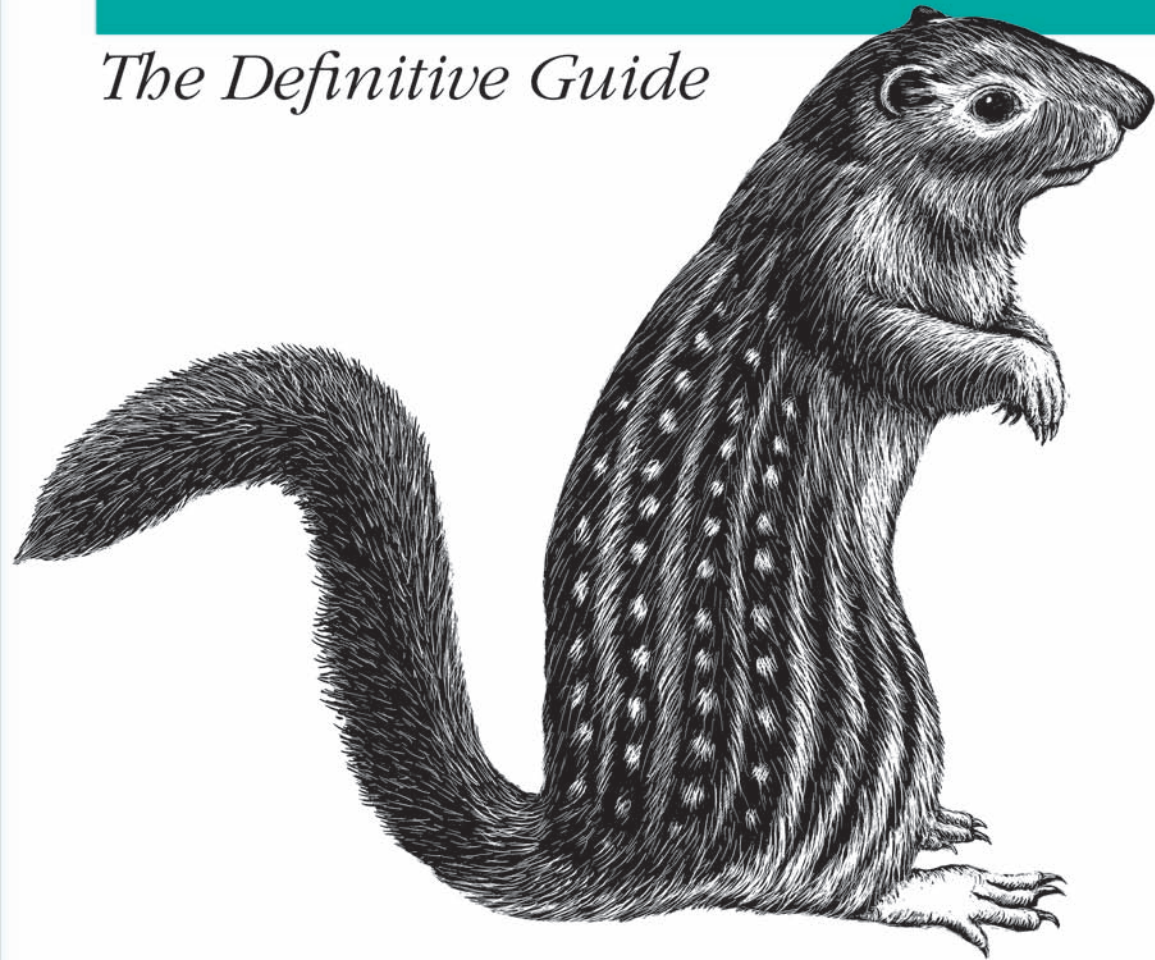


Understanding Web Internals

HTTP

The Definitive Guide



O'REILLY®

*David Gourley & Brian Totty
with Marjorie Sayer, Sailu Reddy & Anshu Aggarwal*

META-1010

IPR2026-00346, IPR2026-00347

HTTP: The Definitive Guide



Behind every successful web transaction lurks the Hypertext Transfer Protocol (HTTP), the language by which web clients and servers exchange documents and information. HTTP is commonly known as the workhorse behind the browsers we use every day to access our company intranets, locate out-of-print books, or research census information.

But HTTP is used for far more than browsing the Web: the simplicity and ubiquity of HTTP also have made it the choice protocol for many other networked applications, most notably through web services such as SOAP and XML-RPC.

As the title suggests, *HTTP: The Definitive Guide* explains the HTTP protocol: how it works and how to use it to develop web-based applications. However, this book is not just about HTTP; it's also about all the other core Internet technologies that HTTP depends on to work effectively. Although HTTP is at the center of the book, the essence of *HTTP: The Definitive Guide* is in understanding how the Web works and how to apply that knowledge to web programming and administration. The book explains the technical workings, motivations, performance considerations, and objectives of HTTP and the technologies around which it revolves.

HTTP: The Definitive Guide is the bible for the HTTP protocol and related web technologies.

Topics covered include:

- HTTP methods, headers, and status codes
- Optimizing proxies and caches
- Strategies for designing web robots and crawlers
- Cookies, authentication, and Secure HTTP
- Internationalization and content negotiation
- Redirection and load-balancing strategies

Written by experts with years of practical experience, this book uses clear, concise language and a plethora of detailed illustrations to help readers visualize what goes on behind the scenes, providing a complete understanding of the story behind each query on the Web.

All web programmers, administrators, and application developers need to be familiar with HTTP in order to work effectively. There are many books that explain how to use the Web, but this is the book that explains how the Web works.

www.oreilly.com

US \$44.95

CAN \$69.95

ISBN: 978-1-56592-509-0

5 4 4 9 5



HTTP

The Definitive Guide

HTTP

The Definitive Guide

David Gourley and Brian Totty
with Marjorie Sayer, Sailu Reddy, and Anshu Aggarwal

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

HTTP: The Definitive Guide

by David Gourley and Brian Totty
with Marjorie Sayer, Sailu Reddy, and Anshu Aggarwal

Copyright © 2002 O'Reilly Media, Inc. All rights reserved.
Printed in the United States of America.

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

O'Reilly Media, Inc. books may be purchased for educational, business, or sales promotional use. On-line 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: Linda Mui
Production Editor: Rachel Wheeler
Cover Designer: Ellie Volckhausen
Interior Designers: David Futato and Melanie Wang

Printing History:

September 2002: First Edition.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. *HTTP: The Definitive Guide*, the image of a thirteen-lined ground squirrel, and related trade dress are trademarks of O'Reilly Media, 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 Media, 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 authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.



This book uses RepKover™, a durable and flexible lay-flat binding.

ISBN-10: 1-56592-509-2

ISBN-13: 978-1-56592-509-0

[C]

[01/08]

Table of Contents

Preface	xiii
----------------------	-------------

Part I. HTTP: The Web's Foundation

1. Overview of HTTP	3
HTTP: The Internet's Multimedia Courier	3
Web Clients and Servers	4
Resources	4
Transactions	8
Messages	10
Connections	11
Protocol Versions	16
Architectural Components of the Web	17
The End of the Beginning	21
For More Information	21
2. URLs and Resources	23
Navigating the Internet's Resources	24
URL Syntax	26
URL Shortcuts	30
Shady Characters	35
A Sea of Schemes	38
The Future	40
For More Information	41
3. HTTP Messages	43
The Flow of Messages	43
The Parts of a Message	44

Methods	53
Status Codes	59
Headers	67
For More Information	73
4. Connection Management	74
TCP Connections	74
TCP Performance Considerations	80
HTTP Connection Handling	86
Parallel Connections	88
Persistent Connections	90
Pipelined Connections	99
The Mysteries of Connection Close	101
For More Information	104
Part II. HTTP Architecture	
5. Web Servers	109
Web Servers Come in All Shapes and Sizes	109
A Minimal Perl Web Server	111
What Real Web Servers Do	113
Step 1: Accepting Client Connections	115
Step 2: Receiving Request Messages	116
Step 3: Processing Requests	120
Step 4: Mapping and Accessing Resources	120
Step 5: Building Responses	125
Step 6: Sending Responses	127
Step 7: Logging	127
For More Information	127
6. Proxies	129
Web Intermediaries	129
Why Use Proxies?	131
Where Do Proxies Go?	137
Client Proxy Settings	141
Tricky Things About Proxy Requests	144
Tracing Messages	150
Proxy Authentication	156

Proxy Interoperation	157
For More Information	160
7. Caching	161
Redundant Data Transfers	161
Bandwidth Bottlenecks	161
Flash Crowds	163
Distance Delays	163
Hits and Misses	164
Cache Topologies	168
Cache Processing Steps	171
Keeping Copies Fresh	175
Controlling Cachability	182
Setting Cache Controls	186
Detailed Algorithms	187
Caches and Advertising	194
For More Information	196
8. Integration Points: Gateways, Tunnels, and Relays	197
Gateways	197
Protocol Gateways	200
Resource Gateways	203
Application Interfaces and Web Services	205
Tunnels	206
Relays	212
For More Information	213
9. Web Robots	215
Crawlers and Crawling	215
Robotic HTTP	225
Misbehaving Robots	228
Excluding Robots	229
Robot Etiquette	239
Search Engines	242
For More Information	246
10. HTTP-NG	247
HTTP's Growing Pains	247
HTTP-NG Activity	248

Modularize and Enhance	248
Distributed Objects	249
Layer 1: Messaging	250
Layer 2: Remote Invocation	250
Layer 3: Web Application	251
WebMUX	251
Binary Wire Protocol	252
Current Status	252
For More Information	253

Part III. Identification, Authorization, and Security

11. Client Identification and Cookies	257
The Personal Touch	257
HTTP Headers	258
Client IP Address	259
User Login	260
Fat URLs	262
Cookies	263
For More Information	276
12. Basic Authentication	277
Authentication	277
Basic Authentication	281
The Security Flaws of Basic Authentication	283
For More Information	285
13. Digest Authentication	286
The Improvements of Digest Authentication	286
Digest Calculations	291
Quality of Protection Enhancements	299
Practical Considerations	300
Security Considerations	303
For More Information	306
14. Secure HTTP	307
Making HTTP Safe	307
Digital Cryptography	309

Symmetric-Key Cryptography	313
Public-Key Cryptography	315
Digital Signatures	317
Digital Certificates	319
HTTPS: The Details	322
A Real HTTPS Client	328
Tunneling Secure Traffic Through Proxies	335
For More Information	336

Part IV. Entities, Encodings, and Internationalization

15. Entities and Encodings	341
Messages Are Crates, Entities Are Cargo	342
Content-Length: The Entity's Size	344
Entity Digests	347
Media Type and Charset	348
Content Encoding	351
Transfer Encoding and Chunked Encoding	354
Time-Varying Instances	359
Validators and Freshness	360
Range Requests	363
Delta Encoding	365
For More Information	369
16. Internationalization	370
HTTP Support for International Content	370
Character Sets and HTTP	371
Multilingual Character Encoding Primer	376
Language Tags and HTTP	384
Internationalized URIs	389
Other Considerations	392
For More Information	392
17. Content Negotiation and Transcoding	395
Content-Negotiation Techniques	395
Client-Driven Negotiation	396
Server-Driven Negotiation	397
Transparent Negotiation	400

Transcoding	403
Next Steps	405
For More Information	406

Part V. Content Publishing and Distribution

18. Web Hosting	411
Hosting Services	411
Virtual Hosting	413
Making Web Sites Reliable	419
Making Web Sites Fast	422
For More Information	423
19. Publishing Systems	424
FrontPage Server Extensions for Publishing Support	424
WebDAV and Collaborative Authoring	429
For More Information	446
20. Redirection and Load Balancing	448
Why Redirect?	449
Where to Redirect	449
Overview of Redirection Protocols	450
General Redirection Methods	452
Proxy Redirection Methods	462
Cache Redirection Methods	469
Internet Cache Protocol	473
Cache Array Routing Protocol	475
Hyper Text Caching Protocol	478
For More Information	481
21. Logging and Usage Tracking	483
What to Log?	483
Log Formats	484
Hit Metering	492
A Word on Privacy	495
For More Information	495

Part VI. Appendixes

A. URI Schemes 499

B. HTTP Status Codes 505

C. HTTP Header Reference 508

D. MIME Types 533

E. Base-64 Encoding 570

F. Digest Authentication 574

G. Language Tags 581

H. MIME Charset Registry 602

Index 617

Overview of HTTP

The world's web browsers, servers, and related web applications all talk to each other through HTTP, the Hypertext Transfer Protocol. HTTP is the common language of the modern global Internet.

This chapter is a concise overview of HTTP. You'll see how web applications use HTTP to communicate, and you'll get a rough idea of how HTTP does its job. In particular, we talk about:

- How web clients and servers communicate
- Where resources (web content) come from
- How web transactions work
- The format of the messages used for HTTP communication
- The underlying TCP network transport
- The different variations of the HTTP protocol
- Some of the many HTTP architectural components installed around the Internet

We've got a lot of ground to cover, so let's get started on our tour of HTTP.

HTTP: The Internet's Multimedia Courier

Billions of JPEG images, HTML pages, text files, MPEG movies, WAV audio files, Java applets, and more cruise through the Internet each and every day. HTTP moves the bulk of this information quickly, conveniently, and reliably from web servers all around the world to web browsers on people's desktops.

Because HTTP uses reliable data-transmission protocols, it guarantees that your data will not be damaged or scrambled in transit, even when it comes from the other side of the globe. This is good for you as a user, because you can access information without worrying about its integrity. Reliable transmission is also good for you as an Internet application developer, because you don't have to worry about HTTP communications

being destroyed, duplicated, or distorted in transit. You can focus on programming the distinguishing details of your application, without worrying about the flaws and foibles of the Internet.

Let's look more closely at how HTTP transports the Web's traffic.

Web Clients and Servers

Web content lives on web servers. Web servers speak the HTTP protocol, so they are often called HTTP servers. These HTTP servers store the Internet's data and provide the data when it is requested by HTTP clients. The clients send HTTP requests to servers, and servers return the requested data in HTTP responses, as sketched in Figure 1-1. Together, HTTP clients and HTTP servers make up the basic components of the World Wide Web.

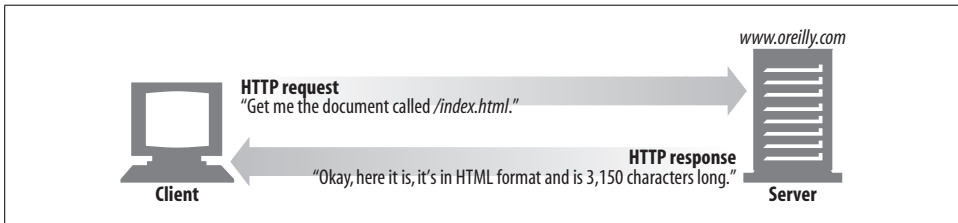


Figure 1-1. Web clients and servers

You probably use HTTP clients every day. The most common client is a web browser, such as Microsoft Internet Explorer or Netscape Navigator. Web browsers request HTTP objects from servers and display the objects on your screen.

When you browse to a page, such as "`http://www.oreilly.com/index.html`," your browser sends an HTTP request to the server `www.oreilly.com` (see Figure 1-1). The server tries to find the desired object (in this case, "`/index.html`") and, if successful, sends the object to the client in an HTTP response, along with the type of the object, the length of the object, and other information.

Resources

Web servers host *web resources*. A web resource is the source of web content. The simplest kind of web resource is a static file on the web server's filesystem. These files can contain anything: they might be text files, HTML files, Microsoft Word files, Adobe Acrobat files, JPEG image files, AVI movie files, or any other format you can think of.

However, resources don't have to be static files. Resources can also be software programs that generate content on demand. These dynamic content resources can generate content based on your identity, on what information you've requested, or on

the time of day. They can show you a live image from a camera, or let you trade stocks, search real estate databases, or buy gifts from online stores (see Figure 1-2).

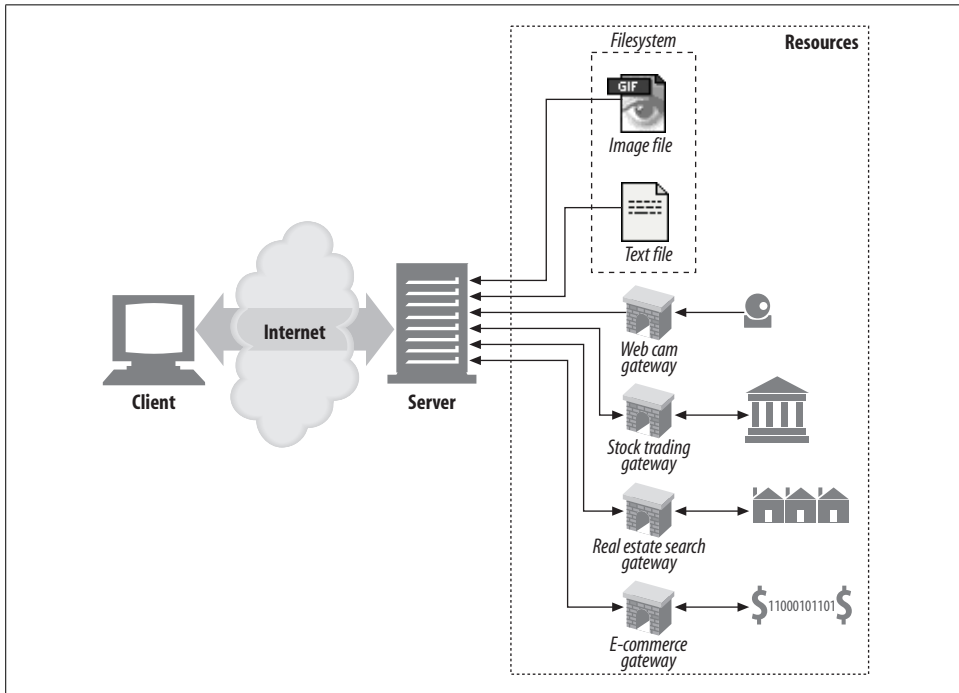


Figure 1-2. A web resource is anything that provides web content

In summary, a resource is any kind of content source. A file containing your company's sales forecast spreadsheet is a resource. A web gateway to scan your local public library's shelves is a resource. An Internet search engine is a resource.

Media Types

Because the Internet hosts many thousands of different data types, HTTP carefully tags each object being transported through the Web with a data format label called a *MIME type*. MIME (Multipurpose Internet Mail Extensions) was originally designed to solve problems encountered in moving messages between different electronic mail systems. MIME worked so well for email that HTTP adopted it to describe and label its own multimedia content.

Web servers attach a MIME type to all HTTP object data (see Figure 1-3). When a web browser gets an object back from a server, it looks at the associated MIME type to see if it knows how to handle the object. Most browsers can handle hundreds of popular object types: displaying image files, parsing and formatting HTML files, playing audio files through the computer's speakers, or launching external plug-in software to handle special formats.

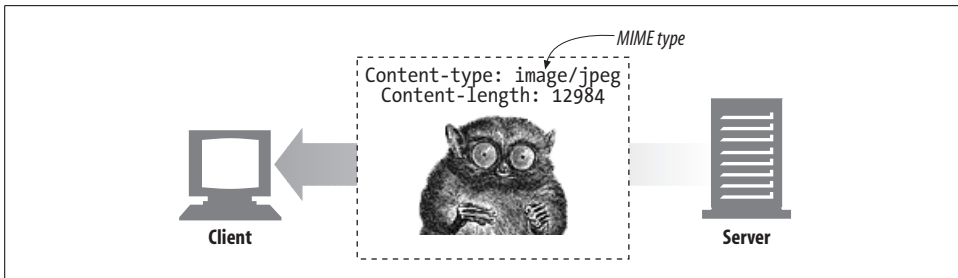


Figure 1-3. MIME types are sent back with the data content

A MIME type is a textual label, represented as a primary object type and a specific subtype, separated by a slash. For example:

- An HTML-formatted text document would be labeled with type `text/html`.
- A plain ASCII text document would be labeled with type `text/plain`.
- A JPEG version of an image would be `image/jpeg`.
- A GIF-format image would be `image/gif`.
- An Apple QuickTime movie would be `video/quicktime`.
- A Microsoft PowerPoint presentation would be `application/vnd.ms-powerpoint`.

There are hundreds of popular MIME types, and many more experimental or limited-use types. A very thorough MIME type list is provided in Appendix D.

URIs

Each web server resource has a name, so clients can point out what resources they are interested in. The server resource name is called a *uniform resource identifier*, or URI. URIs are like the postal addresses of the Internet, uniquely identifying and locating information resources around the world.

Here's a URI for an image resource on Joe's Hardware store's web server:

`http://www.joes-hardware.com/specials/saw-blade.gif`

Figure 1-4 shows how the URI specifies the HTTP protocol to access the saw-blade GIF resource on Joe's store's server. Given the URI, HTTP can retrieve the object. URIs come in two flavors, called URLs and URNs. Let's take a peek at each of these types of resource identifiers now.

URLs

The *uniform resource locator* (URL) is the most common form of resource identifier. URLs describe the specific location of a resource on a particular server. They tell you exactly how to fetch a resource from a precise, fixed location. Figure 1-4 shows how a URL tells precisely where a resource is located and how to access it. Table 1-1 shows a few examples of URLs.

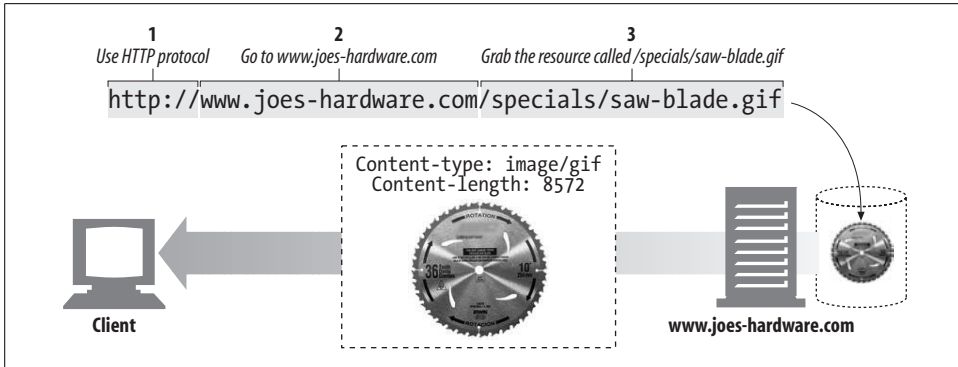


Figure 1-4. URLs specify protocol, server, and local resource

Table 1-1. Example URLs

URL	Description
<code>http://www.oreilly.com/index.html</code>	The home URL for O'Reilly & Associates, Inc.
<code>http://www.yahoo.com/images/logo.gif</code>	The URL for the Yahoo! web site's logo
<code>http://www.joes-hardware.com/inventory-check.cgi?item=12731</code>	The URL for a program that checks if inventory item #12731 is in stock
<code>ftp://joe:tools4u@ftp.joes-hardware.com/locking-pliers.gif</code>	The URL for the <i>locking-pliers.gif</i> image file, using password-protected FTP as the access protocol

Most URLs follow a standardized format of three main parts:

- The first part of the URL is called the *scheme*, and it describes the protocol used to access the resource. This is usually the HTTP protocol (`http://`).
- The second part gives the server Internet address (e.g., `www.joes-hardware.com`).
- The rest names a resource on the web server (e.g., `/specials/saw-blade.gif`).

Today, almost every URI is a URL.

URNs

The second flavor of URI is the *uniform resource name*, or URN. A URN serves as a unique name for a particular piece of content, independent of where the resource currently resides. These location-independent URNs allow resources to move from place to place. URNs also allow resources to be accessed by multiple network access protocols while maintaining the same name.

For example, the following URN might be used to name the Internet standards document “RFC 2141” regardless of where it resides (it may even be copied in several places):

```
urn:ietf:rfc:2141
```

URNs are still experimental and not yet widely adopted. To work effectively, URNs need a supporting infrastructure to resolve resource locations; the lack of such an infrastructure has also slowed their adoption. But URNs do hold some exciting promise for the future. We'll discuss URNs in a bit more detail in Chapter 2, but most of the remainder of this book focuses almost exclusively on URLs.

Unless stated otherwise, we adopt the conventional terminology and use URI and URL interchangeably for the remainder of this book.

Transactions

Let's look in more detail how clients use HTTP to transact with web servers and their resources. An HTTP transaction consists of a request command (sent from client to server), and a response result (sent from the server back to the client). This communication happens with formatted blocks of data called *HTTP messages*, as illustrated in Figure 1-5.

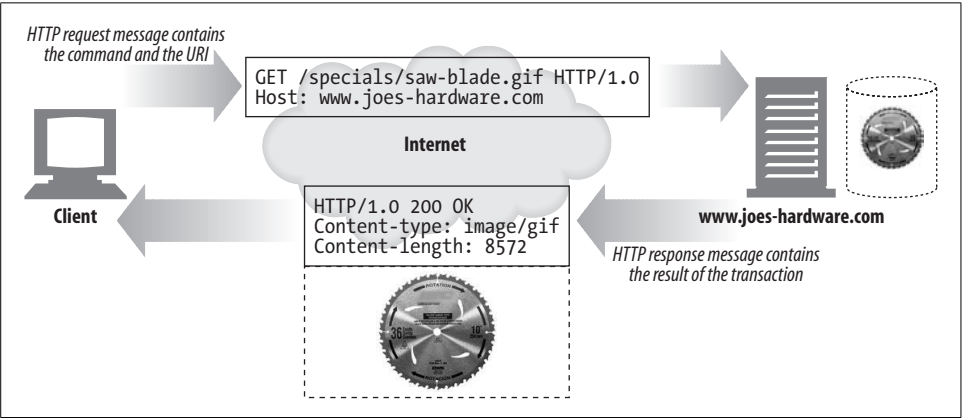


Figure 1-5. HTTP transactions consist of request and response messages

Methods

HTTP supports several different request commands, called *HTTP methods*. Every HTTP request message has a method. The method tells the server what action to perform (fetch a web page, run a gateway program, delete a file, etc.). Table 1-2 lists five common HTTP methods.

Table 1-2. Some common HTTP methods

HTTP method	Description
GET	Send named resource from the server to the client.
PUT	Store data from client into a named server resource.

Table 1-2. Some common HTTP methods (continued)

HTTP method	Description
DELETE	Delete the named resource from a server.
POST	Send client data into a server gateway application.
HEAD	Send just the HTTP headers from the response for the named resource.

We'll discuss HTTP methods in detail in Chapter 3.

Status Codes

Every HTTP response message comes back with a status code. The status code is a three-digit numeric code that tells the client if the request succeeded, or if other actions are required. A few common status codes are shown in Table 1-3.

Table 1-3. Some common HTTP status codes

HTTP status code	Description
200	OK. Document returned correctly.
302	Redirect. Go someplace else to get the resource.
404	Not Found. Can't find this resource.

HTTP also sends an explanatory textual “reason phrase” with each numeric status code (see the response message in Figure 1-5). The textual phrase is included only for descriptive purposes; the numeric code is used for all processing.

The following status codes and reason phrases are treated identically by HTTP software:

```
200 OK
200 Document attached
200 Success
200 All's cool, dude
```

HTTP status codes are explained in detail in Chapter 3.

Web Pages Can Consist of Multiple Objects

An application often issues multiple HTTP transactions to accomplish a task. For example, a web browser issues a cascade of HTTP transactions to fetch and display a graphics-rich web page. The browser performs one transaction to fetch the HTML “skeleton” that describes the page layout, then issues additional HTTP transactions for each embedded image, graphics pane, Java applet, etc. These embedded resources might even reside on different servers, as shown in Figure 1-6. Thus, a “web page” often is a collection of resources, not a single resource.

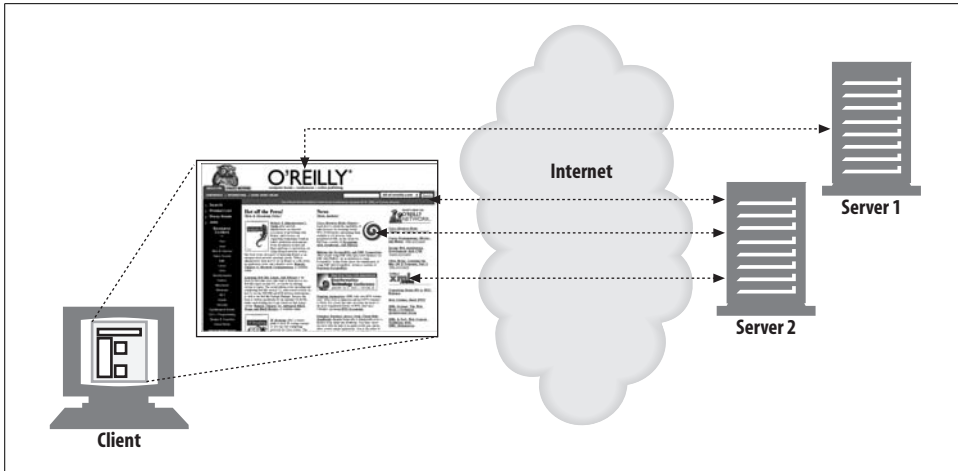


Figure 1-6. Composite web pages require separate HTTP transactions for each embedded resource

Messages

Now let's take a quick look at the structure of HTTP request and response messages. We'll study HTTP messages in exquisite detail in Chapter 3.

HTTP messages are simple, line-oriented sequences of characters. Because they are plain text, not binary, they are easy for humans to read and write.* Figure 1-7 shows the HTTP messages for a simple transaction.

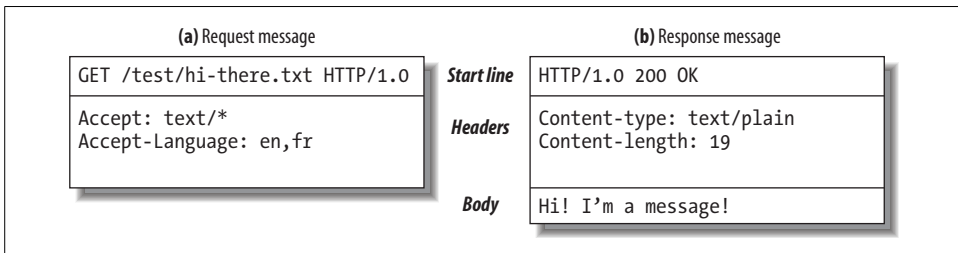


Figure 1-7. HTTP messages have a simple, line-oriented text structure

HTTP messages sent from web clients to web servers are called *request messages*. Messages from servers to clients are called *response messages*. There are no other kinds of HTTP messages. The formats of HTTP request and response messages are very similar.

* Some programmers complain about the difficulty of HTTP parsing, which can be tricky and error-prone, especially when designing high-speed software. A binary format or a more restricted text format might have been simpler to process, but most HTTP programmers appreciate HTTP's extensibility and debuggability.

HTTP messages consist of three parts:

Start line

The first line of the message is the start line, indicating what to do for a request or what happened for a response.

Header fields

Zero or more header fields follow the start line. Each header field consists of a name and a value, separated by a colon (:) for easy parsing. The headers end with a blank line. Adding a header field is as easy as adding another line.

Body

After the blank line is an optional message body containing any kind of data. Request bodies carry data to the web server; response bodies carry data back to the client. Unlike the start lines and headers, which are textual and structured, the body can contain arbitrary binary data (e.g., images, videos, audio tracks, software applications). Of course, the body can also contain text.

Simple Message Example

Figure 1-8 shows the HTTP messages that might be sent as part of a simple transaction. The browser requests the resource `http://www.joes-hardware.com/tools.html`.

In Figure 1-8, the browser sends an HTTP request message. The request has a GET method in the start line, and the local resource is `/tools.html`. The request indicates it is speaking Version 1.0 of the HTTP protocol. The request message has no body, because no request data is needed to GET a simple document from a server.

The server sends back an HTTP response message. The response contains the HTTP version number (HTTP/1.0), a success status code (200), a descriptive reason phrase (OK), and a block of response header fields, all followed by the response body containing the requested document. The response body length is noted in the Content-Length header, and the document's MIME type is noted in the Content-Type header.

Connections

Now that we've sketched what HTTP's messages look like, let's talk for a moment about how messages move from place to place, across Transmission Control Protocol (TCP) connections.

TCP/IP

HTTP is an application layer protocol. HTTP doesn't worry about the nitty-gritty details of network communication; instead, it leaves the details of networking to TCP/IP, the popular reliable Internet transport protocol.

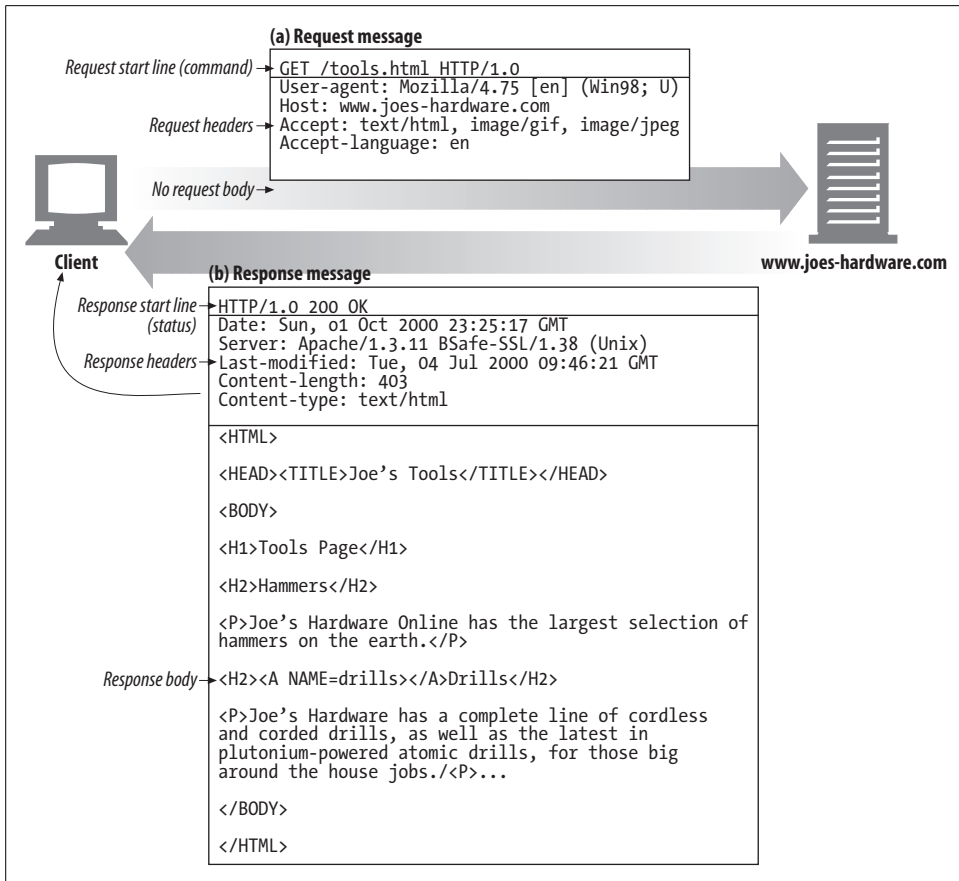


Figure 1-8. Example GET transaction for `http://www.joes-hardware.com/tools.html`

TCP provides:

- Error-free data transportation
- In-order delivery (data will always arrive in the order in which it was sent)
- Unsegmented data stream (can dribble out data in any size at any time)

The Internet itself is based on TCP/IP, a popular layered set of packet-switched network protocols spoken by computers and network devices around the world. TCP/IP hides the peculiarities and foibles of individual networks and hardware, letting computers and networks of any type talk together reliably.

Once a TCP connection is established, messages exchanged between the client and server computers will never be lost, damaged, or received out of order.

In networking terms, the HTTP protocol is *layered* over TCP. HTTP uses TCP to transport its message data. Likewise, TCP is layered over IP (see Figure 1-9).

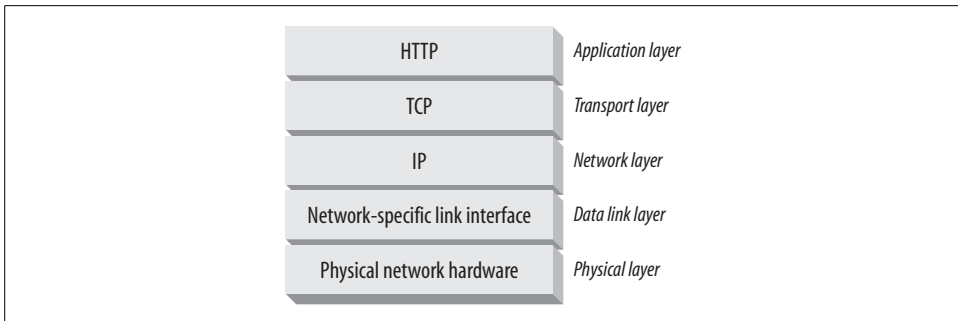


Figure 1-9. HTTP network protocol stack

Connections, IP Addresses, and Port Numbers

Before an HTTP client can send a message to a server, it needs to establish a TCP/IP connection between the client and server using Internet protocol (IP) addresses and port numbers.

Setting up a TCP connection is sort of like calling someone at a corporate office. First, you dial the company's phone number. This gets you to the right organization. Then, you dial the specific extension of the person you're trying to reach.

In TCP, you need the IP address of the server computer and the TCP port number associated with the specific software program running on the server.

This is all well and good, but how do you get the IP address and port number of the HTTP server in the first place? Why, the URL, of course! We mentioned before that URLs are the addresses for resources, so naturally enough they can provide us with the IP address for the machine that has the resource. Let's take a look at a few URLs:

```
http://207.200.83.29:80/index.html
http://www.netscape.com:80/index.html
http://www.netscape.com/index.html
```

The first URL has the machine's IP address, "207.200.83.29", and port number, "80".

The second URL doesn't have a numeric IP address; it has a textual domain name, or *hostname* ("www.netscape.com"). The hostname is just a human-friendly alias for an IP address. Hostnames can easily be converted into IP addresses through a facility called the Domain Name Service (DNS), so we're all set here, too. We will talk much more about DNS and URLs in Chapter 2.

The final URL has no port number. When the port number is missing from an HTTP URL, you can assume the default value of port 80.

With the IP address and port number, a client can easily communicate via TCP/IP. Figure 1-10 shows how a browser uses HTTP to display a simple HTML resource that resides on a distant server.

Here are the steps:

- (a) The browser extracts the server's hostname from the URL.
- (b) The browser converts the server's hostname into the server's IP address.
- (c) The browser extracts the port number (if any) from the URL.
- (d) The browser establishes a TCP connection with the web server.
- (e) The browser sends an HTTP request message to the server.
- (f) The server sends an HTTP response back to the browser.
- (g) The connection is closed, and the browser displays the document.

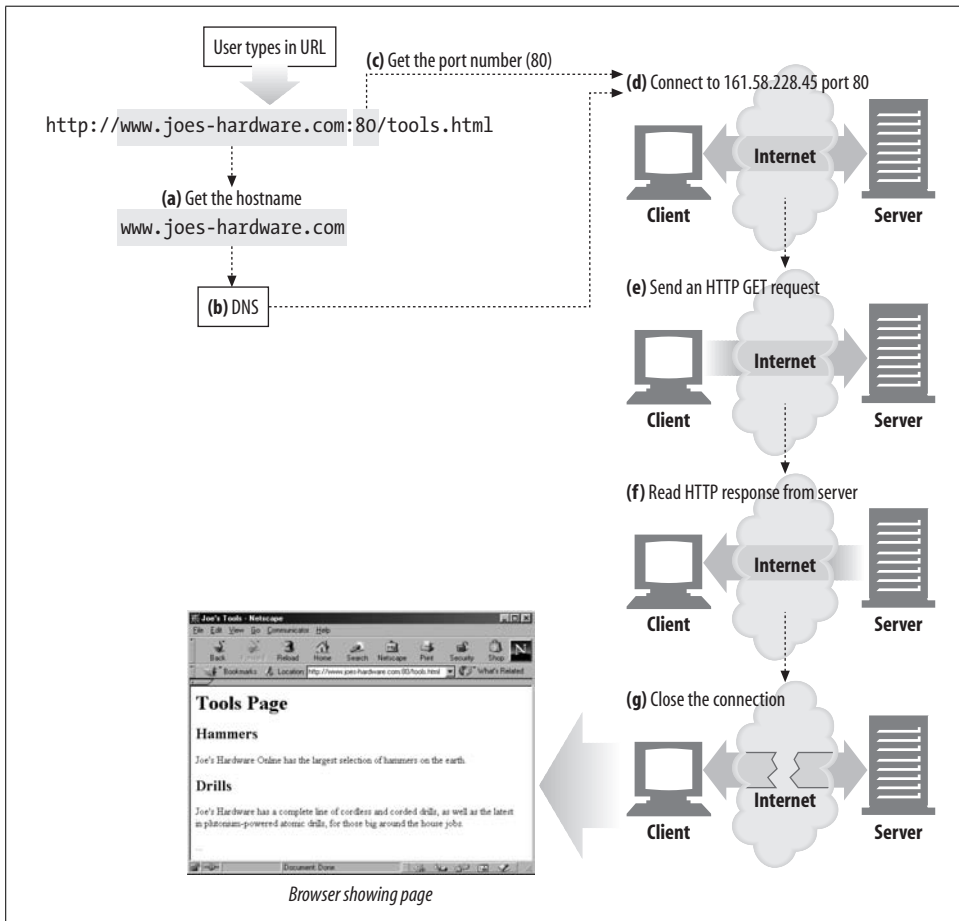


Figure 1-10. Basic browser connection process

Web caches are HTTP devices that automatically keep copies of popular documents. When a web request arrives at a cache, if a local “cached” copy is available, the document is served from the local storage instead of from the origin server. Caches have the following benefits:

- Caches *reduce redundant data transfers*, saving you money in network charges.
- Caches *reduce network bottlenecks*. Pages load faster without more bandwidth.
- Caches *reduce demand on origin servers*. Servers reply faster and avoid overload.
- Caches *reduce distance delays*, because pages load slower from farther away.

In this chapter, we explain how caches improve performance and reduce cost, how to measure their effectiveness, and where to place caches to maximize impact. We also explain how HTTP keeps cached copies fresh and how caches interact with other caches and servers.

Redundant Data Transfers

When multiple clients access a popular origin server page, the server transmits the same document multiple times, once to each client. The same bytes travel across the network over and over again. These redundant data transfers eat up expensive network bandwidth, slow down transfers, and overload web servers. With caches, the cache keeps a copy of the first server response. Subsequent requests can be fulfilled from the cached copy, reducing wasteful, duplicate traffic to and from origin servers.

Bandwidth Bottlenecks

Caches also can reduce network bottlenecks. Many networks provide more bandwidth to local network clients than to remote servers (Figure 7-1). Clients access servers at the speed of the slowest network on the way. If a client gets a copy from a cache on a fast LAN, caching can boost performance—especially for larger documents.

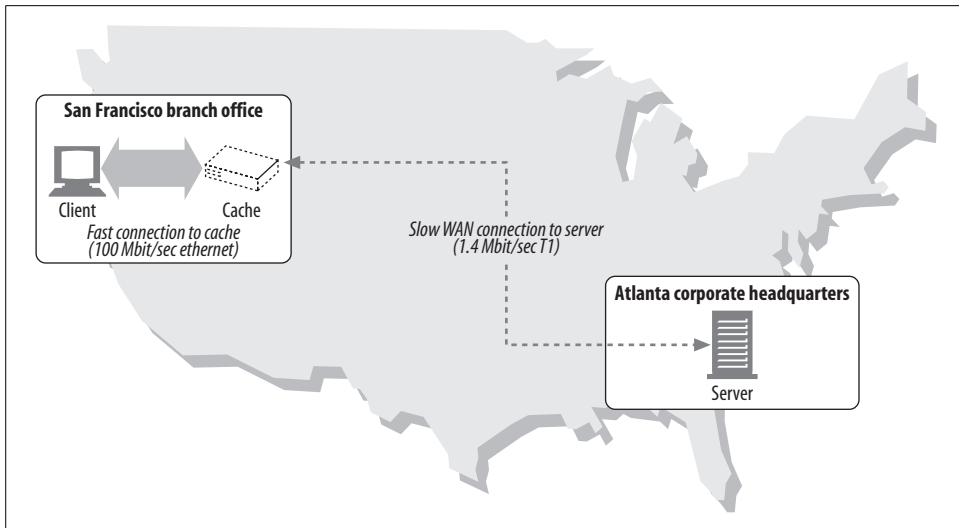


Figure 7-1. Limited wide area bandwidth creates a bottleneck that caches can improve

In Figure 7-1, it might take 30 seconds for a user in the San Francisco branch of Joe’s Hardware, Inc. to download a 5-MB inventory file from the Atlanta headquarters, across the 1.4-Mbps T1 Internet connection. If the document was cached in the San Francisco office, a local user might be able to get the same document in less than a second across the Ethernet connection.

Table 7-1 shows how bandwidth affects transfer time for a few different network speeds and a few different sizes of documents. Bandwidth causes noticeable delays for larger documents, and the speed difference between different network types is dramatic.* A 56-Kbps modem would take 749 seconds (over 12 minutes) to transfer a 5-MB file that could be transported in under a second across a fast Ethernet LAN.

Table 7-1. Bandwidth-imposed transfer time delays, idealized (time in seconds)

	Large HTML (15 KB)	JPEG (40 KB)	Large JPEG (150 KB)	Large file (5 MB)
Dialup modem (56 Kbit/sec)	2.19	5.85	21.94	748.98
DSL (256 Kbit/sec)	.48	1.28	4.80	163.84
T1 (1.4 Mbit/sec)	.09	.23	.85	29.13
Slow Ethernet (10 Mbit/sec)	.01	.03	.12	4.19
DS3 (45 Mbit/sec)	.00	.01	.03	.93
Fast Ethernet (100 Mbit/sec)	.00	.00	.01	.42

* This table shows just the effect of network bandwidth on transfer time. It assumes 100% network efficiency and no network or application processing latencies. In this way, the delay is a lower bound. Real delays will be larger, and the delays for small objects will be dominated by non-bandwidth overheads.

Flash Crowds

Caching is especially important to break up flash crowds. Flash crowds occur when a sudden event (such as breaking news, a bulk email announcement, or a celebrity event) causes many people to access a web document at nearly the same time (Figure 7-2). The resulting redundant traffic spike can cause a catastrophic collapse of networks and web servers.

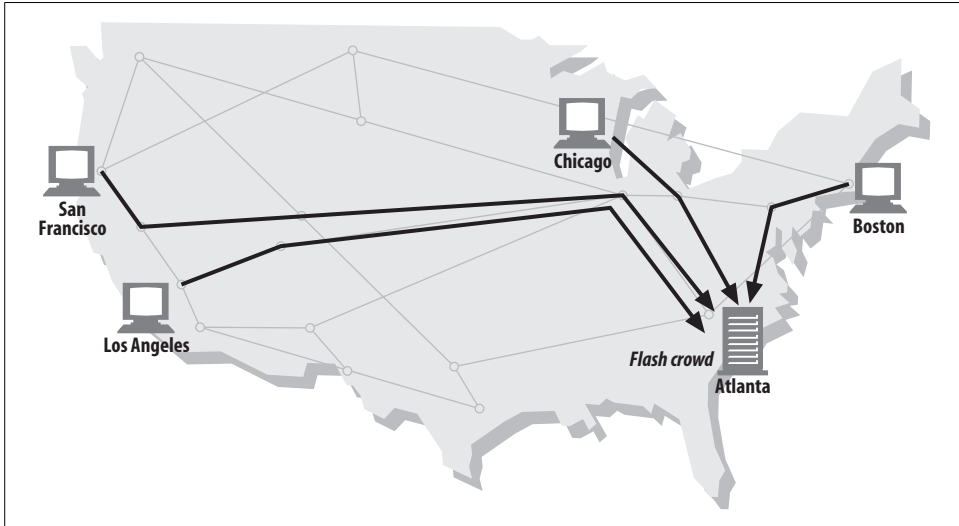


Figure 7-2. Flash crowds can overload web servers

When the “Starr Report” detailing Kenneth Starr’s investigation of U.S. President Clinton was released to the Internet on September 11, 1998, the U.S. House of Representatives web servers received over 3 million requests per hour, 50 times the average server load. One news web site, CNN.com, reported an average of over 50,000 requests every second to its servers.

Distance Delays

Even if bandwidth isn’t a problem, distance might be. Every network router adds delays to Internet traffic. And even if there are not many routers between client and server, the speed of light alone can cause a significant delay.

The direct distance from Boston to San Francisco is about 2,700 miles. In the very best case, at the speed of light (186,000 miles/sec), a signal could travel from Boston to San Francisco in about 15 milliseconds and complete a round trip in 30 milliseconds.*

* In reality, signals travel at somewhat less than the speed of light, so distance delays are even worse.

Say a web page contains 20 small images, all located on a server in San Francisco. If a client in Boston opens four parallel connections to the server, and keeps the connections alive, the speed of light alone contributes almost 1/4 second (240 msec) to the download time (Figure 7-3). If the server is in Tokyo (6,700 miles from Boston), the delay grows to 600 msec. Moderately complicated web pages can incur several seconds of speed-of-light delays.

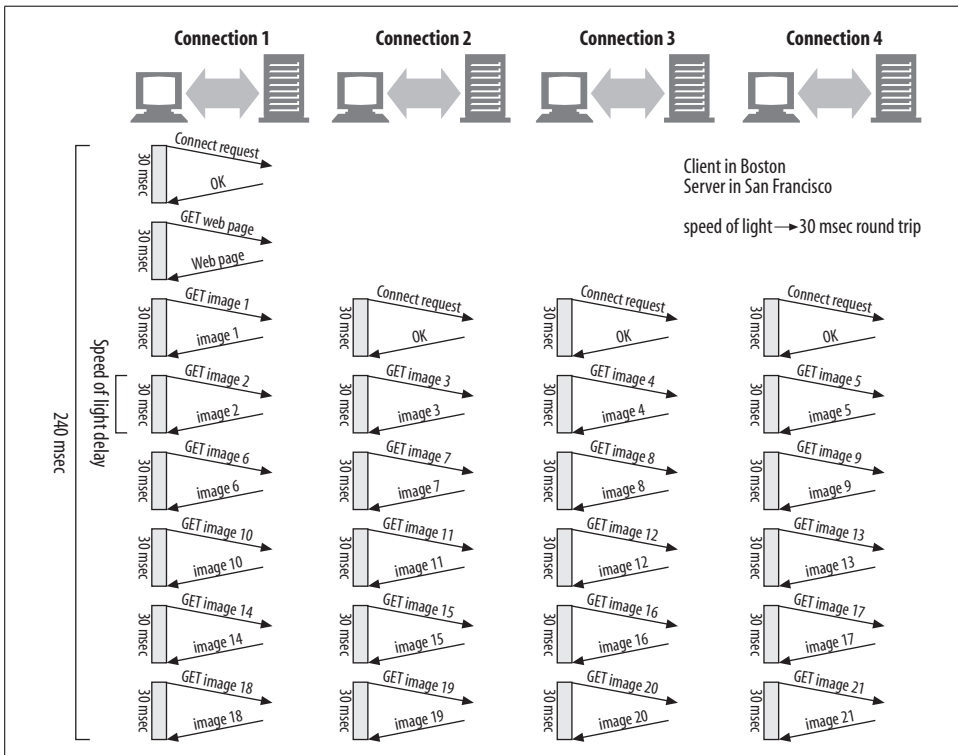


Figure 7-3. Speed of light can cause significant delays, even with parallel, keep-alive connections

Placing caches in nearby machine rooms can shrink document travel distance from thousands of miles to tens of yards.

Hits and Misses

So caches can help. But a cache doesn't store a copy of *every* document in the world.*

* Few folks can afford to buy a cache big enough to hold all the Web's documents. And even if you could afford gigantic "whole-Web caches," some documents change so frequently that they won't be fresh in many caches.

Some requests that arrive at a cache can be served from an available copy. This is called a *cache hit* (Figure 7-4a). Other requests arrive at a cache only to be forwarded to the origin server, because no copy is available. This is called a *cache miss* (Figure 7-4b).

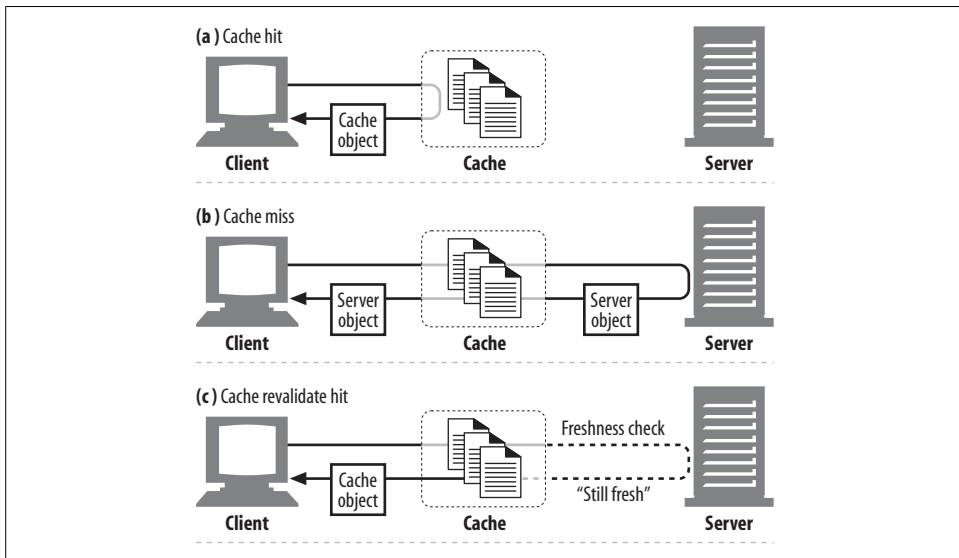


Figure 7-4. Cache hits, misses, and revalidations

Revalidations

Because the origin server content can change, caches have to check every now and then that their copies are still up-to-date with the server. These “freshness checks” are called HTTP *revalidations* (Figure 7-4c). To make revalidations efficient, HTTP defines special requests that can quickly check if content is still fresh, without fetching the entire object from the server.

A cache can revalidate a copy any time it wants, and as often as it wants. But because caches often contain millions of documents, and because network bandwidth is scarce, most caches revalidate a copy only when it is requested by a client and when the copy is old enough to warrant a check. We’ll explain the HTTP rules for freshness checking later in the chapter.

When a cache needs to revalidate a cached copy, it sends a small revalidation request to the origin server. If the content hasn’t changed, the server responds with a tiny 304 Not Modified response. As soon as the cache learns the copy is still valid, it marks the copy temporarily fresh again and serves the copy to the client (Figure 7-5a). This is called a *revalidate hit* or a *slow hit*. It’s slower than a pure cache hit, because it *does* need to check with the origin server, but it’s faster than a cache miss, because no object data is retrieved from the server.