CHAPTER

Hypertext Transfer Protocol

22

What You Will Learn

In this chapter, you will learn about the HTTP protocol used on the Web, including the major message types and HTTP methods. We'll also discuss the status codes and headers used in HTTP.

You will learn how URLs are structured and how to decipher them. We'll also take a brief look at the use of cookies and how they apply to the Web.

After email, the World Wide Web is probably the most common TCP/IP application general users are familiar with. In fact, many users access their email through their Web browser, which is a tribute to the versatility of the protocols used to make the Web such a vital part of the Internet experience.

There is no need to repeat the history of the Web and browser, which are covered in other places. It is enough to note here that the Web browser is a type of "universal client" that can be used to access almost any type of server, from email to the file transfer protocal (FTP) and beyond. The unique addressing and location scheme employed with a browser along with several related protocols combine to make "surfing the Web" (it's really more like fishing or trawling) an essential part of many people's lives around the world.

The protocol used to convey formatted Web pages to the browser is the Hypertext Transfer Protocol (HTTP). Often confused with the Web page formatting standard, the Hypertext Markup Language (HTML), it is HTTP we will investigate in this chapter. The more one learns about how the Hypertext Transfer Protocol and the browser interact with the Web site and TCP/IP, the more impressed people tend to become with the system as a whole. The wonder is not that browsers sometimes freeze or open unwanted windows or let worms wiggle into the host but that it works effectively and efficiently at all.

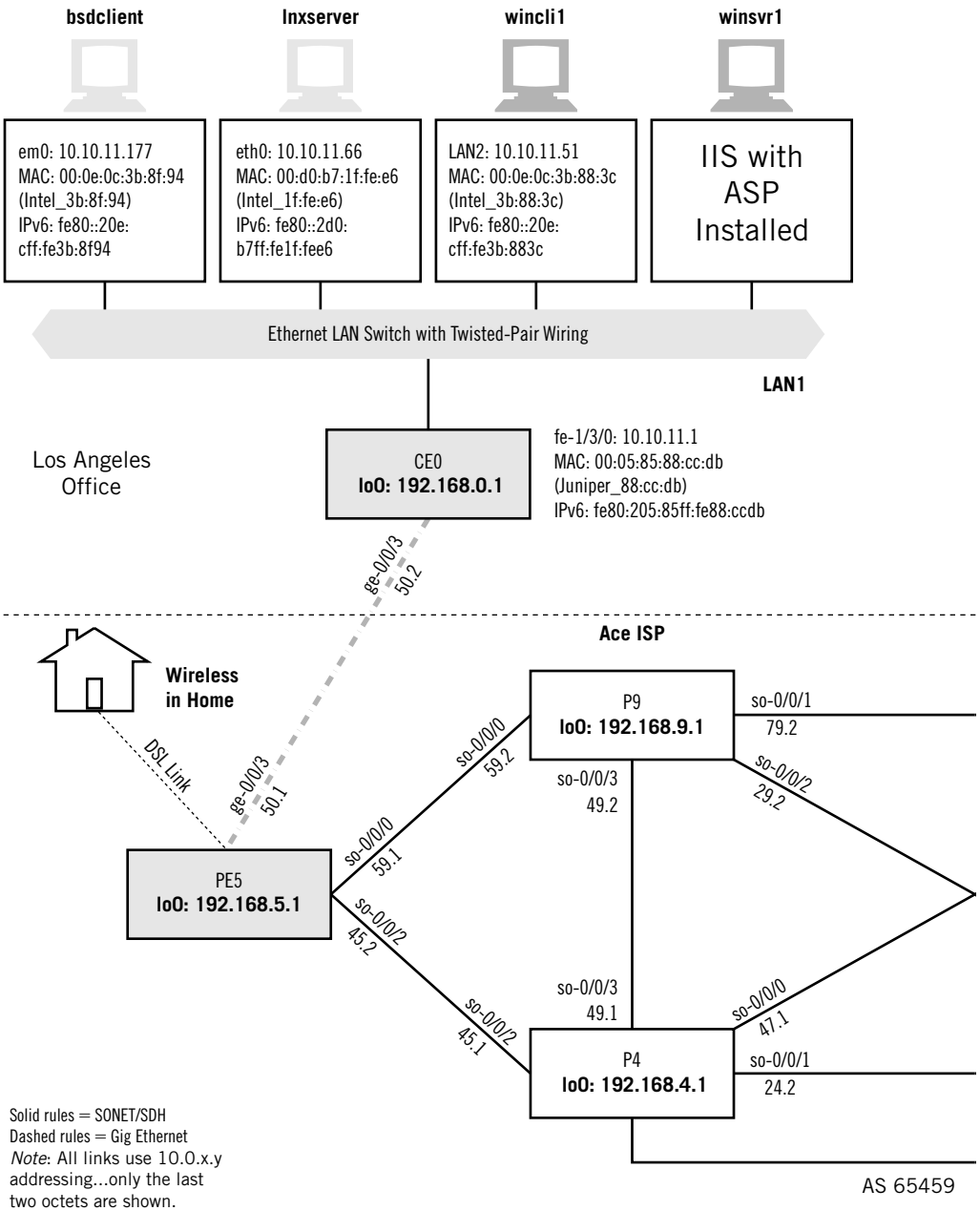
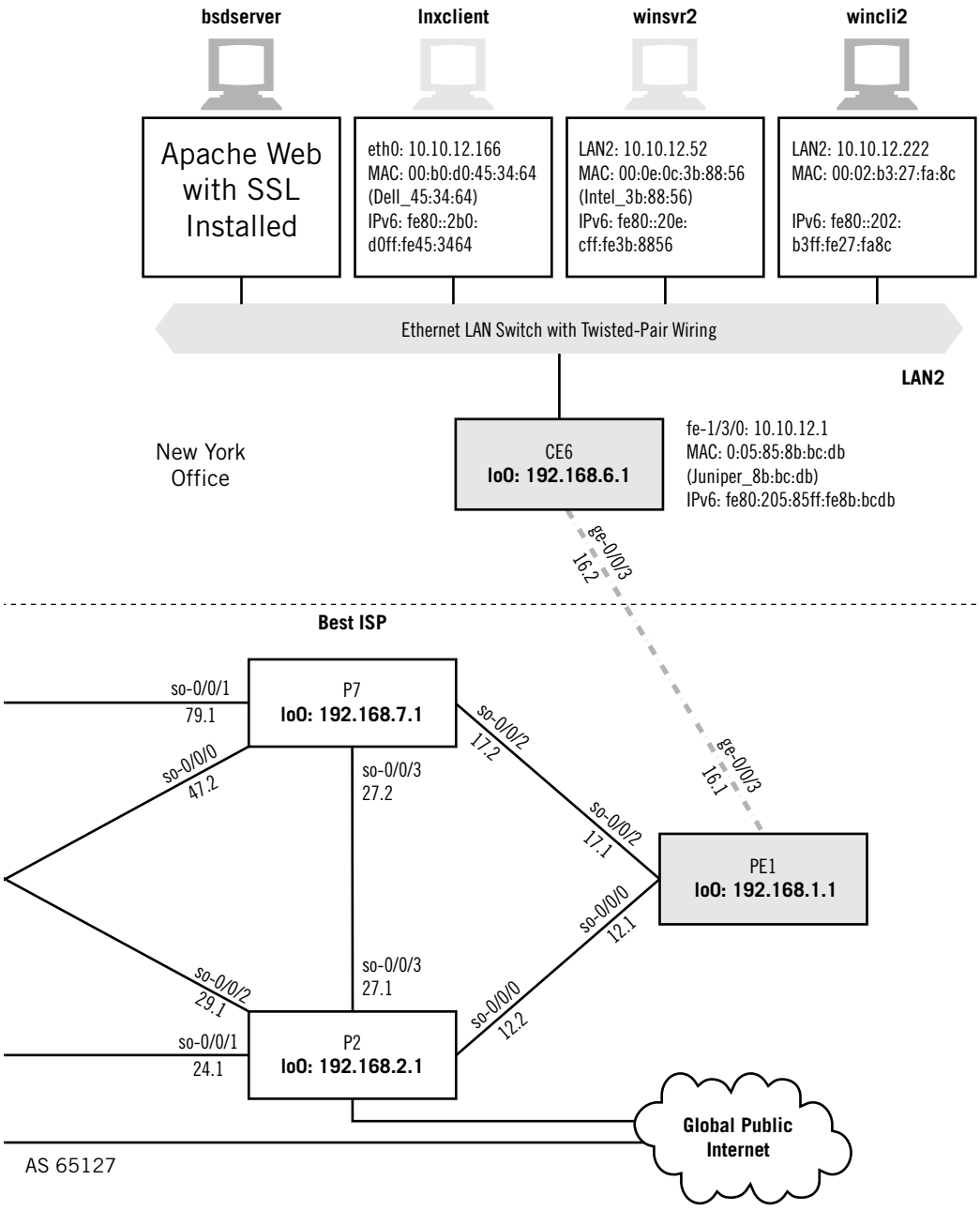


FIGURE 22.1

The Web servers on the Illustrated Network, also showing the major client browser hosts. Note that we'll be using IIS with ASP on the Windows platform and Apache with SSL on the Unix host.


bsdserver
lnxclient
winsvr2
wincli2
Apache Web with SSL Installed
eth0: 10.10.12.166
MAC: 00:b0:d0:45:34:64
(Dell_45:34:64)
IPv6: fe80::2b0:
d0ff:fe45:3464
LAN2: 10.10.12.52
MAC: 00:0e:0c:3b:88:56
(Intel_3b:88:56)
IPv6: fe80::20e:
cff:fe3b:8856
LAN2: 10.10.12.222
MAC: 00:02:b3:27:fa:8c
IPv6: fe80::202:
b3ff:fe27:fa8c
Ethernet LAN Switch with Twisted-Pair Wiring
LAN2
New York Office
CE6
lo0: 192.168.6.1
fe-1/3/0: 10.10.12.1
MAC: 0:05:85:8b:bc:db
(Juniper_8b:bc:db)
IPv6: fe80:205:85ff:fe8b:bcdb
ge-0/0/3
16.2
Best ISP
so-0/0/1
79.1
P7
lo0: 192.168.7.1
so-0/0/2
17.2
so-0/0/0
47.2
so-0/0/3
27.2
ge-0/0/3
16.1
so-0/0/2
17.1
PE1
lo0: 192.168.1.1
so-0/0/0
12.1
so-0/0/3
27.1
so-0/0/2
29.1
so-0/0/0
12.2
so-0/0/1
24.1
P2
lo0: 192.168.2.1
Global Public Internet
AS 65127

HTTP IN ACTION

Web browsers and Web servers are perhaps even more familiar than electronic mail, but nevertheless there are some interesting things that can be explored with HTTP on the Illustrated Network. In this chapter, Windows hosts will be used to maximum effect. Not that the Linux and FreeBSD hosts could not run GUI browsers, but the "purity" of Unix is in the command line (not the GUI).

We'll use the popular Apache Web server software and install it on `bsdserver`. Just to make it interesting (and to prepare for the next chapter), we'll install Apache with the Secure Sockets Layer (SSL) module, which we'll look at in more detail in the next chapter. We'll also be using `winsrv1` and the two Windows clients, `wincli1` and `wincli2`, as shown in Figure 22.1.

We could install Apache for Windows XP as well, because one of the goals of this book is to explore how much can be done with basic Windows XP Professional. But we don't want to go into full-blown server operating systems and build a complete Windows server. It should be noted that many Unix hosts are used exclusively as Web sites or email servers, but here we're only exploring the basics of the protocols and applications, not their ability or relative performance.

The Web has changed a lot since the early days of statically defined content delivered with HTTP. Now it's common for the Web page displayed to be built on fly on the server, based on the user's request. There are many ways to do this, from good old Perl to Java and beyond, all favored and pushed by one vendor or platform group or another. In Windows, the "in-house" dynamic Web page software is called Active Service Pages (ASP). ASP works differently than the others, but all of them vary in large or small ways, so that's not really a criticism.

So, we'll install Integrated Information Services (IIS), available for Windows XP Pro and a few other (free) packages, notably the .NET Framework and Software Development Kit (SDK). This will make it possible for us to build ASP Web pages on `winsrv1` and access them with a browser.

The ASP installation was rather torturous, but there are invaluable Web sites and books that take you through the process step by step. One book includes an extremely simple Web page along the lines of "Hello World!" (but the Web page is also small enough to demonstrate how HTTP fetches the page). Figure 22.2 shows how the page looks in the browser window on `wincli2`.

What does the HTTP exchange look like between the client and server? Let's capture it with Ethereal and see what we come up with. Figure 22.3 shows the result.

Not surprisingly, after the TCP handshake the content is transferred with a single HTTP request and response pair. The entire page fit in one packet, which is detailed in the figure. And just as it should, once TCP acknowledges the transfer the connection stays open (persistent).

Note that the dynamic date and time content is transferred as a static string of text. All of the magic of dynamic content takes place on the server's "back room" and does not involve HTTP in the least.

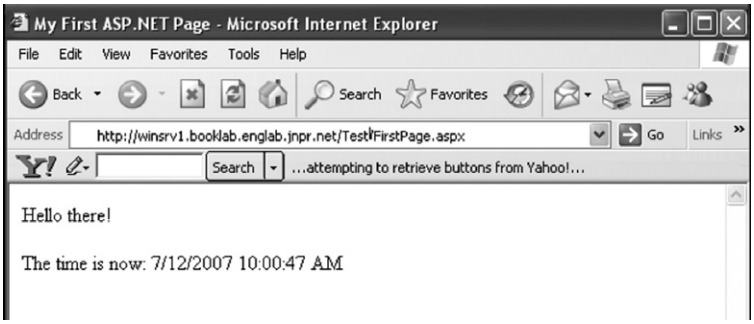


FIGURE 22.2

An ASP page from winsrv1. The "active" component means that the date and time on the page are kept current.

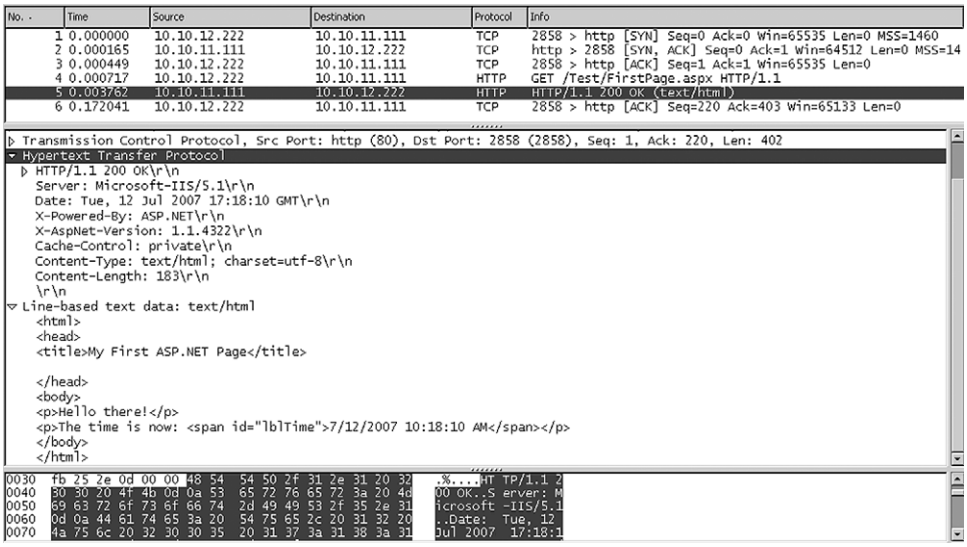


FIGURE 22.3

Capture of the HTTP for the ASP page, showing how the protocol identifies the "make and model" of the Web site (Microsoft IIS using ASP.NET).

What about more involved content? Let's see what the default Apache with SSL page looks like from `wincli2` when we install it on `bsdserver`. This is shown in Figure 22.4.

This is just the default index.html page showing that Apache installed successfully. There is no "real" SSL on this page, however. There is no security or encryption

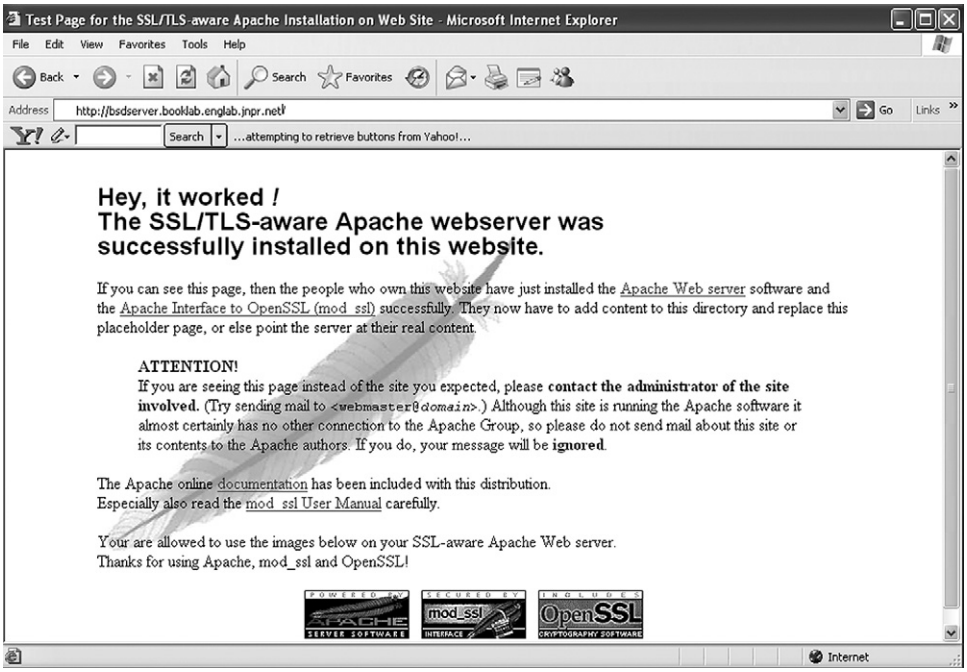


FIGURE 22.4

Apache HTTP "success" page displayed when the software is installed correctly.

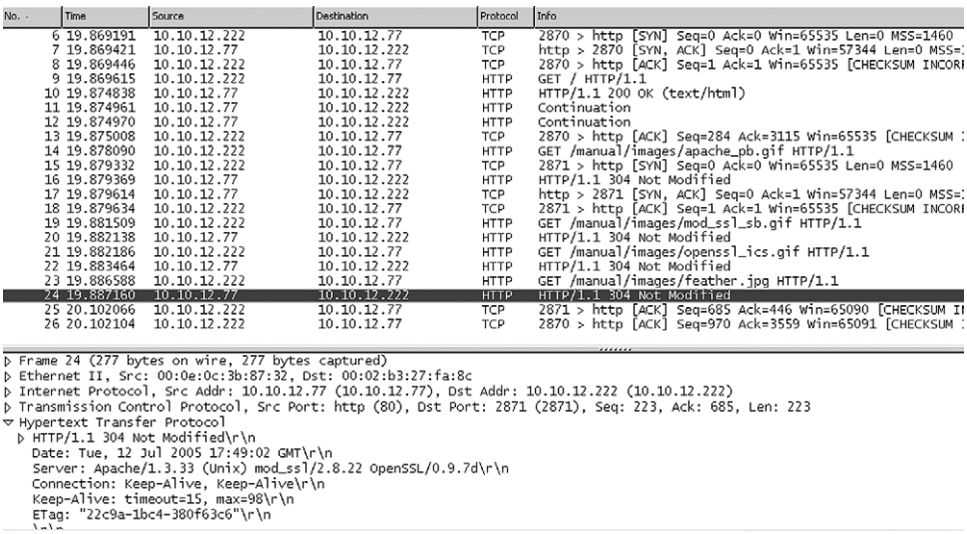


FIGURE 22.5

HTTP Apache capture. Most of the text is transferred in only a few packets.

involved. What does the HTTP capture look like now? It's captured on `wincli2` (shown in Figure 22.5).

This exchange involved 21 packets, and would have been longer if the image had not been cached on the client (a simple "Not Modified" string is all that is needed to fetch it onto the page). Most of the text is transferred in packets 10 through 12, and then the images on the page are "filled in." We'll take a look at the SSL aspects of this Web site in the next chapter.

Before getting into the nuts and bolts of HTTP, there is a related topic that must be investigated first. This is an appreciation of the addressing system used by browsers and Web servers to locate the required information in whatever form it may be stored. There are three closely related systems defined for the Internet (not just the Web). These are uniform resource identifiers (URIs), locators (URLs), and names (URNs).

Uniform Resources

As if it weren't enough to have to deal with MAC addresses, IP addresses, ports, sockets, and email addresses, there is still another layer of addresses used in TCP/IP that has to be covered. These are "application layer" addresses, and unlike most of the other addresses (which are really defined by the needs of the particular protocol) application layer addresses are most useful to humans.

This is not to say that the addresses we are talking about here are the same as those used in DNS, where a simple correspondence between IP address `192.168.77.22` and the name `www.example.com` is established. As is fitting for the generalized Web browser, the addresses used are "universal"—and that was one name for them before someone figured out that they weren't really *universal* quite yet, but they were at least *uniform*.

So, labels were invented not only to tell the browser which host to go to and application use but what *resources* the browser was expecting to find and just where they were located. Let's start with the general form for these labels, the URI.

URIs

The generic term for resource location labels in TCP/IP is *URI*. One specific form of URI, used with the Web, is the URL. The use of URLs as an instance of URIs has become so commonplace that most people don't bother to distinguish the two, but they are technically distinct.

The latest work on URIs is RFC 2396, which updated several older RFCs (including RFC 1738, which defines URLs). In the RFC, a URI is simply defined as "a compact string of characters for identifying an abstract or physical resource." There is no mention of the Web specifically, although it was the popularity of the Web that led to the development of uniform resource notations in the first place.

When a user accesses `http://www.example.com` from a Web browser, that string is a URI as much as a URL. So, what's the difference between the URI and the URL?

URLs

RFC 1738 defined a URL format for use on the Web (although the RFC just says "Internet"). Newer URI rules all respect conventions that have grown up around URLs over the years. URLs are a subset of URIs, and like URIs, consist of two parts: a *method* used to access the resource, and the *location* of the resource itself. Together, the parts of the URL provide a way for users to access files, objects, programs, audio, video, and much more on the Web.

The method is labeled by a scheme, and usually refers to a TCP/IP application or protocol, such as `http` or `ftp`. Schemes can include plus signs (+), periods (.), or hyphens (-), but in practice they contain only letters. Methods are case insensitive, so `HTTP` is the same as `http` (but by convention they are expressed in lowercase letters).

The locator part of the URL follows the scheme and is separated from it by a colon and two forward slashes (`://`). The format or the locator depends on the type of scheme, and if one part of the locator is left out, default values come into play. The scheme-specific information is parsed by the received host based on the actual scheme (method) used in the URL.

Theoretically, each scheme uses an independently defined locator. In practice, because URLs use TCP/IP and Internet conventions many of the schemes share a common syntax. For example, both `http` and `ftp` schemes use the DNS name or IP address to identify the target host and expect to find the resource in a hierarchical directory file structure.

The most general form of URL for the Web is shown in Figure 22.6. There is very little difference between this format and the general format of a URI, and some of these differences are mentioned in the material that follows the figure.

The format changes a bit with method, so an FTP URL has only a `type=<typecode>` field as the single `<params>` field following the `<url-path>`. For example, a type code of `d` is used to request an FTP directory listing. The figure shows the general field for the `http` method.

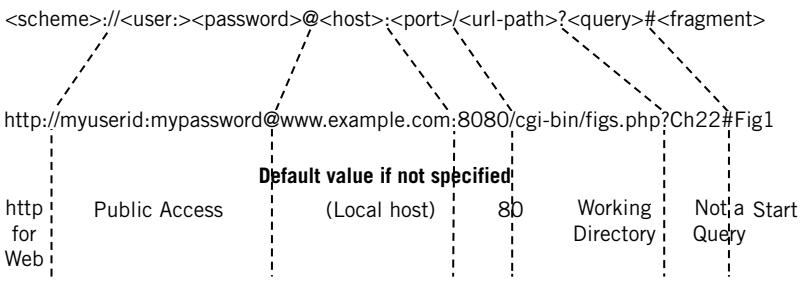


FIGURE 22.6

The fields of a complete URL, showing that the default values used in the fields are absent.

`<scheme>`—The method used to access the resource. The default method for a Web browser is `http`.

`<user>` and `<password>`—In a URI, this is the `authorization` field. A URL's authorization consists of a user ID and password separated by a colon (:). Many private Web sites require user authorization, and if not provided in the URL the user is prompted for this information. When absent, the user defaults to publicly available resource access.

`<host>`—Called the `networkpath` in a URI, the host is specified in a URL by DNS name or IP address (IPv6 works fine for servers using that address form).

`<port>`—This is the TCP or UDP port that together with the host information specifies the socket where the method appropriate to the scheme is found. For `http`, the default port is 80.

`<url-path>`—The URI specification calls this the `absolutepath`. In a URL, this is usually the directory path starting from the default directory to where the resource is to be found. If this field is absent, the Web site has a default directory into which the user is placed. The forward slash (/) before the path is not technically part of the path, but forms the delimiter and must follow the port. If the `url-path` ends in another slash, this means a directory and not a "file" (but most Web sites figure out whether the path ends at a file or directory on their own). A double dot (..) moves the user up one level from the default directory.

`<params>`—These parameters control how the method is used on the resource and are scheme specific. Each parameter has the form <parameter>=<value> and the parameters are separated by semicolons (;). If there are no parameters, the default action for the resource is taken.

`<query>`—This URL field contains information used by the server to form the response. Whereas parameters are scheme specific, query information is resource specific.

`<fragment>`—The field is used to indicate which particular part of the resource the user is interested in. By default, the user is presented with the start of the entire resource.

Most of the time, a simple URL, such as `ftp://ftp.example.com`, works just fine for users. But let's look at a couple of examples of fairly complex URLs to illustrate the use of these fields.

http://myself:mypassword@mail.example.com:32888/mymail/ShowLetter?MsgID-5551212#1

The user `myself`, authenticated with `mypassword`, is accessing the `mail.example.com` server at TCP port `32888`, going to the directory `/mymail`, and running the `ShowLetter`

program. The letter is identified to the program as `MsgID-5551212`, and the first part of the message is requested (this form is typically used for a multipart MIME message).

www.examplephotos.org:8080/cgi-bin/pix.php?WeddingPM#Reception19

The user is going to a publicly accessible part of the site called `www.examplephotos.org`, which is running on TCP port `8080` (a popular alternative or addition to port 80). The resource is the PHP program `pix.php` in the `cgi-bin` directory below the default directory, and the URL asks for a particular page of photographs to be accessed (`WeddingPM`) and for a particular photograph (`Reception19`) to be presented.

www.sample.com/who%20are%20you%3F

File names that have embedded spaces and special characters that are the same as URL delimiters can be a problem. This URL accesses a file named `who are you?` in the default directory at the `www.sample.com` site. There are 21 "unsafe" URL characters that can be represented this way.

There are many other URL "rules" (as for Windows files), and quite a few tricks. For example, if we wanted to make a Web page at `www.loserexample.com` (IP address `192.168.1.1`) appear as if it is located at `www.nobelprizewinners.org`, we can translate the Web site's IP address to decimal (`192.168.1.1` = `0xC0A80101` = 3232235777 decimal), add some "bogus" authentication information in front of it (which will be ignored by the Web site), and hope that no one remembers the URL formatting rules:

http://www.nobelprizewinners.org@3232235777

A lot of evil hackers use this trick to make people think they are pointing and clicking at a link to their bank's Web site when they are really about to enter their account information into the hacker's server! Well, if that's what a URL is for, why is a URN needed?

URNs

URNs extend the URI and URL concept beyond the Web, beyond the Internet even, right into the ordinary world. URIs and URLs proved so popular that the system was extended to become URNs. URNs, first proposed in RFC 2141, would solve a particularly vexing problem with URLs.

It may be a tautology, but a URL specifies resources by *location*. This can be a problem for a couple of reasons. First, the resource (such as a freeware utility program) could exist on many Web servers, but if it is not on the one the URL is pointing to the familiar `HTTP 404 - NOT FOUND` error results. And how many times has a Web site moved, changing name or IP address or both—leaving thousands of pages with embedded links to the stale information? (URLs do not automatically supply a helpful "You are being directed to our new site" message.)

As expected, URNs label resources by a *name* rather than a location. The familiar Web URL is a little like going by address to a particular house on a particular street

and asking for Joe Smith. A URN is like asking for Joe Smith, getting an answer from a "resolver," and going to the current address where good old Joe is found. "Joe Smith" is an example of a URN in the human "namespace." Of course, if this is to work properly there can only be one Joe Smith in the world.

Any namespace that can be used to uniquely identify *any* type of resource can be used as a URN. But before you rush out to invent a URN system for automobiles, for example, keep in mind that designing URNs for new namespaces is not that easy.

Each URN must be recognized by some official body or another, and must be strictly defined by a formal language. It's not enough to say that the URN string will identify a car. It is necessary to define things such as the length of the string and just what is allowed in the string and what isn't (actually, there's a lot more to it than that).

For example, the International Standard Book Number (ISBN) system uniquely identifies books published all over the world. Part of the number identifies region of the world where the book is published, another part the publisher, yet another part the particular book, and finally there is a checksum digit that is computed in case someone makes a mistake writing down one of the other parts. The formal definition of the ISBN namespace would establish the length of these fields, and note that the ISBN must be 10 digits long and can only be made up of the digits 0 through 9, except for the last checksum digit, where the Roman numeral X is used for the checksum 10 (10 is a valid ISBN checksum "digit"). The general format of a URN is `URN:<namespace-ID>:<resource-identifier>`.

Note the lack of any sense of location. The namespace ID is needed to distinguish a 10-digit telephone number from a 10-digit ISBN numbers (for example), and the `URN` literally makes it obvious that the URN notation system is being employed.

Work on URNs has been slow. A resource identified by URN still has a location, and so must still provide one or more URLs (think of all the places where a certain book might be located) to the user. A series of RFCs, from RFC 3401 to RFC 3406, defines a system of URN "resolvers" called the *Dynamic Delegation Discovery System* (DDDS). For now, the Internet will have to make do with URLs.

HTTP

HTTP started out as a very simple protocol, based on the familiar scheme of a small set of commands issued by the client (browser) and reply codes and related information issued by the server (Web site). As indicated by the name, the original HTTP (and HTML) concerned itself with hyper*text*, the idea being to embed active links in textual information and allow users to spontaneously follow their instincts from page to page and site to site around the Internet and around the world. There were also graphics associated with the Web almost immediately, and this was a startling enough innovation to completely change the user perception of the Internet.

The original version of HTTP, now called HTTP 0.9, was just something people did if they wanted their Web sites to work, and nobody bothered to write down much about it. The people who wanted to know found out how it worked. This was fine for a few years, but once the Web got rolling RFC 1945 in 1996 defined HTTP 1.0 (a more

full-blooded protocol)—which made "old" HTTP into HTPP 0.9. Then HTTP 1.1 came along in 1997 with RFC 2068, which was extended in 1999 with RFC 2616. And that was pretty much it. The basic HTTP 1.1 is what we live and work with on the Internet today.

However, it's always good to remember what HTTP is and isn't. HTTP is just a transport mechanism for Web stuff, and not only for varied *content*. HTTP is flexible enough to transport Web features such as cascading style sheets (CCSs), Java Applets, Active Server Pages (ASPs), Perl scripts, and any one of the half dozen of so languages and programming tools that have evolved to make Web servers more complex and paradoxically easier to configure and use.

The Evolution of HTTP

HTTP began as a simple TCP/IP request/response language using TCP to retrieve information from a server in a *stateless* manner (most TCP/IP applications are stateless). Because the server is stateless, the server has no idea of any history of the interaction between client and server. Therefore, any state information has to be stored in the client. We'll talk about *cookies* later, after looking at the basics of HTTP.

With HTTP 0.9, a basic browser accessed a Web page by issuing a `GET` command for the page desired (indicated in the URL), accompanied by a number of HTTP *headers*. This was sent over a TCP connection established between the browser port and port 80 (the default Web port) on the server. The server responded with the text-based Web page marked up in HTML and closed the TCP session. The initial browser command was usually `GET /index.`html.

But what about the graphics and audio in the reply, if included in the Web page? HTML is a *markup* language, meaning that special *tags* are inserted into an ordinary text file to control the appearance of the Web page on the browser screen. Once the initial request transfer was made in HTTP 0.9, the browser parsed the HTML tags and opened a separate TCP connection to the server for every *element* of the page. This is why the location of the graphics and associated media files are so important in HTML: they aren't really "there" on the page in any sense until HTTP is used to fetch them.

Naturally, the TCP overhead involved with all of this shuttling of information was staggering, especially on slow dial-up links and when Web pages grew to include 30 or more elements. Some Web sites shut down as the "listen" queues filled up, router links became saturated with TCP overhead, and browsers hung as frustrated users began pounding and clicking everything in sight (one old Internet Explorer message box begged "Stop doing that!").

Interim solutions were not particularly effective. Many solutions made use of massive caching of Web pages on "intermediate systems" that were closer to the perceived user pool, and many businesses used "proxy servers" (an old Internet security mechanism pressed into service as a caching storehouse). Caching Web pages became so common that Internet gurus felt compelled to remind everyone that the point of TCP was that it was an *end-to-end* protocol and that fetching Web pages from caches from proxy servers was not the same as the real thing.

So, HTTP evolved to make the entire process more efficient. HTTP 1.0 created a true messaging protocol and added support for MIME types, adapted for the Web, and addressed some of the issues with HTTP 0.9 (but not all). In addition, vendors had been incrementally adding features here and there haphazardly. HTTP 1.1 brought all of these changes under one specification. In particular, HTTP 1.1 added:

Persistent connections: A client can send multiple requests for related resources in a single TCP session.

Pipelining—Persistent connections permitted clients to pipeline requests to the server. If the browser requests images 1, 2, and 3 from the server, the client does not have to wait for a response to the image 1 request before requesting file 2. This allows the server to handle requests much more efficiently.

Multiple host name support—Web sites could now run more than one Web server per IP address and host name. Today, one Web server can handle requests for literally hundreds of individual Web sites, all running as "virtual hosts" on the server.

Partial resource selection—A client can ask for only part of a document of resource.

Content negotiation—The client and server can exchange information to allow the client to select the best format for a resource, such as MP3 or WAV format for audio files (the formats must be available on the server, of course). This negotiation is not the same as presenting format options to the user.

Better security—Authentication was added to HTTP interactions with RFC 2617.

Better support for caching and proxying—Rules were added to make caching of Web pages and the operation of proxy servers more uniform.

HTTP 1.1 is the current version of HTTP. With so many millions of Web sites in operation today, any fundamental changes to HTTP would be unthinkable. Instead, changes to HTTP are to be made through extensions to HTTP 1.1. Unfortunately, not everyone agrees about the best way to do this. An HTTP extension "framework" was written as RFC 2774 in 2000 but has never moved beyond the experimental stage.

HTTP Model

The simplest HTTP interaction is for a browser client to send a request directly to the Web site server (running `httpd`) and get a response over a TCP connection between client and server. With HTTP 1.1, the model was extended to allow for *intermediaries* in the path between client and server. These devices can be proxies, gateways, tunnel endpoints, and so on. Proxy servers are especially popular for the Web, and a company frequently uses them to improve response time for job-related queries and to provide security for the corporate LAN.

Like FTP, HTTP invites data from "untrustworthy" sources right in the front door, and the proxy tries to screen harmful pages out. The proxy also protects IP addresses and

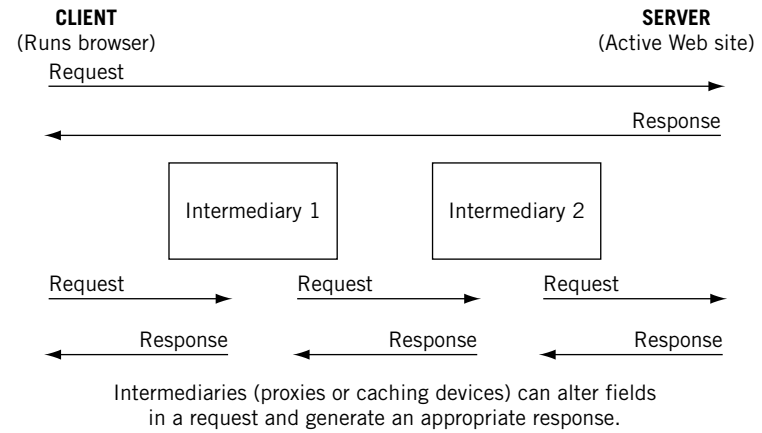


FIGURE 22.7

The HTTP models of interaction, showing how intermediaries can act on a request or response.

other types of information from leaving the site. (Some companies feared that workers would fritter away company time and so tried to limit Web access with proxies as well.) With an intermediary in place, the direct request/response becomes a four-step process.

1. *Browser request:* HTTP client sends the request to the intermediary.
2. *Intermediary request:* The intermediary makes changes to the request and forwards the request to the actual Web server.
3. *Web server response:* The Web site interprets the request and sends the reply back to the intermediary.
4. *Intermediary response:* The intermediary device processes the reply, makes changes, and forwards it to the client browser.

Generally, intermediaries become security devices that can perform a variety of functions, which we will explore later in this book. It is not unusual to find more than one intermediary on the path from HTTP client to server. In these scenarios, the request (and response) is *created* once but sent three times, usually with slightly different information. The difference between direct interactions and those with intermediaries is shown in Figure 22.7.

HTTP Messages

All HTTP messages are either requests or responses. Clients almost always issue requests, and servers almost always issue responses. Intermediaries can do both. The HTTP *generic message format* is similar to a text-based email message and is defined as a series of headers followed by an optional message body and trailer (which consists of more "headers"). The whole is introduced by a "start line."

```
<start-line>
<message-headers>
<empty-line>
[<message-body>]
[<message-trailers>]
```

The start line text identifies the nature of the message. HTTP headers can be presented in any order at all, and they follow a `<header-name>:<header-value>` convention. The message body frequently carries a file (called an *entity* in HTTP) found more often in responses than in requests. Special headers describe the encoding and other characteristics of the entity.

TRAILERS AND DYNAMIC WEB PAGES

Web pages were originally statically defined in HTML and passed out to whoever was allowed to see them. Web pages today are sometimes still created this way, but the most sophisticated Web pages create their content dynamically, on the fly, after a user has requested it. And for reasons of efficiency, the beginning can be streamed toward the browser before the end of the result has been determined. Pages that include current date and time stamps are good examples of dynamic Web page content, but of course many are much more complex.

Dynamic Web pages, however, pose a problem for persistent TCP connections. The browser has to know when the entire Web page response has been received. With a static Web page, the size is announced in a header at the start of the item. But dynamic page headers cannot list the size ahead of time, because the server does not know.

HTTP today uses *chunked encoding* to solve this problem. As soon as it is known, each piece of the response gets it own size (the chunk) and is sent to the browser. The last chunk has size 0, and can include optional "trailer" information consisting of a series of HTTP headers.

HTTP Requests and Responses

HTTP requests are a specific instance of the generic message format. They are introduced by a "request line."

```
<request-line>
<general-headers>
<request-headers>
<entity-headers>
<empty-line>
[<message-body>]
[<message-trailers>]
```

A typical initial request from a browser to the Web site is shown in Figure 22.8.

Request line	GET.index.html HTTP/1.1
General headers	Date: Mon, 04 July 2007 19:12:45 GMT Connection: close
Request headers	Host: www.example.com From: walterg@example.com Accept: text/html, text/plain User-Agent: MSIE6.0 (Windows XP)
Entity headers	
Message body	

FIGURE 22.8

The HTTP request message, showing some details of the general and request headers.

Status line	HTTP/1.1 200 OK
General headers	Date: Mon, 04 July 2007 19:12:48 GMT Connection: close
Response headers	Server: Apache/1/3/27 Accept-Range: bytes
Entity headers	Content-Type: text/html Content-Length: 170 Last-Modified: Fri, 01 July 2007 22:15:32 GMT
Message body	<html> <head> <title>Welcome to the Illustrated Network Site!</title> </head> <body> This site under construction. Check back later... </body> </html>

FIGURE 22.9

The HTTP response message, showing the headers usually included.

If the request is sent to an intermediary, such as a proxy server, the host name would appear in the request line as the resource's full URL: `GET http://www.example.com`. The use of the general, request, and entity headers are fairly self-explanatory. Request headers, however, can be conditional and are only filled if certain criteria are met. Each HTTP request to a server generates a response, and sometimes two (a preliminary response and then the full response). The format is only slightly different from the request.

```
<status-line>
<general-headers>
<response-headers>
<entity-headers>
<empty-line>
[<message-body>]
[<message-trailers>]
```

The status line has two purposes: It tells the client what version of HTTP is in use and summarizes the results of processing the client's request. The results are set as a status code and reason phrase associated with it. The structure of a typical HTTP response, sent in response to the request shown in Figure 22.8, is shown in Figure 22.9. The response headers provide details for the overall status summarized in the first line of the response.

HTTP Methods

HTTP commands, such as GET, are not called commands at all. HTTP is an *object-oriented* language, and instead of pointing out that *all* languages used for programming are to one extent or another object oriented we'll just mention that HTTP commands are called *methods*. (Yes, the URI method `http` has other HTTP methods beneath it.) Most HTTP messages use the first three methods almost exclusively. The HTTP methods are:

`GET`—Requests a resource from a Web site by URL. Sometimes also used to upload form data, but this is not a secure method. When the request headers contain conditionals, this situation is often called a *conditional GET*. When part of a resource is requested, this is sometimes called a *partial GET*.

`HEAD`—Formatted very much like a `GET`, the `HEAD` requests only the HTTP headers from the server (not the target itself). Clients use this to see if the resource is actually there before asking for a potentially monstrous file.

`POST`—Sends a block of data from the browser to the server, usually data from a form the user has filled out or some other application data. The URL sent must identify the function (program) that processes the data on the server.

`PUT`—Also sends data to the server, but asks the server to store the body of the data as a resource (file), which must be named in the URL. This can be used (with authentication) to store a file on the server, but FTP is most often used to accomplish this and thus PUT is not often used (or allowed).

`OPTIONS`—Requests information about communication options available on the Web server, with an asterisk (*) asking for details about the server itself. Not surprisingly, this method can be a security risk.

`DELETE`—Asks the server to delete the resource, which must be named in the URL. Not often used, for the same reasons as PUT.

`TRACE`—Used to debug Web applications, especially when proxy servers and gateways are in use. The client asks for a copy of the request it sent.

`CONNECT`—Reserved for future use with SSL tunneling.

The initial HTTP RFC 2068 also defined `PATCH`, `LINK`, and `UNLINK`, but these have been removed. However, some sources continue to list them. Most of the HTTP methods are

Table 22.1 HTTP Status Codes and Their Meanings

Code	Meaning
1xx	Informational, such as "request received" or "continuing process"
2xx	Successful reception, processing, acceptance, or completion
3xx	Redirection, indicating further action is needed to complete the request
4xx	Client error, such as the familiar 404, not found often, indicating syntax error
5xx	Server error when the Web site fails to fulfill a valid request

"safe" methods that can be repeated by impatient users without harm. The exception is the `POST` method, which should only be done once or side effects will result in inconsistent or just plain wrong information on the server.

HTTP Status Codes

The status codes used to provide status information to the browser are very similar to those used in FTP and email. Only the major (first) digit codes are listed in Table 22.1.

Each status code has an associated reason phrase. The reason phrases in the HTTP specification are "samples" that everyone copies and uses. They are intended as aids to memory and not as a full explanation of what is wrong when an error occurs. But a lot of browsers just display the 404 status code reason phrase, `Not Found`, and deem it adequate.

It's not necessary to list all of the HTTP status codes, but one does require additional comment. The 100 status code (reason phrase Continue) is often seen when a client is going to use the `POST` (or `PUT`) method to store a large amount of data on the server. The client might want to check to see whether the server can accept the data, rather than immediately sending it all. So, the request will have a special `Expect: 100-continue` header in it asking the server to reply with a `100 Continue` preliminary reply if all is well. After this response is received, the client can send the data.

That's the theory, anyway. In practice, it's a little different. Clients usually go ahead and send the data even if they *don't* get the `100 Continue` response from the server (hey, the browser has to do *something* with all of that data). And servers, perhaps thinking about all those users out there holding their breaths just waiting for `100 Continue` responses before they turn blue, often send out `100 Continue` preliminary responses for almost every request they get from a browser. But it was a fine idea.

HTTP Headers

It is not possible or necessary to list every HTTP header. Instead, we can just a take a look at the types of things HTTP headers do. First, some of the headers are *end-to-end* and others are *hop-by-hop*. As might be expected, the end-to-end headers are not changed as they make their way between client and server no matter how many

Table 22.2 HTTP General Headers and Their Uses

Header	Use
Cache-control	These contain a directive that establishes limits on how the request or response in cached. Only one directive can accompany a cache-control header, but multiple cache-control headers can be used.
Connection	These contain instructions that apply only to a particular connection. The headers are hop-by-hop and cannot be retained by proxies and used for other connections. The most common use is with the "close" parameters (Connection: close) to override a persistent connection and terminate the TCP session after the server response.
Date	Date and time the message originated, in RFC 822 email format.
Pragma	Implementation-specific directives similar to Unix programming. Often used for cache control in older versions of HTTP.
Trailer	When the response is chunked, this header is used before the data to indicate the presence of the trailer fields.
Transfer-encoding	Message body encoding, most often used with chunked transfers. This applies to the entire message, not a particular entity.
Upgrade	Clients can list connection protocols they support. If the server supports another in common, it can "upgrade" the connection and inform the client in the response.
Via	Used by intermediaries to allow client and server to trace the exact path.
Warning	Carries additional information about the message, usually from an intermediary device regarding cached information.

intermediary devices are between client and server. Hop-by-hop headers, on the other hand, have information relevant to each intermediary system.

General Headers

General headers are not supposed to be specific to any particular message or component. These convey information about the message itself, not about content. They also control how the message is handled and processed. However, in practice general headers are found in one type of message and not another. Some can have slightly different meanings in a request or response. The general headers are outlined in Table 22.2.

Request Headers

The request headers in an HTTP request message allow clients to supply information about themselves to the server, provide details about the request, and give the client more control over how the server handles the request and how (or if) the response is

Table 22.3 HTTP Request Headers and Their Uses

Header	Use
Accept	What media types the client will accept, including preference (q).
Accept-Charset	Similar to accept, but for character sets.
Accept-Encoding	Similar to accept, but for content encoding (especially compression).
Accept-Language	Similar to accept, but for language tags.
Authorization	Used to present authentication information ("credentials") to the server.
Expect	Tells the server what action the client expects next, usually "Continue."
From	Human user's email address. Optional, and for information only.
Host	Only mandatory header, used to specify DNS name/port of Web site.
If-Match	Usually in GET, server responds with entity only if it matches the value of the entity tags.
If-Modified-Since	Similar to If-Match, but only if the resource has changed in the time interval specified.
If-None-Match	Similar to If-Match, but the exact opposite.
If-Range	Used with Range header to check whether entity has changed and request that part of the entity.
If-Unmodified-Since	Opposite of If-Modified-Since.
Max-Forwards	Limits the number of intermediaries. Used with TRACE and OPTIONS. Value is decremented and when 0 must get a response.
Proxy-Authorization	Similar to Authorization, but used to present authentication information ("credentials") to a proxy server.
Range	Asks for part of an entity.
Referer	Never corrected to "referrer," this is used to supply the URL for the "back" button function to the server (also has privacy implications).
TE	Means "transfer encodings," and is often used with chunking.
User-Agent	Provides server with information about the client (name/version).

returned. This is the largest category of headers, and only the briefest description can be given of each. They are listed in Table 22.3.

Response Headers

HTTP response headers are the opposite of request headers and appear only in messages sent from server to browser. They expand on the information provided in the summary status line, as outlined in Table 22.4. Many response headers are sent only in answer to a specific type of request, or to certain headers within particular requests.

Table 22.4 HTTP Response Headers and Their Uses

Header	Use
Accept-Ranges	Tells client if server accepts partial content requests using Range request header. Typical values are in bytes, or "none" for no support.
Age	Tells the client the approximate age of the resource.
ETag	Gives the entity tag for the entity in the response.
Location	Gives client a new URL to use instead of one requested.
Proxy-Authenticate	Tells client how the proxy requires authentication, both method and parameters needed.
Retry-After	Tells client to try the request again later, seconds or by date/time.
Server	Server version of User-Agent request header, used for server details.
Vary	Used by caching devices to make decisions.
WWW-Authenticate	Tells client how the Web site requires authentication, both method and parameters needed.

Table 22.5 HTTP Entity Headers and Their Uses

Header	Use
Allow	Lists methods that apply to this resource.
Content-Encoding	Describes optional encoding method, usually the compression algorithm used so that the client can decompress the entity.
Content-Language	Specifies the human language used by the entity. It is optional and can specify multiple languages.
Content-Length	Size of the entity in bytes (octets). Not used in chunked transfers.
Content-Location	Resource location as URL. Optional, but used if entity is in multiple places.
Content-MD5	Used for message integrity checking with Message Digest 5.
Content-Range	Used for entities that are part of the complete resource.
Content-Type	Similar to MIME type and subtype, but not exactly the same.
Expires	Data and time after which entity is considered stale.
Last-Modified	Date and time server "believes" entity last changed.

Entity Headers

Finally, entity headers describe the resource carried in the body of the HTTP message. They usually appear in responses, but can appear in `PUT` and `POST` requests. Many of the entity headers have the same names as the MIME types they are based on, but with important differences. The entity headers are outlined in Table 22.5.

Use of the Last-Modified header is complicated by the fact that the server might not know when an entity was last modified, especially if the resource is "virtual." For dynamic content, this header should be the same as the time the message was generated.

Cookies

A Web server gets a request, processes a request, and returns a response in a completely stateless manner. Every request, even from the same client a moment later, looks brand new to the server.

Stateless servers are the easiest to operate. If they fail, just start them up again. No one cares where they left off. You can even transfer processing to another host and everything runs just fine, as long as the resources are there. Stateless servers are best for simple resource-retrieval systems.

That's how the Web started out, but unfortunately this is not how the Web is used today. Web sites have shopping carts that remember content and billing systems that remember credit card information. They also remember log-in information that would otherwise have to be entered every time an HTTP request was made.

How should the state information necessary for the Web today be stored? For better or worse, the answer today is in *cookies*. The term seems to have originated in older programs that required users to supply a "magic cookie" to make the program do something out of the ordinary ("Easter eggs" seem to be the GUI equivalent). According to others, an old computer virus put the image onscreen of Cookie Monster (of *Sesame Street* fame) announcing, "Want cookie!" The user had to type the word *cookie* to continue. The *cookie* term is also used in BOOTP/DHCP.

Cookies were initially developed by Netscape and were formalized as a Web *state management* system in RFC 2965, which replaced RFC 2109. Cookies are not actually part of HTTP, and remain an option, but few Web browsers can afford to reject all cookies out of hand (so to speak).

The idea behind cookies as a method of server state management is simple. If the server can't hold state information about the user and the session, let the client do it. When the server has a function that needs a state to be maintained over time, the server sends a small amount of data to the client (a cookie).

Cookies are presented when the server asks for them, and are updated as the session progresses. Cookies are just text strings and have no standard formats, in that only a particular server has to understand and parse them. In Windows XP, cookies are stored in the `cookies.txt` file under the user's *Documents and Settings* directory. Cookies just accumulate there until users clear them out (few do). If deleted, the file is built again from scratch. Looking at someone's cookies is a quick and dirty way to see where the browser (not necessarily the user) has gone recently.

Cookies, as indispensable as they are on the Web today, tend to have a somewhat unsavory reputation. They aren't perfect: If a cookie is established to allow access to a book-shop Web site at home, the cookie is not present on the user's office computer and the Web site has no idea who the user is because there is no cookie to give to the

server. A lot of users assume they've done something wrong, but that's just the way cookies work.

Most browsers can be set to screen or reject cookies, mainly because cookies are a barely tolerated security risk to many people (many think the browser default should be to *reject* all cookies instead of accepting them). In particular, there are three big issues with cookies.

Sending of sensitive information— Banks routinely store user ID and password in a cookie. Even if it is encrypted when sent, the information is typically sitting on your computer in plain text (waiting for anyone to look at it).

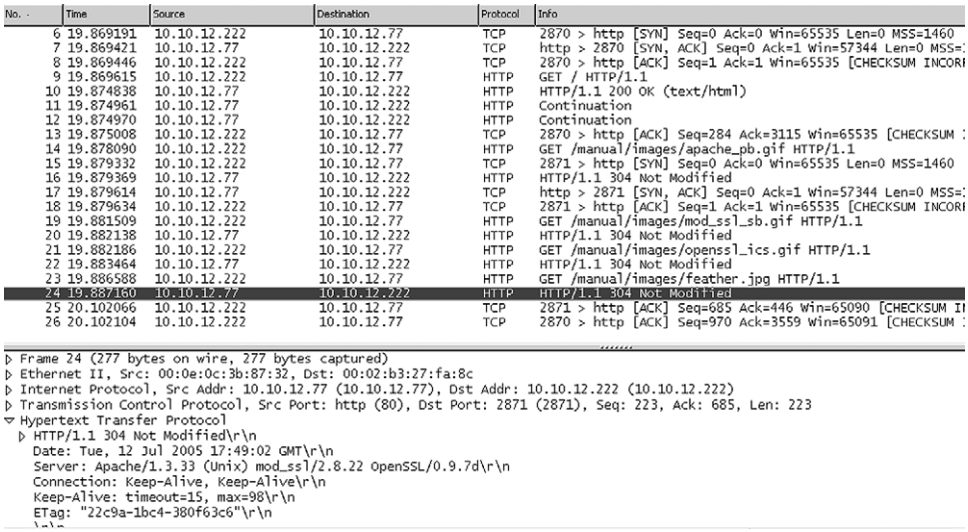
User tracking abuse—Servers can set cookies for any reason, including tracking the sites a user visits rather than storing useful parameters. This is often seen as a violation of the right to privacy, and some Web browsers are silent when a cookie is set.

Third-party cookies—If a Web page contains a link (perhaps to a small image) to another Web site, the *second site* can set a cookie (called a *third-party cookie*) on your machine even though you've never visited (or intend to visit) the site. So, that must be how all those porn-site cookies got there.

Some people regard cookies as much ado about nothing, whereas others busily turn off all cookie support whenever they go on-line. But most people should at least consider disabling third-party cookies, which really have no legitimate use when it comes to HTTP state management.

QUESTIONS FOR READERS

Figure 22.10 shows some of the concepts discussed in this chapter and can be used to answer the following questions.



No.	Time	Source	Destination	Protocol	Info
6	19.869191	10.10.12.222	10.10.12.77	TCP	2870 > http [SYN] Seq=0 Ack=0 Win=65535 Len=0 MSS=1460
7	19.869421	10.10.12.77	10.10.12.222	TCP	http > 2870 [SYN, ACK] Seq=0 Ack=1 Win=57344 Len=0 MSS=
8	19.869446	10.10.12.222	10.10.12.77	TCP	2870 > http [ACK] Seq=1 Ack=1 Win=65535 [CHECKSUM INCORI
9	19.869615	10.10.12.222	10.10.12.77	HTTP	GET / HTTP/1.1
10	19.874838	10.10.12.77	10.10.12.222	HTTP	HTTP/1.1 200 OK (text/html)
11	19.874961	10.10.12.77	10.10.12.222	HTTP	Continuation
12	19.874970	10.10.12.77	10.10.12.222	HTTP	Continuation
13	19.875008	10.10.12.222	10.10.12.77	TCP	2870 > http [ACK] Seq=284 Ack=3115 Win=65535 [CHECKSUM
14	19.878090	10.10.12.222	10.10.12.77	HTTP	GET /manual/images/apache_pb.gif HTTP/1.1
15	19.879332	10.10.12.222	10.10.12.77	TCP	2871 > http [SYN] Seq=0 Ack=0 Win=65535 Len=0 MSS=1460
16	19.879369	10.10.12.77	10.10.12.222	HTTP	HTTP/1.1 304 Not Modified
17	19.879614	10.10.12.77	10.10.12.222	TCP	http > 2871 [SYN, ACK] Seq=0 Ack=1 Win=57344 Len=0 MSS=
18	19.879634	10.10.12.222	10.10.12.77	TCP	2871 > http [ACK] Seq=1 Ack=1 Win=65535 [CHECKSUM INCORI
19	19.881509	10.10.12.222	10.10.12.77	HTTP	GET /manual/images/mod_ssl_sb.gif HTTP/1.1
20	19.882138	10.10.12.77	10.10.12.222	HTTP	HTTP/1.1 304 Not Modified
21	19.882186	10.10.12.222	10.10.12.77	HTTP	GET /manual/images/openssl_ics.gif HTTP/1.1
22	19.883464	10.10.12.77	10.10.12.222	HTTP	HTTP/1.1 304 Not Modified
23	19.886588	10.10.12.222	10.10.12.77	HTTP	GET /manual/images/feather.jpg HTTP/1.1
24	19.887160	10.10.12.77	10.10.12.222	HTTP	HTTP/1.1 304 Not Modified
25	20.102066	10.10.12.222	10.10.12.77	TCP	2871 > http [ACK] Seq=685 Ack=446 Win=65090 [CHECKSUM I
26	20.102104	10.10.12.222	10.10.12.77	TCP	2870 > http [ACK] Seq=970 Ack=3559 Win=65091 [CHECKSUM

```
▷ Frame 24 (277 bytes on wire, 277 bytes captured)
▷ Ethernet II, Src: 00:0e:0c:3b:87:32, Dst: 00:02:b3:27:fa:8c
▷ Internet Protocol, Src Addr: 10.10.12.77 (10.10.12.77), Dst Addr: 10.10.12.222 (10.10.12.222)
▷ Transmission Control Protocol, Src Port: http (80), Dst Port: 2871 (2871), Seq: 223, Ack: 685, Len: 223
▽ Hypertext Transfer Protocol
  ▷ HTTP/1.1 304 Not Modified\r\n
    Date: Tue, 12 Jul 2005 17:49:02 GMT\r\n
    Server: Apache/1.3.33 (Unix) mod_ssl/2.8.22 OpenSSL/0.9.7d\r\n
    Connection: Keep-Alive, Keep-Alive\r\n
    Keep-Alive: timeout=15, max=98\r\n
    ETag: "22c9a-1bc4-380f63c6"\r\n
    \r\n
```

FIGURE 22.10

The Apache server capture.

1. Which version of Apache is the server using?
2. Which ports are the client and server using?
3. Completely parse the following URL: *http://www.examplebooks.com:8888/cgi-bin/ebook.php?HTTPforChimps#page345*.
4. Completely parse the following URL:

   ftp://ftp.freestuff.com/Is%20This%20Really%20Free%3F.
5. What is a cookie used for? Examine your `cookies.txt` file.