CHAPTER 1

Web Essentials
Clients, Servers, and Communication

The essential elements of the World Wide Web are the web browsers used to surf the Web, the server systems used to supply information to these browsers, and the computer networks supporting browser-server communication. This chapter will provide an overview of all of these elements. We'll begin by considering communication, with a focus on the Internet and some of its key communication protocols, especially the Hypertext Transport Protocol used for the bulk of web communication. The chapter also reviews features common to modern web browsers and introduces web servers, the software applications that provide web pages to browsers.

1.1 The Internet

"So, you're into computers. Maybe you can answer a question I've had for a while: I hear people talk about the Internet, and I'm not sure exactly what it is, or where it came from. Can you tell me?"

You may have already been asked a question like this. If not, if you work with computers long enough, you'll probably hear it at least once in your career, and more likely several times. At this point in your career, you may even be curious about the Internet yourself: you use it a lot, but what exactly is it?

The Internet traces its roots to a project of the U.S. Department of Defense's then-named Advanced Research Projects Agency, or ARPA. The ARPANET project was intended to support DoD research on computer networking. As this project began in the late 1960s, there had been only a few small experimental networks providing communication between geographically dispersed computers from different manufacturers running different operating systems. The purpose of ARPANET was to create a larger such network, both in order to electronically connect DoD-sponsored researchers and in order to experiment with and develop tools for heterogeneous computer networking.

The ARPANET computer network was launched in 1969 and by year's end consisted of four computers at four sites running four different operating systems. ARPANET grew steadily, but because it was restricted to DoD-funded organizations and was a research project, it was never large. By 1983, when many ARPANET nodes were split off to form a separate network called MILNET, there were only 113 nodes in the entire network, and these were primarily at universities and other organizations involved in DoD-sponsored research.

Despite the relatively small number of machines actually on the ARPANET, the benefits of networking were becoming known to a wide audience. For example, e-mail was available on ARPANET beginning in 1972, and it soon became an extremely popular

application for those who had ARPANET access. It wasn't long before other networks were being built, both internationally and regionally within the United States. The regional U.S. networks were often cooperative efforts between universities. As one example, SURAnet (Southeastern University Research Association Network) was organized by the University of Maryland beginning in 1982 and eventually included essentially all of the major universities and research institutions in the southeastern United States. Another of these networks, CSNET (Computer Science Network), was partially funded by the U.S. National Science Foundation (NSF) to aid scientists at universities without ARPANET access, laying the groundwork for future network developments that we'll say more about in a moment.

While these other networks were springing up, the ARPANET project continued to fund research on networking. Several of the most widely used Internet protocols—including the File Transfer Protocol (FTP) and Simple Mail Transfer Protocol (SMTP), which underlie many of the Internet's file transfer and e-mail operations, respectively—were initially developed under ARPANET. But perhaps most crucial to the emergence of the Internet as we know it was the development of the TCP/IP (Transmission Control Protocol/Internet Protocol) communication protocol. TCP/IP was designed to be used for host-to-host communication both within *local area networks* (that is, networks of computers that are typically in close proximity to one another, such as within a building) and between networks. ARPANET switched from using an earlier protocol to TCP/IP during 1982. At around the same time, an ARPA Internet was created, allowing computers on some outside networks such as CSNET to communicate via TCP/IP with computers on the ARPANET.

A "connection" from CSNET to the ARPA Internet often meant that a modem connection was made from one computer to another for the purpose of sending along an e-mail message. This form of communication was asynchronous. That is, the e-mail might be delayed some time before it was actually delivered, which precluded interactive communication of any type. Furthermore, each institution connecting to CSNET was largely on its own in determining how it was going to connect to the network. At first, many institutions connected through the so-called PhoneNet (modem) approach for passing e-mail messages. This generally involved long distance calls, and the expense of these calls could be a problem. Other options, such as leasing telephone lines for dedicated use, could be even more expensive. It was obvious to everyone that the CSNET institutions were still not enjoying all the potential benefits of the ARPA Internet.

Beginning in 1985, the NSF began work on a new network based on TCP/IP, called NSFNET. One of the primary goals of this network was to connect the NSF's new regional supercomputing centers. But it was also decided that regional networks should be able to connect to NSFNET, so that the NSFNET would provide a *backbone* through which other networks could interconnect synchronously. Figure 1.1 shows the geographic distribution of the six supercomputer centers connected by the early NSFNET backbone. Regional networks connecting to the backbone included SURAnet as well as NYSERNet (with primary connections through the Ithaca center), JvNCnet (with primary connection through the Princeton center) and SDSCnet (with primary connection through the San Diego center). In addition, many universities and other organizations connected to the NSFNET backbone either directly or through agreements with other institutions that had NSFNET access, either directly or indirectly.

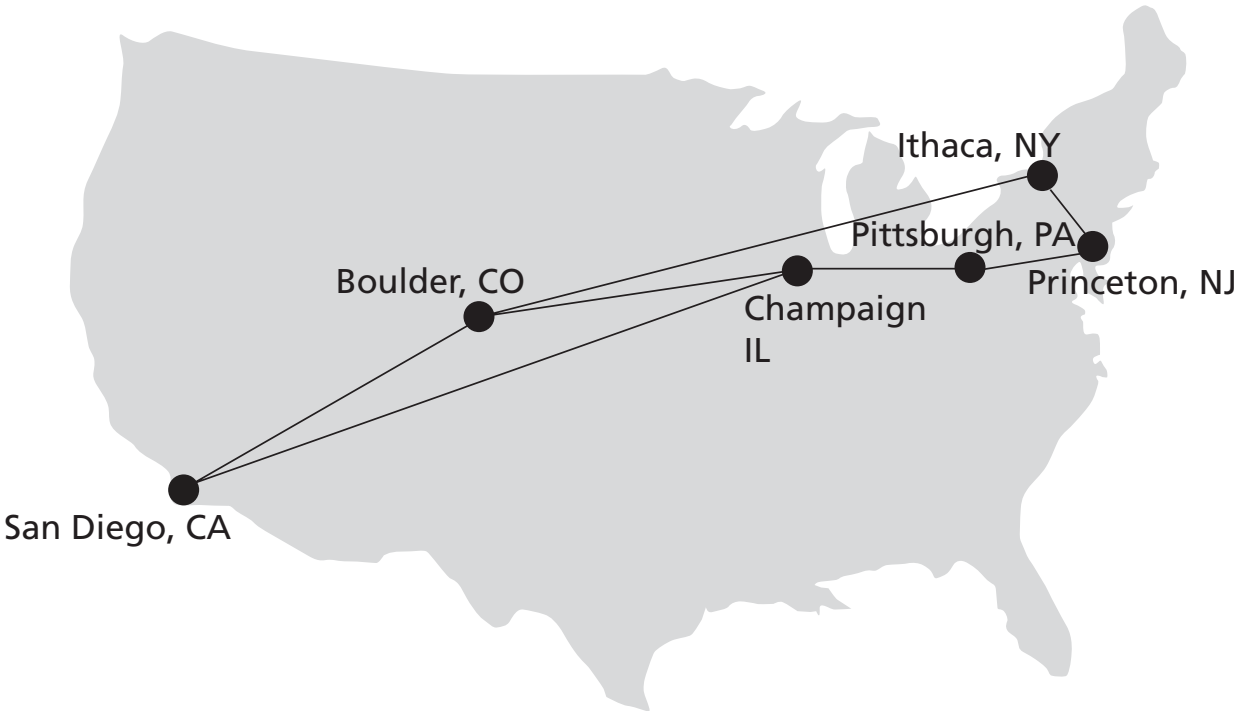

FIGURE 1.1 Geographic distribution of and connections between nodes on the early NSFNET backbone.

The original backbone operated at only 56 kbit/s, the maximum speed of a home dial-up line today. But at the time the primary network traffic was still textual, so this was a reasonable starting point. Once operational, the number of machines connected to NSFNET grew quickly, in part because the NSF directly or indirectly provided significant support—both technically and with monetary grants—to educational and research organizations that wished to connect. The backbone rate was upgraded to 1.5 Mbit/s (T1) in 1988 and then to 45 Mbit/s (T3) in 1991. Furthermore, the backbone was expanded to directly include several research networks in addition to the supercomputer centers, making it that much easier for sites near these research networks to connect to the NSFNET. In 1988, networks in Canada and France were connected to NSFNET; in each succeeding year for the remaining seven years of NSFNET's existence, networks from 10 or so new countries were added per year.

NSFNET quickly supplanted ARPANET, which was officially decommissioned in 1990. At this point, NSFNET was at the center of the *Internet*, that is, the collection of computer networks connected via the public backbone and communicating across networks using TCP/IP. This same year, commercial Internet dial-up access was first offered. But the NSFNET terms of usage stipulated that purely commercial traffic was not to be carried over the backbone: the purpose of the Internet was still, in the eyes of the NSF, research and education.

Increasingly, though, it became clear that there could be significant benefits to allowing commercial traffic on the Internet as well. One of the arguments for allowing commercial traffic was economic: commercial traffic would increase network usage, leading to reduced unit costs through economies of scale. This in turn would provide a less expensive network for research and educational purposes. Whatever the motivation, the restriction on commercial traffic was rescinded in 1991, spurring further growth of the Internet and laying the

groundwork for the metamorphosis of the Internet from a tool used primarily by scientists at research institutions to the conduit for information, entertainment, and commerce that we know today. This also led fairly quickly to the NSF being able to leave its role as the operator of the Internet backbone in the United States. Those responsibilities were assumed by private telecommunication firms in 1995. These firms are paid by other firms, such as some of the larger Internet service providers (ISPs), who connect directly with the Internet backbone. These ISPs, in turn, are paid by their users, which may include smaller ISPs as well as end users.

In summary, the Internet is the collection of computers that can communicate with one another using TCP/IP over an open, global communications network. Before describing how the World Wide Web is related to the Internet, we'll take a closer look at several of the key Internet protocols. This will be helpful in understanding the place of the Web within the wider Internet.

1.2 Basic Internet Protocols

Before covering specific protocols, it may be helpful to explain exactly what the term "protocol" means in the context of networked communication. A computer *communication protocol* is a detailed specification of how communication between two computers will be carried out in order to serve some purpose. For example, as we will learn, the Internet Protocol specifies both the high-level behavior of software implementing the protocol and the low-level details such as the specific fields of information that will be contained in a communication message, the order in which these fields will appear, the number of bits in each field, and how these bits should be interpreted. We are primarily interested in a high-level view of general-purpose Internet protocols in this section; we'll look at a key Web protocol, HTTP, in more detail in the next section.

1.2.1 TCP/IP

Since TCP/IP is fundamental to the definition of the Internet, it's natural to begin our study of Internet protocols with these protocols. Yes, I said protocols (plural), because although so far I have treated TCP/IP as if it were a single protocol, TCP and IP are actually two different protocols. The reason that they are often treated as one is that the bulk of the services we associate with the Internet—e-mail, Web browsing, file downloads, accessing remote databases—are built on top of both the TCP and IP protocols. But in reality, only one of these protocols—IP, the Internet Protocol—is fundamental to the definition of the Internet. So we'll begin our study of Internet protocols with IP.

A key element of IP is the *IP address*, which is simply a 32-bit number. At any given moment, each device on the Internet has one or more IP addresses associated with it (although the device associated with a given address may change over time). IP addresses are normally written as a sequence of four decimal numbers separated by periods (called "dots"), as in 192.0.34.166. Each decimal number represents one byte of the IP address.

The function of IP software is to transfer data from one computer (the *source*) to another computer (the *destination*). When an application on the source computer wants to send information to a destination, the application calls IP software on the source machine

and provides it with data to be transferred along with an IP address for each of the source and destination computers. The IP software running on the source creates a *packet*, which is a sequence of bits representing the data to be transferred along with the source and destination IP addresses and some other header information, such as the length of the data. If the destination computer is on the same local network as the source, then the IP software will send the packet to the destination directly via this network. If the destination is on another network, the IP software will send the packet to a *gateway*, which is a device that is connected to the source computer's network as well as to at least one other network. The gateway will select a computer on one of the other networks to which it is attached and send the packet on to that computer. This process will continue, with the packet going through perhaps a dozen or more *hops*, until the packet reaches the destination computer. IP software on that computer will receive the packet and pass its data up to an application that is waiting for the data.

For example, returning to the Internet as it existed in the mid-1980s, suppose that a computer in the SURAnet network (say, at the University of Delaware) was a packet source and that a computer in a network directly connected to the NSFNET backbone at San Diego (say, at the San Diego Supercomputer Center) was the destination. The IP packet would first go through the Delaware local computer network to a gateway device connecting the Delaware network to SURAnet. The gateway device would then send the packet on to another SURAnet gateway device (how this gateway is chosen is discussed later in this subsection) until it reached a gateway on the NSFNET backbone at Ithaca (the primary SURAnet connection to the NSFNET backbone). As there was no direct connection from Ithaca to San Diego in the NSFNET at the time (Figure 1.1), the packet would need to go through at least one other gateway on the NSFNET backbone before reaching the San Diego node. From there, it would be passed to the San Diego Supercomputer Center local network, and from there on to the destination machine.

The sequence of computers that a packet travels through from source to destination is known as its *route*. How does each computer choose the next computer in the route for a packet? A separate protocol (the current standard is BGP-4, the Border Gateway Protocol) is used to pass network connectivity information between gateways so that each can choose a good next hop for each packet it receives.

IP software also adds some error detection information (a *checksum*) to each packet it creates, so that if a packet is corrupted during transmission, this can usually be detected by the recipient. The IP standard calls for IP software to simply discard any corrupted packets. Thus, IP-based communication is unreliable: packets can be lost. Obviously, IP alone is not a particularly good form of communication for many Internet applications.

TCP, the Transmission Control Protocol, is a higher-level protocol that extends IP to provide additional functionality, including reliable communication based on the concept of a *connection*. A connection is established between TCP software running on two machines by one of the machines (let's call it A) sending a connection-request message via IP to the other (B). That is, the IP message contains a message conforming to the TCP protocol and representing a TCP connection request. If the connection is accepted by B, then B returns a message to A requesting a connection in the other direction. If A responds affirmatively, then the connection is established. Notice that this means that A and B can both send messages to one another at the same time; this is known as *full duplex* communication. When A and

B are both done sending messages to one another (or at least done for the time being), a similar set of three messages is used to close the connection.

Once a connection has been established, TCP provides reliable data transmission by demanding an *acknowledgment* for each packet it sends via IP. Essentially, the software sets a timer after sending each packet. The TCP software on the receiving side sends a packet containing an acknowledgment for every TCP-based packet it receives that passes the checksum test. If the TCP software sending a packet does not receive an acknowledgment packet before its timer expires, then it resends the packet and restarts the timer.

Another important feature that TCP adds to IP is the concept of a *port*. The port concept allows TCP to be used to communicate with many different applications on a machine. For example, a machine connected to the Internet may run a mail server for users on its local network, a file download server, and also a server that allows users to log in to the machine and execute commands from remote locations. As illustrated in Figure 1.2 (which ignores connections and acknowledgments for simplicity), such a server application will make a call to the TCP software on its system to request that any incoming TCP connection requests that specify a certain port number as part of the TCP/IP message be sent to the application. For example, a mail server conforming with SMTP will typically ask TCP to listen for requests to port 25. If at a later time an IP message is received by the machine running the mail server application and that IP message contains a TCP message with port

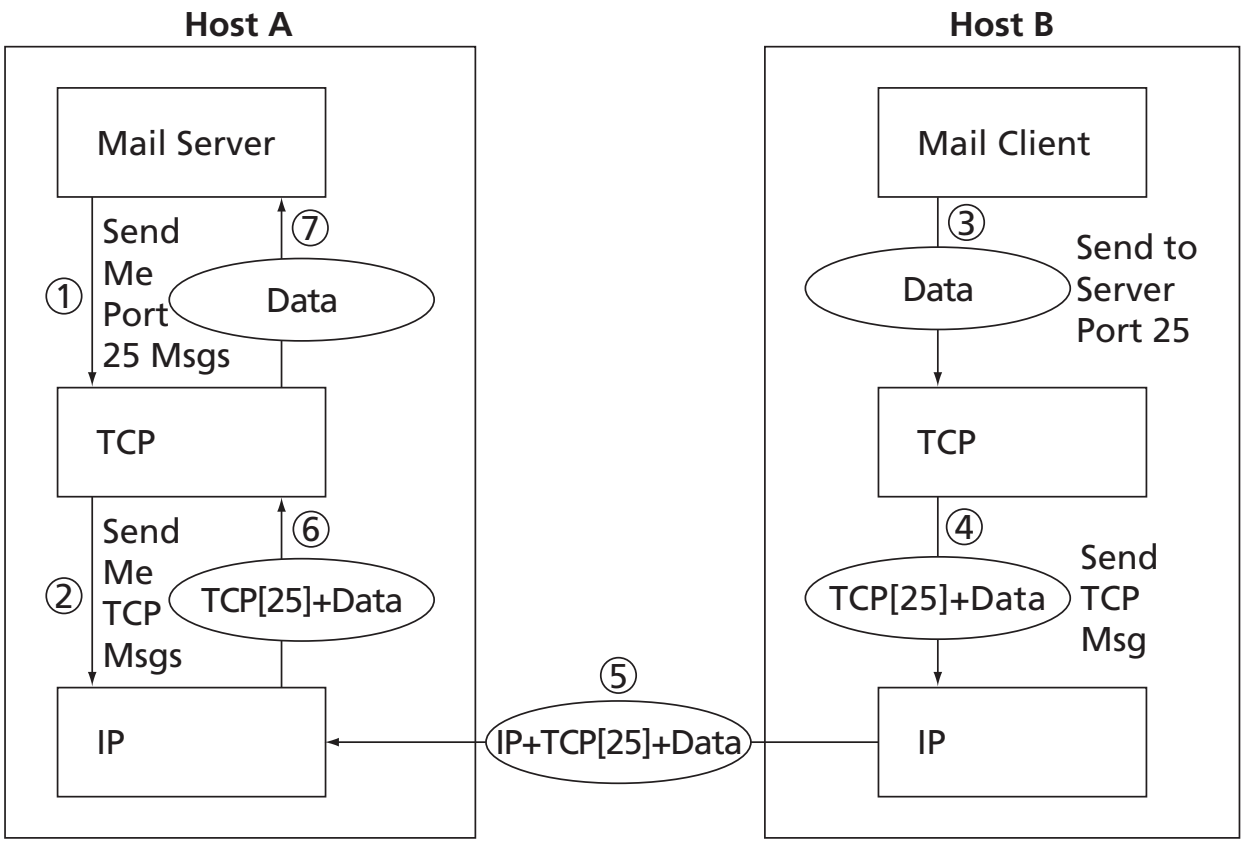

FIGURE 1.2 Simplified view of communication using TCP/IP. Boxes represent software applications on the respective host machines, ovals represent data transmitted between applications, and circled numbers denote the time order of operations. "TCP[25]" represents a TCP header containing 25 as the port number.

25 indicated in its header, then the data contained within the TCP message will be returned to the mail server application. Such an IP message could be generated by a mail client calling on TCP software on another system, as illustrated on the right side of the figure.

Though the connection between port numbers and applications is managed individually by every machine on the Internet, certain broadly useful applications (such as e-mail over SMTP) have had port numbers assigned to them by the Internet Assigned Numbers Authority (IANA) [IANA-PORTS]. These port numbers, in the range 0–1023, can usually be requested only by applications that are run by the system at boot-up or that are run by a user with administrative permissions on the system. Other possible port numbers, from 1024 to 65535, can generally be used by the first application on a system that requests the port.

TCP and IP provide many other functions, such as splitting long messages into shorter ones for transport over the Internet and transparently reassembling them on the receiving side. But this brief overview of TCP/IP covers the essential concepts for our purposes.

1.2.2 UDP, DNS, and Domain Names

UDP (User Datagram Protocol) is an alternative protocol to TCP that also builds on IP. The main feature that UDP adds to IP is the port concept that we have just seen in TCP. However, it does not provide the two-way connection or guaranteed delivery of TCP. Its advantage over TCP is speed for simple tasks. For example, if all you want to do is send a short message to another computer, you're expecting a single short response message, and you can handle resending if you don't receive the response within a reasonable amount of time, then UDP is probably a good alternative to TCP.

One Internet application that is often run using UDP rather than TCP is the Domain Name Service (DNS). While every device on the Internet has an IP address such as 192.0.34.166, humans generally find it easier to refer to machines by names, such as `www.example.org`. DNS provides a mechanism for mapping back and forth between IP addresses and host names. Basically, there are a number of DNS servers on the Internet, each listening through UDP software to a port (port 53 if the server is following the current IANA assignment). When a computer on the Internet needs DNS services—for example, to convert a host name such as `www.example.org` to a corresponding IP address—it uses the UDP software running on its system to send a UDP message to one of these DNS servers, requesting the IP address. If all goes well, this server will then send back a UDP message containing the IP address. Recall that it took three messages just to get a TCP connection set up, so the UDP approach is much more efficient for sporadic DNS queries. (UDP is sometimes referred to as a lightweight communication protocol and TCP as a heavyweight protocol, at least in comparison with UDP. In general, the terms *lightweight* and *heavyweight* in computer science are used to describe alternative software solutions to some problem, with the lightweight solution having less functionality but also less overhead.)

Internet host names consist of a sequence of *labels* separated by dots. The final label in a host name is a *top-level domain*. There are two standard types of top-level domain: generic (such as `.com`, `.edu`, `.org`, and `.biz`) and country-code (such as `.de`, `.il`, and `.mx`). The top-level domain names are assigned by the Internet Corporation for Assigned Names and

Numbers (ICANN), a private nonprofit organization formed to take over technical Internet functions that were originally funded by the U.S. government.

Each top-level domain is divided into subdomains (second-level domains), which may in turn be further divided, and so on. The assignment of second-level domains within each top-level domain is performed (for a fee) by a *registry operator* selected by ICANN. The owner of a second-level domain can then further divide that domain into subdomains, and so on. Ultimately, the subdomains of a domain are individual computers. Such a subdomain, consisting of a local host name followed by a domain name (typically consisting of at least two labels) is sometimes called a *fully qualified domain name* for the computer. For example, `www.example.org` is a fully qualified domain name for a host with local name `www` that belongs to the `example` second-level domain of the `org` top-level domain.

Some user-level tools are available that allow you to query the Internet DNS. For example, on most systems the `nslookup` command can be typed at a command prompt (see *Appendix A* for instructions on obtaining a command prompt on some systems) in order to find the IP address given a fully qualified domain name or vice versa. Typical usage of `nslookup` is illustrated by the following (user input is italicized):

```
C:\>nslookup www.example.org
Server: slave9.dns.stargate.net
Address: 209.166.161.121

Name: www.example.org
Address: 192.0.34.166

C:\>nslookup 192.0.34.166
Server: slave9.dns.stargate.net
Address: 209.166.161.121

Name: www.example.com
Address: 192.0.34.166
```

The first two lines following the command line identify the qualified name and IP address of the DNS server that is providing the domain name information that follows. Also notice that a single IP address can be associated with multiple domain names. In this example, both `www.example.org` and `www.example.com` are associated with the IP address 192.0.34.166. A lookup that specifies an IP address, such as the second lookup in the example, is sometimes referred to as a *reverse lookup*. As shown, even if multiple qualified names are associated with an IP address, only one of the names will be returned by a reverse lookup. This is known as the *canonical name* for the host; all other names are considered *aliases*. The reverse lookup in the example indicates that `www.example.com` is the canonical name for the host with IP address 192.0.34.166.

1.2.3 Higher-Level Protocols

The following analogy may help to relate the computer networking concepts described in Sections 1.2.1 and 1.2.2 with something more familiar: the telephone network. The Internet

is like the physical telephone network: it provides the basic communications infrastructure. UDP is like calling a number and leaving a message rather than actually speaking with the intended recipient. DNS is the Internet version of directory assistance, associating names with numbers. TCP is roughly equivalent to placing a phone call and having the other party answer: you now have a connection and are able to communicate back and forth.

However, in the cases of both TCP and a phone call, different protocols can be used to communicate once a connection has been established. For example, when making a telephone call, the parties must agree on the language(s) that will be used to communicate. Beyond that, there are also conventions that are followed to decide which party will speak first, how the parties will take turns speaking, and so on. Furthermore, different conventions may be used in different contexts: I answer the phone differently at home ("Hello") than I do at work ("Mathematics and Computer Science Department, this is Jeff Jackson"), for example.

Similarly, a variety of *higher-level protocols* are used to communicate once a TCP connection has been established. SMTP and FTP, mentioned earlier, are two examples of widely used higher-level protocols that are used to communicate over TCP connections. SMTP supports transfer of e-mail between different e-mail servers, while FTP is used for transferring files between machines. Another higher-level TCP protocol, Telnet, is used to execute commands typed into one computer on a remote computer. As we will see, Telnet can also be used to communicate directly (via keyboard entries) with some TCP-based applications. As described earlier, which protocol will be used to communicate over a TCP connection is normally determined by the port number used to establish the connection.

The primary TCP-based protocol used for communication between web servers and browsers is called the Hypertext Transport Protocol (HTTP). In some sense, just as IP is a key component in the definition of the Internet, HTTP is a key component in the definition of the World Wide Web. So, before getting into details of HTTP, let's briefly consider what the Web is, and in particular how HTTP figures in its definition.

1.3 The World Wide Web

Public sharing of information has been a part of the Internet since its early days. For example, the Usenet newsgroup service began in 1979 and provided a means of "posting" information that could be read by users on other systems with the appropriate software (the Google Groups™ Usenet discussion forum at `http://www.google.com` provides one of several modern interfaces to Usenet). Large files were (and still are) often shared by running an FTP server application that allowed any user to transfer the files from their origin machine to the user's machine. The first Internet chat software in widespread use, Internet Relay Chat (IRC), provided both public and private chat facilities.

However, as the amount of information publicly available on the Internet grew, the need to locate information also grew. Various technologies for supporting information management and search on the Internet were developed. Some of the more popular information management technologies in the early 1990s were Gopher information servers, which provided a simple hierarchical view of documents; the Wide Area Information System

(WAIS) system for indexing and retrieving information; and the ARCHIE tool for searching online information archives accessible via FTP.

The World Wide Web also was developed in the early 1990s (we'll learn more about its development in the next chapter), and for a while was just one among several Internet information management technologies. To understand why the Web supplanted the other technologies, it will be helpful to know a bit about the mechanics of the Web and other Internet information management technologies. All of these technologies consist of (at least) two types of software: server and client. An Internet-connected computer that wishes to provide information to other Internet systems must run *server* software, and a system that wishes to access the information provided by servers must run *client* software (for the Web, the client software is normally a web browser). The server and client applications communicate over the Internet by following a communication protocol built on top of TCP/IP.

The protocol used by the Web, as just noted, is the Hypertext Transport Protocol, HTTP. As we will learn in the next section, this is a rather generic protocol that for the most part supports a client requesting a document from a server and the server returning the requested document. This generic nature of HTTP gives it the advantage of somewhat more flexibility than is present in the protocols used by WAIS and Gopher.

Perhaps a bigger advantage for the Web is the type of information communicated. Most web pages are written using the Hypertext Markup Language, HTML, which along with HTTP is a fundamental web technology. HTML pages can contain the familiar web links (technically called *hyperlinks*) to other documents on the Web. While certain Gopher pages could also contain links, normal Gopher documents were plain text. WAIS and ARCHIE provided no direct support for links. In addition to hyperlinks, modern versions of HTML also provide extensive page layout facilities, including support for inline graphics, which (as you might guess) has added significantly to the commercial appeal of the Web.

The World Wide Web, then, can be defined in much the same way as the Internet. While the Internet can be thought of as the collection of machines that are globally connected via IP, the World Wide Web can be informally defined as the collection of machines (web servers) on the Internet that provide information via HTTP, and particularly those that provide HTML documents.

Given this overview, we'll now spend some time looking closely at HTTP.

1.3.1 Hypertext Transport Protocol

HTTP is a form of communication protocol, in particular a detailed specification of how web clients and servers should communicate. The basic structure of HTTP communication follows what is known as a *request–response* model. Specifically, the protocol dictates that an HTTP interaction is initiated by a client sending a request message to the server; the server is then expected to generate a response message. The format of the request and response messages is dictated by HTTP. HTTP does not dictate the network protocol to be used to send these messages, but does expect that the request and response are both sent within a TCP-style connection between the client and the server. So most HTTP implementations send these messages using TCP.

Let's relate this to what happens when you browse the Web. Figure 1.3 shows a browser window in which I typed `http://www.example.org` in the Location bar (note that this is technically not a web site address and therefore might not be operational by the time you read this). When I pressed the Enter key after typing this address, the browser created a message conforming to the HTTP protocol, used DNS to obtain an IP address for `www.example.org`, created a TCP connection with the machine at the IP address obtained, sent the HTTP message over this TCP connection, and received back a message containing the information that is shown displayed in the *client area* of the browser (the portion of the browser containing the information received from the web server).

A nice feature of HTTP is that these request and response messages often consist entirely of plain text in a fairly readable form. An HTTP request message consists of a start line followed by a message header and optionally a message body. The start line always consists of printable ASCII characters, and the header normally does as well. What's more, the HTTP response (or at least most of it) is often also a stream of printable characters. So, to see an example of HTTP in action, let's connect to the same web server shown in Figure 1.3 using Telnet. This can be done on most modern systems by entering `telnet` at a command prompt. Specifically, we will Telnet to port 80, the IANA standard port for HTTP web servers, type in an HTTP request message corresponding to the Internet address entered into the browser before, and view the response (the request consists of the three lines beginning with the GET and ending with a blank line, and user input is again italicized):

```
$ telnet www.example.org 80
Trying 192.0.34.166...
Connected to www.example.com (192.0.34.166).
Escape character is '^]'.
GET / HTTP/1.1
Host: www.example.org

HTTP/1.1 200 OK
Date: Thu, 09 Oct 2003 20:30:49 GMT
```

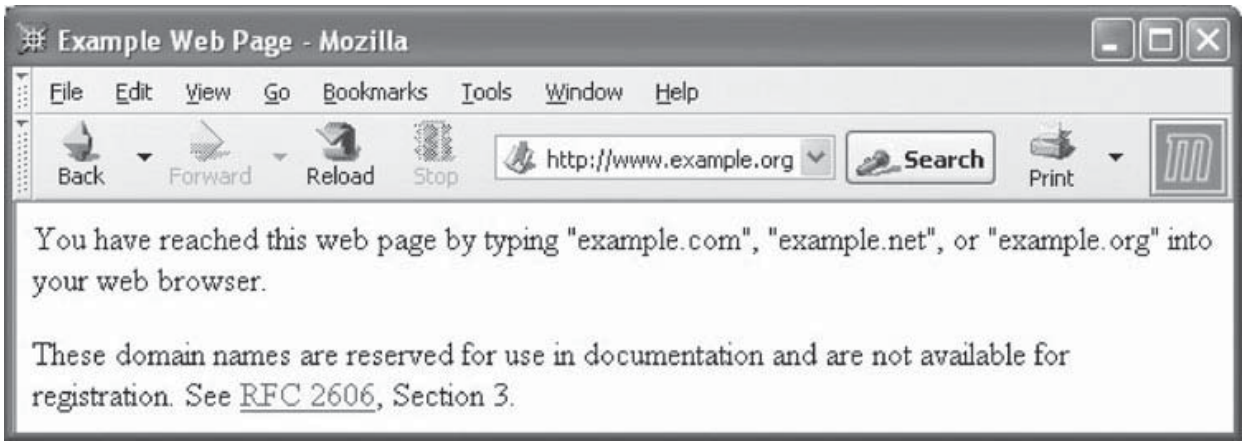

FIGURE 1.3 Web browser displaying information received in an HTTP response message received after the browser sent an HTTP request message to a web server. The content shown is subject to copyright and used by permission of the Internet Assigned Numbers Authority (IANA).

```
Server: Apache/1.3.27 (Unix)  (Red-Hat/Linux)
Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT
ETag: "3f80f-1b6-3e1cb03b"
Accept-Ranges: bytes
Content-Length: 438
Connection: close
Content-Type: text/html

<HTML>
<HEAD>
<TITLE>Example Web Page</TITLE>
</HEAD>
<body>
<p>You have reached this web page by typing &quot;example.com&quot;,
  &quot;example.net&quot;,
  or &quot;example.org&quot; into your web browser.</p>
<p>These domain names are reserved for use in documentation and are
  not available for registration. See
  <a href="http://www.rfc-editor.org/rfc/rfc2606.txt">RFC 2606</a>,
  Section 3.</p>
</BODY>
</HTML>
```

The response message in this case begins with the line

```
HTTP/1.1 200 OK
```

which is known as the *status line* of the response, and continues to the end of the example. The portion of the response between the status line and the first blank line following it is the header of the response. The part following this blank line—from `<HTML>` down—is the body of the response and is written using the HTML language, which will be discussed in the next chapter. For now, just notice that this body contains the information displayed by the browser.

Now that we have an idea of HTTP's basic structure, we'll look at some details of request and response messages.

1.4 HTTP Request Message

1.4.1 Overall Structure

Every HTTP request message has the same basic structure:

Start line
Header field(s) (one or more)
Blank line
Message body (optional)

The start line in the example request in Section 1.3.1 was

```
GET / HTTP/1.1
```

Every start line consists of three parts, with a single space used to separate adjacent parts:

1. Request method
2. Request-URI portion of web address
3. HTTP version

We'll cover each of these parts of the start line—in reverse order—in the next several subsections, then move on to the header fields and body.

1.4.2 HTTP Version

The initial version of HTTP was referred to as HTTP/0.9, and the first Internet RFC (Request for Comments; see the References section (Section 1.9) for more on RFCs) regarding HTTP described HTTP/1.0. In 1997, HTTP/1.1 was formally defined, and is currently an Internet Draft Standard [RFC-2616]. Essentially all operational browsers and servers support HTTP/1.1, including the server that generated the example in Section 1.3.1 (as indicated by the HTTP version portion of the status line). We will therefore focus on HTTP/1.1 in this chapter. If a new version of HTTP is developed in the future, the new standard defining this version will specify a new value for the version portion of the start line (assuming that the new standard has the same start line). The version string for HTTP/1.1 must appear in the start line exactly as shown, with all capital letters and no embedded white space.

1.4.3 Request-URI

The second part of the start line is known as the *Request-URI*. The concatenation of the string `http://`, the value of the Host header field (`www.example.org`, in this example), and the Request-URI (`/` in this example) forms a string known as a *Uniform Resource Identifier* (URI). A URI is an identifier that is intended to be associated with a particular resource (such as a web page or graphics image) on the World Wide Web. Every URI consists of two parts: the *scheme*, which appears before the colon (:), and another part that depends on the scheme. Web addresses, for the most part, use the `http` scheme (the scheme name in URIs is case insensitive, but is generally written in lowercase letters). In this scheme, the URI represents the location of a resource on the Web. A URI of this type is said to be a *Uniform Resource Locator* (URL). Therefore, URIs using the `http` scheme are both URIs and URLs. Some other URI schemes that mark the URI as a URL are shown in Table 1.1. A complete list of the currently registered URI schemes along with references to details on each scheme can be found at [IANA-SCHEMES].

In addition to the URL type of URI, there is one other type, called a *Uniform Resource Name* (URN). While not as common as URLs, URNs are sometimes used in web development (see Section 8.6 for an example). A URN is designed to be a unique name for a resource rather than specifying a location at which to find the resource. For example, an edition of *War and Peace* has an ISBN (International Standard Book Number) of 0-1404-4417-3 associated with it, and this is the only book worldwide with this number.

TABLE 1.1 Some Non-`http` URL Schemes

Scheme Name	Example URL	Type of Resource
`ftp`	`ftp://ftp.example.org/pub/afile.txt`	File located on FTP server
`telnet`	`telnet://host.example.org/`	Telnet server
`mailto`	`mailto:someone@example.org`	Mailbox
`https`	`https://secure.example.org/sec.txt`	Resource on web server supporting encrypted communication
`file`	`file:///C:/temp/localFile.txt`	File accessible from machine processing this URL

So it makes sense to associate information regarding this book, such as bibliographic data, with its ISBN. In fact, this book has an associated URN, which can be written as follows:

```
urn:ISBN:0-1404-4417-3
```

The URI for a URN always consists of three colon-separated parts, as illustrated here. The first part is the scheme name, which is always `urn` for a URN-type URI. The second part is the *namespace identifier*, which in this example is `ISBN`. Other currently registered URN namespace identifiers along with pointers to documentation for each are listed at [IANA-URNS]. The third part is the *namespace-specific string*. The exact format and meaning of this string varies with the namespace. In this example it represents the ISBN of a book and has a format defined by the documentation linked to at [IANA-URNS].

We will have more to say about URLs, particularly those with an `http` scheme, in Section 1.6. For now, we will complete our coverage of the HTTP request start line by examining the first part, the request method.

1.4.4 Request Method

The standard HTTP methods and a brief description of each are shown in Table 1.2. The method part of the start line of an HTTP request must be written entirely in uppercase letters, as shown in the table. In addition to the methods shown, the HTTP/1.1 standard defines a CONNECT method, which can be used to create certain types of secure connections. However, its use is beyond our scope and therefore will not be discussed further here.

The primary HTTP method is GET. This is the method used when you type a URL into the Location bar of your browser. It is also the method that is used by default when you click on a link in a document displayed in your browser and when the browser downloads images for display within an HTML document. The POST method is typically used to send information collected from a form displayed within a browser, such as an order-entry form, back to the web server. The other methods are not frequently used by web developers, and we will therefore not discuss them further here.

TABLE 1.2 Standard HTTP/1.1 Methods

Method	Requests server to . . .
GET	return the resource specified by the Request-URI as the body of a response message.
POST	pass the body of this request message on as data to be processed by the resource specified by the Request-URI.
HEAD	return the same HTTP header fields that would be returned if a GET method were used, but not return the message body that would be returned to a GET (this provides information about a resource without the communication overhead of transmitting the body of the response, which may be quite large).
OPTIONS	return (in Allow header field) a list of HTTP methods that may be used to access the resource specified by the Request-URI.
PUT	store the body of this message on the server and assign the specified Request-URI to the data stored so that future GET request messages containing this Request-URI will receive this data in their response messages.
DELETE	respond to future HTTP request messages that contain the specified Request-URI with a response indicating that there is no resource associated with this Request-URI.
TRACE	return a copy of the complete HTTP request message, including start line, header fields, and body, received by the server. Used primarily for test purposes.

1.4.5 Header Fields and MIME Types

We have already learned that the Host header field is used when forming the URI associated with an HTTP request. The Host header field is required in every HTTP/1.1 request message. HTTP/1.1 also defines a number of other header fields, several of which are commonly used by modern browsers. Each header field begins with a *field name*, such as Host, followed by a colon and then a *field value*. White space is allowed to precede or follow the field value, but such white space is not considered part of the value itself. The following slightly modified example of an actual HTTP request sent by a browser consists of a start line, 10 header fields, and a short message body:

```
POST /servlet/EchoHttpRequest HTTP/1.1
host: www.example.org:56789
user-agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.4)
  Gecko/20030624
accept: text/xml,application/xml,application/xhtml+xml,
  text/html;q=0.9,text/plain;q=0.8,video/x-mng,image/png,image/jpeg,
  image/gif;q=0.2,*/*;q=0.1
accept-language: en-us,en;q=0.5
accept-encoding: gzip,deflate
accept-charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
connection: keep-alive
keep-alive: 300
content-type: application/x-www-form-urlencoded
content-length: 13

doit=Click+me
```

Before describing each of the header fields, it will be helpful to understand some common header field features. First, header names are not case sensitive, although I will throughout this text refer to header field names following the capitalization used by the HTTP/1.1 reference [RFC-2616]. So, while the browser used "host" to name the first header field, I will refer to this as the "Host" header field. Second, a header field value may wrap onto several lines by preceding each continuation line with one or more spaces or tabs, as shown for the User-Agent and Accept fields of the preceding example. This also means that a header field name must begin at the first character of a line, with no preceding white space.

A third common feature is the use of so-called MIME types in several header field values. *MIME* is an acronym standing for Multipurpose Internet Mail Extensions, and refers to a standard that can be used to pass a variety of types of information, including graphics and applications, through e-mail as well as through other Internet message protocols. In particular, as defined in the MIME Internet Draft Standard [RFC-2045], the content of a MIME message is specified using a two-part, case-insensitive string which, in web applications, is known as the *content type* of the message. Two examples of standard MIME content-type strings are text/html and image/jpeg. The substring preceding the slash in these strings is the *top-level type*, and is normally one of a small number of standard types shown in Table 1.3. The substring following the slash, called the *subtype*, specifies the particular type of content relative to the top-level type. A complete list of current registered top-level types and subtypes can be found at [IANA-MIME]. In addition, *private* (unregistered) MIME top-level types and subtypes may be used. A private type or subtype is indicated by an "x-" (or "X-") prefix. Table 1.4 lists some common MIME types.

Yet another common feature of header fields is that many header field values use so-called *quality values* to indicate preferences. A quality value is specified by a string of

TABLE 1.3 Standard Top-level MIME Content Types

Top-level Content Type	Document Content
application	Data that does not fit within another content type and that is intended to be processed by application software, or that is itself an executable binary.
audio	Audio data. Subtype defines audio format.
image	Image data, typically static. Subtype defines image format. Requires appropriate software and hardware in order to be displayed.
message	Another document that represents a MIME-style message. For example, following an HTTP TRACE request message to a server, the server sends a response with a body that is a copy of the HTTP request. The value of the Content-Type header field in the response is message/http.
model	Structured data, generally numeric, representing physical or behavioral models.
multipart	Multiple entities, each with its own header and body.
text	Displayable as text. That is, a human can read this document without the need for special software, although it may be easier to read with the assistance of other software.
video	Animated images, possibly with synchronized sound.

TABLE 1.4 Some Common MIME Content Types

MIME Type	Description
text/html	HTML document
image/gif	Image represented using Graphics Interchange Format (GIF)
image/jpeg	Image represented using Joint Picture Expert Group (JPEG) format
text/plain	Human-readable text with no embedded formatting information
application/octet-stream	Arbitrary binary data (may be executable)
application/x-www-form-urlencoded	Data sent from a web form to a web server for processing

the form `;q=`*num*, where *num* is a decimal number between 0 and 1, with a higher number representing greater preference. Each quality value applies to all of the comma-separated field values preceding it back to the next earlier quality value. So, for example, according to the Accept header field (explained in Section 1.5) the browser in this example prefers text/xml (quality value 0.9) over image/jpeg (quality value 0.2). A final common header field feature is the use of the `*` character in a header field value as a wildcard character. For instance, the string `*/*` in the Accept header field value represents all possible MIME types.

Each of the header fields shown in the example, along with the Referer field (yes, this misspelling of "referrer" is the name of the field in the HTTP/1.1 standard), are briefly described in Table 1.5. The field values for Accept-Charset are discussed in detail in Section 1.5.4. Full details on all of these header fields, along with descriptions of the many other header fields defined in HTTP/1.1 plus an explanation of how you can define your own header fields, are contained in [RFC-2616].

1.5 HTTP Response Message

As we have seen earlier, an HTTP response message consists of a status line, header fields, and the body of the response, in the following format:

Status line
Header field(s) (one or more)
Blank line
Message body (optional)

In this section, we'll begin by describing the status line and then move on to an overview of some of the response header fields and related topics. The message body, if present, is often an HTML document; HTML is covered in the next chapter.

1.5.1 Response Status Line

The example status line shown earlier was

```
HTTP/1.1 200 OK
```

TABLE 1.5 Some Common HTTP/1.1 Request Header Fields

Field Name	Use
Host	Specify *authority* portion of URL (host plus port number; see Section 1.6.2). Used to support *virtual hosting* (running separate web servers for multiple fully qualified domain names sharing a single IP address).
User-Agent	A string identifying the browser or other software that is sending the request.
Accept	MIME types of documents that are acceptable as the body of the response, possibly with indication of preference ranking. If the server can return a document according to one of several formats, it should use a format that has the highest possible preference rating in this header.
Accept-Language	Specifies preferred language(s) for the response body. A server may have several translations of a document, and among these should return the one that has the highest preference rating in this header field. For complete information on registered language tags, see [RFC-3066] and [ISO-639-2].
Accept-Encoding	Specifies preferred encoding(s) for the response body. For example, if a server wishes to send a compressed document (to reduce transmission time), it may only use one of the types of compression specified in this header field.
Accept-Charset	Allows the client to express preferences to a server that can return a document using various character sets (see Section 1.5.4).
Connection	Indicates whether or not the client would like the TCP connection kept open after the response is sent. Typical values are `keep-alive` if connection should be kept open (the default behavior for servers/clients compatible with HTTP/1.1), and `close` if not.
Keep-Alive	Number of seconds TCP connection should be kept open.
Content-Type	The MIME type of the document contained in the message body, if one is present. If this field is present in a request message, it normally has the value shown in the example, application/x-www-form-urlencoded.
Content-Length	Number of bytes of data in the message body, if one is present.
Referer	The URI of the resource from which the browser obtained the Request-URI value for this HTTP request. For example, if the user clicks on a hyperlink in a web page, causing an HTTP request to be sent to a server, the URI of the web page containing the hyperlink will be sent in the Referer field of the request. This field is not present if the HTTP request was generated by the user entering a URI in the browser's Location bar.

Like the start line of a request message, the status line consists of three fields: the HTTP version used by the server software when formatting the response; a numeric *status code* indicating the type of response; and a text string (the *reason phrase*) that presents the information represented by the numeric status code in human-readable form. In this example, the status code is 200 and the reason phrase is `OK`. This particular status code indicates that no errors were detected by the server. The body of a response having this status code should contain the resource requested by the client.

All status codes are three-digit decimal numbers. The first digit represents the general class of status code. The five classes of HTTP/1.1 status codes are given in Table 1.6. The last two digits of a status code define the specific status within the specified class. A few of the more common status codes are shown in Table 1.7. The HTTP standard recommends

TABLE 1.6 HTTP/1.1 Status Code Classes (First Digit of Status Code)

Digit	Class	Standard Use
1	Informational	Provides information to client before request processing has been completed.
2	Success	Request has been successfully processed.
3	Redirection	Client needs to use a different resource to fulfill request.
4	Client Error	Client's request is not valid.
5	Server Error	An error occurred during server processing of a valid client request.

reason phrases for all status codes, but a server may use alternative but equivalent phrases. All status codes and recommended reason phrases are contained in [RFC-2616].

1.5.2 Response Header Fields

Some of the header fields used in HTTP request messages, including Connection, Content-Type, and Content-Length, are also valid in response messages. The Content-Type of a response can be any one of the MIME type values specified by the Accept header field of the corresponding request. Some other common response header fields are shown in Table 1.8.

TABLE 1.7 Some Common HTTP/1.1 Status Codes

Status Code	Recommended Reason Phrase	Usual Meaning
200	OK	Request processed normally.
301	Moved Permanently	URI for the requested resource has changed. All future requests should be made to URI contained in the Location header field of the response. Most browsers will automatically send a second request to the new URI and display the second response.
307	Temporary Redirect	URI for the requested resource has changed at least temporarily. This request should be fulfilled by making a second request to URI contained in the Location header field of the response. Most browsers will automatically send a second request to the new URI and display the second response.
401	Unauthorized	The resource is password protected, and the user has not yet supplied a valid password.
403	Forbidden	The resource is present on the server but is read protected (often an error on the part of the server administrator, but may be intentional).
404	Not Found	No resource corresponding to the given Request-URI was found at this server.
500	Internal Server Error	Server software detected an internal failure.

TABLE 1.8 Some Common HTTP/1.1 Response Header Fields

Field Name	Use
Date	Time at which response was generated. Used for cache control (see Section 1.5.3). This field must be supplied by the server.
Server	Information identifying the server software generating this response.
Last-Modified	Time at which the resource returned by this request was last modified. Can be used to determine whether cached copy of a resource is valid or not (see Section 1.5.3).
Expires	Time after which the client should check with the server before retrieving the returned resource from the client's cache (see Section 1.5.3).
ETag	A hash code of the resource returned. If the resource remains unchanged on subsequent requests, then the ETag value will also remain unchanged; otherwise, the ETag value will change. Used for cache control (see Section 1.5.3).
Accept-Ranges	Clients can request that only a portion (*range*) of a resource be returned by using the Range header field. This might be used if the resource is, say, a large PDF file and only a single page is currently needed. Accept-Ranges specifies the units that may be used by the client in a range request, or `none` if range requests are not accepted by this server for this resource.
Location	Used in responses with redirect status code to specify new URI for the requested resource.

1.5.3 Cache Control

Several of the response header fields described in Table 1.8 are used in conjunction with cache control. In computer systems, a *cache* is a repository for copies of information that originates elsewhere. A copy of information is placed in a cache in order to improve system performance. For example, most personal computer systems use a small, high-speed memory cache to hold copies of some of the data contained in RAM memory, which is slower than cache memory.

Most web browsers automatically cache on the client machine many of the resources that they request from servers via HTTP. For example, if an image such as a button icon is included in a web page, a copy of the image obtained from the server will typically be cached in the client's file system. Then if another page at the same site uses the same image, the image can be retrieved from the client file system rather than sending another HTTP request to the server and waiting for the server's response containing the image. HTTP caching, when successful, generally leads to quicker display by the browser, reduced network communication, and reduced load on the web server.

However, there is a key drawback to using a cache: information in a cache can become *invalid*. For example, if the button image in the preceding example is modified on the server, but a client accesses its cached copy of the older version of the image, then the client will display an invalid version of the image. This problem can be avoided in several ways.

One approach to guaranteeing that a cached copy of a resource is valid is for the client to ask the server whether or not the client's copy is valid. This can be done with relatively little communication by sending an HTTP request for the resource using the HEAD method, which returns only the status line and header portion of the response. If

the response message contains a Last-Modified time, and this time precedes the value of the Date header field returned with the cached resource, then the cached copy is still valid and can be used. Otherwise, the cached copy is invalid and the browser should send a normal GET request for the resource.

A somewhat simpler approach can be used if the server returns an ETag with the resource. The client can then simply compare the ETag returned by a HEAD request with the ETag stored with the cached resource. If the ETag values match, then the cached copy is valid; otherwise, it is not. This approach avoids the complexity of comparing two dates to determine which is larger.

Finally, if the server can determine in advance the earliest time at which a resource will change, the server can return that time in an Expires header. In this case, as long as the Expires time has not been reached, the client may use the cached copy of the resource without the need to validate with the server. If an Expires time is not included in a response, a browser may use a heuristic algorithm to choose an expiration time and then behave as if this time had been passed to it in an Expires header. This behavior can be prevented by sending an Expires time that precedes the Date value (a value of 0 is commonly used for this purpose). If this is done, then an HTTP/1.1-compliant browser will validate before each access to the resource.

The HTTP/1.1 specification provides a variety of other header fields related to caching; see [RFC-2616] for full details.

1.5.4 Character Sets

Finally, a word about how characters are represented in web documents. As you know, characters are represented by integer values within a computer. A *character set* defines the mapping between these integers, or *code points*, and characters. For example, US-ASCII [RFC-1345] is the character set used to represent the characters used in HTTP header field names, and is also used in key portions of many other Internet protocols. Each US-ASCII character can be represented by a 7-bit integer, which is convenient in part because the messages transmitted by the Internet Protocol are viewed as streams of 8-bit bytes, and therefore each character can be represented by a single byte.

However, many characters in common use in modern languages are not contained in the US-ASCII character set. Over the years, a wide variety of other character sets have been defined for use with languages other than U.S. English and also for representing characters that are not associated with human language representation, such as mathematical and graphical symbols.

For web pages, which are meant to be viewed throughout the world, it is vital that a single worldwide character set be used. So, as in the Java™ programming language, the underlying character set used internally by web browsers is defined by the Unicode™ Standard [UNICODE]. The Unicode Standard is an attempt to provide a single character set that encompasses every human language representation as well as all other commonly used symbols. The Unicode Standard's Basic Multilingual Plane (BMP), which covers most of the commonly used characters in every modern language, uses 16-bit character codes, and the full character code space of the Unicode Standard extends to 21-bit integers.

Of course, if the resource requested by a client is written using the US-ASCII character set, then sending 21 (or more) bits per character from the server to the client would take roughly three times as long as sending the ASCII characters. Therefore, most browsers, for purposes of efficiency and compatibility, accept a variety of character sets in addition to those in Unicode. See [IANA-CHARSETS] for a complete (long) list of character sets currently registered for use over the Internet.

More generally, in addition to a variety of character sets, most browsers also accept certain character encodings. A *character encoding* is a bit string that must be decoded into a code-point integer that is then mapped to a character according to the definition provided by some character set. A character encoding often represents characters using variable-length bit strings, with common characters represented using shorter strings and less-common characters using longer strings. For example, UTF-8 and UTF-16 are encodings of the character set in Unicode that use variable numbers of 8- and 16-bit values to encode all possible Unicode Standard characters. (Don't confuse character encoding with the message encoding concept mentioned earlier. Message encoding typically involves applying a general-purpose compression algorithm to the body of a message, regardless of the character encoding used.)

The Accept-Charset header field is used by a client to tell a server the character sets and character encodings that it will accept as well as its preferred character sets or encodings, if more than one is available for the requested document. In our earlier example, the header field

```
accept-charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
```

said that the client would prefer to receive documents using the ISO-8859-1 character set or the UTF-8 encoding of the characters in Unicode, but that it would also accept any other valid Internet character set/encoding. (ISO-8859-1 is an 8-bit superset of US-ASCII that contains many characters found in Latin-based languages but not in English. ISO-8859-1 and UTF-8 are preferred even though they have the same quality value as `*` because specific field values are given preference over the `*` wildcard.)

A web server can inform a client about the character set/encoding used in a returned document by adding a `charset` parameter to the value of the Content-Type header field. For example, the following Content-Type header field in an HTTP response would indicate that the body of the message is an HTML document written using the UTF-8 character encoding:

```
Content-Type: text/html; charset=UTF-8
```

The US-ASCII character set is a subset of both the ISO-8859-1 character set and the UTF-8 character encodings, so the `charset` parameter is set to one of these two values for many US-ASCII documents in order to ensure international compatibility. We will learn other ways to indicate the character set/encoding for a document in later chapters.

Now that we have covered HTTP in some detail, we're ready to look at the primary software applications that communicate using HTTP: web clients and servers. We'll begin with the more familiar client software before moving on to web servers.

1.6 Web Clients

A *web client* is software that accesses a web server by sending an HTTP request message and processing the resulting HTTP response. Web browsers running on desktop or laptop computers are the most common form of web client software, but there are many other forms of client software, including text-only browsers, browsers running on cell phones, and browsers that speak a page (over the phone, for example) rather than displaying the page. In general, any web client that is designed to directly support user access to web servers is known as a *user agent*. Furthermore, some web clients are not designed to be used directly by humans at all. For example, software *robots* are often used to automatically crawl the Web and download information for use by search engines (and, unfortunately, e-mail spammers).

We will focus here on traditional browsers, since they are the most widely used web client software and have features that are generally a superset of those found in other clients. A brief history of these browsers will provide some useful background.

Early web browsers generally either were text-based or ran on specialized platforms, such as computers from Sun Microsystems or the now-defunct NeXT Systems. The Mosaic™ browser, developed at the National Center for Supercomputer Applications (NCSA) in 1993, was the starting point for bringing graphical web browsing to the general public. The developers of Mosaic founded Netscape Communications Corporation, which dedicated a large team to developing and marketing a series of Netscape Navigator® browsers based on Mosaic. Microsoft soon followed with the Microsoft® Internet Explorer (IE) browser, which was originally based on Mosaic.

For a time, a "browser war" was waged between Netscape and Microsoft, with each company trying to add features and performance to its browser in order to increase its market share. Netscape soon found itself at a disadvantage, however, as Microsoft began bundling IE with its popular Windows® operating system. The war soon ended, and Microsoft was victorious. Netscape, acquired by America Online (at the time primarily an Internet service provider), chose to make its source code public and launched the Mozilla project as an open-source approach to developing new core functionality for the Netscape® browser. In particular, Netscape browser releases starting with version 6.0 have been based on software developed as part of the Mozilla project.

At the time of this writing, IE is by far the most widely used browser in the world. However, the Mozilla™ and Firefox™ browsers from the Mozilla Foundation are increasingly popular, and other browsers, including the Opera™ and Safari™ browsers, also have significant user communities.

Despite this diversity, all of the major modern browsers support a common set of basic user features and provide similar support for HTTP communication. A number of common browser features are discussed in the remainder of this section. For concreteness, I will also explain how to access the features described using one particular browser, Mozilla 1.4, and will also use the Mozilla browser for most examples in later chapters. A primary reason for choosing to use Mozilla as a concrete browser example is that it runs on Linux®, Windows, and Macintosh® systems. Also, the fact that it is open source means that if you're curious about details of how a feature operates, you have access to the source code itself. In addition to having essentially all of the features found in IE, Mozilla has some nice

tools for software developers that are not found in basic IE distributions. Finally, as we will learn in later chapters, Mozilla browsers are designed to comply with HTML and other Internet standards, while IE is (at this time, at least) less standards compliant. Instructions for downloading and installing Mozilla 1.4 are found in Appendix A.

1.6.1 Basic Browser Functions

The window of a typical modern browser is split into several rectangular regions, most of which are known as *bars*. Figure 1.4 shows five standard regions in a Mozilla 1.4 window. The primary region is the *client area*, which displays a document. For many documents, the *title bar* displays a title assigned by the document author to the document currently displayed within the client area. The title bar also displays the browser name as well as standard window-management controls. The *menu bar* contains a set of dropdown menus, much like most other applications that incorporate a graphical user interface (GUI). We'll take a closer look at the Mozilla menus in Section 1.6.3. The browser's *Navigation toolbar* contains standard push-button controls that allow the user to return to a previously viewed web page (Back), reverse the effect of pressing Back (Forward), ask the server for an updated version of the page currently viewed (Reload), halt page downloading currently in progress (Stop), and print the client area of the window (Print). Clicking the small down-arrow to the right of some buttons produces a menu allowing users to override the default behavior of the associated button. For example, clicking the arrow to the right of Back produces a menu of titles of a number of documents that have been recently viewed, any of which can be loaded into the client area by selecting its title from the menu. The Navigation toolbar also contains a text box, known as the *Location bar*, where a user can enter a URL and press the Enter key in order to request the browser to display the document located at the specified URL. Clicking the Search button instead of pressing Enter causes the information entered in the text box to be sent to a search engine. Clicking the down-arrow at the right side of the Location bar produces a dropdown menu of recently visited URLs that can be visited again with a single click. Finally, the *status bar* displays messages and icons related to the

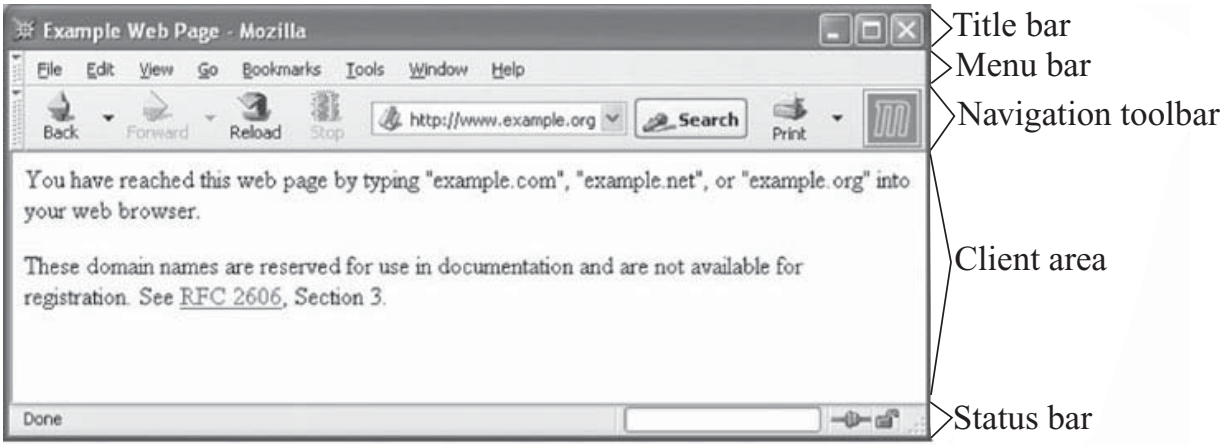

FIGURE 1.4 Some of the standard Mozilla bars. The content shown is subject to copyright and used by permission of IANA.

status of the browser. For example, the two icons in the right portion of the status bar in Figure 1.4 show that the browser is online (left icon) and that the browser is communicating with the server over an insecure communication channel. The messages displayed in the left portion of the status bar are normally information about the communication between client and server (Table 1.9).

A primary task of any browser is to make HTTP requests on behalf of the browser user. If a user types an `http`-scheme URL in Mozilla's Location bar, for example, the browser must perform a number of tasks:

1. Reformat the URL entered as a valid HTTP request message.
2. If the server is specified using a host name (rather than an IP address), use DNS to convert this name to the appropriate IP address.
3. Establish a TCP connection using the IP address of the specified web server.
4. Send the HTTP request over the TCP connection and wait for the server's response.
5. Display the document contained in the response. If the document is not a plain-text document but instead is written in a language such as HTML, this involves *rendering* the document: positioning text and graphics appropriately within the browser window, creating table borders, using appropriate fonts and colors, etc.

Before discussing various features of browsers that can be controlled by users, it will be helpful to have a more complete understanding of URLs.

1.6.2 URLs

An `http`-scheme URL consists of a number of pieces. In order to show the main possibilities, let's consider the following example URL:

```
http://www.example.org:56789/a/b/c.txt?t=win&s=chess#para5
```

The portion of an `http` URL following the `://` string and before the next slash (`/`) (or through the completion of the URL, if there is no trailing slash) is known as the *authority* of the URL. It consists of either a fully qualified domain name (or other name that can be resolved to an IP address, such as an unqualified name of a machine on the local network) or an IP

TABLE 1.9 Some Mozilla Status Messages

Status Message	Meaning
Resolving host www.example.org . . .	Requested IP address from DNS; waiting for response.
Connecting to www.example.org . . .	Creating TCP connection to server.
Waiting for www.example.org . . .	Sent HTTP request to server; waiting for HTTP response.
Transferring data from www.example.org . . .	HTTP response has begun, but has not completed.
Done	HTTP response has been received, although further processing may be needed before the document will be displayed.

address of an Internet web server, optionally followed by a colon (:) and a port number. As indicated earlier, if the port number is omitted, then a TCP connection to port 80 is implied. In this example, the authority is `www.example.org:56789` and consists of the fully qualified domain name `www.example.org` followed by the port number 56789.

The portion from the slash following the authority through the question mark (?) (or through the end of the URL, if there is no question mark) is called the *path* of the URL. The leading slash is part of the path, but the question mark is not. So the path in the example URL just given is `/a/b/c.txt`. The fact that this looks a great deal like a Linux file reference to a file named `c.txt` located within the `b` subdirectory of the `a` directory of the root (`/`) of the file system is not entirely a coincidence. In many cases, the path portion of a URL is in fact concatenated by the server with a base file path in order to form an actual file path on the server's system. We'll learn more about how servers use URL paths later in this chapter as well as in later chapters.

Following the path there may be a question mark followed by information up to a number sign (#). The information between but not including the question mark and number sign is the *query* portion of the URL, and in general a string of the form shown is known as a *query string*. The query portion of the example URL is `t=win&s=chess`. Originally, the query portion of a URL was intended to pass search terms to a web server. So in this example, it might be that the user is seeking a resource with a title containing the string "win" that is related to the subject "chess." As we will learn in later chapters, while query strings are still sometimes used to represent search terms, they are also used for a variety of other purposes in modern web systems. We will also learn that query strings may appear in the body of POST requests, as well as how to encode special characters in the query strings sent to web servers.

A browser forms the Request-URI portion of an HTTP request from a URL by concatenating the path and query portions of the URL with an intervening question mark. Thus, the Request-URI for the example URL would be

```
/a/b/c.txt?t=win&s=chess
```

Syntactically, the query portion of a URL can only be present if the path portion is present. If both the path and query are missing from a URL, then the Request-URI must be set to `/`, which is known as the *root* path. This is why we used a `/` as the Request-URI in the example of Section 1.3.1.

The final optional part of an `http`-scheme URL—the portion following but not including the number sign—is known as the *fragment* of the URL, and the string contained in the fragment is known as a *fragment identifier*. Fragment identifiers are used by browsers to scroll HTML documents; details are given in the next chapter, which covers HTML.

Summarizing, if a user types a URL such as the one considered into a browser's Location bar and presses Enter, the browser will generate an HTTP request message as follows. The request start line will begin with GET. The path and query portions of the URL will be used as the second, Request-URI portion of the start line. Assuming the browser is HTTP/1.1 compliant, the final portion of the start line will be the string `HTTP/1.1` (this string must be uppercase). The request will also contain a Host header field having as its value the authority portion of the URL. The fragment portion of the URL is not sent to

the web server, but is instead used by the browser to modify the way in which it displays any HTML document sent to the browser in the HTTP response returned as a result of this request. Other header fields will also generally be included, as described earlier.

So, given the example URL, the browser would send a request containing the lines (spacing and some capitalization might vary from that shown):

```
GET /a/b/c.txt?t=win&s=chess HTTP/1.1
...
Host: www.example.org:56789
...
```

1.6.3 User-Controllable Features

Graphical browsers also provide many user-controllable features, including:

- *Save:* Most documents can be saved by the user to the client machine's file system. If the document is an HTML page that contains other documents, such as images, then the browser will attempt to save all of these documents locally so that the entire page can be displayed from the local file system. A user saves a document in Mozilla under the **File**|**Save Page As** menu.
- *Find in page:* Standard documents (text and HTML) can be searched with a function that is similar to that provided by most word processors. In Mozilla, the find function is accessed under the **Edit**|**Find in This Page** menu. (Mozilla also provides a "find as you type" feature under Edit that is similar to the incremental search in Emacs, for users familiar with that paradigm.)
- *Automatic form filling:* The browser can "remember" information entered on certain forms, such as billing address, phone numbers, etc. When another form is visited at a later date, the browser can automatically fill in previously saved data. The **Edit**|**Save Form Info** and **Edit**|**Fill in Form** menu options can be used to save and retrieve form information in Mozilla. The **Tools**|**Form Manager** menu can be used to manage saved form information.
- *Preferences:* Users can customize browser functionality in a wide variety of ways. In Mozilla, a window presenting preference options is obtained by selecting **Edit**|**Preferences** (Figure 1.5). The Appearance, Navigator, and Advanced categories (left subwindow) and their subcategories are used to customize Mozilla. Some preference settings directly related to the HTTP topics covered earlier are:
  - *Accept-Language:* The non-* values sent by the browser for this HTTP request header field can be set under the **Navigator**|**Languages** category, **Languages for Web Pages** box.
  - *Default character set/encoding:* The character set/encoding to be assumed for documents that do not specify one is also set under **Navigator**|**Languages** in the **Character Coding** box.
  - *Cache properties:* The amount of local storage allocated to the cache and the conditions controlling when a cached file will be validated are set under **Advanced**|**Cache** in the **Set Cache Options** box.

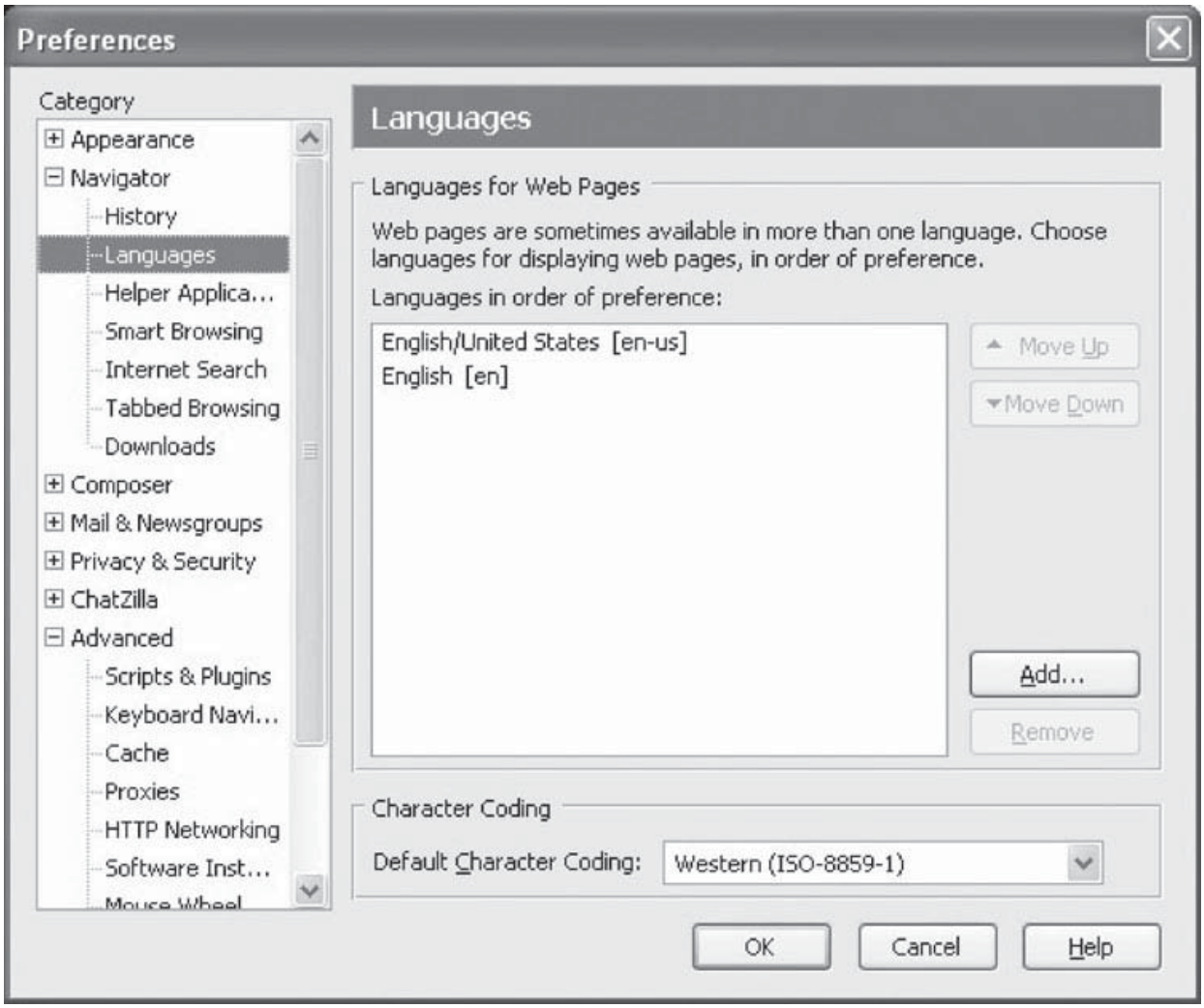

FIGURE 1.5 Preferences window with Languages category selected.

  - *HTTP settings:* The version of HTTP used and whether or not the client will keep connections alive is set under **Advanced|HTTP Networking** in the **Direct Connection Options** box.
- *Style definition:* The user can define certain aspects affecting how the browser renders HTML pages, such as font sizes, background and foreground colors, etc. In Mozilla, the font size can be modified using **View|Text Zoom**. If a page offers alternative styles, they can be selected using the **View|Use Style** menu as discussed in Chapter 3, where methods for changing default browser style settings are also described.
- *Document meta-information:* Interested users can view information about the displayed document, such as the document's MIME type, character encoding, size, and, if the document was written using HTML, the raw HTML source from which the rendering in the client area was produced. In Mozilla, **View|Page Source** is used to view raw HTML, and **View|Page Info** to view other so-called *meta-information*, that is, information about the document rather than information contained in the document itself.
- *Themes:* The look of one or more of the browser bars, particularly the navigation bar, can be modified by applying a certain theme (sometimes called a "skin"). In Mozilla, the

browser scheme can be modified using **View|Apply Theme**. Additional themes can be obtained from **View|Apply Theme|Get New Themes**.

- *History:* The browser will automatically maintain a list of all pages visited within the last several days. Users can use the history list to easily return to any recently visited page. In Mozilla, the history list can be reached by selecting **Go|History**.
- *Bookmarks ("favorites" in Internet Explorer):* Users can explicitly *bookmark* a web page, that is, save the URL for that page for an indefinite length of time. At any later time, the browser's bookmark facility can be used to easily return to any bookmarked page. Options under the **Bookmarks** menu in Mozilla allow users to bookmark a page, return to a bookmarked page, and edit the bookmark list.

1.6.4 Additional Functionality

In addition to the facilities for end users described in the preceding subsection, browsers perform a number of other functions, including:

- *Automatic URL completion:* If the user has entered a URL in the Location bar and begins to type it again (within the next several days), the URL will be completed automatically by the browser.
- *Script execution:* In addition to displaying documents, browsers can run programs (scripts). These programs can perform a variety of tasks, from validating data entered on a form before sending it to a web server to creating various dynamic effects on web pages, such as drop-down menus.
- *Event handling:* When the user performs an action, such as clicking on a link or a button in a web page, the browser treats this as the occurrence of an *event*. Browsers recognize a number of different types of events, including mouse button clicks, mouse movement, and even events not directly under user control such as the completion of the browser's rendering of a document. A browser can perform a variety of actions in response to an event—loading a document from a URL, clearing a form, or calling a script function defined by the document author, for example.
- *Management of form GUI:* If a web page contains a form with fill-in fields, the browser must allow the user to perform standard text-editing functions within these fields. It also needs to automatically provide certain graphical feedback, such as changing a button image when it is pressed or providing a text cursor in a text field that will receive keyboard input.
- *Secure communication:* When the user sends sensitive information, such as a credit card number, to a web server, the browser can encode this information in a way the prevents any machines along the IP route from the client to the server from obtaining the information.
- *Plug-in execution:* While the browser itself normally understands only a limited number of MIME types, most browsers support some form of *plug-in* protocol that allows the browser's capabilities to be supplemented by other software. If a browser has a plug-in for displaying, say, a document conforming to the application/pdf MIME type, then when the browser receives such a document it will pass it—via the plug-in protocol—to the appropriate plug-in for display. Some plug-ins may display the document within the

browser's client area, while others may display in a separate window that is controlled by the plug-in itself. Plug-ins are often installed automatically, after user permission is obtained, when an unsupported MIME type is encountered. To see a list of plug-ins installed in your copy of Mozilla, select **Help|About Plug-ins**.

Some other standard browser features, such as a facility for managing so-called *cookies*, are described in later chapters. In addition to standard browser features, Mozilla also provides a number of tools specifically designed for use by software developers, such as a script console and debugging tools. Some of these tools will also be described in later chapters.

This completes our coverage of web browsers. It's now time to move to the software running on the other end of the HTTP communications pipeline: web servers.

1.7 Web Servers

In this section, we'll cover basic functionality found in most web servers as well as some specific instructions for accessing and modifying the parameters for one particular web server, Tomcat 5.0. We'll also briefly look at how web servers support secure communication with browsers.

1.7.1 Server Features

The primary feature of every web server is to accept HTTP requests from web clients and return an appropriate resource (if available) in the HTTP response. Even this basic functionality involves a number of steps (the quoted terms used in this list are defined in subsequent paragraphs):

1. The server calls on TCP software and waits for connection requests to one or more ports.
2. When a connection request is received, the server dedicates a "subtask" to handling this connection.
3. The subtask establishes the TCP connection and receives an HTTP request.
4. The subtask examines the Host header field of the request to determine which "virtual host" should receive this request and invokes software for this host.
5. The virtual host software maps the Request-URI field of the HTTP request start line to a resource on the server.
6. If the resource is a file, the host software determines the MIME type of the file (usually by a mapping from the file-name extension portion of the Request-URI), and creates an HTTP response that contains the file in the body of the response message.
7. If the resource is a program, the host software runs the program, providing it with information from the request and returning the output from the program as the body of an HTTP response message.
8. The server normally logs information about the request and response—such as the IP address of the requester and the status code of the response—in a plain-text file.

9. If the TCP connection is kept alive, the server subtask continues to monitor the connection until a certain length of time has elapsed, the client sends another request, or the client initiates a connection close.

A few definitions will be helpful before proceeding to more detailed coverage of web server features. First, all modern servers can concurrently process multiple requests. It is as if multiple copies of the server were running simultaneously, each devoted to handling the requests received over a single TCP connection. The specifics of how this concurrency is actually implemented on a system may depend on many factors, including the number of processors available in the system, the programming language used, and programmer choices. We will learn more about concurrent server processing in Chapter. 6. For now, I will simply use the term *subtask* to refer to the concept of a single "copy" of the server software handling a single client connection.

Another term that may need some explanation is *virtual host*. As noted earlier, every HTTP request must include a Host header field. The reason for this requirement is that multiple host names may all be mapped by the Internet DNS system to a single IP address. For example, a single server machine within a college may host web sites for multiple departments. Each web site would be assigned its own fully qualified domain name, such as `www.cs.example.edu`, `www.physics.example.edu`, and so on. But DNS would be configured to map all of these domain names to a single IP address. When an HTTP request is received by the web server at this address, it can determine which *virtual* host is being requested by examining the Host header. Separately configured software can then be used to handle the requests for each virtual host.

Finally, as noted in point 7, the documents returned by web servers are often produced by executing software at the time of the HTTP request rather than being generated beforehand and stored in the server's file system for later retrieval. One significant difference between web servers concerns the support that each has for executing software written in various traditional programming languages as well as in scripting languages. We'll touch on some of these differences in the next subsection, which briefly surveys the history of web server development.

1.7.2 Server History

Just as the NCSA Mosaic™ browser was the starting point for subsequent browser development efforts by Netscape and Microsoft, NCSA's *httpd* web server was also a starting point for server development. httpd was used on a large fraction of the early web servers, but the NCSA discontinued development of the server in the mid-1990s. When this happened, several individuals who were running httpd at their sites joined forces and began developing their own updates to the open-source httpd software. Their updates were called "patches," and this led to calling their work "a patchy server," which soon became known as "the Apache server." They made the first public release of their free, open-source server in April 1995, and within a year Apache was the most widely used server on the Web. It has held that distinction to this day, although many large corporate and government sites tend to use commercial server software instead.

As with web browsers, Microsoft began development of web servers well after others had begun, but quickly caught up. Microsoft's Internet Information Server (IIS) provides

essentially all of the features found in Apache, although IIS does have the drawback of running only on Windows systems, while Apache runs on Windows, Linux, and Macintosh systems. IIS and Apache are, at the time of this writing, by far the most widely used servers on the market.

Both servers can be configured to run a variety of types of programs, although certain programming languages tend to be used more frequently on one system than the other. For example, many IIS servers run programs written in VBScript (a derivative of Visual Basic), while a typical Apache server might run programs written in either Perl or the PHP scripting language (PHP stands for "PHP Hypertext Processor"; yes, the definition is infinitely recursive). A number of IIS and Apache servers also run Java programs. When running a Java program, both Apache and IIS servers are usually configured to run the program by using separate software called a *servlet container*. The servlet container provides the Java Virtual Machine that runs the Java program (known as a *servlet*), and also provides communication between the servlet and the Apache or IIS web server.

Tomcat is a popular, free, and open-source servlet container developed and maintained by the Apache Software Foundation, the same organization that is continuing development of the Apache web server. In addition to running as a servlet container called on by web servers, Tomcat can also be run as a standalone web server that communicates directly with web clients. Furthermore, the standalone Tomcat server can serve documents stored in the server machine's file system and run programs written in non-Java languages.

To provide a concrete illustration of server configuration, we will next cover configuration of a Tomcat 5.0 server in some detail (this is the server you will have if you follow the instructions for installing JWSDP in Appendix A). The Tomcat material presented here is not meant to be a comprehensive reference, but is primarily intended to introduce you to some key terms and concepts that are encountered when setting up any web server, not just Tomcat. Since we will be using Java servlets and related technologies to illustrate server-side programming in later chapters, it is natural for us to focus on Tomcat rather than non-Java servers in this chapter. If you understand Tomcat configuration well, configuring a basic IIS or Apache server should not be particularly difficult.

1.7.3 Server Configuration and Tuning

Modern servers have a large number of configuration parameters. In this section, we will cover many of the key configuration items found in Tomcat. Similar features, along with some not found in Tomcat, are included in the Apache and IIS servers.

Broadly speaking, server configuration can be broken into two areas: external communication and internal processing. In Tomcat, this corresponds to two separate Java packages: Coyote, which provides the HTTP/1.1 communication, and Catalina, which is the actual servlet container. Some of the Coyote parameters, affecting external communication, include the following:

- IP addresses and TCP ports that may be used to connect to this server.
- Number of subtasks (called *threads* in Java) that will be created when the server is initialized. This many TCP connections can be established simultaneously with minimal overhead.

- Maximum number of threads that will be allowed to exist simultaneously. If this is larger than the previous value, then the number of threads maintained by the server may change, either up or down, over time.
- Maximum number of TCP connection requests that will be queued if the server is already running its maximum number of threads. Connection requests received if the queue is full will be refused.
- Length of time the server will wait after serving an HTTP request over a TCP connection before closing the connection if another request is not received.

The settings of these parameters can have a significant influence on the performance of a server; changing the values of these and similar parameters in order to optimize performance is often referred to as *tuning* the server. As with all optimization problems, there are various trade-offs involved in attempting to tune a server. For example, increasing the maximum number of simultaneous threads that may execute increases memory requirements and thread-management overhead, and may lead to slower responses to individual requests, due to sharing CPU cycles among the large number of threads. On the other hand, lower values for this parameter may lead to some clients having their connection requests refused, which may lead some users to believe that the site is down. Tuning is therefore often performed by trial and error: if a server seems to be running poorly by some measure, the server administrator may try to vary one or more of these parameters and observe the impact, retaining parameter values that seem to help. *Load generation* or *stress test* tools can be used to simulate requests to a web server, and can therefore be helpful for experimenting with tuning parameters based on anticipated traffic patterns even before a web site "goes live." A fuller discussion of server tuning is beyond the scope of this book.

The internal Catalina portion of Tomcat also has a number of parameter settings that affect functionality. These settings can determine:

- Which client machines may send HTTP requests to the server.
- Which virtual hosts are listening for TCP connections on a given port.
- What logging will be performed.
- How the path portion of Request-URIs will be mapped to the server's file system or other resources.
- Whether or not the server's resources will be password protected.
- Whether or not resources will be cached in the server's memory.

The Tomcat 5.0 server you have installed if you followed the instructions in Appendix A has a web interface for setting most of these parameters. If your server is installed at the default port 8080 and you open a browser on the machine running the server, then browsing to the URL

```
http://localhost:8080
```

(more on `localhost` in Section 1.7.4) and clicking the Server Administration link (you may need to scroll down to find this link) should cause a log-in page to be displayed. Otherwise, if the server is not on the machine you are browsing from, or if your browser is not at port

8080, modify the URL to contain the correct host and/or port number. You were asked for a user name and password when you installed Tomcat; enter them on this log-in page. You should then see a page such as the one in Figure 1.6.

Because your copy of Tomcat was included in the Java Web Services Developer Pack (JWSDP), there is already a JWSDP Service entry in the list on the left side of the browser window. Each Service in Tomcat is almost its own web server, except that a Service cannot be individually stopped and started (only the underlying server can be stopped and started, as described in Appendix A). We will only cover here how to change parameters of the JWSDP Service; the procedures for creating a new Service are similar.

First, click on the handle icon next to the JWSDP Service entry in order to reveal its associated server components (Figure 1.7). This Service has five components: one each of Connector, Host, Logger, Realm, and Valve. A Connector is a Coyote component that handles HTTP communications directed to a particular port. Clicking on the Connector item in the JWSDP Service list will produce a window such as the one shown in Figure 1.8. The panel on the right in this figure is typical of the panels displayed for creating and editing Tomcat components. At the top of the panel is a dropdown menu of possible actions that can be performed for this component, such as creating subcomponents or deleting a component (there are no actions for this particular component). Below this menu is a Save button that must be clicked after entering data in the fields further below in order to save this data. This temporarily saves the data from these fields in memory, but any changes made are not saved permanently to disk until the Commit Changes button at the top of the window is clicked. Furthermore, the server will, in general, ignore the committed changes until it is restarted.

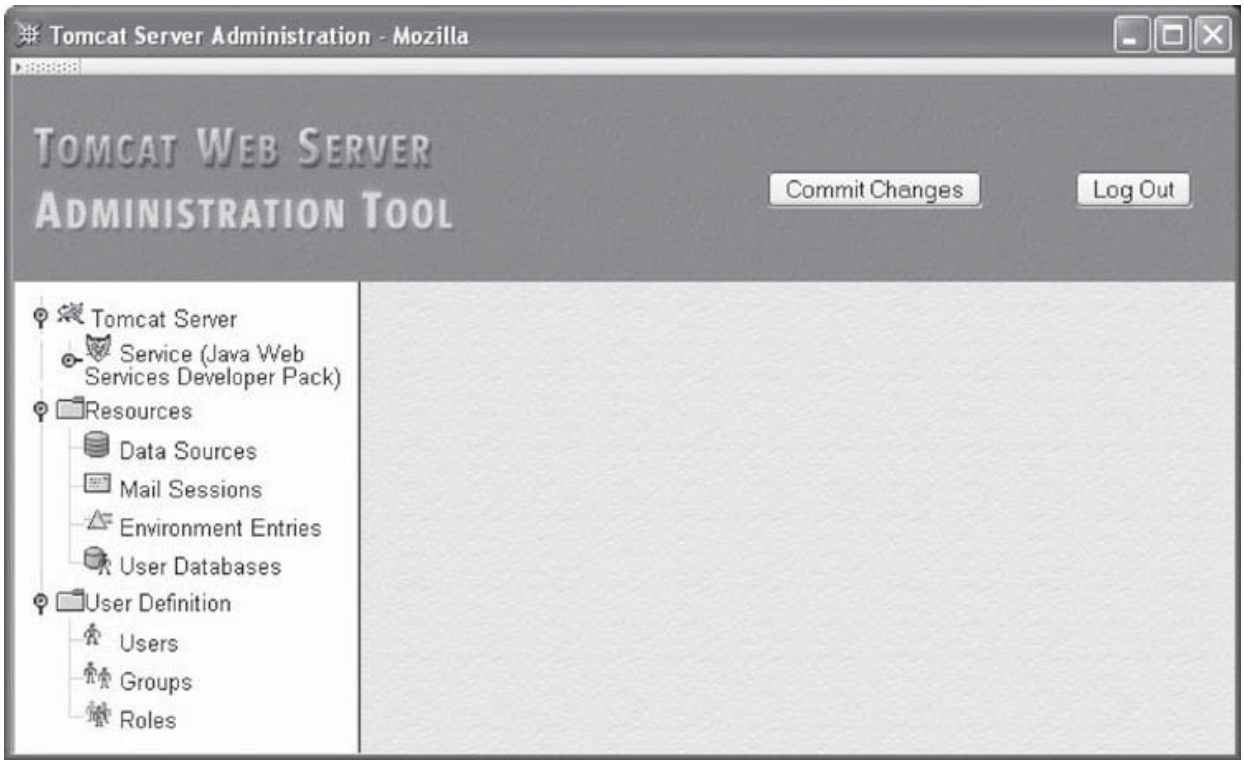

FIGURE 1.6 Tomcat administration tool entry page. The content of this screen shot is reproduced by permission of the Apache Software Foundation.

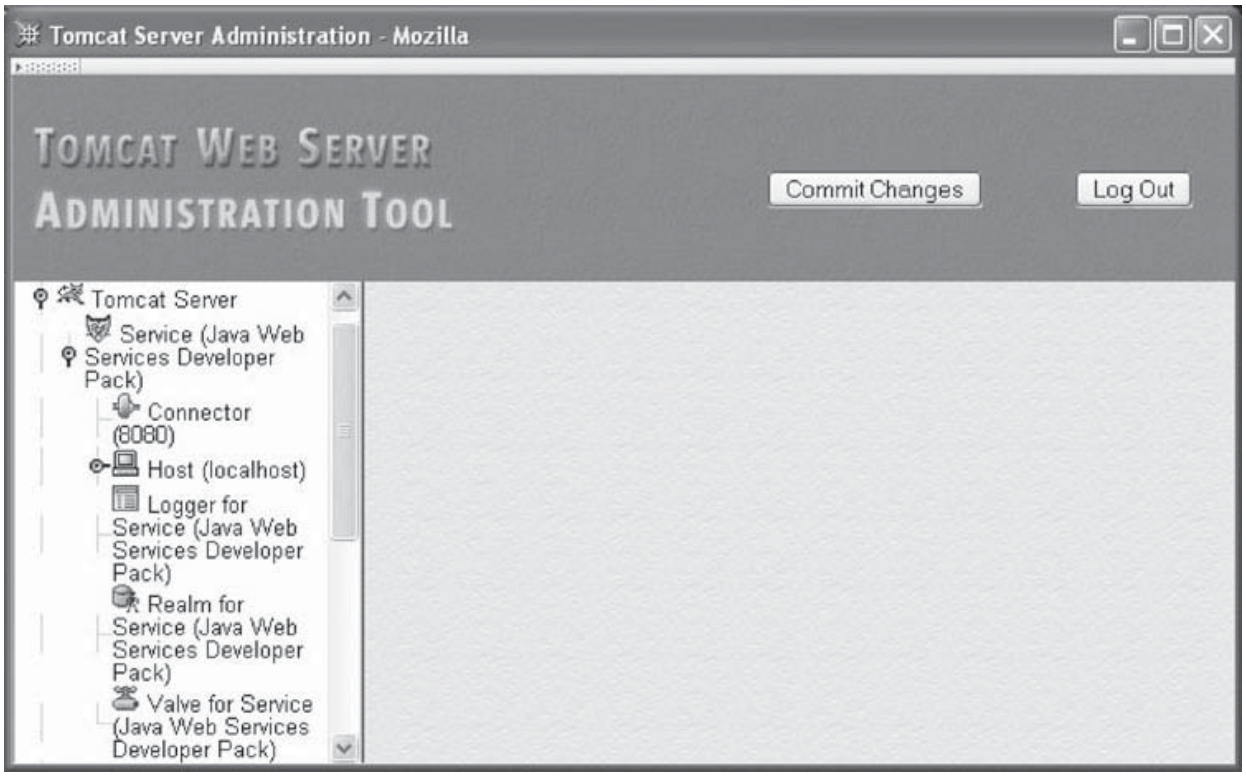

FIGURE 1.7 List of Service components produced by clicking on Service "handle" icon. The content of this screen shot is reproduced by permission of the Apache Software Foundation.

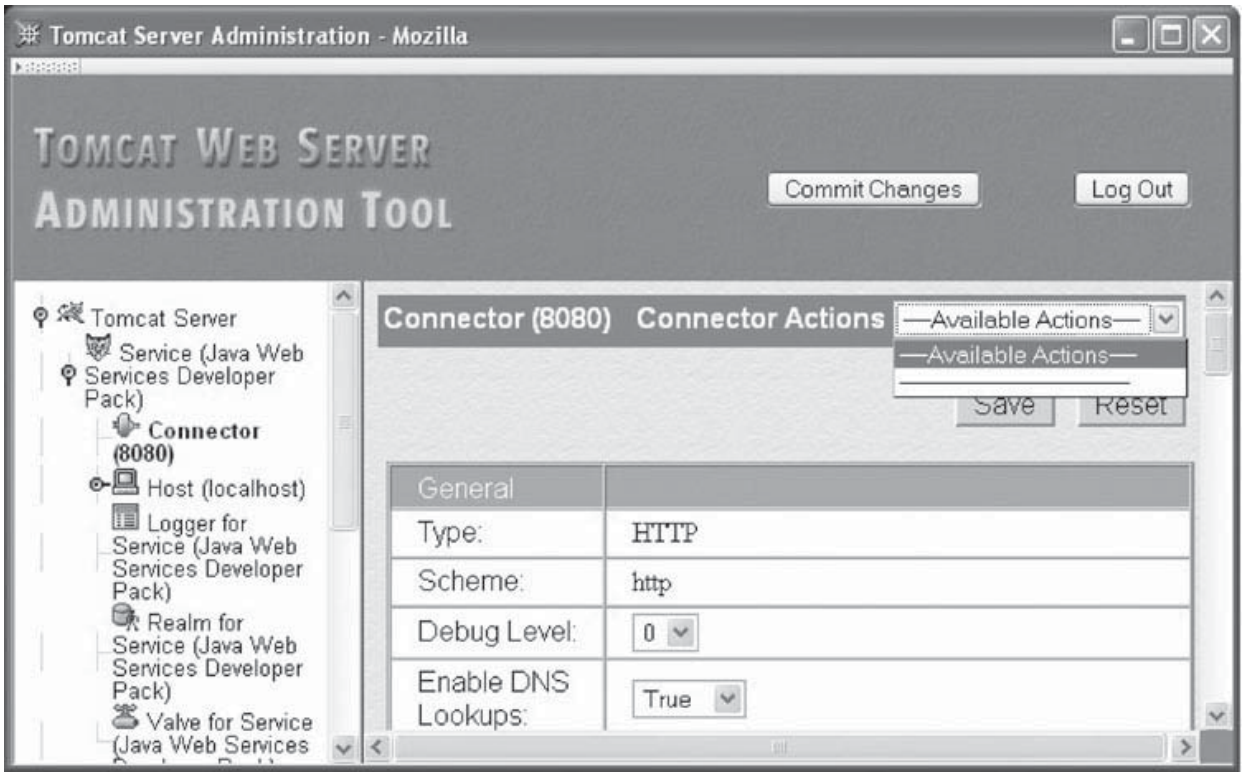

FIGURE 1.8 Connector edit page. The content of this screen shot is reproduced by permission of the Apache Software Foundation.

Some of the data fields in a panel, such as Edit Connector, have fixed values, while others can be edited (if we were creating a Connector, all fields would be editable). Some of the key fields for the Connector component type are listed in Table 1.10. Notice that the Port Number field value (8080 in this example) is used as the name of the Connector in the Service list. This is because the Port Number value for this Connector will be unique to this Connector, since each IP port can "belong" to, at most, one application on a system. On the other hand, multiple Connectors can be associated with a single Service, so a Service can potentially be accessed through multiple ports.

1.7.4 Defining Virtual Hosts

The Host component (Figure 1.9) is used to define a virtual host. Some of the key fields are described in Table 1.11. The virtual host name should normally be a fully qualified domain name that would be used by visitors to your web site, although the Host supplied as part of the JWSDP Service is given the unqualified name `localhost`. This is a special name that the DNS system treats as a reference to a special IP address, 127.0.0.1. If an IP message is sent to this address, the IP software causes the message to loop back to itself for receipt. In short, browsing to a URL with domain name `localhost` causes the browser to send the HTTP request to a web server on the machine running the browser. So it would seem that this virtual host should only be accessible if the browser runs on the server machine. However, clicking on the JWSDP Service link in the left panel reveals (in the right panel) that the value of the Default Hostname field for this Service is `localhost`. This means that if a user browses to this Service using a URL with a host name other than `localhost`, the request will be passed to the `localhost` virtual host. In essence, this Host component will respond to any HTTP request sent to the Service's Connector (at port 8080), regardless of the value of the request's Host header field.

TABLE 1.10 Some of the Fields for the Connector Component

Field Name	Description
Accept Count	Length of the TCP connection wait queue.
Connection Timeout	Server will close connection if it is idle for this many milliseconds.
IP Address	Blank indicates that this Connector will accept TCP connections directed to any IP address associated with this machine. Specifying an address restricts connections to requests for that address.
Port Number	Port number on which this Connection will listen for TCP connection requests.
Min Spare Threads	Initial number of threads that will be allocated to process TCP connections associated with this Connector. Once connections are established with the Connector, the server will maintain at least this many *idle* processing threads, that is, threads waiting for new connections but otherwise unused.
Max Threads	Maximum number of threads that will be allocated to process TCP connections associated with this Connector.
Max Spare Threads	Maximum number of idle threads allowed to exist at any one time. The server will begin stopping threads if the number of idle threads exceeds this value.

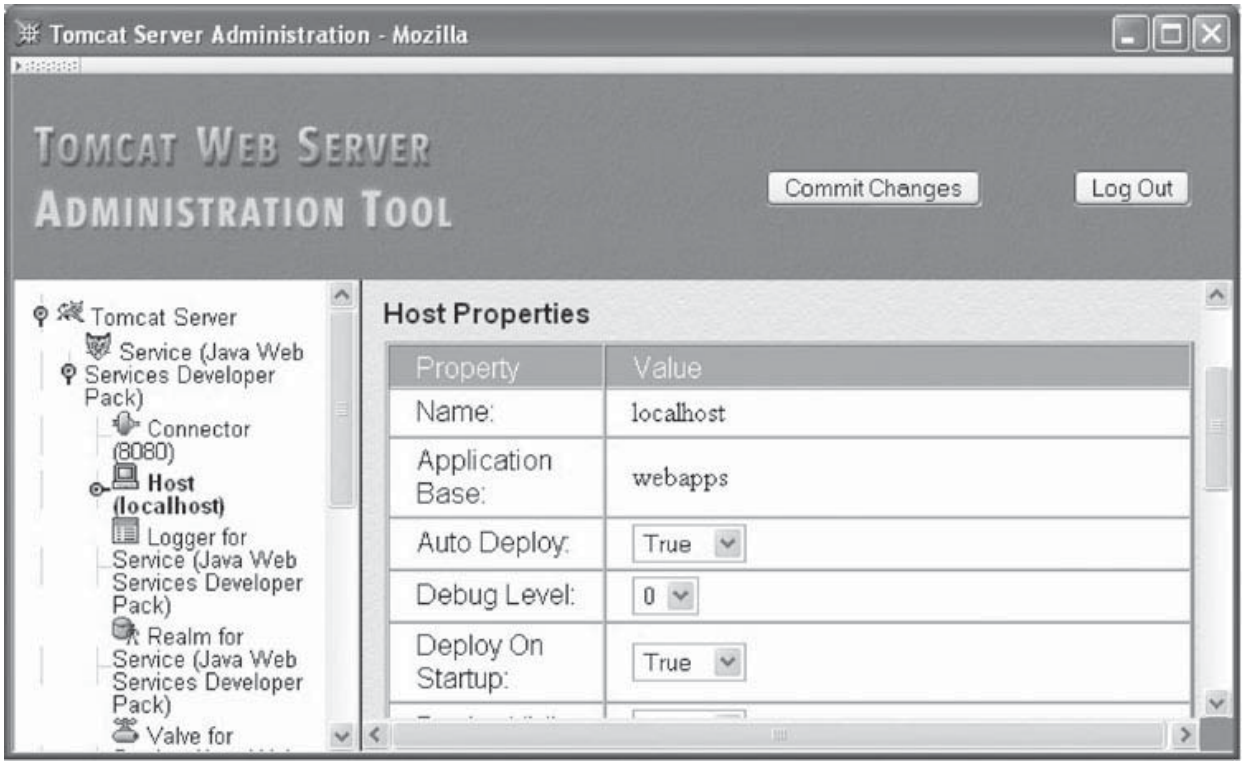

FIGURE 1.9 Host component panel for the JWSDP Service. The content of this screen shot is reproduced by permission of the Apache Software Foundation.

Now let's assume an additional Host component with name, say, `www.example.org` was added to this Service (through the Tomcat Administration Tool by clicking on the Service in the left panel of the web page and then selecting the Create New Host item from the Service Actions menu in the right panel). Then this new virtual host would handle requests containing a Host header field with value `www.example.org`, while all requests with any other Host value would continue to be handled by the default `localhost` virtual host.

Several of the fields listed in Table 1.11 are associated with web applications. A *web application* is a collection of files and programs that work together to provide a particular function to web users. For example, a Web site might run two web applications: one for

TABLE 1.11 Key Fields for Host Component

Field Name	Description
Name	Usually the fully qualified domain name (or `localhost`) that clients will use to access this virtual host.
Application Base	Directory containing *web applications* for this virtual host (see text).
Deploy on Startup	Boolean value indicating whether or not web applications should be automatically initialized when the server starts.
Auto Deploy	Boolean value indicating whether or not web applications added to the Application Base while the server is running should be automatically initialized.

use by administrators of the site that provides maintenance functionality, and another for use by external clients that provides customer functionality. In Tomcat, a web application is represented by a Context component. Clicking on a Host handle icon will reveal the list of Contexts provided with that virtual host. If you open the `localhost` Host, you will find that it has several contexts predefined (Figure 1.10).

Each Host and Context is associated with a directory in the server's file system. The directory associated with a Host is specified by the value of the Application Base field. If this value is a *relative pathname*—a pathname that does not begin with a `/` in Linux or with a drive specification such as `C:\` in Windows—then it is taken as relative to the directory in which JWSDP 1.3 (and therefore Tomcat 5.0) was installed. For example, on my Linux machine I installed JWSDP 1.3 at `/usr/java/jwsdp-1.3`, so the relative pathname `webapps` given in Figure 1.9 corresponds on my machine to the directory `/usr/java/jwsdp-1.3/webapps`. [I will normally use forward slash (/) as the separator in file paths; change this to backslash (\) if you are using Windows.] This is known as the *absolute pathname* for the directory, and could have been specified instead of the relative pathname. Using a relative pathname for the Application Base value is generally recommended, since this allows your JWSDP 1.3 installation to be moved to another location within the server file system without the need to change the Application Base value.

The directory associated with a Context is specified by the value of the Document Base field (Figure 1.10). The figure shows an absolute pathname value (on a Windows system), but again the pathname can be relative instead. However, if a relative pathname is specified, it will be relative to the Application Base, not relative to the JWSDP 1.3 installation directory. So, assuming that the Application Base is at `C:\jwsdp-1.3\webapps`, the Document Base in Figure 1.10 could have been specified as simply `ROOT`. If you create a Context (by selecting Create New Context from the Host Action menu for a Host), be sure to create the directory that will be specified in the Document Base field before clicking the Save button for the Context.

As we will discuss in some detail in Chapter 8, a Context associates certain URLs with the specified Document Base. Figure 1.10, for example, shows that the root URL path (/) is associated with a directory named ROOT. And, in fact, if you examine the `webapps/ROOT` directory of your JWSDP 1.3 installation, you will find a file `THIRDPARTYLICENSEREADME.html` that contains the text (and some other information, discussed in the next chapter) that is displayed when you navigate to `http://localhost:8080/THIRDPARTYLICENSEREADME.html` (or the equivalent URL for your server). Similarly, if you navigate to `http://localhost:8080/`, you will see the contents of the `webapps/ROOT/index.html` file. This is because the server by default displays certain "welcome" files (such as `index.html`) if you do not explicitly specify a file name at the end of the path portion of the URL used to visit the server. What's more, navigating to `http://localhost:8080/servlets-examples` will display the contents of the `webapps/servlets-examples/index.html` file, because (as you can verify by clicking on the `/servlets-examples` Context object) the Document Base for URLs with paths beginning with `/servlets-examples` is the `webapps/servlets-examples` subdirectory of your JWSDP 1.3 installation. Note that the URL path for the Context object is specified using the Path field of the edit page.

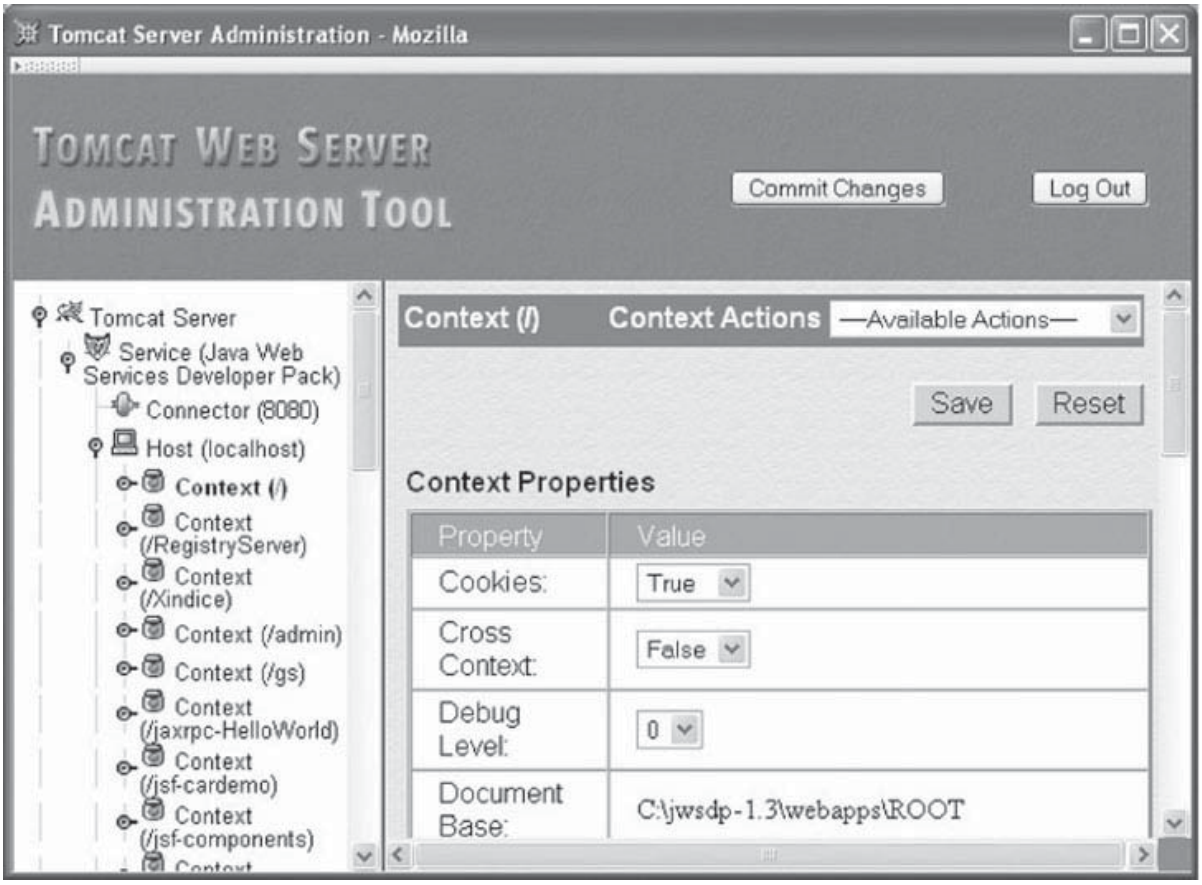

FIGURE 1.10 Context edit page. The content of this screen shot is reproduced by permission of the Apache Software Foundation.

This brief introduction to virtual host concepts is intended to provide you with enough information to be able to set up your own virtual host that will serve simple text files. Again, we will have much more to say about associating URLs with server resources in later chapters. For now, we will move on to some other server capabilities.

1.7.5 Logging

Web server *logs* record information about server activity. The primary web server log recording normal activity is an *access log*, a file that records information about every HTTP request processed by the server. A web server may also produce one or more *message logs* containing a variety of debugging and other information generated by web applications as well as possibly by the web server itself. Finally, information written to the standard output and error streams by the web server or applications may also be logged. We will cover Tomcat's handling of these types of logs as well as some general logging concepts in this subsection.

Access logging in Tomcat is performed by adding a Valve component to a Service. For example, Figure 1.7 shows that the JWSDP Service includes a Valve, and if you click on it, you will find that it is of type AccessLogValve (some other types of Valves are discussed in the next subsection). The primary fields for an AccessLogValve are shown in Table 1.12.

TABLE 1.12 Key Fields for Valve Component of Type AccessLogValve

Field Name	Description
Directory	Directory (relative to Tomcat installation directory or absolute) where log file will be written
Pattern	Information to be written to the log (see text)
Prefix	String that will be used to begin log file name
Resolve Hosts	Whether IP addresses (`False` value) or host names (`True` value) should be written to the log file
Rotatable	Whether or not date should be added to file name and file should be automatically rotated each day
Suffix	String that will be used to end log file name

The combination of values for the Directory, Prefix, Rotatable, and Suffix fields determine the file system path to the access log. The JWSDP Service settings for the values of these Valve fields cause the access log for this Service to be written to the `logs` directory under the JWSDP 1.3 installation directory in a file that starts with the string `access_log.` and ends with the string `.txt`. In between these strings, because Rotatable is given the value `True`, Tomcat inserts the current date, in YYYY-MM-DD (year-month-day) format. So an example JWSDP access log name might be `access_log.2005-07-20.txt`. If you have started and browser to your Tomcat server, you should see one or more files of this form in the `logs` directory under your JWSDP 1.3 installation directory.

The Tomcat server writes one line of information per HTTP request processed to the access log, with the information to be output and its format specified by the Pattern field. The Pattern for the JWSDP Service access log Valve is

```
%h %l %u %t "%r" %s %b
```

This corresponds to what is often called the *common* access log format (in fact, the word `common` can be specified as the value of the Pattern field to specify this log format). The following is an example access log line in common format (this example is split into two lines for readability):

```
www.example.org - admin [20/Jul/2005:08:03:22 -0500]
"GET /admin/frameset.jsp HTTP/1.1" 200 920
```

The following information is contained in this log entry:

- Host name (or IP address; see Table 1.12) of client machine making the request
- User name used to log in, if server password protection is enabled (user "admin" logged in here)
- Date and time of response, plus the time zone (offset from GMT) of the time
- Start line of HTTP request (quoted)
- HTTP status code of response (200 in this example)
- Number of bytes sent in body of response

The Tomcat 5.0 server always returns the hyphen character (-) as the value of the `%l` pattern.

An advantage of using this log format is that a variety of *log analyzers* have been developed that can read logs in this (and some other) formats and produce reports on various aspects of a site's usage. For example, a log analyzer might report on the number of accesses per day, the percentage of requests that received error status codes, or a breakdown of accesses by domain. Such information can be useful for server tuning, locating software problems, or modifying site content to better target a desired audience. Analog (`http://www.analog.cx`) is one popular free log analyzer available at the time of this writing.

Another standard log format can be obtained by specifying the value `combined` for the Pattern. The `combined` format is the same as the common format but also has the Referer and User-Agent HTTP header field values appended. Custom log formats can also be created; see the section on the Valve component in the Tomcat 5 Server Configuration Reference [APACHE-TOMCAT-5-CONFIG] for details.

The Tomcat Logger component can be used to create a message log for a Service such as the JWSDP service (see Figure 1.7). A message log records informational, debugging, and error messages passed to logging methods by either servlets or Tomcat itself. Some of the key fields for File Loggers (the standard type of message log) are described in Table 1.13.

The JWSDP service sets the values of these fields so that the message log produced is written to the `logs` directory under the JWSDP 1.3 installation directory in a file that starts with the string `jwsdp_log.` followed by the date (this is not an option for message logs in Tomcat) and ends with the string `.txt`. If you look at the contents of one of these files, you will see lines such as

```
2005-08-02 07:38:54 createObjectName with StandardEngine[Catalina]
```

Because the JWSDP service has its Timestamp property set to `true`, the beginning of each message log entry begins with a *timestamp*, that is, with the date and time at which the entry was written to the log. Timestamps can be useful, particularly when trying to debug an application. One thing to be aware of when using timestamps is that some applications may write timestamps in universal (GMT) time, whereas others, including Tomcat, use local time.

Loggers can be associated with different levels of the Tomcat object hierarchy: with a Service (such as JWSDP, the example just given); with a Host within a Service (such as `localhost`); and even with a Context, or web application, within a Host (such as `admin`,

TABLE 1.13 Key Fields for Logger Component of Type File Logger

Field Name	Description
Directory	Directory (relative to Tomcat installation directory or absolute) where log file will be written
Prefix	String that will be used to begin log file name
Suffix	String that will be used to end log file name
Timestamp	Whether or not date and time should be added to beginning of each message written to the log file

the web application that implements the Tomcat administration tool). For example, if you examine the `admin` Context under the `localhost` Host using the Tomcat administration tool, you will see that this Context has its own Logger that produces files beginning with `localhost_admin_log.`, also within the `logs` directory. Messages sent to logging methods by servlets within the `admin` web application will go to this message log file rather than to the JWSDP Service's logger. In general, logging methods will search for a Logger beginning in the Context, then the Host, and finally the Service, sending the log message to the first Logger found. Access logs in Tomcat can also be associated with different levels of the server object hierarchy, although typically there is only one access log per service.

Finally, Tomcat itself or servlets it runs may write directly to the Java standard output and error streams `System.out` and `System.err`. The JWSDP 1.3 installation of Tomcat redirects both of these streams to a file named `launcher.server.log` in the `logs` subdirectory of the JWSDP 1.3 installation directory. Thus, if you write an application that prints an exception stack trace, this is likely where you will find it.

1.7.6 Access Control

Tomcat can provide automatic password protection for resources that it serves. At its heart, this is a two-stage process. First, a database of user names is created. Each user name is assigned a password and a list of *roles*. Think of a role as a user's functional relationship to a web application: administrator, developer, end user, etc. Some users may be assigned to multiple roles. The second stage is to tell Tomcat that certain resources can only be accessed by users who belong to certain roles and who have authenticated themselves as belonging to one of these roles by logging in with an appropriate user name and password. For example, the Tomcat administration tool application (`admin` Context) can only be accessed by users who have logged in and who belong to the admin role.

The second stage of this process—associating resources with required roles—is normally performed by web application developers, as described in Section 8.3.3. The first stage—defining one or more user databases—can be performed by web system administrators, application developers, or both. The JWSDP Service contains an example of a database defined at the Service level through the use of a Realm component, which associates a user database with a Service (Figure 1.11). This particular type of Realm indicates that a Tomcat Resource—an object representing a file or other static resource on the server—will be used to store the user database. The Realm's Resource Name field contains the name of the Resource, which in this case is UserDatabase. If you click on the User Databases link in the Resources list in the left panel of your Tomcat administration tool window and then click on the UserDatabase link in the User Databases panel, you will see that this Resource is associated with a file located at `conf/tomcat-users.xml` (this is relative to the Tomcat installation directory). The administration tool also automatically loads the contents of this file under the User Definition folder in the left panel. Clicking on the Users link under this folder shows that there is one user name in this user database: the user name that you chose for the Tomcat administrator when you installed Tomcat. Finally, clicking on this user name in the right panel shows the roles to which a user logged in with this user name belongs: admin and manager.

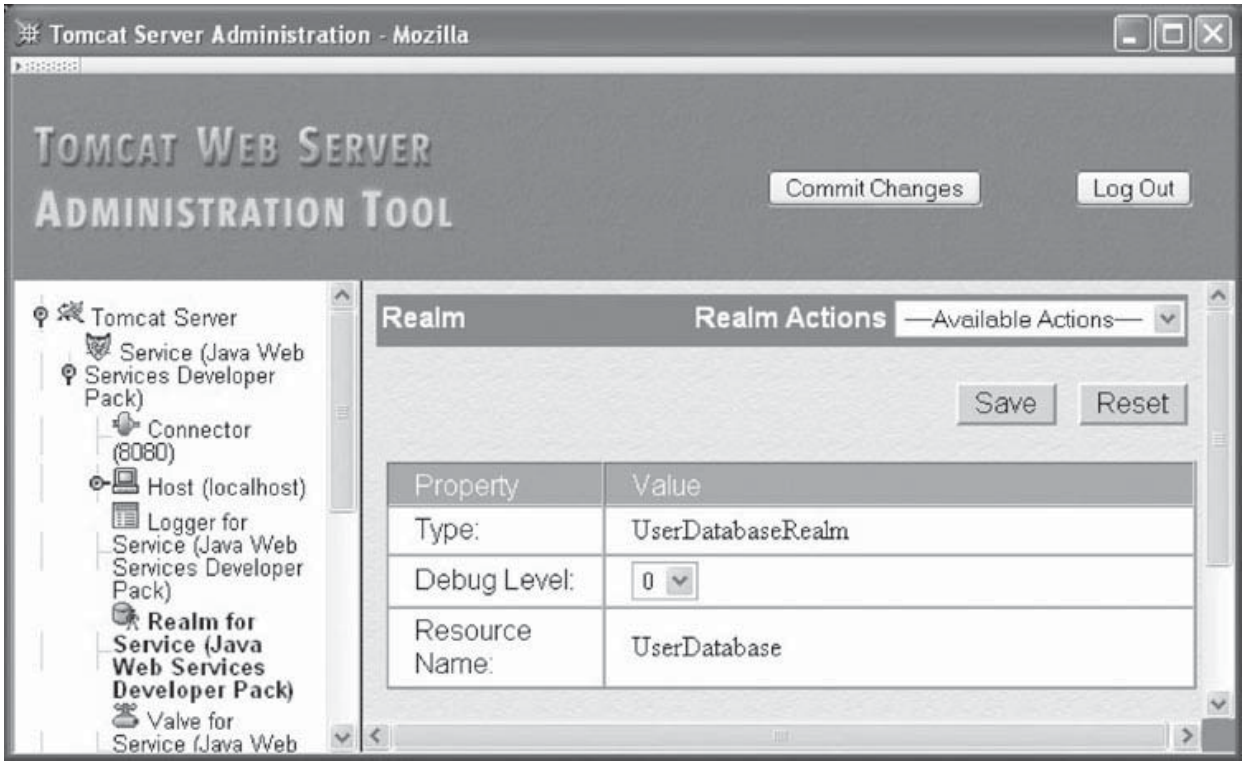

FIGURE 1.11 Realm component panel for the JWSDP Service. The content of this screen shot is reproduced by permission of the Apache Software Foundation.

As mentioned, the admin role is the role required to run the Tomcat administration tool. So, if you wanted to allow another user to run this tool (and other web applications accessible in the admin role), you would simply create that user by selecting Create New User from the Actions dropdown menu of the Users panel and be sure to check the admin role for that user.

A coarser-grained access control can be provided by using Valve objects of type RemoteHostValve and RemoteAddressValve. Both are used to specify client machines that should be rejected if they request a connection to the server. They differ only in whether client machine host names or IP addresses are specified. Each type of Valve has two possible lists of clients: an Allow list and a Deny list. If one or more host names (comma-separated) is entered in the Allow list, then only these hosts can access the server. You can use the * wildcard in place of any label within a host name. So, for example, to allow access only from machines in the `example.org` and `example.net` domains, you would enter in the Allow list

```
*.example.org,*.example.net
```

In addition, whether or not any names are entered in the Allow list, any hosts (possibly wildcarded) entered in the Deny list will be prevented from accessing the server. So, to exclude a single machine from the `example.org` domain while allowing all of the others, we might enter in the Deny list something like

```
baduser.example.org
```

1.7.7 Secure Servers

Normally, the HTTP request and response messages are sent as simple text files. Because these messages are carried by TCP/IP, each message may travel through a number of machines before reaching its destination. It is possible that some machine along the route will extract information from the IP messages it forwards for nefarious purposes. Furthermore, it is often possible for other machines sharing a local network with the sending or receiving machine to snoop the network and view messages associated with other machines as if they were sent to the snooper. In general, any machine other than the sender or receiver that extracts information from network messages is known as an *eavesdropper*.

To prevent eavesdroppers from obtaining sensitive information, such as credit card numbers, all such sensitive information should be *encrypted* before being transmitted over any public communication network. The standard means of indicating to a browser that it should encrypt an HTTP request is to use the `https` scheme on the URL for the request. For example, entering the URL

```
https://www.example.org
```

in Mozilla's Location bar will cause the browser to attempt to send an encrypted HTTP GET request to `www.example.org`.

Various protocols have been used to support encryption of HTTP messages. Many browsers and servers support one or more versions of the Secure Socket Layer (SSL) protocol as well as the newer Transport Layer Security (TLS) protocol, which is based on SSL 3.0. The following description of HTTP encryption is derived from the TLS 1.0 specification [RFC-2246], but the same general ideas apply to the earlier SSL protocols as well.

A client browser that wishes to communicate securely with a server begins by initiating (over TCP/IP) a *TLS Handshake* with the server. During the Handshake process, the server and client agree on various parameters that will be used to encrypt messages sent between them. The server also sends a *certificate* to the client. The certificate enables the client to be sure that the machine it is communicating with is the one the client intends (as specified by the host name in the URL the browser is requesting). Certificates are necessary to avoid so-called *man-in-the-middle attacks*, in which some machine intercepts a message intended for another machine (the target), prevents the message from further forwarding, and returns an HTTP reply to the sender pretending to be from the target. Such an interception could occur at a rogue Internet bridge device on the route between client and server, or through unauthorized alteration of the DNS system, for example.

At the conclusion of the TLS Handshake, the client uses the cryptographic parameter information obtained to encrypt its HTTP request message before sending it to the server over TCP/IP. The server's TLS software decrypts this request before any other server processing is performed. The server similarly encrypts its response before sending it to the client, and the client immediately decrypts the received message. Therefore, other HTTP-processing software running on the client and server are, for the most part, unaffected by the encryption process.

One small point involves the port used for the TCP/IP communication of TLS data. Since the TLS protocol begins with a TLS Handshake, and not with an HTTP request start line, different communication ports are used for the two types of communication. Whereas the default port for HTTP communication is 80, the default for TLS/SSL is 443. This port can be overridden just as the HTTP port can be overridden, by explicitly adding a port number after the host name in an `https`-scheme URL. So, for example, to access the root of a secure server on localhost at port 8443, you would use the URL

```
https://localhost:8443/
```

Tomcat supports the TLS 1.0 and earlier protocols. To enable the secure server Tomcat features, you must do two things:

1. Obtain and install a certificate.
2. Configure the server to listen for TLS connections on some port.

For test purposes, you can generate your own "self-signed" certificates using the `keytool` program distributed with Sun Java™ JDK™ development software. This program is located in the same directory as the `javac` and `java` programs. Assuming that this directory is included in your PATH environment variable, you can begin to create a self-signed certificate suitable for use with Tomcat 5.0 by entering the following at a command prompt:

```
keytool -genkey -alias tomcat -keyalg RSA
```

This says that you want to generate a self-signed certificate that can be referenced by the name `tomcat` and that the encryption/decryption keys generated for use with this certificate should be compatible with the RSA encryption/decryption algorithm (which is the algorithm Tomcat uses). You will be prompted to enter several pieces of information. Since this certificate is self-signed and will be used for test purposes only, for the most part, it does not matter what you enter. However, I suggest entering the fully qualified domain name of your machine when asked to enter your first and last name, as this will prevent a warning later when you try to use the certificate. Also, I suggest using the password `changeit` when asked, which will allow you to use defaults when you configure the server to use this certificate (but use this password for testing purposes only).

Configuring the server to listen for TLS connections simply involves adding a second Connector to a Service (by selecting Create New Connector from the Service's Action drop-down menu). The Type field of the new Connector must be set to HTTPS. On the resulting Connector panel, make sure that the Secure field is set to `True` (since this is a secure connection), and fill in the port number (say 8443) to be used for this connection. Other fields can retain their default values if you run `keytool` with its defaults. After Saving and Committing the changes made in order to create your new Connector, stop your server. If you have not already performed the JWSDP 1.3 postinstallation tasks described in the appendix (Section A.4.2), do so at this time. Now restart your Tomcat server, close and reopen your browser, and then browse to `https://localhost:8443` (modify this as appropriate for the host name and port number for your secure server). If you created a self-signed certificate,

you should see a message asking you whether or not you wish to accept the certificate. After accepting it, you should see the default JWSDP web page produced by your server. Note that a small padlock icon at the bottom of your browser window is shown locked, indicating that the page is being viewed securely.

Since there is no independent validation of self-signed certificates, anyone can generate a self-signed certificate for your machine. This is why browsers will typically display a warning message if a self-signed certificate is presented by a server: while it is syntactically a certificate, it does not prevent a man-in-the-middle attack, because an attacker could easily have generated the certificate. In order for your server to provide transparent secure communication using certificates that browsers will trust automatically, you must have your certificate verified and then digitally "signed" (for a fee) by a certificate authority, such as VeriSign. Details are provided in the SSL section of the Tomcat User Guide [APACHE-TOMCAT-5-UG].

1.8 Case Study

To provide some context for the various technologies we will be covering, an ongoing case study will be part of most chapters. Specifically, we'll create a simple tool for writing and reading a web log (*blog*). One user, the *blogger*, will be able to add text entries to the blog. The most recent entry will appear at the beginning of a web page, followed by the next most recent, and so on for all entries made during the current month. Links elsewhere on the page will provide access to entries made in earlier months (Figure 1.12). Other capabilities will be described in later chapters.

Although we're not ready to start developing any software for this application, we can make some decisions related to the material covered in this chapter:

- Which browsers will we support?
- Which web server(s) will we use?
- How extensive will our security measures be?

At the time of this writing, if our application runs well with IE6 and Mozilla-like browsers, then we will have covered a large percentage of browsers in use, so we will test our application against Mozilla 1.4 and IE6. We will use the Tomcat server distributed with JWSDP 1.3 because it is freely available, runs on multiple platforms, is simple to configure (compared with running, say, both Apache and Tomcat), is sufficiently fast for our needs, and supports the technologies that we will be covering in later chapters.

The question of security is somewhat more difficult. The key security task for the case study application is to prevent everyone but the blogger from adding entries to the blog. A preceding task—one that is beyond the scope of this textbook—is to make sure that the machine running the web server is itself secure from unauthorized access. Obviously, if someone can gain administrative privileges on the server machine, then no amount of work we put into securing the application itself will make it truly secure.

Assuming a secure server machine, the weakest level of security would be to have a "secret" URL that the blogger visits in order to add an entry to the blog. This approach is open to several attacks, one of which is to simply try a variety of reasonable URLs. For

FIGURE 1.12 Example blog page with entries on left and links on right.

example, if the blog can be read by visiting the URL `http://www.example.com/blog/read`, we might guess that the "secret" URL is `http://www.example.com/blog/add`. Somewhat more security can be achieved by requiring the blogger to log in before adding an entry. A weakness with this approach, as pointed out in the preceding section, is that an eavesdropper might be able to learn the log-in information (eavesdropping might also be used to learn a "secret" URL). This weakness can be overcome by using a secure server and visiting only `https`-scheme URLs when logging in and adding entries to the blog. Even this level of security can be defeated if the log-in information can be guessed, so for even more security we would force all passwords to conform with certain conventions (e.g., consist of at least eight characters including both a lowercase and an uppercase letter plus at least one digit).

The application we develop will require that the blogger log in before adding an entry, but we won't require a secure server or password conventions. This is probably an appropriate level of security for this application: we want to discourage people from impersonating the blogger, but the damage if someone does is probably not so great that it requires the additional development effort (and potential problems for users) of additional security measures.

1.9 References

Unlike most topics in this textbook, there is no one definitive reference for the history of the Internet. The brief history presented here was culled from a number of sources. A good starting point for further reading is the Internet Society's list of links to online Internet

histories (http://www.isoc.org/internet/history/). One of the more comprehensive histories is Hobbes' Internet Timeline (http://www.zakon.org/robert/internet/timeline/). An outstanding Internet history through 1992 is available at the Computer History Museum (http://www.computerhistory.org/exhibits/internet_history/). Finally, a good starting point for early World Wide Web history is the World Wide Web Consortium's "A Little History of the World Wide Web" at http://www.w3.org/History.html.

A particularly nice feature of the Internet is that from the earliest days of ARPANET, electronic communication has been used to support technical discussions about and documentation of network standards. Much of this documentation is in the form of RFCs (requests for comment), which are basically numbered memos written by and to the Internet technical community. The RFC collection is maintained by an organization known, appropriately enough, as the RFC Editor (http://rfc-editor.org).

Most of the key standards for the Internet are documented via one or more RFCs. An organization known as the Internet Engineering Steering Group (IESG) is responsible for deciding which RFCs become standards. A list of the RFCs describing all current Internet standards is maintained at http://www.rfc-editor.org/rfcxx00.html, and periodic snapshots of this information are published in document form [STD-1]. Section 1 of this document gives an overview of the standards process.

RFCs themselves are never changed once published. However, the RFC Editor does maintain an errata list at http://www.rfc-editor.org/errata.html. Furthermore, it is not uncommon for later RFCs to update or even obsolete earlier RFCs. Searching for an RFC number using the RFC Editor's RFC-Search function will provide a list of RFCs that update or obsolete the given RFC.

Many of the RFCs and STDs (Internet standards) on which the information in this chapter is based are given in Table 1.14 for easy reference. The Bibliography provides the title and a URL for each STD (or RFC if no STD is available).

Most of the end-user reference material for the Mozilla browser is contained in its built-in help files. The home page for web developers who are writing software to be run by Mozilla is currently at http://www.mozilla.org/docs/web-developer/. This is, to some extent, a list of links to documentation for various standards, since Mozilla is one of the

TABLE 1.14 RFCs Related to Topics in this Chapter

Topic	RFCs
IP	STD 5/RFC 791
TCP	STD 7/RFC 793
UDP	STD 6/RFC 768
DNS	STD 13/RFCs 1034, 1035
HTTP 1.1	RFC 2616
URI/URL	STD 66/RFC 3986
URN	RFC 2141
https	RFC 2818
MIME	RFCs 2045–2047, 2049, 2077, 4288, 4289
UTF-8	STD 63/RFC 3629
TLS	RFC 2246

more standards-compliant browsers available at this time. References to these standards will be presented in later chapters as we learn about the related technologies.

The various components of Tomcat 5 servers—Service, Connector, Host, etc.—are documented in the Tomcat 5 Server Configuration Reference [APACHE-TOMCAT-5-CONFIG]. Appendix A of the Java Web Services Tutorial [SUN-JWS-TUTORIAL-1.3] provides a full description of the Tomcat web server administration tool covered briefly in Section. 1.7.3. The Tomcat 5 User Guide [APACHE-TOMCAT-5-UG] provides an overview of many Tomcat concepts, including SSL in Chapter. 12. SSL is also covered in Chapter 24 of the JWSDP Tutorial [SUN-JWS-TUTORIAL-1.3].

Exercises

1.1. Using `nslookup` (or some other mechanism), determine IP addresses for three Internet hosts assigned by your instructor.

1.2. Send HTTP requests using `telnet` (or some other mechanism) in order to determine the Server header field value for three Internet hosts assigned by your instructor. You may want to include the header field "Connection: close" in your requests in order to tell the server to immediately close the TCP connection rather than keeping it open (most servers will otherwise keep the connection open for a minute or so, tying up your command prompt). Also note that some systems may not echo the characters you type while executing `telnet`. (Hint: Don't forget that HTTP requests end with a blank line.)

1.3. For three Internet hosts assigned by your instructor, list the names of the header fields that each host returns in response to a HEAD request for the root document (Request-URI of `/`). As in the previous question, you may want to use the "Connection: close" header in your requests.

1.4. For each host assigned by your instructor, give a list of the HTTP methods allowed by the host. (Hint: You may want to try using an OPTIONS HTTP request, although not all web servers support this method.)

1.5. Given the header field

```
accept: text/xml,application/xml,application/xhtml+xml,
  text/html;q=0.9,text/plain;q=0.8,video/x-mng,image/png,
  image/jpeg,image/gif;q=0.2,*/*;q=0.1
```

place the following MIME types in order from high to low preference: `image/png`, `application/pdf`, `text/plain`, `application/xhtml+xml`.

1.6. Explain how a web site could learn something about your browsing habits outside its site from an HTTP request sent to the site by your browser. Assume that the request has only the headers listed in Table 1.5.

1.7. In Java J2SE™ version 1.5, characters are represented using the UTF-16 encoding. Specifically, each `char` value consists of 16 bits representing a UTF-16 *character code unit*. Every character in the Unicode Standard can be represented by either one or two UTF-16 code units, with virtually all characters in widespread use requiring only a single code unit. Give an argument for and one against this design for representing characters in Java versus using 8-bit `char`'s and the UTF-8 encoding.

1.8. Can a web browser load an HTML document from a web server running on a different host if DNS is not operational? Explain.

1.9. Give a complete minimal HTTP GET request corresponding to the URL

```
http://www.ThisIsATest.net:2012/hmm/oh/well?isThis=right#now
```

1.10. Modify your browser preferences to specify a language other than the one that you normally use as your preferred language (for example, if you normally use English, you might specify German as your preferred language). Then browse to `www.google.com` or another web site that returns different documents based on the setting of the Accept-Language HTTP request header field. Print the web page to verify that you successfully modified your language preference.

1.11. The Host field of an HTTP request can contain a port number as well as a host name. Based on the discussion in Section 1.7.1, explain how a web server can determine the port number of the request even if it is not included in the Host field, as long as the HTTP request is transmitted via TCP.

The following questions assume that you have installed JWSDP 1.3 or otherwise set up a Tomcat web server.

1.12. What is the connection timeout (in seconds) for the 8080 Connector to the JWSDP Service of your server?

1.13. Change the connection timeout of your 8080 Connector to 10 seconds, and test your change (for example, by using Telnet to connect to the server and then verifying that the server closes the connection in 10 seconds). Submit the host name and port number of your server to your instructor so that the change can be independently verified.

1.14. Add a virtual host named `www.example.org` to the JWSDP Service of your Tomcat server with Application Base `virtualhost`. Then create a Context within this virtual host with Document Base `docs` and Path `/`. (Hint: Don't forget to create the directory before saving the Context object.) After committing your changes, create a short text file named `test.txt` in your `docs` directory (you should have no other files in this directory). Finally, test that you have created your virtual host properly by using Telnet to visit it using the Request-URI `/test.txt` and an appropriate value for the Host field. You should see the contents of your `test.txt` file. Submit the host name and port number of your server to your instructor so that your work can be verified.

1.15. Explain why in the previous question you needed to use Telnet rather than standard browser navigation in order to test that the virtual host was set up properly.

1.16. Heuristic estimation of cache expiration.

(a) Determine whether or not your Tomcat server returns an Expires header field when the root (`/`) document is requested. If it does return this header field, give the value returned. Repeat for the Last-Modified header field.

(b) Assuming that you are using Mozilla 1.4 as your browser, select the **View|Page Info** menu item. In the pop-up window under the **General** tab, what are the values of the Modified and Expires fields?

(c) The HTTP/1.1 specification (RFC 2616) says that if an Expires header field (or similar information) is not provided by a web server in its response to a request for a resource, then browsers may use a heuristic to determine how long to wait before validating a cached copy of the resource. It also says that if the server provides a Last-Modified time, then this waiting period should be no more than a certain percentage of the difference between the current time and the Last-Modified time. Based on the information gathered in the first two parts of this question, give a reasonable explanation for how the browser produced the Modified and Expires field values displayed in the **View|Page Info** pop-up window. [Hint: Note that

time values in header fields are often given in Greenwich Mean Time (GMT), while the browser generally displays local times.] In particular, estimate the percentage the browser might be using in any heuristic it employs to compute the displayed Expires value.

1.17. This question explores the interaction between browsers and cache. First, open Mozilla 1.4 and select the **Edit** | **Preferences** menu item. In the Preferences window that appears, click on the + to the left of **Advanced** in the Category panel, and select **Cache** under **Advanced**. In the Cache panel make sure that, under "Compare the page in the cache to the page on the network," "When the page is out of date" has been selected. If not, select this button, click OK, and then close and reopen your browser. Next, locate the access log for your server in the server's local file system. View the last three or so lines of this log. Next, make sure that your Tomcat server is running, and navigate your Mozilla 1.4 browser to `http://localhost:8080/`. You should see a JWSDP 1.3 welcome page. Now answer the following questions:

(a) Reexamine the final three or so lines of the server access log. Have they changed? (Yes or no is sufficient for this question.)

(b) Scroll down to the bottom of the web page; then click your mouse in the Location bar of your browser and press the Enter key on the keyboard. This causes your browser to navigate to the `http://localhost:8080/` URL again, as indicated by the fact that the top of the web page is again shown in the browser. Now again reexamine the final three or so lines of the server access log. Have they changed? Explain why this has (or has not) happened.

(c) Now click the browser's Reload button. Once again, reexamine the final three or so lines of the server access log. Have they changed? Again, explain.

1.18. Configure your Tomcat server to deny access to the `localhost` Host of the JWSDP Service from an IP address supplied by your instructor. What status code is returned by your server when it is accessed from a host that is denied access? What message does the Tomcat administration tool produce if you try to deny access to the IP address of the machine running the browser through which you are accessing the tool? Explain why this message is generated.

1.19. Create a self-signed certificate, and use it to set up secure access to the JWSDP Service running on your Tomcat web server. Send the host name and port number of your server to your instructor for verification.

Research and Exploration

1.20. Learn about your educational institution's network history by answering questions such as the following: In what year was your institution first connected to the Internet? What was the original connection type: PhoneNet or something else? What was the original connection speed? Was your institution a member of a regional or other specialized network, such as SURAnet or CSNET? Answer these questions for your institution currently.

1.21. Use the `tracert` (Windows) or `traceroute` (Linux) command to determine the number of hops from your machine to the Internet host(s) assigned by your instructor. Each command can be run by typing the command name followed by a host name. (Note: If you attempt to run `traceroute` on a Linux system and get the message "Command not found," try using the command `whereis traceroute` to locate it. Then prefix the command name with the directory where it is located.) Provide a copy of the output and briefly explain what it means.

1.22. The standard port number for HTTP is 80. What is the standard port number for an initial connection to an FTP server? For a DNS request? Name and give a standard port number for one IANA-registered UDP service and one TCP service not mentioned in this chapter.

1.23. List all of the generic Internet top-level domains.

1.24. Which country is associated with the top-level domain `de`? What is the top-level domain for Bolivia?

1.25. How could you determine whether or not a TCP service is running at port 13 of a given Internet host? Test this for the host(s) assigned by your instructor. What is the standard IANA-registered higher-level protocol associated with this port?

1.26. Refer to RFC 1436, and then write a short example Gopher directory (menu) file. How does the protocol used for communicating with Gopher servers differ from HTTP?

1.27. Give a `mailto`-scheme URL to send e-mail with subject `Test Message` to a user named `Kim` at host `www.example.net`.

1.28. What is one of the MIME types used to represent a sound file? What type of data is represented using the MIME type `model/vrml`?

1.29. Write an Accept-Language header indicating a preference for documents in English, then in French, and finally in German.

1.30. Compare the basic features of HTTP status codes with those of the FTP reply codes given by RFC 640. What is one way in which these codes are similar and one way in which they are different?

1.31. Refer to [IANA-CHARSETS] to find three alternative registered names for the US-ASCII character set. For which character set is latin1 an alias? Name a character set tailored to Danish.

1.32. Research and report on current browser usage statistics. In particular, give approximate percentages of users of Internet Explorer, Firefox, Safari, Opera, and other popular browsers. Cite your source(s). Why should you be aware of browser usage statistics when developing web documents?

1.33. Research and report on current web server usage statistics. In particular, give approximate usage percentages for Apache, IIS, and other popular servers. Cite your source(s).

1.34. A *robot* (also known as a *bot* or *spider* or *crawler*) is a program that accesses web documents automatically rather than in direct response to a user input. For example, the Google search engine uses a program called `googlebot` to automatically crawl the World Wide Web and build its searchable index of Web pages. An indexing robot such as `googlebot` begins by reading some Web document, then reading documents linked to by the initial document, and recursively continuing this process on previously unread documents. Some informal standards have been developed to allow Web site administrators and document authors to request robots not to read certain documents.

(a) Read the first part of Section 4.1 of Appendix B of the HTML 4.01 Recommendation [W3C-HTML-4.01], and explain what you would do in order to request that robots not crawl the documents accessible from your Tomcat web server. (See `http://www.robotstxt.org/wc/norobots.html` for more information on the Robot Exclusion Standard.)

(b) For one or more Web sites as directed by your instructor, list for each the robots (if any) that are explicitly excluded from crawling one or more of the files at that site.

Projects

1.35. Write a simple web browser. Specifically, write a Java program that meets one or more of the following requirements, as specified by your instructor:

(a) Input a URL from the user, and output the complete HTTP response produced by visiting this URL. This is relatively easy if you use the classes `HttpURLConnection` and `URL` in the `java.net` package of the Java API. For example, if `url` is a String variable containing a URL, then the code

```
HttpURLConnection connection =
  (HttpURLConnection)(new URL(url).openConnection());
connection.connect();
```

opens a TCP connection with the server specified in the `url` variable, sends an appropriate HTTP GET request over this connection, and receives back the HTTP response. The methods `getHeaderFieldKey()` and `getHeaderField()` can then be called on the `connection` variable in order to retrieve header field names and values, respectively (and even the status line, on many systems), while the `getInputStream()` method provides access to the body of the HTTP response. See the Java API [SUN-JAVA-API-1.4.2] for details on these and other methods of these classes.

(b) By default, when the `connect()` method of the `HttpURLConnection` class is called, if the initial response from the server is a redirect (first digit of status code is a 3), then the method automatically issues a request for the URL contained in the redirect response. This automatic redirection is applied to subsequent responses until a nonredirect response is finally received. The application calling the method only sees the final response. Modify the original program so that it overrides this default (using the `setInstanceFollowRedirects()` method of `HttpURLConnection`) before connecting with the server. Then modify your program so that, if it receives a redirection response, it outputs the URL to which it is redirected (and only that URL) and then sends a request to that URL. Note that you must create a separate `HttpURLConnection` instance for each request. Your program should repeat this process until a nonredirect response is received; this final response should be printed in its entirety. (Hint: In order to test this program, you'll need a URL that returns a redirect response. If your instructor does not supply such a URL, find any URL that ends with a `/` and visit the URL obtained by removing the trailing `/`. Many servers will respond to such a URL with a redirect status code.)

(c) In order to appreciate some of the HTTP protocol complexities handled by the `HttpURLConnection` class, write your program without using this class or the `openConnection()` method of `URL`. Instead, write your program using the `java.net.Socket` class. Creating an instance of this class using the `Socket(String host, int port)` constructor causes Java to open a TCP/IP connection between your program and the specified host at the specified port. You can then call the `getOutputStream()` method on this Socket instance in order to get a stream to which you can write an HTTP request message. Notice that you will need to extract some of the information needed for this request, such as the Request-URI and the port, from the URL input by the user. If values are not supplied by the URL, then your program must supply appropriate defaults. You will find that the `URL` Java API class has many methods that are useful for extracting the appropriate information. Be sure to *flush* the output stream after writing the request message to it: this moves

the message you wrote from your system's memory to the actual TCP/IP connection. After flushing the output stream, you can call `getInputStream()` on your Socket instance to get a stream through which the server will send the HTTP response. If you include a "Connection: close" header field in your request, then you should be able to obtain the entire response by simply reading from the input stream until the end of the stream is reached (note that this stream contains the entire response, including the headers and the body, while the `HttpURLConnection`'s input stream provides only the body).

1.36. Write a simple web server. Specifically, write a Java program that meets one or more of the following requirements, as specified by your instructor:

(a) Write a server that listens for HTTP requests on port 8080 (or other port specified by your instructor) and accepts one request at a time. This is relatively easy using classes from the `java.net` package of the Java API. In particular, the first line of the code

```
ServerSocket mySocket = new ServerSocket(8080);
Socket yourSocket = mySocket.accept();
```

creates a socket on port 8080 of the machine running this code. The second line then causes the program to listen for a connection to this port. The program will not execute the line of code following the `accept()` call until a connection is made to the port. When the connection is made, `yourSocket` will provide communication with the connecting program. Specifically, the `getInputStream()` method on this object will return a stream that can be read to obtain the HTTP request being sent, and `getOutputStream()` will return a stream to which the server program can write its response. If a valid HTTP/1.1 request for the root (`/`) document is received, then send back a response with status code 200 (OK) and containing a short text document (such as "Success!"). Otherwise, send a response with status code 404 (Not Found) and a short text document giving further information (such as "Failure ..."). In either case, the response should contain at least the header fields Date, Content-Type (with value "text/plain"), and Content-Length. Don't forget to flush your output after you have written the entire response. You can then call `close()` on `yourSocket` followed by a call to `accept()` to await the next connection. Your server can continue iterating in this way until it is killed. Test your program by starting it (first make sure that no other program that uses port 8080, such as Tomcat, is running) and then browsing to `http://localhost:8080/` and `http://localhost:8080/fail`. Visiting the first URL should display your successful response; the second should fail. (Hint: Section 3.3.1 of the HTTP/1.1 specification [RFC-2616] requires that the Date header field value generated by a web server follow a particular format. You can produce a String `dateTime` representing the current date and time in the appropriate format using the code

```
import java.util.Date;
import java.util.Locale;
import java.util.TimeZone;
import java.text.SimpleDateFormat;
import java.text.DateFormatSymbols;
. . .
```

```
SimpleDateFormat formatter =
    new SimpleDateFormat("E, dd MMM yyyy HH:mm:ss zzz",
                         new DateFormatSymbols(Locale.US));
formatter.setTimeZone(TimeZone.getTimeZone("GMT"));
String dateTime = formatter.format(new Date());
```

(b) Modify the server program described in (a) so that if the Request-URI of an HTTP request corresponds to a file within the server's file system, the server will return that file. Otherwise, the server should return a 404 response as before. In particular, if the Request-URI is of the form `/filename.ext` and a file named `filename.ext` exists in the directory from which the server is being run, then this file should be returned in the response. You may assume for simplicity that every requested file is character-oriented (rather than the more general case of treating a file as a stream of bytes). The Date, Content-Type, and Content-Length header fields should all be set appropriately in the response. The static method `getFileNameMap()` of the `java.net.URLConnection` class can be used to get a `java.net.FileNameMap`, which in turn provides a method `getContentTypeFor()` that maps a filename to a corresponding MIME type based on its extension `ext`. The resulting MIME type is appropriate for use as the value of a Content-Type header field. Test your program by creating small text files named `test.txt` and `test.xml` in the directory from which your server runs and then browsing to these files using URLs such as `http://localhost:8080/test.xml`. The Type field of Mozilla's **View|Page Info** pop-up window will display the MIME type of the document.

(c) Modify your server to produce an access log in common log format. Output the IP address of the client rather than the host name, and output a hypen (-) for the user-name field. The date and time can be produced using the formatter

```
SimpleDateFormat formatter =
    new SimpleDateFormat("dd/MMM/yyyy:HH:mm:ss Z",
                         new DateFormatSymbols(Locale.US));
```

Write the log to a file named `access.log`. Be sure to flush the buffer after each output to the log file so that each access is immediately visible in the file.