes Squid won't have any clues (e.g., if you disable ICP/HTCP with `no-query`). In these cases, Squid looks for a parent that has been marked as a default choice.

round-robin

This option is a simple load-sharing technique. It makes sense only when you mark two or more parent caches as `round-robin`. Squid keeps a counter for each parent. When it needs to forward a cache miss, Squid selects the parent with the lowest counter.

multicast-responder

This option is specific to multicast ICP. See the section “Multicast ICP.”

closest-only

This option is specific to ICP/HTCP. See the section “cache_peer options for ICP clients.”

no-digest

This option is specific to Cache Digests. See the section “Cache Digests.”

no-netdb-exchange

This option tells Squid not to request the neighbor's *netdb* database (see the section “The Network Measurement Database”). Note, this refers to the bulk transfer of the RTT measurements, not the inclusion of these measurements in ICP miss replies.

no-delay

This option tells Squid to ignore any delay pools settings for requests to the neighbor. See Appendix C for more information on delay pools.

login=credentials

This option instructs Squid to send HTTP authentication credentials to the neighbor. It has three different formats:

login=user:password

This is the most commonly used form. It causes Squid to add the same username and password in every request going to the neighbor. Your users don't need to enter any authentication information.

login=PASS

Setting the value to `PASS` causes Squid to pass the user's authentication credentials to the neighbor cache. It works only for HTTP basic authentication. Squid doesn't add or modify any authentication information.

If your Squid is configured to require proxy authentication (i.e., with a *proxy_auth* ACL), the neighbor cache must use the same username and password database. In other words, you should use the PASS form only for a group of caches owned and operated by a single organization. This feature is dangerous because Squid doesn't remove the authentication credentials from forwarded requests.

`login=*:password`

With this form, Squid changes the password, but not the username, in requests that it forwards. It allows the neighbor cache to identify individual users, but doesn't expose their passwords. This form is less dangerous than using PASS, but does have some privacy implications.

Use this feature with extreme caution. Even if you ignore the privacy issues, this feature may cause undesirable side effects with upstream proxies. For example, I know of at least one other caching product that only looks at the credentials of the first request on a persistent connection. It apparently assumes (incorrectly) that all requests on a single connection come from the same user.

`connect-timeout=n`

This option specifies how long Squid should wait when establishing a TCP connection to the neighbor. Without this option, the timeout is taken from the global *connect_timeout* directive, which has a default value of 120 seconds. By using a lower timeout, Squid gives up on the neighbor quickly and may try to send the request to another neighbor or directly to the origin server.

`digest-url=url`

This option is specific to Cache Digests. See the section "Cache Digests."

`allow-miss`

This option instructs Squid to omit the Cache-Control: *only-if-cached* directive for requests sent to a sibling. You should use this only if the neighbor has enabled the *icp_hit_stale* directive and isn't using a *miss_access* list.

`max-conn=n`

This option places a limit on the number of simultaneous connections that Squid can open to the neighbor. When this limit is reached, Squid excludes the neighbor from its selection algorithm.

`htcp`

This option designates the neighbor as an HTCP server. In other words, Squid sends HTCP queries, instead of ICP, to the neighbor. Note that Squid doesn't accept ICP and HTCP queries on the same port. When you add this option, don't forget to change the *icp-port* value as well. See the section "Configuring Squid for HTCP." HTCP support requires the `--enable-htcp` option when running `./configure`.

`carp-load-factor=f`

This option makes the neighbor, which must be a parent, a member of a CARP array. The sum of all *f* values, for all parents, must equal 1. I cover CARP in the section “Cache Array Routing Protocol.” CARP support requires the `--enable-carp` option when running `./configure`.

Neighbor State

Squid keeps a variety of statistics and state information about each of its neighbors. One of the most important is whether Squid thinks the neighbor is *alive* (up) or *dead* (down). The neighbor’s alive/dead state affects many aspects Squid’s selection procedures. The algorithm for determining the alive/dead state is a little bit complicated, so I’ll go through it here. If you want to follow along in the source code, look at the `neighborUp()` function.

Squid uses both TCP (HTTP) and UDP (ICP/HTCP) communication to determine the state. The TCP state defaults to alive, but changes to dead if 10 consecutive TCP connections fail. When this happens, Squid initiates probe connections, no more than once every *connect_timeout* time period (the global directive, not the *cache_peer* option). The state remains dead until one of the probe connections succeeds.

If the `no-query` option isn’t set (meaning Squid is sending ICP/HTCP queries to the neighbor), the UDP layer communication also factors into the alive/dead algorithm. The UDP state defaults to alive, but changes to dead if Squid doesn’t get any ICP/HTCP replies for a certain amount of time—the value of the *dead_peer_timeout* directive.

Squid also marks a neighbor dead if its hostname doesn’t resolve to any IP addresses. When Squid determines a neighbor is dead, it writes an entry in *cache.log*. Here’s an example:

```
2003/09/29 01:13:46| Detected DEAD Sibling: bo2.us.ircache.net/3128/3130
```

When communication with the neighbor is reestablished, Squid logs a message like this:

```
2003/09/29 01:13:49| Detected REVIVED Sibling: bo2.us.ircache.net/3128/3130
```

A neighbor’s state affects neighbor-selection algorithms in the following ways:

- Squid doesn’t expect to receive ICP/HTCP replies from dead neighbors. Squid sends ICP queries to dead neighbors no more than once each *dead_peer_timeout* interval. See Appendix A.
- A dead parent is excluded from the following algorithms: Cache Digests, round-robin parent, first up parent, default parent, and closest parent.
- CARP is special: any failed TCP connections (not the 10 required to become dead) excludes the parent from the CARP algorithm.

There is no way to force Squid to send HTTP requests to a dead neighbor. If all neighbors are dead, Squid will try connecting to the origin server. If you don't allow Squid to talk to the origin server (with *never_direct*, for example), Squid returns a cannot forward error message:

This request could not be forwarded to the origin server or to any parent caches. The most likely cause for this error is that:

- * The cache administrator does not allow this cache to make direct connections to origin servers, and
- * All configured parent caches are currently unreachable.

Altering the Relationship

The *neighbor_type_domain* directive allows you to change the relationship with your neighbor based on the origin server's hostname. This is useful, for example, if your neighbor is willing to serve cache hits for any request but misses only for certain nearby domains. The syntax is:

```
neighbor_type_domain neighbor.host.name relationship [!]domain ...
```

For example:

```
cache_peer squid.uk.web-cache.net sibling 3128 3130
neighbor_type_domain squid.uk.web-cache.net parent .uk
```

Of course, the *squid.uk.web-cache.net* cache in this example should utilize appropriate *miss_access* rules to enforce the sibling relationship for non-UK requests. Note that domain names are matched to hostnames as described in Chapter 6.

Restricting Requests to Neighbors

Many people who use hierarchical caching need to control or limit requests that Squid sends to its neighbors. Squid has seven different directives that affect request routing: *cache_peer_access*, *cache_peer_domain*, *never_direct*, *always_direct*, *hierarchy_stoplist*, *nonhierarchical_direct*, and *prefer_direct*.

cache_peer_access

The *cache_peer_access* directive defines an access list for a neighbor cache. That is, it determines which requests may, or may not, be sent to the neighbor.

You can use this, for example, to split the flow of FTP and HTTP requests. You can send all FTP URIs to one parent and all HTTP URIs to another:

```
cache_peer A-parent.my.org parent 3128 3130
cache_peer B-parent.my.org parent 3128 3130
acl FTP proto FTP
acl HTTP proto HTTP
```

```
cache_peer_access A-parent allow FTP
cache_peer_access B-parent allow HTTP
```

This configuration ensures that A-parent receives only requests for FTP URIs, while B-parent receives only requests for HTTP URIs. This includes ICP/HTCP queries as well.

You might also use *cache_peer_access* to enable or disable a neighbor cache during certain times of the day:

```
cache_peer A-parent.my.org parent 3128 3130
acl DayTime time 07:00-18:00
cache_peer_access A-parent.my.org deny DayTime
```

cache_peer_domain

The *cache_peer_domain* directive is an earlier form of *cache_peer_access*. Rather than using the full access control feature set, it only uses domain names in URIs. It is often used to partition a group of parent caches by domain name. For example, if you have a global intranet, you may want to send requests to caches located on each continent:

```
cache_peer europe-cache.my.org parent 3128 3130
cache_peer asia-cache.my.org parent 3128 3130
cache_peer aust-cache.my.org parent 3128 3130
cache_peer africa-cache.my.org parent 3128 3130
cache_peer na-cache.my.org parent 3128 3130
cache_peer sa-cache.my.org parent 3128 3130
cache_peer_domain europe-cache.my.org parent .ch .dk .fr .uk .nl .de .fi ...
cache_peer_domain asia-cache.my.org parent .jp .kr .cn .sg .tw .vn .hk ...
cache_peer_domain aust-cache.my.org parent .nz .au .aq ...
cache_peer_domain africa-cache.my.org parent .dz .ly .ke .mz .ma .mg ...
cache_peer_domain na-cache.my.org parent .mx .ca .us ...
cache_peer_domain sa-cache.my.org parent .br .cl .ar .co .ve ...
```

Of course, this scheme doesn't address the popular global top-level domains, such as *.com*.

never_direct

The *never_direct* directive is an access list for requests that must never be sent directly to an origin server. When a request matches this access list, it must be sent to a neighbor (usually parent) cache.

For example, if Squid is behind a firewall, it may be able to talk to your "internal" servers directly but must send all requests for external servers via the firewall proxy (a parent). You can tell Squid "never connect directly to sites outside the firewall." To do so, tell Squid what is inside the firewall:

```
acl InternalSites dstdomain .my.org
never_direct allow !InternalSites
```

The syntax is a little strange. `never_direct allow foo` means Squid will not go directly for requests that match “foo.” Since the set of internal sites is easy to specify, I used the negation operator (!) to match external sites, which Squid must never directly contact.

Note that this example doesn’t force Squid to connect directly to sites that match the *InternalSites* ACL. The *never_direct* access rule can only force Squid not to contact certain origin servers. You must use the *always_direct* rule to force direct connections to origin servers.

You must take care when using *never_direct* in combination with the other directives that control request routing. You can easily create an impossible situation. Here’s an example:

```
cache_peer A-parent.my.org parent 3128 3130
acl COM dstdomain .com
cache_peer_access A-parent.my.org deny COM
never_direct allow COM
```

This configuration creates a contradiction because any request whose domain name ends with *.com* must go through a neighbor cache. However, I defined only one neighbor cache, and don’t allow the *.com* requests to go there. When this happens, Squid emits the “cannot forward” error message mentioned earlier in the “Neighbor State” section.

always_direct

As you can probably guess, the list of *always_direct* rules tell Squid that some requests must be forwarded directly to the origin server. For example, many organizations want to keep their local traffic local. An easy way to do this is to define an IP address-based ACL and put it in the *always_direct* rule list:

```
acl OurNetwork src 172.16.3.0/24
always_direct allow OurNetwork
```

hierarchy_stoplist

Internally, Squid flags each client request as either hierarchical or nonhierarchical. A nonhierarchical request is one that is unlikely to result in a cache hit. For example, responses to POST requests are almost never cachable. Forwarding requests for uncachable objects to neighbors is a waste of resources when Squid can simply connect to the origin server.

Some of the rules for differentiating hierarchical and nonhierarchical requests are hardcoded in Squid. For example, the POST and PUT methods are always nonhierarchical. However, the *hierarchy_stoplist* directive allows you to customize the

algorithm. It contains a list of strings that, when found in a URI, make the request nonhierarchical. The default list is:

```
hierarchy_stoplist ? cgi-bin
```

Thus, any request that contains a question mark or the `cgi-bin` string matches the stoplist and becomes nonhierarchical.

By default, Squid prefers to send nonhierarchical requests directly to origin servers. Because they are unlikely to result in cache hits, they are generally an extra burden on neighbor caches. However, the *never_direct* access control rules override *hierarchy_stoplist*. In particular, Squid:

- Never sends ICP/HTCP queries for nonhierarchical requests unless the request matches a *never_direct* rule
- Never sends ICP/HTCP queries to sibling caches for nonhierarchical requests
- Never looks in neighbor cache digests for nonhierarchical requests

nonhierarchical_direct

This directive controls the way that Squid forwards nonhierarchical (i.e., probably uncachable) requests. By default, Squid prefers to send nonhierarchical requests directly to origin servers. This is because such requests are unlikely to result in cache hits. I feel it is always better to get them directly from the origin server, rather than waste time looking for them in neighbor caches. If, for some reason, you want to route such requests through the hierarchy, disable this directive:

```
nonhierarchical_direct off
```

prefer_direct

This directive controls the way that Squid forwards hierarchical (i.e., probably cacheable) requests. By default, Squid prefers to send such requests to a neighbor cache first and then directly to the origin server. You can reverse this behavior by enabling the directive:

```
prefer_direct on
```

In this way, your neighbor caches become a backup if communication with the origin server fails.

The Network Measurement Database

Squid's network measurement database (*netdb*) is designed to measure the proximity of origin servers. In other words, by querying this database, Squid knows how close it is to the origin server. The database includes ICMP round-trip time (RTT)

measurements and hop counts. Squid normally uses only the RTT measurements but can also use the hop counts in some situations.

To enable *netdb*, you must configure Squid with the `--enable-icmp` option. You must also install the *pinger* program with superuser permissions, as described in Chapter 3. When everything is working correctly, you should see a message like this in *cache.log*:

```
2003/09/29 00:01:03| Pinger socket opened on FD 28
```

When *netdb* is enabled, Squid sends ICMP “pings” to origin servers. The ICMP messages are actually sent and received by the *pinger* program, which runs as *root*. Squid is careful not to send pings too frequently, which may annoy web site administrators. By default, Squid waits at least five minutes before sending another ping to the same host, or to any other host on the same /24 subnet. You can adjust the interval with the *netdb_ping_period* directive.

The ICMP pings are generally small in size (less than 100 bytes). Squid includes the origin server hostname in the payload of the ICMP message, along with a timestamp.

To reduce memory requirements, Squid aggregates the *netdb* data by /24 subnets. Squid assumes that all hosts within the subnet have similar RTT and hop-count measurements. This scheme also allows Squid to estimate the proximity of a new origin server when other servers in the subnet have already been measured.

Along with the RTT and hop-count measurements, Squid also stores a list of hostnames associated with the subnet. A typical record may look something like this:

```
Subnet 140.98.193.0
RTT    76.5
Hops   20.0
Hosts  services1.ieee.org
      www.spectrum.ieee.org
      www.ieee.org
```

The *netdb* measurements are primarily used by ICP and HTCP. When you enable the *query_icmp* directive in *squid.conf*, Squid sets a flag in the ICP/HTCP queries that it sends to neighbors. This flag is a request to include proximity measurements in the ICP/HTCP reply. If your neighbors also enabled *netdb*, their replies should include RTT and hop-count measurements if available. Note that Squid always sends ICP replies immediately. It doesn’t wait for an ICMP measurement before replying to the query. See the section “ICP and netdb” for details on how ICP uses *netdb*.

Squid remembers the RTT values it learns from ICP/HTCP replies. These values may be used later to optimize forwarding decisions. Squid also supports a “bulk transfer” of *netdb* measurements via what is called *netdb exchange*. Squid periodically makes an HTTP request to a neighbor for its *netdb* data. You can disable these requests with the *no-netdb-exchange* option on the *cache_peer* line.

The *netdb_low* and *netdb_high* directives control the size of the measurement database. When the number of stored subnets reaches *netdb_high*, Squid deletes the least recently used entries until the count is less than *netdb_low*.

The *minimum_direct_hops* and *minimum_direct_rtt* directives instruct Squid to connect directly to origin servers that are no more than some number of hops, or milliseconds, away. Requests that meet this criteria are logged with `CLOSEST_DIRECT` in *access.log*.

The cache manager's *netdb* page displays the entire network measurement database, including values from neighbor caches. For example:

Network DB Statistics:					
Network	recv/sent	RTT	Hops	Hostnames	
63.241.84.0	1/ 1	25.0	9.0	www.xyzyz.com	
sd.us.ircache.net		21.5	15.0		
bo1.us.ircache.net		27.0	13.0		
pb.us.ircache.net		70.0	11.0		
206.100.24.0	5/ 5	25.0	3.0	wcarchive.cdrom.com ftp.cdrom.com	
uc.us.ircache.net		23.5	11.0		
bo1.us.ircache.net		27.7	7.0		
pb.us.ircache.net		35.7	10.0		
sd.us.ircache.net		72.9	10.0		
146.6.135.0	1/ 1	25.0	13.0	www.cm.utexas.edu	
bo1.us.ircache.net		32.0	11.0		
sd.us.ircache.net		55.0	8.0		
216.234.248.0	2/ 2	25.0	8.0	postfuture.com www1.123india.com	
pb.us.ircache.net		44.0	14.0		
216.148.242.0	1/ 1	25.0	9.0	images.worldres.com	
sd.us.ircache.net		25.2	15.0		
bo1.us.ircache.net		27.0	13.0		
pb.us.ircache.net		69.5	11.0		

Here you can see that the server *www.xyzyz.com* has an IP address in the 63.241.84.0/24 block. The RTT from this cache to the origin server is 25 milliseconds. The neighbor cache *sd.us.ircache.net* is a little closer, at 21.5 milliseconds.

Internet Cache Protocol

ICP is a lightweight object location protocol invented as a part of the Harvest project.* An ICP client sends a query message to one or more ICP servers, asking if they have a particular URI cached. Each server replies with an `ICP_HIT`, `ICP_MISS`, or other type of ICP message. The ICP client uses the information in the ICP replies to make a forwarding decision.

* For more information, see the following papers: "A Hierarchical Internet Object Cache," by Danzig, Chankhunthod, et al, *USENIX Annual Technical Conference, 1995*, and "The Harvest information discovery and access system," by C. Mic Bowman, Peter B. Danzig, Darren R. Hardy, Udi Manber, and Michael F. Schwartz, *Proceedings of the Second International World Wide Web Conference*.

In addition to predicting cache hits, ICP is also useful for providing hints about network conditions between Squid and the neighbor. ICP messages are similar to ICMP pings in this regard. By measuring the query/response round-trip time, Squid can estimate network congestion. In the extreme case, ICP messages may be lost, indicating that the path between the two is down or congested. From this, Squid decides to avoid the neighbor for that particular request.

Increased latency is perhaps the most significant drawback to using ICP. The query/response exchange takes some time. Caching proxies are supposed to decrease response time, not add more latency. If ICP helps us discover cache hits in neighbors, then it may lead to an overall reduction in response time. See the section “Putting It All Together” for a description of the query algorithm implemented in Squid.

ICP also suffers from a number of design deficiencies: security, scalability, false hits, and the lack of a request method. The protocol doesn’t include any security features. In general, Squid can’t verify that an ICP message is authentic; it relies on address-based access controls to filter out unwanted ICP messages.

ICP has poor scaling properties. The number of ICP messages (and bandwidth) grows in proportion to the number of neighbors. Unless you use some kind of partitioning scheme, this places a practical limit on the number of neighbors you can have. I don’t recommend having more than five or six neighbors.

ICP queries contain only URIs, with no additional request headers. This makes it difficult to predict cache hits with perfect accuracy. An HTTP request may include additional headers (such as `Cache-Control: max-stale=N`) that turn a cache hit into a cache miss. These false hits are particularly awkward for sibling relationships.

Also missing from the ICP query message is the request method. ICP assumes that all queries are for GET requests. A caching proxy can’t use ICP to locate cached objects for non-GET request methods.

You can find additional information about ICP by reading:

- My book *Web Caching* (O’Reilly)
- RFCs 2186 and 2187
- My article with kc claffy: “ICP and the Squid Web Cache” in the *IEEE Journal on Selected Areas in Communication*, April 1998
- <http://icp.ircache.net/>

Being an ICP Server

When you use the `icp_port` directive, Squid automatically becomes an ICP server. That is, it listens for ICP messages on the port you’ve specified, or port 3130 by default. Be sure to tell your sibling and/or child caches if you decide to use a non-standard port.

By default, Squid denies all ICP queries. You must use the *icp_access* rule list to allow queries from your neighbors. It's usually easiest to do this with *src* ACLs. For example:

```
acl N1 src 192.168.0.1
acl N2 src 172.16.0.2
acl All src 0/0
icp_access allow N1
icp_access allow N2
icp_access deny All
```

Note that only ICP_QUERY messages are subject to the *icp_access* rules. ICP client functions, such as sending queries and receiving replies, don't require any special access controls. I also recommend that you take advantage of your operating system's packet filtering features (e.g., ipfw, iptables, and pf) if possible. Allow UDP messages on the ICP port from your trusted neighbors and deny them from all other hosts.

When Squid denies an ICP query due to the *icp_access* rules, it sends back an ICP_DENIED message. However, if Squid detects that more than 95% of the recent queries have been denied, it stops responding for an hour. When this happens, Squid writes a message in *cache.log*:

```
WARNING: Probable misconfigured neighbor at foo.web-cache.com
WARNING: 150 of the last 150 ICP replies are DENIED
WARNING: No replies will be sent for the next 3600 seconds
```

If you see this message, you should contact the administrator responsible for the misconfigured cache.

Squid was designed to answer ICP queries immediately. That is, Squid can tell whether or not it has a fresh, cached response by checking the in-memory index. This is also why Squid is a bit of a memory hog. When an ICP query comes in, Squid calculates the MD5 hash of the URI and looks for it in the index. If not found, Squid sends back an ICP_MISS message. If found, Squid checks the expiration time. If the object isn't fresh, Squid returns ICP_MISS. For fresh objects, Squid returns ICP_HIT.

By default, Squid logs all ICP queries (but not responses) to *access.log*. If you have a lot of busy neighbors, your log file may become too large to manage. Use the *log_icp_queries* directive to prevent logging of these queries. Although you'll lose the detailed logging for ICP, you can still get some aggregate stats via the cache manager (see Chapter 14).

If you have sibling neighbors, you'll probably want to use the *miss_access* directive to enforce the relationship. It specifies an access rule for cache misses. It is similar to *http_access* but is checked only for requests that must be forwarded. The default rule is to allow all cache misses. Unless you add some *miss_access* rules, any sibling cache can become a child cache and forward cache misses through your network connection, thus stealing your bandwidth.

Your *miss_access* rules can be relatively simple. Don't forget to include your local clients (i.e., web browsers) as well. Here's a simple example:

```
acl Browsers src 10.9.0.0/16
acl Child1 src 172.16.3.4
acl Child2 src 192.168.2.0/24
acl All src 0/0
miss_access allow Browsers
miss_access allow Child1
miss_access allow Child2
miss_access deny All
```

Note that I haven't listed any siblings here. The child caches are allowed to request misses through us, but the siblings are not. Their cache miss requests are denied by the *deny All* rule.

The *icp_hit_stale* directive

One of the problems with ICP is that it returns *ICP_MISS* for cached but stale responses. This is true even if the response is stale, but valid (such that a validation request returns "not modified"). Consider a simple hierarchy with a child and two parent caches. An object is cached by one parent but not the other. The cached response is stale, but unchanged, and needs validation. The child's ICP query results in two *ICP_MISS* replies. Not knowing that the stale response exists in the first parent, the child forwards its request to the second parent. Now the object is stored in both parents, wasting resources.

You might find the *icp_hit_stale* directive useful in this situation. It tells Squid to return an *ICP_HIT* for any cached object, even if it is stale. This is perfectly safe for parent relationships but can create problems for siblings.

Recall that in a sibling relationship, the client cache is only allowed to make requests that are cache hits. Enabling the *icp_hit_stale* directive increases the number of false hits because Squid must validate the stale responses. Squid normally handles false hits by adding the *Cache-Control: only-if-cached* directive to HTTP requests sent to siblings. If the sibling can't satisfy the HTTP request as a cache hit, it returns an HTTP 504 (Gateway Timeout) message instead. When Squid receives the 504 response, it forwards the request again, but only to a parent or the origin server.

It makes little sense to enable *icp_hit_stale* for sibling relationships if all the false hits must be reforwarded. This is where the ICP client's *allow-miss* option to *cache_peer* becomes useful. When the *allow-miss* option is set, Squid omits the *only-if-cached* directive in HTTP requests it sends to siblings.

If you enable *icp_hit_stale*, you also need to make sure that *miss_access* doesn't deny cache-miss requests from siblings. Unfortunately, there is no way to make Squid allow only cache-misses for cached, stale objects. Allowing cache misses for siblings also leaves your cache open to potential abuse. The administrator of the sibling cache may change it to a parent relationship without your knowledge or permission.

The ICP_MISS_NOFETCH feature

The command-line `-Y` option to Squid causes it to return `ICP_MISS_NOFETCH`, instead of `ICP_MISS`, while rebuilding the in-memory indexes. ICP clients that receive `ICP_MISS_NOFETCH` responses should not send HTTP requests for those objects. This reduces the load placed on Squid and allows the rebuild process to complete sooner.

The `test_reachability` directive

If you enable the `netdb` feature (see the section “The Network Measurement Database”), you might also be interested in enabling the `test_reachability` directive. The goal behind it is to accept only requests for origin servers Squid can reach. Enabling `test_reachability` causes Squid to return `ICP_MISS_NOFETCH`, instead of `ICP_MISS`, for origin server sites that don’t respond to ICMP pings. This can help reduce the number of failed HTTP requests and increase the chance that the end user receives the data promptly. However, a significant percentage of origin server sites intentionally filter out ICMP traffic. For these, Squid returns `ICP_MISS_NOFETCH` even though an HTTP connection would succeed.

Enabling `test_reachability` also causes Squid to make `netdb` measurements in response to ICP queries. If Squid doesn’t have any RTT measurements for the origin server in question, it sends out an ICMP ping (subject to the rate limiting mentioned previously).

Being an ICP Client

First, you must use the `cache_peer` directive to define your neighbor caches. See the section “Telling Squid About Your Neighbors.”

Second, you must also use the `icp_port` directive, even if your Squid is only an ICP client. This is because Squid uses the same socket for sending and receiving ICP messages. It is perhaps a bad design decision in retrospect. If you are a client only, use `icp_access` to block queries. For example:

```
acl All src 0/0
icp_access deny All
```

Squid sends ICP queries to its neighbors for most requests by default. See the section “Putting It All Together” for a complete description of the way that Squid decides when, and when not, to query its neighbors.

After sending one or more queries, Squid waits some amount of time for ICP replies to arrive. If Squid receives an `ICP_HIT` from one of its neighbors, it forwards the request there immediately. Otherwise, Squid waits until all replies arrive or until a timeout occurs. The timeout is calculated dynamically, based on the following algorithm.

Squid knows the average round-trip time between itself and each neighbor, taken from recent ICP transactions. When querying a group of neighbors, Squid calculates the mean of all the neighbor ICP RTTs, and then doubles it. In other words, the query timeout is twice the mean of RTTs for each neighbor queried. Squid ignores neighbors that appear to be down when calculating the timeout.

In some cases, the algorithm doesn't work well, especially if you have neighbors with widely varying RTTs. You can change the upper limit on the timeout using the *maximum_icp_query_timeout* directive. Alternatively, you can make Squid always use a constant timeout value with the *icp_query_timeout* directive.

cache_peer options for ICP clients

weight=n allows you to weight parent caches artificially when using ICP/HTCP. It comes into play only when all parents report a cache miss. Normally, Squid selects the parent whose reply arrives first. In fact, it remembers which parent has the best RTT for the query. Squid actually divides the RTT by the weight, so that a parent with *weight=2* is treated as if it's closer to Squid than it really is.

no-query disables ICP/HTCP for the neighbor. That is, your cache won't send any queries to the neighbor for cache misses. It is often used with the default option.

closest-only refers to one of Squid's *netdb* features. It instructs Squid to select the parent based only on *netdb* RTT measurements and not the order in which replies arrive. This option requires *netdb* at both ends.

ICP and netdb

As mentioned in the section "The Network Measurement Database," *netdb* is mostly used with ICP queries. In this section, we'll follow all the steps involved in this process.

1. A Squid cache, acting as an ICP client, prepares to send a query to one or more neighbors. If *query_icmp* is set, Squid sets the SRC_RTT flag in the ICP query. This informs the ICP server that Squid would like to receive an RTT measurement in the ICP reply.
2. The neighbor receives the query with the SRC_RTT flag set. If the neighbor is configured to make *netdb* measurements, it searches the database for the origin server hostname. Note that the neighbor doesn't query the DNS for the origin server's IP address. Thus, it finds a *netdb* entry only if that particular host has already been measured.
3. If the host exists in the *netdb* database, the neighbor includes the RTT and hop count in the ICP reply. The SRC_RTT flag is set in the reply to indicate the measurement is present.

4. When Squid receives the ICP reply with the `SRC_RTT` flag set, it extracts the RTT and hop count. These are added to the local *netdb* so that, in the future, Squid knows the approximate RTT from the neighbor to the origin server.
5. An `ICP_HIT` reply causes Squid to forward the HTTP request immediately. If, on the other hand, Squid receives only `ICP_MISS` replies, it selects the parent with the smallest (nonzero) measured RTT to the origin server. The request is logged to *access.log* with `CLOSEST_PARENT_MISS`.
6. If none of the parent `ICP_MISS` replies contain RTT values, Squid selects the parent whose ICP reply arrived first. In this case, the request is logged with `FIRST_PARENT_MISS`. However, if the `closest-only` option is set for a parent cache, Squid never selects it as a “first parent.” In other words, the parent is selected only if it is the closest parent to the origin server.

Multicast ICP

As you already know, ICP has poor scaling properties. The number of messages is proportional to the number of neighbors. Because Squid sends identical `ICP_QUERY` messages to each neighbor, you can use multicast to reduce the number of messages transmitted on the network. Rather than send *N* messages to *N* neighbors, Squid sends one message to a multicast address. The multicast routing infrastructure makes sure each neighbor receives a copy of the message. See the book *Interdomain Multicast Routing: Practical Juniper Networks and Cisco Systems Solutions* by Brian M. Edwards, Leonard A. Giuliano, and Brian R. Wright (Addison Wesley) for more information on the inner workings of multicast.

Note that ICP replies are always sent via unicast. This is because ICP replies may be different (e.g., hit versus miss) and because the unicast and multicast routing topologies may differ. Because ICP is also used to indicate network conditions, an ICP reply should follow the same path an HTTP reply takes. The bottom line is that multicast only reduces message counts for queries.

Historically, I’ve found multicast infrastructure unstable and unreliable. It seems to be a low priority for many ISPs. Even though it works one day, something may break a few days or weeks later. You’re probably safe using multicast entirely within your own network, but I don’t recommend using it for ICP on the public Internet.

Multicast ICP server

A multicast ICP server joins one or more multicast group addresses to receive messages. The `mcast_groups` directive specifies these group addresses. The values must be multicast IP addresses or hostnames that resolve to multicast addresses. The IPv4 multicast address range is 224.0.0.0–239.255.255.255. For example:

```
mcast_groups 224.11.22.45
```

An interesting thing about multicast is that hosts, rather than applications, join a group. When a host joins a multicast group, it receives all packets that are transmitted to that group. This means that you need to be a little bit careful when selecting a multicast group to use for ICP. You don't want to select an address that's already being used by another application. When this kind of group overlap occurs, the two groups become joined and receive each other's traffic.

Multicast ICP client

Multicast ICP clients transmit queries to one or more multicast group addresses. Thus, the *hostname* argument of the *cache_peer* line must be, or resolve to, a multicast address. For example:

```
cache_peer 224.0.14.1 multicast 3128 3130 ttl=32
```

The HTTP port number (e.g., 3128) is irrelevant in this case because Squid never makes HTTP connections to a multicast neighbor.

Realize that multicast groups don't have any access controls. Any host can join any multicast group address. This means that, unless you're careful, others may be able to receive the multicast ICP queries sent by your Squid. IP multicast has two ways to prevent packets from traveling too far: TTLs and administrative scoping. Because ICP queries may carry sensitive information (i.e., URIs that your users access), I recommend using an administratively scoped address and properly configured routers. See RFC 2365 for more information.

The *ttl=n* option is for multicast neighbors only. It is the multicast TTL value to use for ICP queries. It controls how far away the ICP queries can travel. The valid range is 0–128. A larger value allows the multicast queries to travel farther, and possibly be intercepted by outsiders. Use a lower number to keep the queries close to the source and within your network.

Multicast ICP clients must also tell Squid about the neighbors that will be responding to queries. Squid doesn't blindly trust any cache that happens to send an ICP reply. You must tell Squid about legitimate, trusted neighbors. The *multicast-responder* option to *cache_peer* identifies such neighbors. For example, if you know that 172.16.2.3 is a trusted neighbor on the multicast group, you should add this line to *squid.conf*:

```
cache_peer 172.16.3.2 parent 3128 3130 multicast-responder
```

You can, of course, use a hostname instead of an IP address. ICP replies from foreign (unlisted) neighbors are ignored, but logged in *cache.log*.

Squid normally expects to receive an ICP reply for each query that it sends. This changes, however, with multicast because one query may result in multiple replies. To account for this, Squid periodically sends out "probes" on the multicast group address. These probes tell Squid how many servers are out there listening. Squid counts the number of replies that arrive within a certain amount of time. That

amount of time is given by the `mcast_icp_query_timeout` directive. Then, when Squid sends a real ICP query to the group, it adds this count to the number of ICP replies to expect.

Multicast ICP example

Since multicast ICP is tricky, here's another example. Let's say your ISP has three parent caches that listen on a multicast address for ICP queries. The ISP needs only one line in its configuration file:

```
mcast_groups 224.0.14.255
```

The configuration for you (the child cache) is a little more complicated. First, you must list the multicast neighbor to which Squid should send queries. You must also list the three parent caches with their unicast addresses so that Squid accepts their replies:

```
cache_peer 224.0.14.225 multicast 3128 3130 ttl=16
cache_peer parent1.yourisp.net parent 3128 3130 multicast-responder
cache_peer parent2.yourisp.net parent 3128 3130 multicast-responder
cache_peer parent3.yourisp.net parent 3128 3130 multicast-responder
mcast_icp_query_timeout 2 sec
```

Keep in mind that Squid never makes HTTP requests to multicast neighbors, and it never sends ICP queries to multicast-responder neighbors.

Cache Digests

One of the most common complaints about ICP is the additional delay added for each request. In many cases, Squid waits for all ICP replies to arrive before making a forwarding decision. Squid's Cache Digest feature offers similar functionality but without per-request network delays.

Cache Digests are based on a technique first published by Pei Cao, called Summary Cache. The fundamental idea is to use a Bloom filter to represent the cache contents. Neighboring caches download each other's Bloom filters, or digests in this terminology. Then, they can query the digest to determine whether or not a particular URI is in the neighbor's cache.

Compared to ICP, Cache Digests trade time for space. Whereas ICP queries incur time penalties (latency), digests incur space (memory, disk) penalties. In Squid, a neighbor's digest is stored entirely in memory. A typical digest requires about 625 KB of memory for every million objects.

The Bloom filter is an interesting data structure that provides lossy encoding of a collection of items. The filter itself is simply a large array of bits. Given a Bloom filter (and the parameters used to generate it), you can find, with some uncertainty, if a particular item is in the collection. In Squid, items are URIs, and the digest is sized at

5 bits per cached object. For example, you can represent the collection of 1,000,000 cached objects with a filter of 5,000,000 bits, or 625,000 bytes.

Due to their nature, Bloom filters aren't a perfect representation of the collection. They sometimes incorrectly indicate that a particular item is present in the collection because two or more items may turn on the same bit. In other words, the filter can indicate that object *X* is in the cache, even though *X* was never cached or requested. These false positives occur with a certain probability you can control by adjusting the parameters of the filter. For example, increasing the number of bits per object decreases the false positive probability. See my O'Reilly book, *Web Caching*, for many more details about Cache Digests.

Configuring Squid for Cache Digests

First of all, you must compile Squid with the Cache Digest code enabled. Simply add the `--enable-cache-digests` option when running `./configure`. Taking this step causes two things to happen when you run Squid:

- Your Squid cache generates a digest of its own contents. Your neighbors may request this digest if they are also configured to use Cache Digests.
- Your Squid requests a Cache Digest from each of its neighbors.

If you don't want to request digests for a particular neighbor, use the `no-digest` option on the `cache_peer` line. For example:

```
cache_peer neighbor.host.name parent 3128 3130 no-digest
```

Squid stores its own digest under the following URL: `http://my.host.name:port/squid-internal-periodic/store_digest`. When Squid requests a neighbor's digest, it simply requests `http://neighbor.host.name:port/squid-internal-periodic/store_digest`. Obviously, this naming scheme is specific to Squid. If you have a non-Squid neighbor that supports Cache Digests, you may need to tell your Squid that the neighbor's digest has a different address. The `digest-url=url` option to `cache_peer` allows you to configure the URL for the neighbor's Cache Digest. For example:

```
cache_peer neighbor.host.name parent 3128 3130 digest-url=http://blah/digest
```

`squid.conf` has a number of directives that control the way in which Squid generates its own Cache Digest. First, the `digest_generation` directive controls whether or not Squid generates a digest of its cache. You might want to disable digest generation if your cache is a child to a parent, but not a parent or sibling to any other caches. The remaining directives control low-level underlying details of digest generation. You should change them only if you fully understand the Cache Digest implementation.

The `digest_bits_per_entry` determines the size of the digest. The default value is 5. Increasing the value results in larger digests (consuming more memory and bandwidth) and lower false-hit probabilities. A lower setting results in smaller digests and more false hits. I feel that the default setting is a very nice tradeoff. A setting of 3 or

lower has too many false hits to be useful, and a setting of 8 or higher simply wastes bandwidth.

Squid uses a two-step process to create a cache digest. First, it builds the cache digest data structure. This is basically a large Bloom filter and small header that contains the digest parameters. Once the data structure is filled, Squid creates a cached HTTP response for the digest. This simply involves prepending some HTTP headers and storing the response on disk with the other cached responses.

A Cache Digest represents a snapshot in time of the cache's contents. The *digest_rebuild_period* controls how frequently Squid rebuilds the digest data structure (but not the HTTP response). The default is once per hour. More frequent rebuilds mean Squid's digest is more up to date, at the expense of higher CPU utilization. The rebuild procedure is relatively CPU-intensive. Your users may experience a slow-down while Squid rebuilds its digest.

The *digest_rebuild_chunk_percentage* directive controls how much of the cache to add to the digest each time the rebuild procedure is called. The default is 10%. During each invocation of the rebuild function, Squid adds some percentage of the cache to the digest. Squid doesn't process user requests while this function runs. After adding the specified percentage, the function reschedules itself and then exits so that Squid can process normal HTTP requests. After processing pending requests, Squid returns to the rebuild function and adds another chunk of the cache to the digest. Decreasing this value should give better response time to your users, while increasing the total time needed to rebuild the digest.

The *digest_rewrite_period* directive controls how often Squid creates an HTTP response from the digest data structure. In most cases, this should match the *digest_rebuild_period* value. The default is one hour. The rewrite procedure consists of numerous calls to a function that simply appends some amount of the digest data structure to the cache entry (as though Squid were reading an origin server response from the network). Each time this function is called, Squid appends *digest_swapout_chunk_size* bytes of the digest.

Hypertext Caching Protocol

HTCP and ICP have many common characteristics, although HTCP is broader in scope and generally more complex. Both use UDP for transport, and both are per-request protocols. However, HTCP addresses a number of problems with ICP, namely:

- An ICP query contains only a URI, without even a request method. HTCP queries contain full HTTP request headers.

- ICP provides no security. HTCP has optional message authentication via shared secret keys, although it isn't yet implemented in Squid. Neither protocol supports encrypted messages.
- ICP uses a simple, fixed-sized binary message format that is difficult to extend. HTCP uses a complex, variable-sized binary message format.

HTCP supports four basic opcodes:

TST

Tests for the presence of a cached response

SET

Tells a neighbor to update cached object headers

CLR

Tells a neighbor to remove an object from its cache

MON

Monitors a neighbor cache's activity

In Squid, only the TST opcode is currently implemented. This book won't cover the others.

The primary advantage of using HTCP over ICP is fewer false hits. HTCP has fewer false hits because the query messages include full HTTP request headers, including any Cache-Control requirements from the client. The primary disadvantages are that HTCP queries are larger, and they require additional CPU processing to generate and parse. Measurements indicate that HTCP queries are about six times larger than ICP queries, due to the presence of HTTP request headers. However, Squid's HTCP replies are typically smaller than ICP replies.

HTCP is documented as an experimental protocol in RFC 2756. For more information about the message format, see the RFC at <http://www.htcp.org> or my O'Reilly book, *Web Caching*.

Configuring Squid for HTCP

To use HTCP, you must configure Squid with the `--enable-htcp` option. With this option enabled, Squid becomes an HTCP server by default. The `htcp_port` specifies the HTCP port number, which defaults to 4827. Setting the port to 0 disables the HTCP server mode.

To become an HTCP client, you need to add the `htcp` option to a `cache_peer` line. When you add this option, Squid always sends HTCP messages, instead of ICP, to the neighbor. You can't use both HTCP and ICP with a single neighbor. The ICP port number field actually becomes an HTCP port number, so you need to change that as well. For example, let's say you want to convert an ICP neighbor to HTCP. Here's the neighbor configured for ICP:

```
cache_peer neighbor.host.name parent 3128 3130
```

To switch over to HTCP, the line becomes:

```
cache_peer neighbor.host.name parent 3128 4827 htcp
```

Sometimes people forget to change the port number, and they end up sending HTCP messages to the ICP port. When this happens, Squid writes warnings to *cache.log*:

```
2003/09/29 02:28:55| WARNING: Unused ICP version 23 received from 64.216.111.20:4827
```

Squid doesn't currently log HTCP queries as it does for ICP queries. HTCP queries aren't tracked in the *client_list* page either. However, when you enable HTCP for a peer, the cache manager *server_list* page (see Chapter 14) shows the count and percentage of HTCP replies that were hits and misses:

```
Histogram of PINGS ACKED:
Misses      5085  98%
Hits        92    2%
```

Note that none of the current Squid versions support HTCP authentication yet.

Cache Array Routing Protocol

CARP is an algorithm that partitions *URI-space* among a group of caching proxies. In other words, each URI is assigned to one of the caches. CARP maximizes hit ratios and minimizes duplication of objects among the group of caches. The protocol consists of two major components: a Routing Function and a Proxy Array Membership Table. Unlike ICP, HTCP, and Cache Digests, CARP can't predict whether a particular request will be a cache hit. Thus, you can't use CARP with siblings—only parents.

The basic idea behind CARP is that you have a group, or array, of parent caches to handle all the load from users or child caches. A cache array is one way to handle ever-increasing loads. You can add more array members whenever you need more capacity. CARP is a deterministic algorithm. That is, the same request always goes to the same array member (as long as the array size doesn't change). Unlike ICP and HTCP, CARP doesn't use query messages.

Another interesting thing about CARP is that you have the choice to deploy it in a number of different places. For example, one approach is to make all user-agents execute the CARP algorithm. You could probably accomplish this with a Proxy Auto-Configuration (PAC) function, written in JavaScript (see Appendix F). However, you're likely to have certain web agents on your network that don't implement or support PAC files. Another option is to use a two-level cache hierarchy. The lower level (child caches) accept requests from all user-agents, and they execute the CARP algorithm to select the parent cache for each request. However, unless your network is very large, many caches can be more of a burden than a benefit. Finally, you can also implement CARP within the array itself. That is, user-agents connect to a random member of the cache array, but each member forwards cache misses to another member of the array based on the CARP algorithm.

CARP was designed to be better than a simple hashing algorithm, which typically works by applying a hash function, such as MD5, to URIs. The algorithm then calculates the modulus for the number of array members. It might be as simple as this pseudocode:

```
N = MD5(URI) % num_caches;  
next_hop = Caches[N];
```

This technique uniformly spreads the URIs among all the caches. It also provides a consistent mapping (maximizing cache hits), as long as the number of caches remains constant. When caches are added or removed, however, this algorithm changes the mapping for most of the URIs.

CARP's Routing Function improves on this technique in two ways. First, it allows for unequal sharing of the load. For example, you can configure one parent to receive twice as many requests as another. Second, adding or removing array members minimizes the fraction of URIs that get reassigned.

The downside to CARP is that it is relatively CPU-intensive. For each request, Squid calculates a "score" for each parent. The request is sent to the parent cache with the highest score. The complexity of the algorithm is proportional to the number of parents. In other words, CPU load increases in proportion to the number of CARP parents. However, the calculations in CARP have been designed to be faster than, say, MD5, and other cryptographic hash functions.

In addition to the load-sharing algorithm, CARP also has a protocol component. The Membership Table has a well-defined structure and syntax so that all clients of a single array can have the same configuration. If some clients are configured differently, CARP becomes less useful because not all clients send the same request to the same parent. Note that Squid doesn't currently implement the Membership Table feature.

Squid's CARP implementation is lacking in another way. The protocol says that if a request can't be forwarded to the highest-scoring parent cache, it should be sent to the second-highest-scoring member. If that also fails, the application should give up. Squid currently uses only the highest-scoring parent cache.

CARP was originally documented as an Internet Draft in 1998, which is now expired. It was developed by Vinod Valloppillil of Microsoft and Keith W. Ross of the University of Pennsylvania. With a little searching, you can still find the old document out there on the Internet. You may even be able to find some documentation on the Microsoft sites. You can also find more information on CARP in my O'Reilly book *Web Caching*.

Configuring Squid for CARP

To use CARP in Squid, you must first run the `./configure` script with the `--enable-carp` option. Next, you must add `carp-load-factor` options to the `cache_peer` lines for parents that are members of the array. The following is an example.

```
cache_peer neighbor1.host.name parent 3128 0 carp-load-factor=0.3
cache_peer neighbor2.host.name parent 3128 0 carp-load-factor=0.3
cache_peer neighbor3.host.name parent 3128 0 carp-load-factor=0.4
```

Note that all `carp-load-factor` values must add up to 1.0. Squid checks for this condition and complains if it finds a discrepancy. Additionally, the `cache_peer` lines must be listed in order of increasing load factor values. Only recent versions of Squid check that this condition is true.

Remember that CARP is treated somewhat specially with regard to a neighbor's alive/dead state. Squid normally declares a neighbor dead (and ceases sending requests to it) after 10 failed connections. In the case of CARP, however, Squid skips a parent that has one or more failed connections. Once Squid is working with CARP, you can monitor it with the `carp` cache manager page. See Chapter 14 for more information.

Putting It All Together

As you probably realize by now, Squid has many different ways to decide how and where requests are forwarded. In many cases, you can employ more than one protocol or technique at a time. Just by looking at the configuration file, however, you'd probably have a hard time figuring out how Squid uses the different techniques in combination. In this section I'll explain how Squid actually makes the forwarding decision.

Obviously, it all starts with a cache miss. Any request that is satisfied as an unvalidated cache hit doesn't go through the following sequence of events.

The goal of the selection procedure is to create a list of appropriate next-hop locations. A next-hop location may be a neighbor cache or the origin server. Depending on your configuration, Squid may select up to three possible next-hops. If the request can't be satisfied by the first, Squid tries the second, and so on.

Step 1: Determine Direct Options

The first step is to determine if the request may, must, or must not be sent directly to the origin server. Squid evaluates the `never_direct` and `always_direct` access rule lists for the request. The goal is to set a flag to one of three values: `DIRECT_YES`, `DIRECT_MAYBE`, or `DIRECT_NO`. This flag later determines whether Squid should, or should not, try to select a neighbor cache for the request. Squid checks the following conditions in order. If any condition is true, it sets the direct flag and proceeds to the next step. If you're following along in the source code, this step corresponds to the beginning of the `peerSelectFoo()` function:

1. Squid looks at the `always_direct` list first. If the request matches this list, the direct flag is set to `DIRECT_YES`.