Chapter 2: Application Layer

In this chapter we study both the conceptual and implementation aspects of network applications. We begin by defining key application-layer concepts, including application-layer protocols, clients and servers, processes, sockets, and transport-layer interfaces. We also discuss the services that an application needs, and survey the services that are provided by the Internet transport-layer protocols. We then examine several applications in detail. We begin with HTTP and the Web, discussing nonpersistent and persistent connections, HTTP message formats, authentication and cookies, and Web caching. We then cover FTP, which uses both control and data connections. We then explore Internet e-mail in detail, covering the protocols SMTP, POP3, IMAP as well as the MIME standard for message formats. We also examine DNS, which is an application-layer protocol that provides services to other applications. The last three sections of this chapter provide an introduction to application development and socket programming.

Online Book

2.1: Principles of Application Layer Protocols

Network applications are the *raisons d'etre* of a computer network. If we couldn't conceive of any useful applications, there wouldn't be any need to design networking protocols to support them. But over the past thirty years, many people have devised numerous ingenious and wonderful networking applications. These applications include the classic text-based applications that became popular in the 1980s, including remote access to computers, electronic mail, file transfers, newsgroups, and chat. But they also include more recently conceived multimedia applications, such as the World Wide Web, Internet telephony, video conferencing, and audio and video on demand.

Although network applications are diverse and have many interacting components, software is almost always at their core. Recall from Section 1.2 that a network application's software is distributed among two or more end systems (that is, host computers). For example, with the Web there are two pieces of software that communicate with each other: the browser software in the user's host (PC, Mac, or workstation), and the Web server software in the Web server. With Telnet, there are again two pieces of software in two hosts: software in the local host and software in the remote host. With multiparty video conferencing, there is a software piece in each host that participates in the conference.

In the jargon of operating systems, it is not actually software pieces (that is, programs) that are communicating but in truth processes that are communicating. A process can be thought of as a program that is *running* within an end system. When communicating processes are running on the same end system, they communicate with each other using interprocess communication. The rules for interprocess communication are governed by the end system's operating system. But in this book we are not interested in how processes on the same host communicate, but instead in how processes running on *different* end systems (with potentially different operating systems) communicate. Processes on two different end systems communicate with each other by exchanging messages across the computer network. A sending process creates and sends messages into the network; a receiving process receives these messages and possibly responds by sending messages back, see Figure 2.1. Networking applications have application-layer protocols that define the format and order of the messages exchanged between processes, as well as define the actions taken on the transmission or receipt of a message.

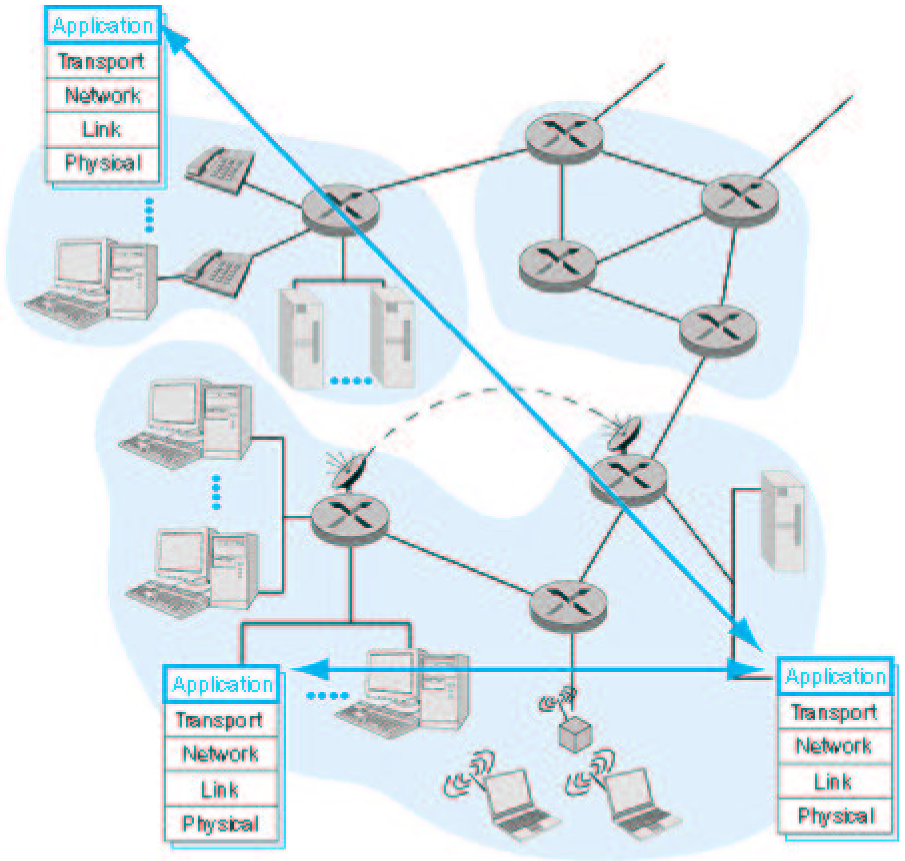

Figure 2.1: Communicating applications

The application layer is a particularly good place to start our study of protocols. It's familiar ground. We're acquainted with many of the applications that rely on the protocols we will study. It will give us a good feel for what protocols are all about and will introduce us to many of the same issues that we'll see again when we study transport, network, and data-link layer protocols.

2.1.1: Application-Layer Protocols

It is important to distinguish between network applications and application-layer protocols. An application-layer protocol is only one piece (albeit, a big piece) of a network application. Let's look at a couple of examples. The Web is a network application that allows users to obtain "documents" from Web servers on demand. The Web application consists of many components, including a standard for document formats (that is, HTML), Web browsers (for example, Netscape Navigator and Microsoft Internet Explorer), Web servers (for example, Apache, Microsoft, and Netscape servers), and an application-layer protocol. The Web's application-layer protocol, HTTP (the HyperText Transfer Protocol [RFC 2616]), defines how messages are passed between browser and Web server. Thus, HTTP is only one piece of the Web application. As another example, consider the Internet electronic mail application. Internet electronic mail also has many components, including mail servers that house user mailboxes, mail readers that allow users to read and create messages, a standard for defining the structure of an e-mail message and application-layer protocols that define how messages are passed between servers, how messages are passed between servers and mail readers, and how the contents of certain parts of the mail message (for example, a mail message header) are to be interpreted. The principal application-layer protocol for electronic mail is SMTP (Simple Mail Transfer Protocol [RFC 821]). Thus, SMTP is only one piece (albeit, a big piece) of the e-mail application.
As noted above, an application-layer protocol defines how an application's processes, running on different end systems, pass messages to each other. In particular, an application-layer protocol defines:

- the types of messages exchanged, for example, request messages and response messages
- the syntax of the various message types, such as the fields in the message and how the fields are delineated
- the semantics of the fields, that is, the meaning of the information in the fields
- rules for determining when and how a process sends messages and responds to messages

Some application-layer protocols are specified in RFCs and are therefore in the public domain. For example, HTTP is available as an RFC. If a browser developer follows the rules of the HTTP RFC, the browser will be able to retrieve Web pages from any Web server that has also followed the rules of the HTTP RFC. Many other application-layer protocols are proprietary and intentionally not available in the public domain. For example, many of the existing Internet phone products use proprietary application-layer protocols.
Clients and Servers
A network application protocol typically has two parts or "sides," a client

side and a server side. See Figure 2.2. The client side in one end system communicates with the server side in another end system. For example, a Web browser implements the client side of HTTP, and a Web server implements the server side of HTTP. In e-mail, the sending mail server implements the client side of SMTP, and the receiving mail server implements the server side of SMTP.

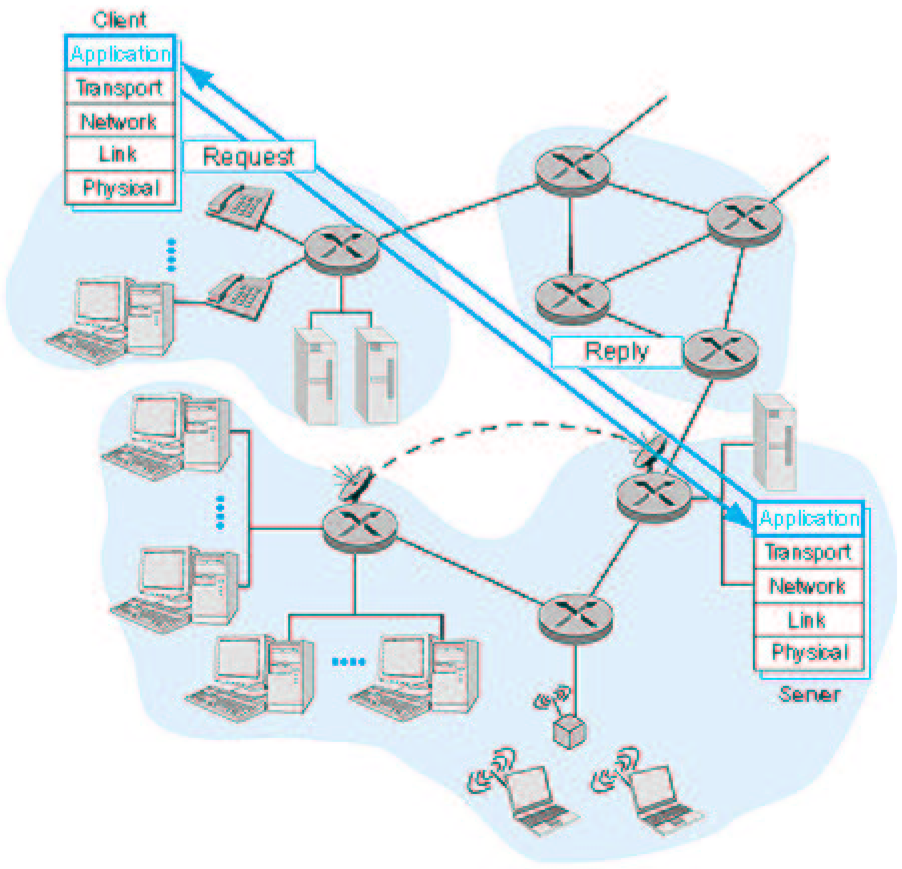

Figure 2.2: Client/server interaction

For many applications, a host will implement both the client and server sides of an application. For example, consider a Telnet session between Hosts A and B. (Recall that Telnet is a popular remote login application.) If Host A initiates the Telnet session (so that a user at Host A is logging onto Host B), then Host A runs the client side of the application and Host B runs the server side. On the other hand, if Host B initiates the Telnet session, then Host B runs the client side of the application. FTP, used for transferring files between two hosts, provides another example. When an FTP session exists between two hosts, then either host can transfer a file to the other host during the session. However, as is the case for almost all network applications, *the host that initiates the session is labeled the client.* Furthermore, a host can actually act as both a client and a server at the same time for a given application. For example, a mail server host runs the client side of SMTP (for sending mail) as well as the server side of SMTP (for receiving mail).

Processes Communicating Across a Network

As noted above, an application involves two processes in two different hosts communicating with each other over a network. (Actually, a multicast application can involve communication among more than two hosts. We shall address this issue in Chapter 4.) The two processes communicate with each other by sending and receiving messages through their *sockets.* A process's socket can be thought of as the process's door: A process sends messages into, and receives message from, the network through its socket. When a process wants to send a message to another process on another host, it shoves the message out its door. The process assumes that there is a transportation infrastructure on the other side of the door that will transport the message to the door of the destination process.

Figure 2.3 illustrates socket communication between two processes that communicate over the Internet. (Figure 2.3 assumes that the underlying transport protocol is TCP, although the UDP protocol could be used as well in the Internet.) As shown in this figure, a socket is the interface between the application layer and the transport layer within a host. It is also referred to as the API (application programmers' interface) between the application and the network, since the socket is the programming interface with which networked applications are built in the Internet. The application developer has control of everything on the application-layer side of the socket but has little control of the transport-layer side of the socket. The only control that the application developer has on the transport-layer side is (1) the choice of transport protocol and (2) perhaps the ability to fix a few transport-layer parameters such as maximum buffer and maximum segment sizes. Once the application developer chooses a transport protocol (if a choice is available), the application is built using the transport layer of the services offered by that protocol. We will explore sockets in some detail in Sections 2.6 and 2.7.

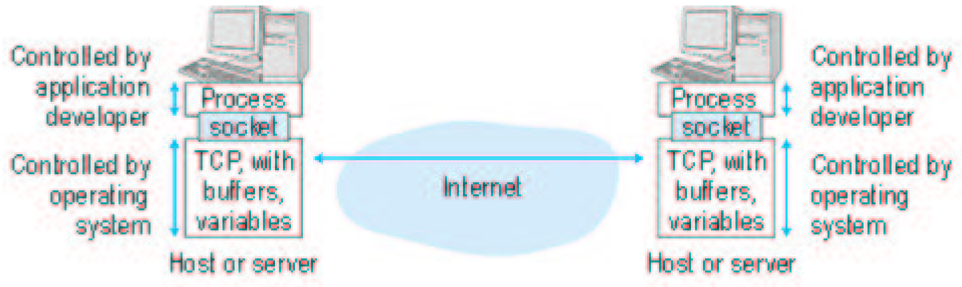
Controlled by application developer
Process
socket
Controlled by operating system
TCP, with buffers, variables
Host or server
Internet
Process
socket
TCP, with buffers, variables
Controlled by application developer
Controlled by operating system
Host or server

Figure 2.3: Application processes, sockets, and the underlying transport protocol

Addressing Processes

In order for a process on one host to send a message to a process on another host, the sending process must identify the receiving process. To identify the receiving process, one must typically specify two pieces of information: (1) the name or address of the host machine, and (2) an identifier that specifies the identity of the receiving process on the destination host.

Let us first consider host addresses. In Internet applications, the destination host is specified by its IP address. We will discuss IP addresses in great detail in Chapter 4. For now, it suffices to know that the IP address is a 32-

bit quantity that *uniquely* identifies the end system (more precisely, it uniquely identifies the interface that connects that host to the Internet). Since the IP address of any end system connected to the public Internet must be *globally* unique, the assignment of IP addresses must be carefully managed, as discussed in Section 4.4. ATM networks have a different addressing standard. The ITU-T has specified telephone number-like addresses, called E.164 addresses [ITU 1997], for use in public ATM networks.

In addition to knowing the address of the end system to which a message is destined, a sending application must also specify information that will allow the receiving end system to direct the message to the appropriate process on that system. A receive-side port number serves this purpose in the Internet. Popular application-layer protocols have been assigned specific port numbers. For example, a Web server process (that uses the HTTP protocol) is identified by port number 80. A mail server (using the SMTP) protocol is identified by port number 25. A list of well-known port numbers for all Internet standard protocols can be found in RFC 1700. When a developer creates a new network application, the application must be assigned a new port number.

User Agents

Before we begin a more detailed study of application-layer protocols, it is useful to discuss the notion of a user agent. The user agent is an interface between the user and the network application. For example, consider the Web. For this application, the user agent is a browser such as Netscape Navigator or Microsoft Internet Explorer. The browser allows a user to view Web pages, to navigate in the Web, to provide input to forms, to interact with Java applets, and so on. The browser also implements the client side of the HTTP protocol. Thus, when activated, the browser is a process that, along with providing an interface to the user, sends/receives messages via a socket. As another example, consider the electronic mail application. In this case, the user agent is a "mail reader" that allows a user to compose and read messages. Many companies market mail readers (for example, Eudora, Netscape Messenger, Microsoft Outlook) with a graphical user interface that can run on PCs, Macs, and workstations. Mail readers running on PCs also implement the client side of application-layer protocols; typically they implement the client side of SMTP for sending mail and the client side of a mail retrieval protocol, such as POP3 or IMAP (see Section 2.4), for receiving mail.

2.1.2: What Services Does an Application Need?

Recall that a socket is the interface between the application process and the transport protocol. The application at the sending side sends messages through the door. At the other side of the door, the transport protocol has the responsibility of moving the messages across the network to the door at the receiving process. Many networks, including the Internet, provide more than one transport protocol. When you develop an application, you must choose one of the available transport protocols. How do you make this choice? Most likely, you will study the services that are provided by the available transport protocols, and you will pick the protocol with the services that best match the needs of your application. The situation is similar to choosing either train or airplane transport for travel between two cities (say, New York and Boston). You have to choose one or the other, and each transport mode offers different services. (For example, the train offers downtown pick up and drop off, whereas the plane offers shorter transport time.)

What services might a network application need from a transport protocol? We can broadly classify an application's service requirements along three dimensions: data loss, bandwidth, and timing.

Data Loss

Some applications, such as electronic mail, file transfer, remote host access, Web document transfers, and financial applications require fully reliable data transfer, that is, no data loss. In particular, a loss of file data, or data in a financial transaction, can have devastating consequences (in the latter case, for either the bank or the customer!). Other loss-tolerant applications, most notably multimedia applications such as real-time audio/video or stored audio/video, can tolerate some amount of data loss. In these latter applications, lost data might result in a small glitch in the played-out audio/video--not a crucial impairment. The effects of such loss on application quality, and actual amount of tolerable packet loss, will depend strongly on the application and the coding scheme used.

Bandwidth

Some applications must be able to transmit data at a certain rate in order to be effective. For example, if an Internet telephony application encodes voice at 32 Kbps, then it must be able to send data into the network and have data delivered to the receiving application at this rate. If this amount of bandwidth is not available, the application needs to encode at a different rate (and receive enough bandwidth to sustain this different coding rate) or it should give up, since receiving half of the needed bandwidth is of no use to such a bandwidth-sensitive application. Many current multimedia applications are bandwidth sensitive, but future multimedia applications may use adaptive coding techniques to encode at a rate that matches the currently available bandwidth. While bandwidth-sensitive applications require a given amount of bandwidth, elastic applications can make use of as much or as little bandwidth as happens to be available. Electronic mail, file transfer, remote access, and Web transfers are all elastic applications. Of course, the more bandwidth, the better. There's an adage that says that one cannot be too rich, too thin, or have too much bandwidth.

Timing

The final service requirement is that of timing. Interactive real-time applications, such as Internet telephony, virtual environments, teleconferencing, and multiplayer games require tight timing constraints on data delivery in order to be effective. For example, many of these applications require that end-to-end delays be on the order of a few

hundred milliseconds or less. (See Chapter 6, [Gauthier 1999; Ramjee 1994].) Long delays in Internet telephony, for example, tend to result in unnatural pauses in the conversation; in a multiplayer game or virtual interactive environment, a long delay between taking an action and seeing the response from the environment (for example, from another player at the end of an end-to-end connection) makes the application feel less realistic. For non-real-time applications, lower delay is always preferable to higher delay, but no tight constraint is placed on the end-to-end delays.
Figure 2.4 summarizes the reliability, bandwidth, and timing requirements of some popular and emerging Internet applications. Figure 2.4 outlines only a few of the key requirements of a few of the more popular Internet applications. Our goal here is not to provide a complete classification, but simply to identify some of the most important axes along which network application requirements can be classified.

Application	**Data loss**	**Bandwidth**	**Time sensitive**
File transfer	No Loss	Elastic	No
E-mail	No Loss	Elastic	No
Web Documents	No Loss	Elastic (few Kbps)	No
Real-time Audio/Video	Loss-tolerant	Audio: Few Kbps - 1Mb Video: 10Kb - 5 Mb	Yes: 100s of msec
Stored Audio/Video	Loss-tolerant	Same as Above	Yes: Few Seconds
Interactive games	Loss-tolerant	Few Kbps - 10Kb	Yes: 100s of msec
Financial Applications	No Loss	Elastic	

Yes and No

Figure 2.4: Requirements of selected network applications

2.1.3: Services Provided by the Internet Transport Protocols

The Internet (and, more generally, TCP/IP networks) makes available two transport protocols to applications, namely, UDP (User Datagram Protocol) and TCP (Transmission Control Protocol). When a developer creates a new application for the Internet, one of the first decisions that the developer must make is whether to use UDP or TCP. Each of these protocols offers a different service model to the invoking applications.

TCP Services

The TCP service model includes a connection-oriented service and a reliable data transfer service. When an application invokes TCP for its transport protocol, the application receives both of these services from TCP.

Connection-oriented service: TCP has the client and server exchange transport-layer control information with each other *before* the application-level messages begin to flow. This so-called handshaking procedure alerts the client and server, allowing them to prepare for an onslaught of packets. After the handshaking phase, a TCP connection is said to exist between the sockets of the two processes. The connection is a full-duplex connection in that the two processes can send messages to each other over the connection at the same time. When the application is finished sending messages, it must tear down the connection. The service is referred to as a "connection-oriented" service rather than a "connection" service (or a "virtual circuit" service), because the two processes are connected in a very loose manner. In Chapter 3 we will discuss connection-oriented service in detail and examine how it is implemented.

Reliable transport service: The communicating processes can rely on TCP to deliver all data sent without error and in the proper order. When one side of the application passes a stream of bytes into a socket, it can count on TCP to de -liver the same stream of data to the receiving socket, with no missing or duplicate bytes.

TCP also includes a congestion-control mechanism, a service for the general welfare of the Internet rather than for the direct benefit of the communicating processes. The TCP congestion-control mechanism throttles a process (client or server) when the network is congested. In particular, as we will see in Chapter 3, TCP congestion control attempts to limit each TCP connection to its fair share of network bandwidth.

The throttling of the transmission rate can have a very harmful effect on real-time audio and video applications that have a minimum required bandwidth constraint. Moreover, real-time applications are loss-tolerant and do not need a fully reliable transport service. For these reasons, developers of real-time applications usually run their applications over UDP rather than TCP.

Having outlined the services provided by TCP, let us say a few words about the services that TCP does *not* provide. First, TCP does not guarantee a minimum transmission rate. In particular, a sending process is not permitted to transmit at any rate it pleases; instead the sending rate is regulated by TCP congestion control, which may force the sender to send at a low average rate. Second, TCP does not provide any delay guarantees. In particular, when a sending process passes data into a TCP socket, the data will eventually arrive at the receiving socket, but TCP guarantees absolutely no limit on how long the data may take to get there. As many of us have experienced with the world-wide wait, one can sometimes wait tens of seconds or even minutes for TCP to deliver a message (containing, for example, an HTML file) from Web server to Web client. In summary, TCP guarantees delivery of all data, but provides no guarantees on the rate of delivery or on the delays experienced.

UDP Services

UDP is a no-frills, lightweight transport protocol with a minimalist service model. UDP is connectionless, so there is no handshaking before the two processes start to communicate. UDP provides an unreliable data transfer service; that is, when a process sends a message into a UDP socket, UDP provides *no* guarantee that the message will ever reach the receiving socket. Furthermore, messages that do arrive to the receiving socket may arrive out of order.

On the other hand, UDP does not include a congestion-control mechanism, so a sending process can pump data into a UDP socket at any rate it pleases. Although all the data may not make it to the receiving socket, a large fraction of the data may arrive. Developers of real-time applications often choose to run their applications over UDP. Similar to TCP, UDP provides no guarantee on delay.

Figure 2.5 indicates the transport protocols used by some popular Internet applications. We see that e-mail, remote terminal access, the Web, and file transfer all use TCP. These applications have chosen TCP primarily because TCP provides the reliable data transfer service, guaranteeing that all data will eventually get to its destination. We also see that Internet telephony typically runs over UDP. Each side of an Internet phone application needs to send data across the network at some minimum rate (see Figure 2.4); this is more likely to be possible with UDP than with TCP. Also, Internet phone applications are loss-tolerant, so they do not need the reliable data transfer service provided by TCP.

Applications	**Application-layer Protocol**	**Underlying Transport Protocol**
Electronic Mail	SMTP [RFC 821]	TCP
Remote Terminal Access	Telnet [RFC 854]	

		TCP
Web	HTTP [RFC 2068]	TCP
File Transfer	FTP [RFC 959]	TCP
Remote File Server	NFS [McKusik 1996]	UDP or TCP
Streaming Multimedia	Proprietary (for example, Real Networks)	UDP or TCP
Internet Telephony	Proprietary (for example, Vocaltec)	Typically UDP

Figure 2.5: Popular Internet applications, their application-layer protocols, and their underlying transport protocols

As noted earlier, neither TCP nor UDP offer timing guarantees. Does this mean that time-sensitive applications cannot run in today's Internet? The answer is clearly no--the Internet has been hosting time-sensitive applications for many years. These applications often work pretty well because they have been designed to cope, to the greatest extent possible, with this lack of guarantee. We will investigate several of these design tricks in Chapter 6. Nevertheless, clever design has its limitations when delay is excessive, as is often the case in the public Internet. In summary, today's Internet can often provide satisfactory service to time-sensitive applications, but it cannot provide any timing or bandwidth guarantees. In Chapter 6, we will also discuss emerging Internet service models that provide new services, including guaranteed delay service for time-sensitive applications.

2.1.4: Network Applications Covered in This Book

New public domain and proprietary Internet applications are being developed everyday. Rather than treating a large number of Internet applications in an encyclopedic manner, we have chosen to focus on a small number of important and popular applications. In this chapter we discuss in some detail four popular applications: the Web, file transfer, electronic mail, and directory service. We first discuss the Web, not only because the Web is an enormously popular application, but also because its application-layer protocol, HTTP, is relatively simple and illustrates many key principles of network protocols. We then discuss file transfer, as it provides a nice contrast to HTTP and enables us to highlight some additional principles. We also discuss electronic mail, the Internet's first highly popular application. We shall see that modern electronic mail makes use of not one, but of several, application-layer protocols. The Web, file transfer, and electronic mail have common service requirements: They all require a reliable transfer service, none of them have special timing requirements, and they all welcome an elastic bandwidth offering. The services provided by TCP are largely sufficient for these three applications. The fourth application, Domain Name System (DNS), provides a directory service for the Internet. Most users do not interact with DNS directly; instead, users invoke DNS indirectly through other applications (including the Web, file transfer, and electronic mail). DNS illustrates nicely how a distributed database can be implemented in the Internet. None of the four applications discussed in this chapter are particularly time sensitive; we will defer our discussion of such time-sensitive applications until Chapter 6.

Online Book

2.2: The World Wide Web: HTTP

Until the 1990s the Internet was used primarily by researchers, academics, and university students to log-in to remote hosts, to transfer files from local hosts to remote hosts and vice versa, to receive and send news, and to receive and send electronic mail. Although these applications were (and continue to be) extremely useful, the Internet was essentially unknown outside the academic and research communities. Then, in the early 1990s, the Internet's killer application arrived on the scene--the World Wide Web [Berners-Lee 1994]. The Web is the Internet application that caught the general public's eye. It is dramatically changing how people interact inside and outside their work environments. It has spawned thousands of start up companies. It has elevated the Internet from just one of many data networks (including online networks such as Prodigy, America OnLine, and Compuserve, national data networks such as Minitel/Transpac in France, private X.25, and frame relay networks) to essentially the one and only data network.

History is sprinkled with the arrival of electronic communication technologies that have had major societal impacts. The first such technology was the telephone, invented in the 1870s. The telephone allowed two persons to orally communicate in real-time without being in the same physical location. It had a major impact on society--both good and bad. The next electronic communication technology was broadcast radio/television, which arrived in the 1920s and 1930s. Broadcast radio/ television allowed people to receive vast quantities of audio and video information. It also had a major impact on society--both good and bad. The third major communication technology that has changed the way people live and work is the Web. Perhaps what appeals the most to users about the Web is that it operates *on demand.* Users receive what they want, when they want it. This is unlike broadcast radio and television, which force users to "tune in" when the content provider makes the content available. In addition to being on demand, the Web has many other wonderful features that people love and cherish. It is enormously easy for any individual to make any information available over the Web; everyone can become a publisher at extremely low cost. Hyperlinks and search engines help us navigate through an ocean of Web sites. Graphics and animated graphics stimulate our senses. Forms, Java applets, Active X components, as well as many other devices enable us to interact with pages and sites. And more and more, the Web provides a menu interface to vast quantities of audio and video material stored in the Internet, audio and video that can be accessed on demand.

Case History

The Browser War

In April 1994, Marc Andreesen, a computer scientist who had earlier led the development of the Mosaic browser at the University of Illinois Urbana-Champagne, and Jim Clark, founder of Silicon Graphics and a former Stanford professor, founded Netscape Communication Corporation. Netscape recruited much of the original Mosaic team from Illinois and released its Beta version of Navigator 1.0 in October of 1994. In the years to come, Netscape would make significant enhancements to its browsers, develop Web servers, commerce servers, mail servers, news servers, proxy servers, mail readers, and many other application-layer software products. They were certainly one of the most innovative and successful Internet companies in the mid 1990s. In January 1995, Jim Barksdale became the CEO of Netscape, and in August of 1995 Netscape went public with much fanfare.

Microsoft, originally slow to get into the Internet business, introduced its own browser, Internet Explorer 1.0, in August 1995. Internet Explorer 1.0 was clunky and slow, but Microsoft invested heavily in its development, and by late 1997 Microsoft and Netscape were running head-to-head in the browser war. On June 11, 1997, Netscape released the version 4.0 of its browser, and on September 30, Microsoft released its version 4.0. There was no consensus at that time as to which of the two browsers was better, and Microsoft, with its monopoly on the Windows operating system, continued to gain more market share. In 1997, Netscape made several major mistakes, such as failing to recognize the importance of its Web site as a potential portal site, and launching a major development effort for an all-Java browser (when Java was not quite ripe for the task) [Cusumano 1998]. Throughout 1998, Netscape continued to lose market share for its browser and other products, and in late 1998 it was acquired by America Online. Marc Andreesen and most of the original Netscape people have since left.

2.2.1: Overview of HTTP

The Hypertext Transfer Protocol (HTTP), the Web's application-layer protocol, is at the heart of the Web. HTTP is implemented in two programs: a client program and a server program. The client program and server program, executing on different end systems, talk to each other by exchanging HTTP messages. HTTP defines the structure of these messages and how the client and server exchange the messages. Before

explaining HTTP in detail, it is useful to review some Web terminology.
A Web page (also called a document) consists of objects. An object is simply a file--such as an HTML file, a JPEG image, a GIF image, a Java applet, an audio clip, and so on--that is addressable by a single URL. Most Web pages consist of a base HTML file and several referenced objects. For example, if a Web page contains HTML text and five JPEG images, then the Web page has six objects: the base HTML file plus the five images. The base HTML file references the other objects in the page with the objects' URLs. Each URL has two components: the host name of the server that houses the object and the object's path name. For example, the URL
www.someSchool.edu/someDepartment/picture.gif
has www.someSchool.edu for a host name and /someDepartment/picture.gif for a path name. A browser is a user agent for the Web; it displays the requested Web page and provides numerous navigational and configuration features. Web browsers also implement the client side of HTTP. Thus, in the context of the Web, we will interchangeably use the words "browser" and "client." Popular Web browsers include Netscape Communicator and Microsoft Internet Explorer. A Web server houses Web objects, each addressable by a URL. Web servers also implement the server side of HTTP. Popular Web servers include Apache, Microsoft Internet Information Server, and the Netscape Enterprise Server. (Netcraft provides a nice survey of Web server penetration [Netcraft 2000].)
HTTP defines how Web clients (that is, browsers) request Web pages from servers (that is, Web servers) and how servers transfer Web pages to clients. We discuss the interaction between client and server in detail below, but the general idea is illustrated in Figure 2.6. When a user requests a Web page (for example, clicks on a hyperlink), the browser sends HTTP request messages for the objects in the page to the server. The server receives the requests and responds with HTTP response messages that contain the objects. Through 1997 essentially all browsers and Web servers implemented version HTTP/1.0, which is defined in RFC 1945. Beginning in 1998, some Web servers and browsers began to implement version HTTP/1.1, which is defined in RFC 2616. HTTP/1.1 is backward compatible with HTTP/1.0; a Web server running 1.1 can "talk" with a browser running 1.0, and a browser running 1.1 can "talk" with a server running 1.0.

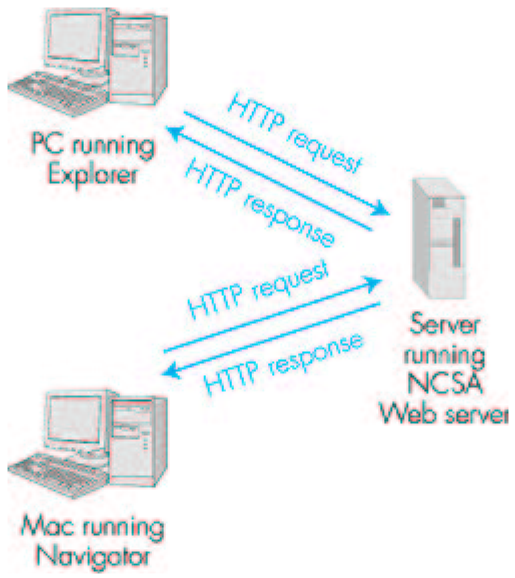

Figure 2.6: HTTP request-response behavior

Both HTTP/1.0 and HTTP/1.1 use TCP as their underlying transport protocol (rather than running on top of UDP). The HTTP client first initiates a TCP connection with the server. Once the connection is established, the browser and the server processes access TCP through their socket interfaces. As described in Section 2.1, on the client side the socket interface is the "door" between the client process and the TCP connection; on the server side it is the "door" between the server process and the TCP connection. The client sends HTTP request messages into its socket interface and receives HTTP response messages from its socket interface. Similarly, the HTTP server receives request messages from its socket interface and sends response messages into the socket interface. Once the client sends a message into its socket interface, the message is "out of the client's hands" and is "in the hands of TCP." Recall from Section 2.1 that TCP provides a reliable data transfer service to HTTP. This implies that each HTTP request message emitted by a client process eventually arrives intact at the server; similarly, each HTTP response message emitted by the server process eventually arrives intact at the client. Here we see one of the great advantages of a layered architecture--HTTP need not worry about lost data, or the details of how TCP recovers from loss or reordering of data within the network. That is the job of TCP and the protocols in the lower layers of the protocol stack.

TCP also employs a congestion control mechanism that we shall discuss in detail in Chapter 3. We mention here only that this mechanism forces each new TCP connection to initially transmit data at a relatively slow rate, but then allows each connection to ramp up to a relatively high rate when the network is uncongested. The initial slow-transmission phase is referred to as slow start.

It is important to note that the server sends requested files to clients without storing any state information about the client. If a particular client asks for the same object twice in a period of a few seconds, the server does not respond by saying that it just served the object to the client; instead, the server resends the object, as it has completely forgotten what it did earlier. Because an HTTP server maintains no information about the clients, HTTP is said to be a stateless protocol.

2.2.2: Nonpersistent and Persistent Connections

HTTP can use both nonpersistent connections and persistent connections. HTTP/ 1.0 uses nonpersistent connections. Conversely, use of persistent connections is the default mode for HTTP/1.1.

Nonpersistent Connections

Let us walk through the steps of transferring a Web page from server to client for the case of nonpersistent connections. Suppose the page consists of a base HTML file and 10 JPEG images, and that all 11 of these objects reside on the same server. Suppose the URL for the base HTML file is www.someSchool.edu/someDepartment/home.index.

Here is what happens:

1. The HTTP client initiates a TCP connection to the server www.someSchool. edu. Port number 80 is used as the default port number at which the HTTP server will be listening for HTTP clients that want to retrieve documents using HTTP.

2. The HTTP client sends a HTTP request message to the server via the socket associated with the TCP connection that was established in step 1. The request message includes the path name /someDepartment/home.index. (We will discuss the HTTP messages in some detail below.)

3. The HTTP server receives the request message via the socket associated with the connection that was established in step 1, retrieves the object /someDepartment/home.index from its storage (RAM or disk), encapsulates the object in an HTTP response message, and sends the response message to the client via the socket.

4. The HTTP server tells TCP to close the TCP connection. (But TCP doesn’t actually terminate the connection until the client has received the response message intact.)

5. The HTTP client receives the response message. The TCP connection terminates. The message indicates that the encapsulated object is an HTML file. The client extracts the file from the response message, parses the HTML file, and finds references to the 10 JPEG objects.

6. The first four steps are then repeated for each of the referenced JPEG objects.

As the browser receives the Web page, it displays the page to the user.

Two different browsers may interpret (that is, display to the user) a Web page in somewhat different ways. HTTP has nothing to do with how a Web page is interpreted by a client. The HTTP specifications ([RFC 1945] and [RFC 2616]) only define the communication protocol between the client HTTP program and the server HTTP program.

The steps above use nonpersistent connections because each TCP connection is closed after the server sends the object--the connection does not persist for other objects. Note that each TCP connection transports exactly one request message and one response message. Thus, in this example, when a user requests the Web page, 11 TCP connections are generated.

In the steps described above, we were intentionally vague about whether the client obtains the 10 JPEGs over 10 serial TCP connections, or whether some of the JPEGs are obtained over parallel TCP connections. Indeed, users can configure modern browsers to control the degree of parallelism. In their default modes, most browsers open 5 to 10 parallel TCP connections, and each of these connections handles one request-response transaction. If the user prefers, the maximum number of parallel connections can be set to 1, in which case the 10 connections are established serially. As we shall see in the next chapter, the use of parallel connections shortens the response time.

Before continuing, let’s do a back-of-the-envelope calculation to estimate the amount of time from a client requesting the base HTML file until the file is received by the client. To this end we define the round-trip time (RTT), which is the time it takes for a small packet to travel from client to server and then back to the client. The RTT includes packet-propagation delays, packet-queuing delays in intermediate routers and switches, and packet-processing delays. (These delays were discussed in Section 1.6.) Now consider what happens when a user clicks on a hyperlink. This causes the browser to initiate a TCP connection between the browser and the Web server; this involves a "three-way handshake"--the client sends a small TCP message to the server, the server acknowledges and responds with a small message, and, finally, the client acknowledges back to the server. One RTT elapses after the first two parts of the three-way handshake. After completing the first two parts of the handshake, the client sends the HTTP request message into the TCP connection, and TCP "piggybacks" the last acknowledgment (the third part of the three-way handshake) onto the request message. Once the request message arrives at the server, the server sends the HTML file into the TCP connection. This HTTP request/response eats up another RTT. Thus, roughly, the total response time is two RTTs plus the transmission time at the server of the HTML file.

Persistent Connections

Nonpersistent connections have some shortcomings. First, a brand new connection must be established and maintained for *each requested object.* For each of these connections, TCP buffers must be allocated and TCP variables must be kept in both the client and server. This can place a

serious burden on the Web server, which may be serving requests from hundreds of different clients simultaneously. Second, as we just described, each object suffers two RTTs--one RTT to establish the TCP connection and one RTT to request and receive an object. Finally, each object suffers from TCP slow start because every TCP connection begins with a TCP slow-start phase. The impact of RTT and slow-start delays can be partially mitigated, however, by the use of parallel TCP connections.
With persistent connections, the server leaves the TCP connection open after sending a response. Subsequent requests and responses between the same client and server can be sent over the same connection. In particular, an entire Web page (in the example above, the base HTML file and the 10 images) can be sent over a single persistent TCP connection; moreover, multiple Web pages residing on the same server can be sent over a single persistent TCP connection. Typically, the HTTP server closes a connection when it isn't used for a certain time (the timeout interval), which is often configurable. There are two versions of persistent connections: without pipelining and with pipelining. For the version without pipelining, the client issues a new request only when the previous response has been received. In this case, each of the referenced objects (the 10 images in the example above) experiences one RTT in order to request and receive the object. Although this is an improvement over nonpersistent's two RTTs, the RTT delay can be further reduced with pipelining. Another disadvantage of no pipelining is that after the server sends an object over the persistent TCP connection, the connection hangs--does nothing--while it waits for another request to arrive. This hanging wastes server resources. The default mode of HTTP/1.1 uses persistent connections with pipelining. In this case, the HTTP client issues a request as soon as it encounters a reference. Thus the HTTP client can make back-to-back requests for the referenced objects. When the server receives the requests, it can send the objects back to back. If all the requests are sent back to back and all the responses are sent back to back, then only one RTT is expended for all the referenced objects (rather than one RTT per referenced object when pipelining isn't used). Furthermore, the pipelined TCP connection hangs for a smaller fraction of time. In addition to reducing RTT delays, persistent connections (with or without pipelining) have a smaller slow-start delay than nonpersistent connections. The reason is that after sending the first object, the persistent server does not have to send the next object at the initial slow rate since it continues to use the same TCP connection. Instead, the server can pick up at the rate where the first object left off. We shall quantitatively compare the performance of nonpersistent and persistent connections in the homework problems of Chapter 3. The interested reader is also encouraged to see [Heidemann 1997; Nielsen 1997].

2.2.3: HTTP Message Format

The HTTP specifications 1.0 ([RFC 1945] and 1.1 [RFC 2616]) define the HTTP message formats. There are two types of HTTP messages, request messages and response messages, both of which are discussed below.

HTTP Request Message

Below we provide a typical HTTP request message:

```
GET /somedir/page.html HTTP/1.1
Host: www.someschool.edu
Connection: close
User-agent: Mozilla/4.0
Accept-language:fr
(extra carriage return, line feed)
```

We can learn a lot by taking a good look at this simple request message. First of all, we see that the message is written in ordinary ASCII text, so that your ordinary computer-literate human being can read it. Second, we see that the message consists of five lines, each followed by a carriage return and a line feed. The last line is followed by an additional carriage return and line feed. Although this particular request message has five lines, a request message can have many more lines or as few as one line. The first line of an HTTP request message is called the request line; the subsequent lines are called the header lines. The request line has three fields: the method field, the URL field, and the HTTP version field. The method field can take on several different values, including GET, POST, and HEAD. The great majority of HTTP request messages use the GET method. The GET method is used when the browser requests an object, with the requested object identified in the URL field. In this example, the browser is requesting the object /somedir/page.html. The version is self-explanatory; in this example, the browser implements version HTTP/1.1.

Now let's look at the header lines in the example. The header line Host: www. someschool.edu specifies the host on which the object resides. By including the Connection: close header line, the browser is telling the server that it doesn't want to use persistent connections; it wants the server to close the connection after sending the requested object. Although the browser that generated this request message implements HTTP/1.1, it doesn't want to bother with persistent connections. The User-agent: header line specifies the user agent, that is, the browser type that is making the request to the server. Here the user agent is Mozilla/4.0, a Netscape browser. This header line is useful because the server can actually send different versions of the same object to different types of user agents. (Each of the versions is addressed by the same URL.) Finally, the Accept-language: header indicates that the user prefers to receive a French version of the object, if such an object exists on the server; otherwise, the server should send its default version. The Accept-language: header is just one of many content negotiation headers available in HTTP.

Having looked at an example, let us now look at the general format for a request message, as shown in Figure 2.7.

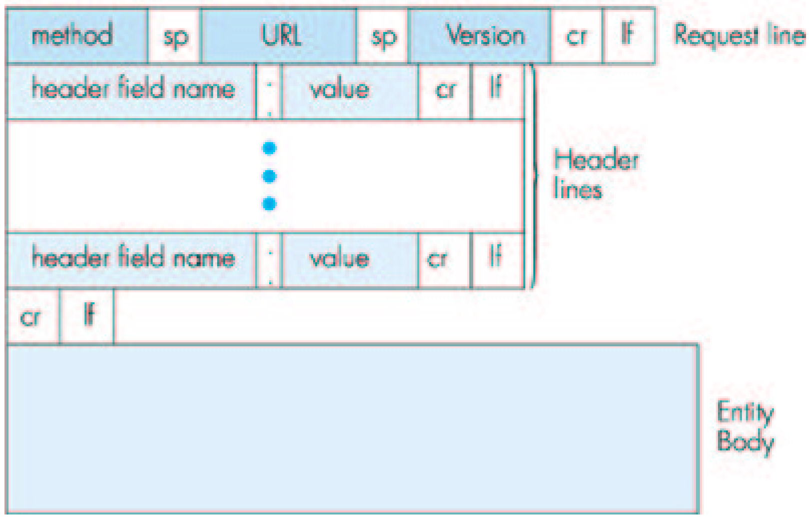

Figure 2.7: General format of a request message

We see that the general format of a request message closely follows our earlier example. You may have noticed, however, that after the header lines (and the additional carriage return and line feed) there is an "entity body." The entity body is not used with the GET method, but is used with the POST method. The HTTP client uses the POST method when the user fills out a form--for example, when a user gives search words to a search engine such as Altavista. With a POST message, the user is still requesting a Web page from the server, but the specific contents of the Web page depend on what the user entered into the form fields. If the value of the method field is POST, then the entity body contains what the user entered into the form fields. The HEAD method is similar to the GET method. When a server receives a request with the HEAD method, it responds with an HTTP message but it leaves out the requested object. The HEAD method is often used by HTTP server developers for debugging.

HTTP Response Message

Below we provide a typical HTTP response message. This response message could be the response to the example request message just discussed.

```
HTTP/1.1 200 OK
Connection: close
Date: Thu, 06 Aug 1998 12:00:15 GMT
Server: Apache/1.3.0 (Unix)
Last-Modified: Mon, 22 Jun 1998 09:23:24 GMT
Content-Length: 6821
Content-Type: text/html

(data data data data data . . .)
```

Let's take a careful look at this response message. It has three sections: an initial status line, six header lines, and then the entity body. The entity body is the meat of the message--it contains the requested object itself (represented by *data data data data data . . .).* The status line has three fields: the protocol version field, a status code, and a corresponding status message. In this example, the status line indicates that the server is using HTTP/1.1 and that everything is OK (that is, the server has found, and is sending, the requested object).

Now let's look at the header lines. The server uses the Connection: close header line to tell the client that it is going to close the TCP connection after sending the message. The Date: header line indicates the time and date when the HTTP response was created and sent by the server. Note that this is not the time when the object was created or last modified; it is the time when the server retrieves the object from its file system, inserts the object into the response message, and sends the response message. The Server: header line indicates that the message was generated by an Apache Web server; it is analogous to the User-agent: header line in the HTTP request message. The Last-Modified: header line indicates the time and date when the object was created or last modified. The Last-Modified: header, which we will cover in more detail, is critical for object caching, both in the local client and in network cache servers (also known as proxy servers). The Content-Length: header line indicates the number of bytes in the object being sent. The Content-Type: header line indicates that the object in the entity body is HTML text. (The object type is officially indicated by the Content-Type: header and not by the file extension.)

Note that if the server receives an HTTP/1.0 request, it will not use persistent connections, even if it is an HTTP/1.1 server. Instead, the HTTP/1.1 server will close the TCP connection after sending the object. This is necessary because an HTTP/1.0 client expects the server to close the connection.

Having looked at an example, let us now examine the general format of a response message, which is shown in Figure 2.8. This general format of the response message matches the previous example of a response message.

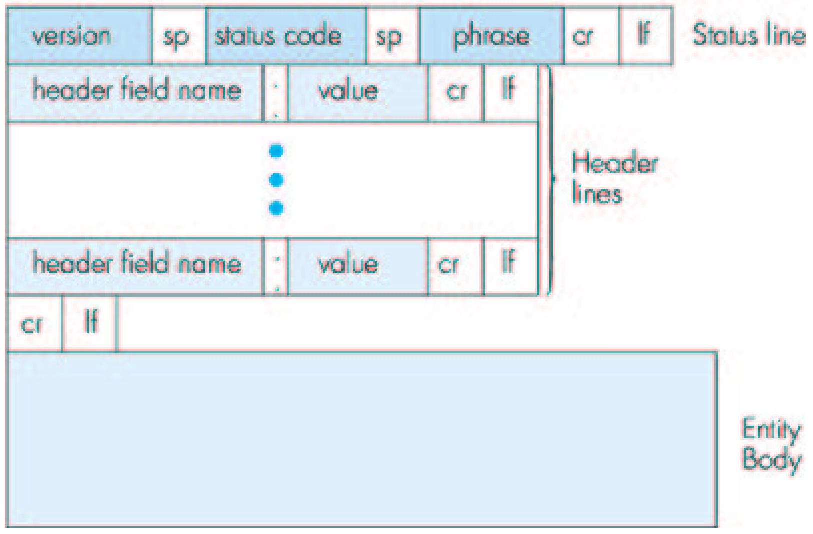

Figure 2.8: General format of a response message

Let's say a few additional words about status codes and their phrases. The status code and associated phrase indicate the result of the request. Some common status codes and associated phrases include:

- `200 OK`: Request succeeded and the information is returned in the response.
- `301 Moved Permanently`: Requested object has been permanently moved; new URL is specified in `Location:` header of the response message. The client software will automatically retrieve the new URL.
- `400 Bad Request`: A generic error code indicating that the request could not be understood by the server.
- `404 Not Found`: The requested document does not exist on this server.
- `505 HTTP Version Not Supported`: The requested HTTP protocol version is not supported by the server.

How would you like to see a real HTTP response message? This is very easy to do! First Telnet into your favorite Web server. Then type in a one-line request message for some object that is housed on the server. For example, if you can logon to a Unix machine, type:

```
telnet www.eurecom.fr 80
GET /~ross/index.html HTTP/1.0
```

(Hit the carriage return twice after typing the second line.) This opens a TCP connection to port 80 of the host `www.eurecom.fr` and then sends the HTTP GET command. You should see a response message that includes the base HTML file of Professor Ross's homepage. If you'd rather just see the HTTP message lines and not receive the object itself, replace `GET` with `HEAD`. Finally, replace `/~ross/ index.html` with `/~ross/banana.html` and see what kind of response message you get.

In this section we discussed a number of header lines that can be used within HTTP request and response messages. The HTTP specification (especially HTTP/ 1.1) defines many, many more header lines that can be inserted by browsers, Web servers, and network cache servers. We have only covered a small number of the totality of header lines. We will cover a few more below and another small number when we discuss network Web caching at the end of this chapter. A readable and comprehensive discussion of HTTP headers and status codes is given in [Luotonen 1998]. An excellent introduction to the technical issues surrounding the Web is [Yeager 1996].

How does a browser decide which header lines to include in a request message? How does a Web server decide which header lines to include in a response message? A browser will generate header lines as a function of the browser type and version (for example, an HTTP/1.0 browser will not generate any 1.1 header lines), the user configuration of the browser (for example, preferred language), and whether the browser currently has a cached, but possibly out-of-date, version of the object. Web servers behave similarly: There are different products, versions, and configurations, all of which influence which header lines are included in response messages.

2.2.4: User-Server Interaction: Authentication and Cookies

We mentioned above that an HTTP server is stateless. This simplifies server design, and has permitted engineers to develop very high-performing Web servers. However, it is often desirable for a Web site to identify users, either because the server wishes to restrict user access or because it wants to serve content as a function of the user identity. HTTP provides two mechanisms to help a server identify a user: authentication and cookies.

Authentication

Many sites require users to provide a username and a password in order to access the documents housed on the server. This requirement is referred to as authentication. HTTP provides special status codes and headers to help sites perform authentication. Let's walk through an example to get a feel for how these special status codes and headers work. Suppose a client requests an object from a server, and the server requires user authorization.

The client first sends an ordinary request message with no special header lines. The server then responds with empty entity body and with a `401 Authorization Required` status code. In this response message the server includes the `WWW-Authenticate:` header, which specifies the details about how to perform authentication. (Typically, it indicates that the user needs to provide a username and a password.)

The client receives the response message and prompts the user for a username and password. The client resends the request message, but this time includes an `Authorization:` header line, which includes the username and password.

After obtaining the first object, the client continues to send the username and password in subsequent requests for objects on the server. (This typically continues until the client closes the browser. However, while the browser remains open, the username and password are cached, so the user is not prompted for a username and password for each object it requests!) In this manner, the site can identify the user for every request.

We will see in Chapter 7 that HTTP performs a rather weak form of authentication, one that would not be difficult to break. We will study more secure and robust authentication schemes later in Chapter 7.

Cookies

Cookies are an alternative mechanism that sites can use to keep track of users. They are defined in RFC 2109. Some Web sites use cookies and others don't. Let's walk through an example. Suppose a client contacts a

Web site for the first time, and this site uses cookies. The server's response will include a Set-cookie: header. Often this header line contains an identification number generated by the Web server. For example, the header line might be:

```
Set-cookie: 1678453
```

When the HTTP client receives the response message, it sees the Set-cookie: header and identification number. It then appends a line to a special cookie file that is stored in the client machine. This line typically includes the host name of the server and user's associated identification number. In subsequent requests to the same server, say one week later, the client includes a Cookie: request header, and this header line specifies the identification number for that server. In the current example, the request message includes the header line:

```
Cookie: 1678453
```

In this manner, the server does not know the username of the user, but the server does know that this user is the same user that made a specific request one week ago.

Web servers use cookies for many different purposes:

- If a server requires authentication but doesn't want to hassle a user with a username and password prompt every time the user visits the site, it can set a cookie.
- If a server wants to remember a user's preferences so that it can provide targeted advertising during subsequent visits, it can set a cookie.
- If a user is shopping at a site (for example, buying several CDs), the server can use cookies to keep track of the items that the user is purchasing, that is, to create a virtual shopping cart.

We mention, however, that cookies pose problems for a nomadic user who accesses the same site from different machines. The site will treat the user as a different user for each different machine used. We conclude by pointing the reader to the page Persistent Client State HTTP Cookies [Netscape Cookie 1999], which provides an in-depth but readable introduction to cookies. We also recommend Cookies Central [Cookie Central 2000], which includes extensive information on the cookie controversy.

2.2.5: The Conditional GET

By storing previously retrieved objects, Web caching can reduce object-retrieval delays and diminish the amount of Web traffic sent over the Internet. Web caches can reside in a client or in an intermediate network cache server. We will discuss network caching at the end of this chapter. In this subsection, we restrict our attention to client caching.

Although Web caching can reduce user-perceived response times, it introduces a new problem--a copy of an object residing in the cache may be *stale.* In other words, the object housed in the Web server may have been modified since the copy was cached at the client. Fortunately, HTTP has a mechanism that allows the client to employ caching while still ensuring that all objects passed to the browser are up to date. This mechanism is called the conditional GET. An HTTP request message is a so-called conditional GET message if (1) the request message uses the GET method and (2) the request message includes an `If-Modified-Since:` header line.

To illustrate how the conditional get operates, let's walk through an example. First, a browser requests an uncached object from some Web server:

```
GET /fruit/kiwi.gif HTTP/1.0
User-agent: Mozilla/4.0
```

Second, the Web server sends a response message with the object to the client:

```
HTTP/1.0 200 OK
Date: Wed, 12 Aug 1998 15:39:29
Server: Apache/1.3.0 (Unix)
Last-Modified: Mon, 22 Jun 1998 09:23:24
Content-Type: image/gif

(data data data data data ...)
```

The client displays the object to the user but also saves the object in its local cache. Importantly, the client also caches the last-modified date along with the object. Third, one week later, the user requests the same object and the object is still in the cache. Since this object may have been modified at the Web server in the past week, the browser performs an up-to-date check by issuing a conditional get. Specifically, the browser sends

```
GET /fruit/kiwi.gif HTTP/1.0
User-agent: Mozilla/4.0
If-modified-since: Mon, 22 Jun 1998 09:23:24
```

Note that the value of the `If-modified-since:` header line is exactly equal to the value of the `Last-Modified:` header line that was sent by the server one week ago. This conditional get is telling the server to send the object only if the object has been modified since the specified date. Suppose the object has not been modified since `22 Jun 1998 09:23:24`. Then, fourth, the Web server sends a response message to the client:

```
HTTP/1.0 304 Not Modified
Date: Wed, 19 Aug 1998 15:39:29
Server: Apache/1.3.0 (Unix)
```

(empty entity body)

We see that in response to the conditional get, the Web server still sends a response message, but does not include the requested object in the response message. Including the requested object would only waste bandwidth and increase user-perceived response time, particularly if the object is large. Note that this last response message has in the status line 304 Not Modified, which tells the client that it can go ahead and use its cached copy of the object.

2.2.6: Web Caches

A Web cache--also called a proxy server--is a network entity that satisfies HTTP requests on the behalf of a client. The Web cache has its own disk storage and keeps in this storage copies of recently requested objects. As shown in Figure 2.9, users configure their browsers so that all of their HTTP requests are first directed to the Web cache. (This is a straightforward procedure with Microsoft and Netscape browsers.) Once a browser is configured, each browser request for an object is first directed to the Web cache.

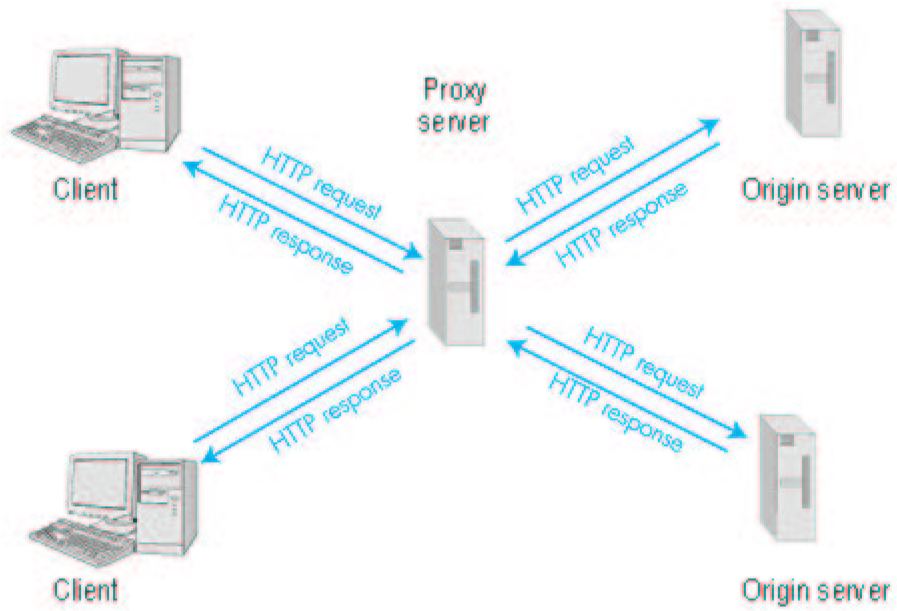

Figure 2.9: Clients requesting objects through a Web cache

As an example, suppose a browser is requesting the object http://www.someschool.edu/campus.gif.

- The browser establishes a TCP connection to the Web cache and sends an HTTP request for the object to the Web cache.
- The Web cache checks to see if it has a copy of the object stored locally. If it does, the Web cache forwards the object within an HTTP response message to the client browser.
- If the Web cache does not have the object, the Web cache opens a TCP connection to the origin server, that is, to www.someschool.edu. The Web cache then sends an HTTP request for the object into the TCP connection. After receiving this request, the origin server sends the object within an HTTP response to the Web cache.
- When the Web cache receives the object, it stores a copy in its local

storage and forwards a copy, within an HTTP response message, to the client browser (over the existing TCP connection between the client browser and the Web cache).

Note that a cache is both a server and a client at the same time. When it receives requests from and sends responses to a browser, it is a server. When it sends requests to and receives responses from an origin server it is a client.

So why bother with a Web cache? What advantages does it have? Web caches are enjoying wide-scale deployment in the Internet for at least three reasons. First, a Web cache can substantially reduce the response time for a client request, particularly if the bottleneck bandwidth between the client and the origin server is much less than the bottleneck bandwidth between the client and the cache. If there is a high-speed connection between the client and the cache, as there often is, and if the cache has the requested object, then the cache will be able to rapidly deliver the object to the client. Second, as we will soon illustrate with an example, Web caches can substantially reduce traffic on an institution's access link to the Internet. By reducing traffic, the institution (for example, a company or a university) does not have to upgrade bandwidth as quickly, thereby reducing costs. Furthermore, Web caches can substantially reduce Web traffic in the Internet as a whole, thereby improving performance for all applications. In 1998, over 75 percent of Internet traffic was Web traffic, so a significant reduction in Web traffic can translate into a significant improvement in Internet performance [Claffy 1998]. Third, an Internet dense with Web caches--such as, at institutional, regional, and national levels--provides an infrastructure for rapid distribution of content, even for content providers who run their sites on low-speed servers behind low-speed access links. If such a "resource-poor" content provider suddenly has popular content to distribute, this popular content will quickly be copied into the Internet caches, and high user demand will be satisfied.

To gain a deeper understanding of the benefits of caches, let us consider an example in the context of Figure 2.10. This figure shows two networks--the institutional network and the rest of the public Internet. The institutional network is a high-speed LAN. A router in the institutional network and a router in the Internet are connected by a 1.5 Mbps link. The origin servers are attached to the Internet, but located all over the globe. Suppose that the average object size is 100 Kbits and that the average request rate from the institution's browsers to the origin servers is 15 requests per second. Also suppose that the amount of time it takes from when the router on the Internet side of the access link in Figure 2.10 forwards an HTTP request (within an IP datagram) until it receives the IP datagram (typically, many IP datagrams) containing the corresponding response is two seconds on average. Informally, we refer to this last delay as the "Internet delay."

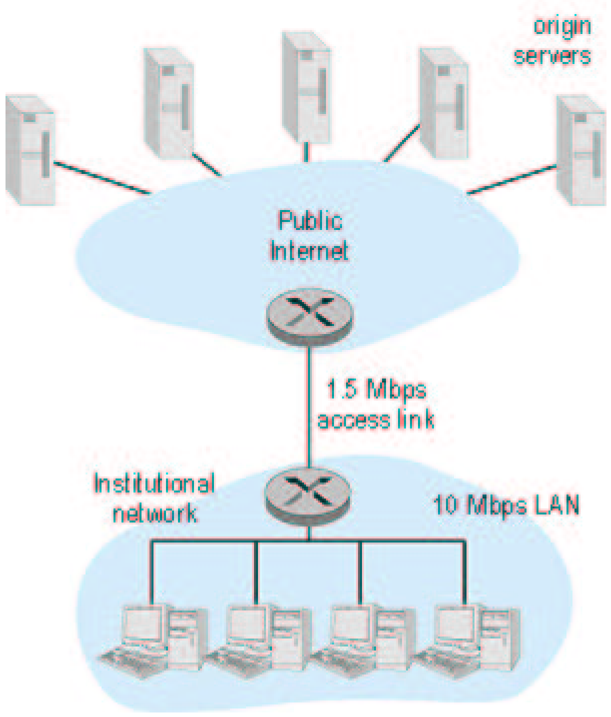

Figure 2.10: Bottleneck between an institutional network and the Internet

The total response time--that is, the time from the browser's request of an object until its receipt of the object--is the sum of the LAN delay, the access delay (that is, the delay between the two routers), and the Internet delay. Let us now do a very crude calculation to estimate this delay. The traffic intensity on the LAN (see Section 1.6) is

(15 requests/sec) * (100 Kbits/request)/(10Mbps) = 0.15

whereas the traffic intensity on the access link (from the Internet router to institution router) is

(15 requests/sec) * (100 Kbits/request)/(1.5 Mbps) = 1

A traffic intensity of 0.15 on a LAN typically results in, at most, tens of milliseconds of delay; hence, we can neglect the LAN delay. However, as discussed in Section 1.6, as the traffic intensity approaches 1 (as is the case of the access link in Figure 2.10), the delay on a link becomes very large and grows without bound. Thus, the average response time to satisfy requests is going to be on the order of minutes, if not more, which is unacceptable for the institution's users. Clearly something must be done. One possible solution is to increase the access rate from 1.5 Mbps to, say, 10 Mbps. This will lower the traffic intensity on the access link to 0.15, which translates to negligible delays between the two routers. In this case, the total response time will roughly be 2 seconds, that is, the Internet delay. But this solution also means that the institution must upgrade its access link from 1.5 Mbps to 10 Mbps, which can be very costly.

Now consider the alternative solution of not upgrading the access link but instead installing a Web cache in the institutional network. This solution is illustrated in Figure 2.11.

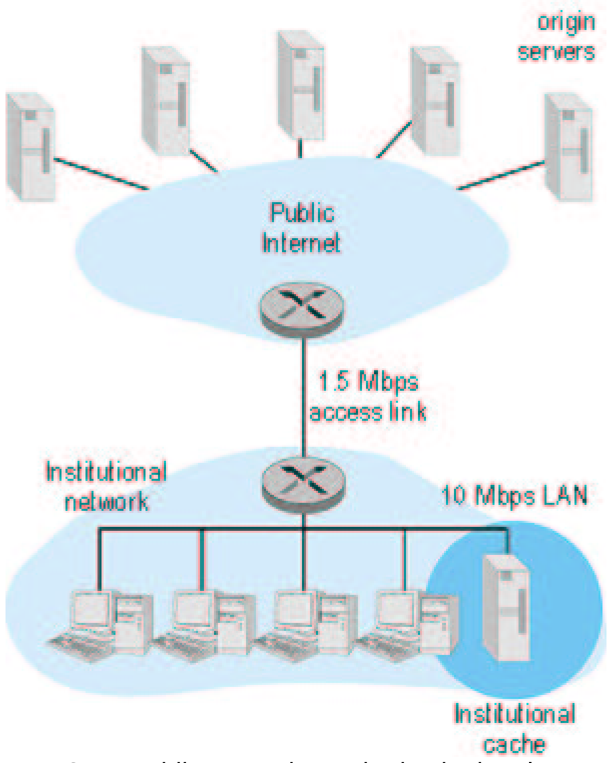

Figure 2.11: Adding a cache to the institutional network

Hit rates--the fraction of requests that are satisfied by a cache-- typically range from 0.2 to 0.7 in practice. For illustrative purposes, let us suppose that the cache provides a hit rate of 0.4 for this institution. Because the clients and the cache are connected to the same high-speed LAN, 40 percent of the requests will be satisfied almost immediately, say, within 10 milliseconds, by the cache. Nevertheless, the remaining 60 percent of the requests still need to be satisfied by the origin servers. But with only 60 percent of the requested objects passing through the access link, the traffic intensity on the access link is reduced from 1.0 to 0.6. Typically, a traffic intensity less than 0.8 corresponds to a small delay, say, tens of milliseconds, on a 1.5 Mbps link. This delay is negligible compared with the 2-second Internet delay. Given these considerations, average delay therefore is

$$0.4 * (0.010 \text{ seconds}) + 0.6 * (2.01 \text{ seconds})$$

which is just slightly greater than 1.2 seconds. Thus, this second solution provides an even lower response time than the first solution, and it doesn't require the institution to upgrade its link to the Internet. The institution does, of course, have to purchase and install a Web cache. But this cost is low--many caches use public-domain software that run on inexpensive servers and PCs.

Cooperative Caching

Multiple Web caches, located at different places in the Internet, can cooperate and improve overall performance. For example, an institutional cache can be configured to send its HTTP requests to a cache in a backbone ISP at the national level. In this case, when the institutional

cache does not have the requested object in its storage, it forwards the HTTP request to the national cache. The national cache then retrieves the object from its own storage or, if the object is not in storage, from the origin server. The national cache then sends the object (within an HTTP response message) to the institutional cache, which in turn forwards the object to the requesting browser. Whenever an object passes through a cache (institutional or national), the cache keeps a copy in its local storage. The advantage of passing through a higher-level cache, such as a national cache, is that it has a larger user population and therefore higher hit rates.
An example of a cooperative caching system is the NLANR caching system, which consists of a number of backbone caches in the United States providing ser vice to institutional and regional caches from all over the globe [NLANR 1999]. The NLANR caching hierarchy is shown in Figure 2.12 [Huffaker 1998]. The caches obtain objects from each other using a combination of HTTP and ICP (Internet Caching Protocol). ICP is an application-layer protocol that allows one cache to quickly ask another cache if it has a given document [RFC 2186]; a cache can then use HTTP to retrieve the object from the other cache. ICP is used extensively in many cooperative caching systems and is fully supported by Squid, a popular public-domain software for Web caching [Squid 2000]. If you are interested in learning more about ICP, you are encouraged to see [Luotonen 1998], [Ross 1998], and the ICP RFC [RFC 2186].

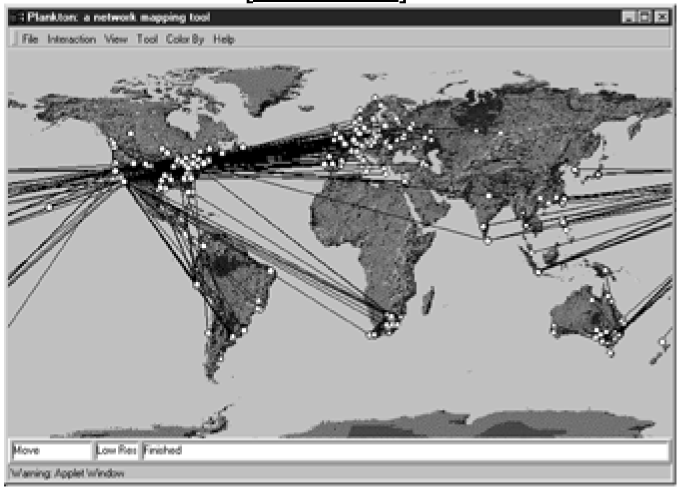
Plankton: a network mapping tool
File Interaction View Tool Color By Help
Move
Low Res
Finished
Warning: Applet Window

Figure 2.12: The NLANR caching hierarchy (Courtesy of [Huffaker and Jung 1998]

An alternative form of cooperative caching involves clusters of caches, often co-located on the same LAN. A single cache is often replaced with a cluster of caches when the single cache is not sufficient to handle the traffic or provide sufficient storage capacity. Although cache clustering is a natural way to scale as traffic increases, clusters introduce a new problem: When a browser wants to request a particular object, to which cache in the cache cluster should it send the request? This problem can be elegantly solved using hash routing. (If you are not familiar with hash functions, you can read about them in Chapter 7.) In the simplest form of hash routing, the browser hashes the URL, and depending on the result of the hash, the browser directs its request message to one of the caches in the cluster. By having

all the browsers use the same hash function, an object will never be present in more than one cache in the cluster, and if the object is indeed in the cache cluster, the browser will always direct its request to the correct cache. Hash routing is the essence of the Cache Array Routing Protocol (CARP). Additional information is available if you are interested in learning more about hash routing or CARP [Valloppillil 1997; Luotonen 1998; Ross 1997, 1998].
Web caching is a rich and complex subject. Web caching has also enjoyed extensive research and product development in recent years. Furthermore, caches are now being built to handle streaming audio and video. Caches will likely play an important role as the Internet begins to provide an infrastructure for the large-scale, on-demand distribution of music, television shows, and movies in the Internet.

Online Book

2.3: File Transfer: FTP

FTP (File Transfer Protocol) is a protocol for transferring a file from one host to another host. The protocol dates back to 1971 (when the Internet was still an experiment), but remains enormously popular. FTP is described in RFC 959. Figure 2.13 provides an overview of the services provided by FTP.

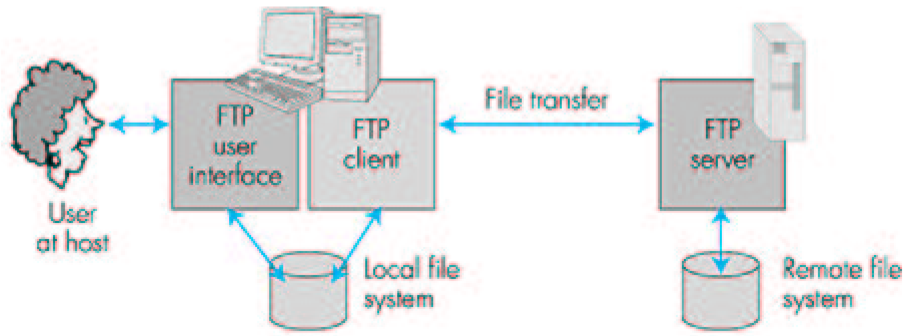

Figure 2.13: FTP moves files between local and remote file systems

In a typical FTP session, the user is sitting in front of one host (the local host) and wants to transfer files to or from a remote host. In order for the user to access the remote account, the user must provide a user identification and a password. After providing this authorization information, the user can transfer files from the local file system to the remote file system and vice versa. As shown in Figure 2.13, the user interacts with FTP through an FTP user agent. The user first provides the hostname of the remote host, causing the FTP client process in the local host to establish a TCP connection with the FTP server process in the remote host. The user then provides the user identification and password, which get sent over the TCP connection as part of FTP commands. Once the server has authorized the user, the user copies one or more files stored in the local file system into the remote file system (or vice versa).

HTTP and FTP are both file transfer protocols and have many common characteristics; for example, they both run on top of TCP. However, the two application-layer protocols have some important differences. The most striking difference is that FTP uses two parallel TCP connections to transfer a file, a control connection and a data connection. The control connection is used for sending control information between the two hosts--information such as user identification, password, commands to change remote directory, and commands to "put" and "get" files. The data connection is used to actually send a file. Because FTP uses a separate control connection, FTP is said to send its control information out-of-band. In Chapter 6 we shall see that the RTSP protocol, which is used for controlling the transfer of continuous media such as audio and video, also sends its control information out-of-band. HTTP, as you recall, sends request and response header lines into the same TCP connection that carries the transferred file itself. For this reason, HTTP is said to send its control information in-band. In the next section we shall see that SMTP, the main protocol for electronic mail, also sends control information in-band. The FTP control and data connections are illustrated in Figure 2.14.

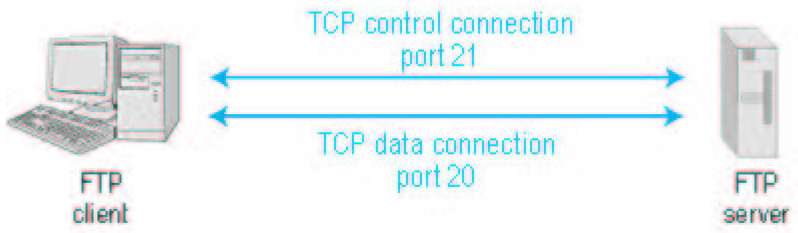

Figure 2.14: Control and data connecctions

When a user starts an FTP session with a remote host, FTP first sets up a control TCP connection on server port number 21. The client side of FTP sends the user identification and password over this control connection. The client side of FTP also sends, over the control connection, commands to change the remote directory. When the user requests a file transfer (either to, or from, the remote host), FTP opens a TCP data connection on server port number 20. FTP sends exactly one file over the data connection and then closes the data connection. If, during the same session, the user wants to transfer another file, FTP opens another data connection. Thus, with FTP, the control connection remains open throughout the duration of the user session, but a new data connection is created for each file transferred within a session (that is, the data connections are nonpersistent).

Throughout a session, the FTP server must maintain state about the user. In particular, the server must associate the control connection with a specific user account, and the server must keep track of the user's current directory as the user wanders about the remote directory tree. Keeping track of this state information for each ongoing user session significantly constrains the total number of sessions that FTP can maintain simultaneously. HTTP, on the other hand, is stateless--it does not have to keep track of any user state.

2.3.1: FTP Commands and Replies

We end this section with a brief discussion of some of the more common FTP commands. The commands, from client to server, and replies, from server to client, are sent across the control connection in seven-bit ASCII format. Thus, like HTTP commands, FTP commands are readable by people. In order to delineate successive commands, a carriage return and line feed end each command (and reply). Each command consists of four uppercase ASCII characters, some with optional arguments. Some of the more common commands are given below (with options in italics):

- USER username: Used to send the user identification to server.
- PASS password: Used to send the user password to the server.
- LIST: Used to ask the server to send back a list of all the files in the current remote directory. The list of files is sent over a (new and nonpersistent) data connection rather than the control TCP connection.
- RETR filename: Used to retrieve (that is, get) a file from the current directory of the remote host.
- STOR filename: Used to store (that is, put) a file into the current directory of the remote host.

There is typically a one-to-one correspondence between the command that the user issues and the FTP command sent across the control connection. Each command is followed by a reply, sent from server to client. The replies are three-digit numbers, with an optional message following the number. This is similar in structure to the status code and phrase in the status line of the HTTP response message; the inventors of HTTP intentionally included this similarity in the HTTP response messages. Some typical replies, along with their possible messages, are as follows:

- 331 Username OK, password required
- 125 Data connection already open; transfer starting
- 425 Can't open data connection

- 452 Error writing file

Readers who are interested in learning about the other FTP commands and replies are encouraged to read RFC 959.

Online Book

2.4: Electronic Mail in the Internet

Along with the Web, electronic mail is one of the most popular Internet applications. Just like ordinary "snail mail," e-mail is asynchronous--people send and read messages when it is convenient for them, without having to coordinate with other peoples' schedules. In contrast with snail mail, electronic mail is fast, easy to distribute, and inexpensive. Moreover, modern electronic mail messages can include hyperlinks, HTML formatted text, images, sound, and even video. In this section we will examine the application-layer protocols that are at the heart of Internet electronic mail. But before we jump into an in-depth discussion of these protocols, let's take a bird's eye view of the Internet mail system and its key components.

Figure 2.15 presents a high-level view of the Internet mail system. We see from this diagram that it has three major components: user agents, mail servers, and the Simple Mail Transfer Protocol (SMTP). We now describe each of these components in the context of a sender, Alice, sending an e-mail message to a recipient, Bob. User agents allow users to read, reply to, forward, save, and compose messages. (User agents for electronic mail are sometimes called *mail readers,* although we will generally avoid this term in this book.) When Alice is finished composing her message, her user agent sends the message to her mail server, where the message is placed in the mail server's outgoing message queue. When Bob wants to read a message, his user agent obtains the message from his mailbox in his mail server. In the late 1990s, GUI (graphical user interface) user agents became popular, allowing users to view and compose multimedia messages. Currently, Eudora, Microsoft's Outlook, and Netscape's Messenger are among the popular GUI user agents for e-mail. There are also many text-based e-mail user interfaces in the public domain, including mail, pine, and elm.

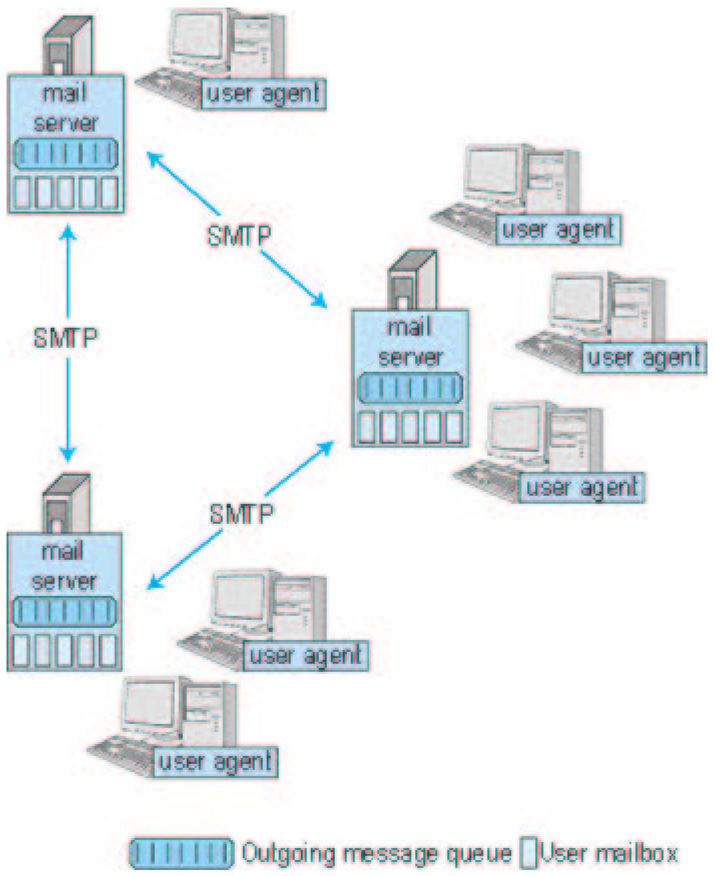

Figure 2.15: A bird's eye view of the Internet e-mail system

Mail servers form the core of the e-mail infrastructure. Each recipient, such as Bob, has a mailbox located in one of the mail servers. Bob's mailbox manages and maintains the messages that have been sent to him. A typical message starts its journey in the sender's user agent, travels to the sender's mail server, and then travels to the recipient's mail server, where it is deposited in the recipient's mailbox. When Bob wants to access the messages in his mailbox, the mail server containing the mailbox authenticates Bob (with user names and passwords). Alice's mail server must also deal with failures in Bob's mail server. If Alice's server cannot deliver mail to Bob's server, Alice's server holds the message in a message queue and attempts to transfer the message later. Reattempts are often done every 30 minutes or so; if there is no success after several days, the server removes the message and notifies the sender (Alice) with an e-mail message.

The Simple Mail Transfer Protocol (SMTP) is the principal application-layer protocol for Internet electronic mail. It uses the reliable data transfer service of TCP to transfer mail from the sender's mail server to the recipient's mail server. As with most application-layer protocols, SMTP has two sides: a client side, which executes on the sender's mail server, and a server side, which executes on the recipient's mail server. Both the client and server sides of SMTP run on every mail server. When a mail server sends mail (to other mail servers), it acts as an SMTP client. When a mail server receives mail (from other mail servers) it acts as an SMTP server.

Case History

Hotmail

In December 1995, Sabeer Bhatia and Jack Smith visited the Internet venture capitalist Draper Fisher Jurvetson and proposed developing a free Web-based e-mail system. The idea was to give a free e-mail account to anyone who wanted one, and to make the accounts accessible from the Web. With Web-based e-mail, anyone with access to the Web--say, from a school or community library--could read and send e-mails. Furthermore, web-based e-mail would offer great mobility to its subscribers. In exchange for 15 percent of the company, Draper Fisher Jurvetson financed Bhatia and Smith, who formed a company called Hotmail. With three full-time people and 12 to 14 part-time people who worked for stock options, they were able to develop and launch the service in July 1996. Within a month after launch they had 100,000 subscribers. The number of subscribers continued to grow rapidly, with all of their subscribers being exposed to advertising banners while reading their e-mail. In December 1997, less than 18 months after launching the service, Hotmail had over 12 million subscribers and was acquired by Microsoft, reportedly for $400 million dollars.

The success of Hotmail is often attributed to its "first-mover advantage" and to the inherent "viral marketing" of e-mail. Hotmail had a first-mover advantage because it was the first company to offer Web-based e-mail. Other companies, of course, copied Hotmail's idea, but Hotmail had a six-month lead on them. The coveted first-mover advantage is obtained by having an original idea, and then developing quickly and secretly. A service or a product is said to have viral marketing if it markets itself. E-mail is a classic example of a service with viral marketing--the sender sends a message to one or more recipients, and then all the recipients become aware of the service. Hotmail demonstrated that the combination of first-mover advantage and viral marketing can produce a killer application. Perhaps some of the students reading this book will be among the new entrepreneurs who conceive and develop first-mover Internet services with inherent viral marketing.

2.4.1: SMTP

SMTP, defined in RFC 821, is at the heart of Internet electronic mail. As mentioned above, SMTP transfers messages from senders' mail servers to the recipients' mail servers. SMTP is much older than HTTP. (The SMTP RFC dates back to 1982, and SMTP was around long before that.)
Although SMTP has numerous wonderful qualities, as evidenced by its ubiquity in the Internet, it is nevertheless a legacy technology that possesses certain "archaic" characteristics. For example, it restricts the body (not just the headers) of all mail messages to be in simple seven-bit ASCII. This restriction made sense in the early 1980s when transmission capacity was scarce and no one was e-mailing large attachments or large image, audio, or video files. But today, in the multimedia era, the seven-bit ASCII restriction is a bit of a pain--it requires binary multimedia data to be encoded to ASCII before being sent over SMTP; and it requires the corresponding ASCII message to be decoded back to binary after SMTP transport. Recall from Section 2.3 that HTTP does not require multimedia data to be ASCII encoded before transfer.
To illustrate the basic operation of SMTP, let's walk through a common

scenario. Suppose Alice wants to send Bob a simple ASCII message:

- Alice invokes her user agent for e-mail, provides Bob's e-mail address (for example, bob@someschool.edu), composes a message, and instructs the user agent to send the message.
- Alice's user agent sends the message to her mail server, where it is placed in a message queue.
- The client side of SMTP, running on Alice's mail server, sees the message in the message queue. It opens a TCP connection to an SMTP server, running on Bob's mail server.
- After some initial SMTP handshaking, the SMTP client sends Alice's message into the TCP connection.
- At Bob's mail server host, the server side of SMTP receives the message. Bob's mail server then places the message in Bob's mailbox.
- Bob invokes his user agent to read the message at his convenience.

The scenario is summarized in Figure 2.16.

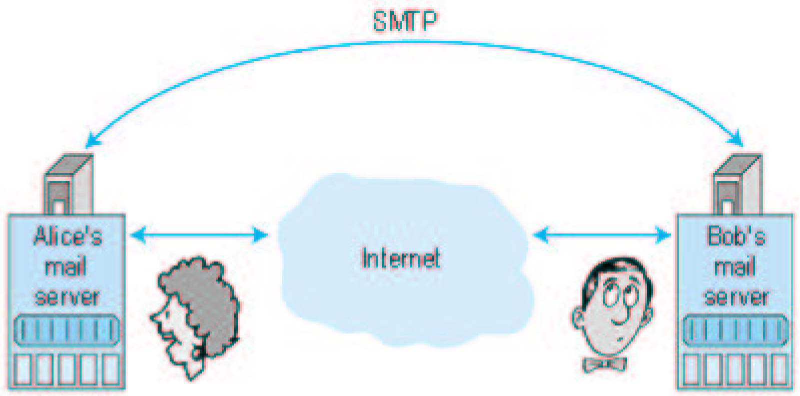

Figure 2.16: Alice's mail server transfers Alice's message to Bob's mail server

It is important to observe that SMTP does not normally use intermediate mail servers for sending mail, even when the two mail servers are located at opposite ends of the world. If Alice's server is in Hong Kong and Bob's server is in Mobile, Alabama, the TCP "connection" is a direct connection between the Hong Kong and Mobile servers. In particular, if Bob's mail server is down, the message remains in Alice's mail server and waits for a new attempt--the message does not get placed in some intermediate mail server.

Let's now take a closer look at how SMTP transfers a message from a sending mail server to a receiving mail server. We will see that the SMTP protocol has many similarities with protocols that are used for face-to-face human interaction. First, the client SMTP (running on the sending mail server host) has TCP establish a connection on port 25 to the server SMTP (running on the receiving mail server host). If the server is down, the client tries again later. Once this connection is established, the server and client perform some application-layer handshaking. Just as humans often introduce themselves before transferring information from one to another, SMTP clients and servers introduce themselves before transferring information. During this SMTP handshaking phase, the SMTP client indicates the e-mail address of the sender (the person who generated the message) and the e-mail address of the recipient. Once the SMTP client and server have introduced themselves to each other, the client sends the message. SMTP can count on the reliable data transfer service of TCP to get the message to the server without errors. The client then repeats this process over the same TCP connection if it has other messages to send to the server; otherwise, it instructs TCP to close the connection.

Let us take a look at an example transcript between client (C) and server (S). The host name of the client is crepes.fr and the host name of the server is hamburger.edu. The ASCII text lines prefaced with **C:** are exactly the lines the client sends into its TCP socket, and the ASCII text lines prefaced with **S:** are exactly the lines the server sends into its TCP socket. The following transcript begins as soon as the TCP connection is established:

S: 220 hamburger.edu

C: HELO crepes.fr

S: 250 Hello crepes.fr, pleased to meet you

C: MAIL FROM: <alice@crepes.fr>

S: 250 alice@crepes.fr... Sender ok

C: RCPT TO: <bob@hamburger.edu>

S: 250 bob@hamburger.edu ... Recipient ok

C: DATA

S: 354 Enter mail, end with "." on a line by itself

C: Do you like ketchup?

C: How about pickles?

C: .

S: 250 Message accepted for delivery

C: QUIT

S: 221 hamburger.edu closing connection

In the above example, the client sends a message ("Do you like ketchup? How about pickles?") from mail server crepes.fr to mail server hamburger.edu. The client issued five commands: HELO (an abbreviation for HELLO), MAIL FROM, RCPT TO, DATA, and QUIT. These commands are self explanatory. The server issues replies to each command, with each reply having a reply code and some (optional) English-language explanation. We mention here that SMTP uses persistent connections: If the sending mail server has several messages to send to the same receiving mail server, it can send all of the messages over the same TCP connection. For each message, the client begins the process with a new MAIL FROM: and only issues QUIT after all messages have been sent.

It is highly recommended that you use Telnet to carry out a direct dialogue with an SMTP server. To do this, issue telnet serverName 25 where serverName is the name of the remote mail server. When you do this, you are simply establishing a TCP connection between your local host and the mail server. After typing this line, you should immediately receive the 220 reply from the server. Then issue the SMTP commands HELO, MAIL FROM, RCPT TO, DATA, and QUIT at the appropriate times. If you Telnet into your friend's SMTP server, you should be able to send mail to your friend in this manner (that is, without using your mail user agent).

2.4.2: Comparison with HTTP

Let us now briefly compare SMTP to HTTP. Both protocols are used to transfer files from one host to another; HTTP transfers files (or objects) from Web server to Web user agent (that is, the browser); SMTP transfers files (that is, e-mail messages) from one mail server to another mail server. When transferring the files, both persistent HTTP and SMTP use persistent connections. Thus, the two protocols have common characteristics. However, there are important differences. First, HTTP is principally a pull protocol--someone loads information on a Web server and users use HTTP to pull the information from the server at their convenience. In particular, the TCP connection is initiated by the machine that wants to receive the file. On the other hand, SMTP is primarily a push protocol--the sending mail server pushes the file to the receiving mail server. In particular, the TCP connection is initiated by the machine that wants to send the file.
A second important difference, which we alluded to earlier, is that SMTP requires each message, including the body of each message, to be in seven-bit ASCII format. Furthermore, the SMTP RFC requires the body of every message to end with a line consisting of only a period--that is, in ASCII jargon, the body of each message ends with "CRLF.CRLF," where CR and LF stand for carriage return and line feed, respectively. In this manner, while the SMTP server is receiving a series of messages from an SMTP client, the server can delineate the messages by searching for "CRLF.CRLF" in the byte stream. Now suppose that the body of one of the messages is not ASCII text but instead binary data (for example, a JPEG image). It is possible that this binary data might accidentally have the bit pattern associated with ASCII representation of "CRLF.CRLF" in the middle of the bit stream. This would cause the SMTP server to incorrectly conclude that the message has terminated. To get around this and related problems, binary data is first encoded to ASCII in such a way that certain ASCII characters (including " . ") are not used. Returning to our comparison with HTTP, we note that neither nonpersistent nor persistent HTTP has to bother with the ASCII conversion. For nonpersistent HTTP, each TCP connection transfers exactly one object; when the server closes the connection, the client knows it has received one entire response message. For persistent HTTP, each response message includes a Content-length: header line, enabling the

client to delineate the end of each message.
A third important difference concerns how a document consisting of text and images (along with possibly other media types) is handled. As we learned in Section 2.3, HTTP encapsulates each object in its own HTTP response message. Internet mail, as we shall discuss in greater detail below, places all of the message's objects into one message.

2.4.3: Mail Message Formats and MIME

When Alice sends an ordinary snail-mail letter to Bob, she puts the letter into an envelope, on which there are all kinds of peripheral information such as Bob's address, Alice's return address, and the date (supplied by the postal service). Similarly, when an e-mail message is sent from one person to another, a header containing peripheral information precedes the body of the message itself. This peripheral information is contained in a series of header lines, which are defined in RFC 822. The header lines and the body of the message are separated by a blank line (that is, by CRLF). RFC 822 specifies the exact format for mail header lines as well as their semantic interpretations. As with HTTP, each header line contains readable text, consisting of a keyword followed by a colon followed by a value. Some of the keywords are required and others are optional. Every header must have a From: header line and a To: header line; a header may include a Subject: header line as well as other optional header lines. It is important to note that these header lines are *different* from the SMTP commands we studied in Section 2.4.1 (even though they contain some common words such as "from" and "to"). The commands in that section were part of the SMTP handshaking protocol; the header lines examined in this section are part of the mail message itself.
A typical message header looks like this:

```
From: alice@crepes.fr
To: bob@hamburger.edu
Subject: Searching for the meaning of life.
```

After the message header, a blank line follows, then the message body (in ASCII) follows. The message terminates with a line containing only a period, as discussed above. You should use Telnet to send to a mail server a message that contains some header lines, including the Subject: header line. To do this, issue `telnet serverName 25`.

The MIME Extension for Non-ASCII Data

While the message headers described in RFC 822 are satisfactory for sending ordinary ASCII text, they are not sufficiently rich enough for multimedia messages (for example, messages with images, audio, and video) or for carrying non-ASCII text formats (for example, characters used by languages other than English). To send content different from ASCII text, the sending user agent must include additional headers in the message. These extra headers are defined in RFC 2045 and RFC 2046, the MIME (Multipurpose Internet Mail Extensions) extension to RFC 822.
Two key MIME headers for supporting multimedia are the Content-Type:

header and the Content-Transfer-Encoding: header. The Content-Type: header allows the receiving user agent to take an appropriate action on the message. For example, by indicating that the message body contains a JPEG image, the receiving user agent can direct the message body to a JPEG decompression routine. To understand the need of the Content-Transfer-Encoding: header, recall that non-ASCII text messages must be encoded to an ASCII format that isn't going to confuse SMTP. The Content-Transfer-Encoding: header alerts the receiving user agent that the message body has been ASCII-encoded and to the type of encoding used. Thus, when a user agent receives a message with these two headers, it first uses the value of the Content-Transfer-Encoding: header to convert the message body to its original non-ASCII form, and then uses the Content-Type: header to determine what actions it should take on the message body.
Let's take a look at a concrete example. Suppose Alice wants to send Bob a JPEG image. To do this, Alice invokes her user agent for e-mail, specifies Bob's e-mail address, specifies the subject of the message, and inserts the JPEG image into the message body of the message. (Depending on the user agent Alice uses, she might insert the image into the message as an "attachment.") When Alice finishes composing her message, she clicks on "Send." Alice's user agent then generates a MIME message, which might look something like this:

```
From: alice@crepes.fr
To: bob@hamburger.edu
Subject: Picture of yummy crepe.
MIME-Version: 1.0
Content-Transfer-Encoding: base64
Content-Type: image/jpeg

(base64 encoded data .....
........................
......base64 encoded data)
```

We observe from the above MIME message that Alice's user agent encoded the JPEG image using base64 encoding. This is one of several encoding techniques standardized in the MIME [RFC 2045] for conversion to an acceptable seven-bit ASCII format. Another popular encoding technique is quoted-printable content-transfer-encoding, which is typically used to convert an ordinary ASCII message to ASCII text void of undesirable character strings (for example, a line with a single period).
When Bob reads his mail with his user agent, his user agent operates on this same MIME message. When Bob's user agent observes the Content-Transfer- Encoding: base64 header line, it proceeds to decode the base64-encoded message body. The message also includes a Content-Type: image/jpeg header line; this indicates to Bob's user agent that the message body should be JPEG decompressed. Finally, the message includes the

MIME-Version: header, which, of course, indicates the MIME version that is being used. Note that the message otherwise follows the standard RFC 822/SMTP format. In particular, after the message header there is a blank line and then the message body; and after the message body, there is a line with a single period.

Let's now take a closer look at the Content-Type: header. According to the MIME specification [RFC 2046], this header has the following format:

```
Content-Type: type/subtype; parameters
```

where the "parameters" (along with the semicolon) is optional.

Paraphrasing [RFC 2046], the Content-Type field is used to specify the nature of the data in the body of a MIME entity, by giving media type and subtype names. After the type and subtype names, the remainder of the header field consists of a set of parameters. In general, the top-level type is used to declare the general type of data, whereas the subtype specifies a specific format for that type of data. The parameters are modifiers of the subtype and as such do not fundamentally affect the nature of the content. The set of meaningful parameters depends on the type and subtype. Most parameters are associated with a single specific subtype. MIME has been carefully designed to be extensible, and it is expected that the set of media type/subtype pairs and their associated parameters will grow significantly over time. In order to ensure that the set of such types/subtypes is developed in an orderly, well-specified, and public manner, MIME sets up a registration process that uses the Internet Assigned Numbers Authority (IANA) as a central registry for MIME's various areas of extensibility. The registration process for these areas is described in RFC 2048.

Currently there are seven top-level types defined. For each type, there is a list of associated subtypes, and the lists of subtypes are growing every year. We describe five of these types below:

- text: The text type is used to indicate to the receiving user agent that the message body contains textual information. One extremely common type/subtype pair is text/plain. The subtype plain indicates plain text containing no formatting commands or directives. Plain text is to be displayed as is; no special software is required to get the full meaning of the text, aside from support for the indicated character set. If you take a glance at the MIME headers in some of the messages in your mailbox, you will almost certainly see content type header lines with text/plain; charset=us-ascii or text/plain; charset="ISO-8859-1". The parameters indicate the character set used to generate the message. Another type/subtype pair that is gaining popularity is text/html. The html subtype indicates to the mail reader that it should interpret the embedded HTML tags that are included in the message. This allows the receiving user agent to display the message as a Web page, which might include a variety of fonts, hyperlinks, applets, and so on.

- image: The image type is used to indicate to the receiving user agent that the message body is an image. Two popular type/subtype pairs are `image/gif` and `image/jpeg`. When the receiving user agent encounters `image/gif`, it knows that it should decode the GIF image and then display it.
- audio: The audio type requires an audio output device (such as a speaker or a telephone) to render the contents. Some of the standardized subtypes include `basic` (basic eight-bit μ-law encoded) and `32kadpcm` (a 32 Kbps format defined in RFC 1911).
- video: The video type includes `mpeg`, and `quicktime` for subtypes.
- application: The application type is for data that does not fit in any of the other categories. It is often used for data that must be processed by an application before it is viewable or usable by a user. For example, when a user attaches a Microsoft Word document to an e-mail message, the sending user agent typically uses `application/msword` for the type/subtype pair. When the receiving user agent observes the content type `application/msword`, it launches the Microsoft Word application and passes the body of the MIME message to the application. A particularly important subtype for the application type is `octet-stream`, which is used to indicate that the body contains arbitrary binary data. Upon receiving this type, a mail reader will prompt the user, providing the option to save the message to disk for later processing.

There is one MIME type that is particularly important and requires special discussion, namely, the multipart type. Just as a Web page can contain many objects (such as text, images, applets), so too can an e-mail message. Recall that the Web sends each of the objects within independent HTTP response messages. Internet e-mail, on the other hand, places all the objects (or "parts") in the same message. In particular, when a multimedia message contains more than one object (such as multiple images or some ASCII text and some images) the message typically has `Content-type: multipart/mixed`. This content type header line indicates to the receiving user agent that the message contains multiple objects. With all the objects in the same message, the receiving user agent needs a means to determine (1) where each object begins and ends, (2) how each non-ASCII object was transfer-encoded, and (3) the content type of each message. This is done by placing *boundary characters* between each object and preceding each object in the message with `Content-type:` and `Content-Transfer-Encoding:` header lines.

To obtain a better understanding of `multipart/mixed`, let's look at an example. Suppose that Alice wants to send a message to Bob consisting of some ASCII text, followed by a JPEG image, followed by more ASCII text.

Using her user agent, Alice types some text, attaches a JPEG image, and then types some more text. Her user agent then generates a message something like this:

```
From: alice@crepes.fr
To: bob@hamburger.edu
Subject: Picture of yummy crepe with commentary
MIME-Version: 1.0
Content-Type: multipart/mixed; Boundary=StartOfNextPart

--StartOfNextPart
Dear Bob,
Please find a picture of an absolutely scrumptious crepe.
--StartOfNextPart
Content-Transfer-Encoding: base64
Content-Type: image/jpeg
base64 encoded data .....
.........................
......base64 encoded data
--StartOfNextPart
Let me know if you would like the recipe.
```

Examining the above message, we note that the Content-Type: line in the header indicates how the various parts in the message are separated. The separation always begins with two dashes and ends with CRLF.

The Received Message

As we have discussed, an e-mail message consists of many components. The core of the message is the message body, which is the actual data being sent from sender to receiver. For a multipart message, the message body itself consists of many parts, with each part preceded by one or more lines of identifying information. Preceding the message body is a blank line and then a number of header lines. These header lines include RFC 822 header lines such as From:, To:, and Subject: header lines. The header lines also include MIME header lines such as Content-type: and Content-transfer-encoding: header lines. But we would be remiss if we didn't mention another class of header lines that are inserted by the SMTP *receiving* server. Indeed, the receiving server, upon receiving a message with RFC 822 and MIME header lines, appends a Received: header line to the top of the message; this header line specifies the name of the SMTP server that sent the message ("from"), the name of the SMTP server that received the message ("by") and the time at which the receiving server received the message. Thus, the message seen by the destination user takes the following form:

```
Received: from crepes.fr by hamburger.edu; 12 Oct 98 15:27:39 GMT
From: alice@crepes.fr
To: bob@hamburger.edu
```

```
Subject: Picture of yummy crepe.
MIME-Version: 1.0
Content-Transfer-Encoding: base64
Content-Type: image/jpeg

base64 encoded data .......
.......................................
.......base64 encoded data
```

Almost everyone who has used electronic mail has seen the `Received:` header line (along with the other header lines) preceding e-mail messages. (This line is often directly seen on the screen or when the message is sent to a printer.) You may have noticed that a single message sometimes has multiple `Received:` header lines and a more complex `Return-Path:` header line. This is because a message may be forwarded to more than one SMTP server in the path between sender and recipient. For example, if Bob has instructed his e-mail server `hamburger.edu` to forward all his messages to `sushi.jp`, then the message read by Bob's user agent would begin with something like:

```
Received: from hamburger.edu by sushi.jp; 12 Oct 98 15:30:01 GMT
Received: from crepes.fr by hamburger.edu; 12 Oct 98 15:27:39 GMT
```

These header lines provide the receiving user agent a trace of the SMTP servers visited as well as timestamps of when the visits occurred. You can learn more about the syntax of these header lines in the SMTP RFC, which is one of the more readable of the many RFCs.

2.4.4: Mail Access Protocols

Once SMTP delivers the message from Alice's mail server to Bob's mail server, the message is placed in Bob's mailbox. Throughout this discussion we have tacitly assumed that Bob reads his mail by logging onto the server host and then executes a mail reader directly on that host. Up until the early 1990s this was the standard way of doing things. But today a typical user reads mail with a user agent that executes on his or her local PC (or Mac), whether that PC be an office PC, a home PC, or a portable PC. By executing the user agent on a local PC, users enjoy a rich set of features, including the ability to view multimedia messages and attachments.

Given that Bob (the recipient) executes his user agent on his local PC, it is natural to consider placing a mail server on his local PC as well. There is a problem with this approach, however. Recall that a mail server manages mailboxes and runs the client and server sides of SMTP. If Bob's mail server were to reside on his local PC, then Bob's PC would have to remain constantly on, and connected to the Internet, in order to receive new mail, which can arrive at any time. This is impractical for the great majority of Internet users. Instead, a typical user runs a user agent on the local PC but accesses a mailbox from a shared mail server--a mail server that is always connected to the Internet, and that is shared with other users. The mail

server is typically maintained by the user's ISP (e.g., university or company).
With user agents running on users' local PCs and mail servers hosted by ISPs or institutional networks, a protocol is needed to allow the user agent and the mail server to communicate. Let us first consider how a message that originates at Alice's local PC makes its way to Bob's SMTP mail server. This task could simply be done by having Alice's user agent communicate directly with Bob's mail server in the language of SMTP. Alice's user agent would initiate a TCP connection to Bob's mail server, issue the SMTP handshaking commands, upload the message with the DATA command, and then close the connection. This approach, although perfectly feasible, is not commonly employed, primarily because it doesn't offer Alice's user agent any recourse to a crashed destination mail server. Instead, Alice's user agent initiates a SMTP dialogue to upload Alice's message to her own mail server (rather than with the recipient's mail server). Alice's mail server then establishes a new SMTP session with Bob's mail server and *relays* the message to Bob's mail server. If Bob's mail server is down, then Alice's mail server holds the message and tries again later. The SMTP RFC defines how the SMTP commands can be used to relay a message across multiple SMTP servers.
But there is still one missing piece to the puzzle! How does a recipient like Bob, running a user agent on his local PC, obtain his messages, which are sitting on a mail server within Bob's ISP? The puzzle is completed by introducing a special mail access protocol that transfers messages from Bob's mail server to his local PC. There are currently two popular mail access protocols: POP3 (Post Office Protocol-- Version 3) and IMAP (Internet Mail Access Protocol). We shall discuss both of these protocols below. Note that Bob's user agent can't use SMTP to obtain the messages because obtaining the messages is a pull operation whereas SMTP is a push protocol. Figure 2.17 provides a summary of the protocols that are used for Internet mail: SMTP is used to transfer mail from the sender's mail server to the recipient's mail server; SMTP is also used to transfer mail from the sender's user agent to the sender's mail server. POP3 or IMAP are used to transfer mail from the recipient's mail server to the recipient's user agent.

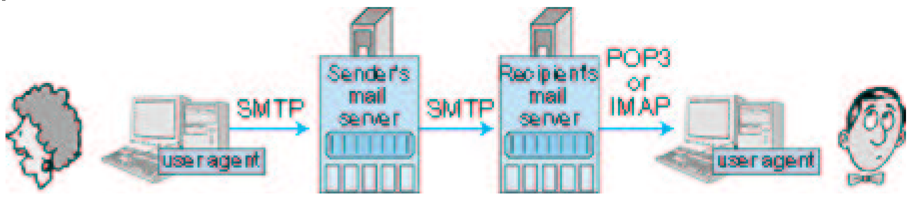

Figure 2.17: E-mail protocols and their communicating entities

POP3

POP3, defined in RFC 1939, is an extremely simple mail access protocol. Because the protocol is so simple, its functionality is rather limited. POP3 begins when the user agent (the client) opens a TCP connection to the mail server (the server) on port 110. With the TCP connection established, POP3 progresses through three phases: authorization, transaction, and update. During the first phase, authorization, the user agent sends a user name and a password to authenticate the user downloading the mail. During the second phase, transaction, the user agent retrieves messages. During the transaction phase, the user agent can also mark messages for deletion, remove deletion marks, and obtain mail statistics. The third phase, update, occurs after the client has issued the `quit` command, ending the POP3 session; at this time, the mail server deletes the messages that were marked for deletion.

In a POP3 transaction, the user agent issues commands, and the server responds to each command with a reply. There are two possible responses: `+OK` (sometimes followed by server-to-client data), whereby the server is saying that the previous command was fine; and `-ERR`, whereby the server is saying that something was wrong with the previous command.

The authorization phase has two principle commands: `user <user name>` and `pass <password>`. To illustrate these two commands, we suggest that you Telnet directly into a POP3 server, using port 110, and issue these commands. Suppose that `mailServer` is the name of your mail server. You will see something like:

```
telnet mailServer 110
+OK POP3 server ready
user alice
+OK
pass hungry
+OK user successfully logged on
```

If you misspell a command, the POP3 server will reply with an `-ERR` message.

Now let’s take a look at the transaction phase. A user agent using POP3 can often be configured (by the user) to "*download and delete*" or to "*download and keep.*" The sequence of commands issued by a POP3 user agent depend on which of these two modes the user agent is operating in. In the download-and-delete mode, the user agent will issue the `list`, `retr`, and `dele` commands. As an example, suppose the user has two messages in his or her mailbox. In the dialogue below, **C:** (standing for client), is the user agent and **S:** (standing for server) is the mail server. The transaction will look something like:

C: list

S: 1 498

S: 2 912

S: .

C: retr 1

S: (blah blah ...

S: ....

S: ....blah)

```
S: .
C: dele 1
C: retr 2
S: (blah blah ...
S: ................
S: ..........blah)
S: .
C: dele 2
C: quit
S: +OK POP3 server signing off
```

The user agent first asks the mail server to list the size of each of the stored messages. The user agent then retrieves and deletes each message from the server. Note that after the authorization phase, the user agent employed only four commands: list, retr, dele, and quit. The syntax for these commands is defined in RFC 1939. After processing the quit command, the POP3 server enters the update phase and removes messages 1 and 2 from the mailbox.

A problem with this download-and-delete mode is that the recipient, Bob, may be nomadic and want to access his mail from multiple machines, including the office PC, the home PC, and a portable computer. The download-and-delete mode scatters Bob's mail over all of his machines; in particular, if Bob first reads a message on a home PC, he will not be able to reread the message on his portable later in the evening. In the download-and-keep mode, the user agent leaves the messages on the mail server after downloading them. In this case, Bob can reread messages from different machines; he can access a message from work, and then access it again later in the week from home.

During a POP3 session between a user agent and the mail server, the POP3 server maintains some state information; in particular, it keeps track of which messages have been marked deleted. However, the POP3 server does not carry state information across POP3 sessions. For example, no message is marked for deletion at the beginning of each session. This lack of state information across sessions greatly simplifies the implementation of a POP3 server.

IMAP

Once Bob has downloaded his messages to the local machine using POP3, he can create mail folders and move the downloaded messages into the folders. Bob can then delete messages, move messages across folders, and search for messages (by sender name or subject). But this paradigm--folders and messages in the local machine--poses a problem for the nomadic user, who would prefer to maintain a folder hierarchy on a remote server that can be accessed from any computer. This is not possible with POP3.

To solve this and other problems, the Internet Mail Access Protocol (IMAP), defined in RFC 2060, was invented. Like POP3, IMAP is a mail access protocol. It has many more features than POP3, but it is also significantly more complex. (And thus the client and server side implementations are significantly more complex.) IMAP is designed to allow users to manipulate remote mailboxes as if they were local. In particular, IMAP enables Bob to create and maintain multiple message folders at the mail server. Bob can put messages in folders and move messages from one folder to another. IMAP also provides commands that allow Bob to search remote folders for messages matching specific criteria. One reason why an IMAP implementation is much more complicated than a POP3 implementation is that the IMAP server must maintain a folder hierarchy for each of its users. This state information persists across a particular user's successive accesses to the IMAP server. Recall that a POP3 server, by contrast, does not maintain anything about a particular user once the user quits the POP3 session.

Another important feature of IMAP is that it has commands that permit a user agent to obtain components of messages. For example, a user agent can obtain just the message header of a message or just one part of a multipart MIME message. This feature is useful when there is a low-bandwidth connection between the user agent and its mail server, for example, a wireless or slow-speed modem connection. With a low-bandwidth connection, the user may not want to download all the messages in its mailbox, particularly avoiding long messages that might contain, for example, an audio or video clip.

An IMAP session consists of the establishment of a connection between the client (that is, the user agent) and the IMAP server, an initial greeting from the server, and client/server interactions. The client/server interactions are similar to, but richer than, those of POP3. They consist of a client command, server data, and a server completion result response. The IMAP server is always in one of four states. In the *nonauthenticated state,* the initial state when the connection begins, the user must supply a user name and password before most commands will be permitted. In the *authenticated state,* the user must select a folder before sending commands that affect messages. In the *selected state,* the user can issue commands that affect messages (retrieve, move, delete, retrieve a part in a multipart message, and so on). Finally, the *logout state* is when the session is being terminated. The IMAP commands are organized by the state in which the command is permitted. You can read all about IMAP at the official IMAP site [IMAP 1999].

HTTP

More and more users today are using browser-based e-mail services such as Hotmail or Yahoo! Mail. With these servers, the user agent is an ordinary Web browser and the user communicates with its mailbox on its mail server via HTTP. When a recipient, such as Bob, wants to access the messages in his mailbox, the messages are sent from Bob's mail server to Bob's browser using the HTTP protocol rather than the POP3 or IMAP protocol. When a sender with an account on an HTTP-based e-mail server, such as Alice, wants to send a message, the message is sent from her browser to her mail server over HTTP rather than over SMTP. The mail server, however, still sends messages to, and receives messages from, other mail servers using SMTP. This solution to mail access is enormously convenient for the user on the go. The user need only to be able to access a browser in order to send and receive messages. The browser can be in an Internet cafe, in a friend's house, in a hotel room with a Web TV, and so on. As with IMAP, users can organize their messages in a hierarchy of folders on the remote server. In fact, Web-based e-mail is so convenient that it may replace POP3 and IMAP access in the upcoming years. Its principle disadvantage is that it can be slow, as the server is typically far from the client and interaction with the server is done through CGI scripts.

2.4.5: Continuous-media E-mail

Continuous-media (CM) e-mail is e-mail that includes audio or video. CM e-mail is appealing for asynchronous communication among friends and family. For example, a young child who cannot type would prefer to send an audio message to his or her grandparents. Furthermore, CM e-mail can be desirable in many corporate contexts, as an office worker may be able to record a CM message more quickly than typing a text message. (English can be spoken at a rate of 180 words per minute, whereas the average office worker types at a much slower rate.) Continuous-media e-mail resembles ordinary telephone voice-mail messaging in some respects. However, continuous-media e-mail is much more powerful. Not only does it provide the user with a graphical interface to the user's mailbox, but it also allows the user to annotate and reply to CM messages and to forward CM messages to a large number of recipients.

CM e-mail differs from traditional text mail in many ways. CM e-mail can have much larger messages, more stringent end-to-end delay requirements, and greater sensitivity to recipients with highly heterogeneous Internet access rates and local storage capabilities. Unfortunately, the current e-mail infrastructure has several inadequacies that obstruct the widespread adoption of CM e-mail [Turner 1999]. First, many existing mail servers do not have the capacity to store large CM objects; recipient mail servers typically reject such messages, which makes sending CM messages to such recipients impossible. Second, the existing mail paradigm of transporting *entire* messages to the recipient's mail server before recipient rendering can lead to excessive waste of bandwidth and storage. Indeed, stored CM is often not rendered in its entirety [Padhye 1999], so that bandwidth and recipient storage is wasted by receiving data that is never rendered. (For example, one can imagine listening to the first 15 seconds of a long audio e-mail from a rather long-winded colleague and then deciding to delete the remaining 20 minutes of the message without listening to it.) Third, current mail access protocols (POP3, IMAP, and HTTP) are inappropriate for streaming CM to recipients. (Streaming CM is discussed in detail in Chapter 6.) In particular, the current mail access protocols do not provide functionality that allows a user to pause/resume a message or to reposition within a message; furthermore, streaming over TCP often leads to poor reception (see Chapter 6). These inadequacies will hopefully be addressed in the upcoming years. Possible solutions are discussed in the literature [Gay 1997; Hess 1998; Schurmann 1996; and Turner 1999].

Online Book

2.5: DNS--The Internet's Directory Service

We human beings can be identified in many ways. For example, we can be identified by the names that appear on our birth certificates. We can be identified by our social security numbers. We can be identified by our driver's license numbers. Although each of these identifiers can be used to identify people, within a given context one identifier may be more appropriate than another. For example, the computers at the IRS (the infamous tax collecting agency in the United States) prefers to use fixed-length social security numbers rather than birth-certificate names. On the other hand, ordinary people prefer the more mnemonic birth-certificate names rather than social security numbers. (Indeed, can you imagine saying, "Hi. My name is 132-67-9875. Please meet my husband, 178-87-1146.")

Just as humans can be identified in many ways, so too can Internet hosts. One identifier for a host is its hostname. Hostnames--such as cnn.com, www.yahoo.com, gaia.cs.umass.edu, and surf.eurecom.fr--are mnemonic and are therefore appreciated by humans. However, hostnames provide little, if any, information about the location within the Internet of the host. (A hostname such as surf.eurecom.fr, which ends with the country code .fr, tells us that the host is probably in France, but doesn't say much more.) Furthermore, because hostnames can consist of variable-length alphanumeric characters, they would be difficult to process by routers. For these reasons, hosts are also identified by so-called IP addresses. We will discuss IP addresses in some detail in Chapter 4, but it is useful to say a few brief words about them now. An IP address consists of four bytes and has a rigid hierarchical structure. An IP address looks like 121.7.106.83, where each period separates one of the bytes expressed in decimal notation from 0 to 255. An IP address is hierarchical because as we scan the address from left to right, we obtain more and more specific information about where (that is, within which network, in the network of networks) the host is located in the Internet. (Similarly, when we scan a postal address from bottom to top, we obtain more and more specific information about where the residence is located.)

2.5.1: Services Provided by DNS

We have just seen that there are two ways to identify a host--by a hostname and by an IP address. People prefer the more mnemonic hostname identifier, while routers prefer fixed-length, hierarchically structured IP addresses. In order to reconcile these different preferences, we need a directory service that translates hostnames to IP addresses. This is the main task of the Internet's Domain Name System (DNS). The DNS is (1) a distributed database implemented in a hierarchy of name servers and (2) an application-layer protocol that allows hosts and name servers to communicate in order to provide the translation service. Name servers are often Unix machines running the Berkeley Internet Name Domain (BIND) software. The DNS protocol runs over UDP and uses port 53. Following this section, we provide interactive links to DNS programs that allow you to translate arbitrary hostnames, among other things. Click here to open it in a new window, or select it from the menu bar at the left. DNS is commonly employed by other application-layer protocols--including HTTP, SMTP, and FTP--to translate user-supplied host names to IP addresses. As an example, consider what happens when a browser (that is, an HTTP client), running on some user's machine, requests the URL www.someschool.edu/ index.html. In order for the user's machine to be able to send an HTTP request message to the Web server www.someschool.edu, the user's machine must obtain the IP address of www.someschool.edu. This is done as follows. The same user machine runs the client-side of the DNS application. The browser extracts the hostname, www.someschool.edu, from the URL and passes the hostname to the client-side of the DNS application. As part of a DNS query message, the DNS client sends a query containing the hostname to a DNS server. The DNS client eventually receives a reply, which includes the IP address for the hostname. The browser then opens a TCP connection to the HTTP server process located at that IP address. We see from this example that DNS adds an additional delay--sometimes substantial--to the Internet applications that use DNS. Fortunately, as we shall discuss below, the desired IP address is often cached in a "nearby" DNS name server, which helps to reduce the DNS network traffic as well as the average DNS delay.

Principles in Practice

DNS: Critical network functions via the client-server paradigm

Like HTTP, FTP, and SMTP, the DNS protocol is an application-layer protocol since it (1) runs between communicating end using the client-server paradigm, and (2) relies on an underlying end-to-end transport protocol to transfer DNS messages between communicating end systems. In another sense, however, the role of the DNS is quite different from Web, file transfer, and e-mail applications. Unlike these applications, the

DNS is not an application with which a user directly interacts. Instead, the DNS provides a core Internet function--namely, translating hostnames to their underlying IP addresses, for user applications and other software in the Internet. We noted earlier in Section 1.2 that much of the complexity in the Internet architecture is located at the "edges" of the network. The DNS, which implements the critical name-to-address translation process using clients and servers located at the edge of the network, is yet another example of that design philosophy.

DNS provides a few other important services in addition to translating hostnames to IP addresses:

- Host aliasing. A host with a complicated hostname can have one or more alias names. For example, a hostname such as relay1.west-coast. enterprise.com could have, say, two aliases such as enterprise.com and www.enterprise.com. In this case, the hostname relay1.west-coast.enterprise.com is said to be a canonical hostname. Alias hostnames, when present, are typically more mnemonic than a canonical hostname. DNS can be invoked by an application to obtain the canonical hostname for a supplied alias hostname as well as the IP address of the host.

- Mail server aliasing. For obvious reasons, it is highly desirable that e-mail addresses be mnemonic. For example, if Bob has an account with Hotmail, Bob's e-mail address might be as simple as bob@hotmail.com. However, the hostname of the Hotmail mail server is more complicated and much less mnemonic than simply hotmail.com (for example, the canonical hostname might be something like relay1.west-coast.hotmail.com). DNS can be invoked by a mail application to obtain the canonical hostname for a supplied alias hostname as well as the IP address of the host. In fact, DNS permits a company's mail server and Web server to have identical (aliased) hostnames; for example, a company's Web server and mail server can both be called enterprise.com.

- Load distribution. Increasingly, DNS is also being used to perform load distribution among replicated servers, such as replicated Web servers. Busy sites, such as cnn.com, are replicated over multiple servers, with each server running on a different end system, and having a different IP address. For replicated Web servers, a *set* of IP addresses is thus associated with one canonical hostname. The DNS database contains this set of IP addresses. When clients make a DNS query for a name mapped to a set of addresses, the server responds with the entire set of IP addresses, but rotates the ordering of the addresses within each reply. Because a client typically sends its HTTP request message to the IP address that is listed first in the set, DNS rotation distributes the traffic among all the replicated servers. DNS rotation is also used for e-mail so that multiple mail servers can have the same alias name. Recently, companies such as Akamai [Akamai 2000] have used DNS in more sophisticated ways to provide Web content distribution.

The DNS is specified in RFC 1034 and RFC 1035, and updated in several additional RFCs. It is a complex system, and we only touch upon key aspects of its operation here. The interested reader is referred to these RFCs and a book by Abitz and Liu [Abitz 1993].

2.5.2: Overview of How DNS Works

We now present a high-level overview of how DNS works. Our discussion will focus on the hostname-to-IP-address translation service. From the client's perspective, the DNS is a black box. The client sends a DNS query message into the black box, specifying the hostname that needs to be translated to an IP address. On many Unix-based machines, `gethostbyname()` is the library routine that an application calls in order to issue the query message. In Section 2.7, we will present a Java program that begins by issuing a DNS query. After a delay, ranging from milliseconds to tens of seconds, the client receives a DNS reply message that provides the desired mapping. Thus, from the client's perspective, DNS is a simple, straightforward translation service. But in fact, the black box that implements the service is complex, consisting of a large number of name servers distributed around the globe, as well as an application-layer protocol that specifies how the name servers and querying hosts communicate.

A simple design for DNS would have one Internet name server that contains all the mappings. In this centralized design, clients simply direct all queries to the single name server, and the name server responds directly to the querying clients. Although the simplicity of this design is attractive, it is completely inappropriate for today's Internet, with its vast (and growing) number of hosts. The problems with a centralized design include:

- A single point of failure. If the name server crashes, so too does the entire Internet!
- Traffic volumes. A single name server would have to handle all DNS queries (for all the HTTP requests and e-mail messages generated from millions of hosts).
- Distant centralized database. A single name server cannot be "close to" all the querying clients. If we put the single name server in New York City, then all queries from Australia must travel to the other side of the globe, perhaps over slow and congested links. This can lead to significant delays (thereby increasing the "world wide wait" for the Web and other applications).
- Maintenance. The single name server would have to keep records for all Internet hosts. Not only would this centralized database be huge, but it would have to be updated frequently to account for every

new host. There are also authentication and authorization problems associated with allowing any user to register a host with the centralized database.

In summary, a centralized database in a single name server simply *doesn't scale.* Consequently, the DNS is distributed by design. In fact, the DNS is a wonderful example of how a distributed database can be implemented in the Internet.
In order to deal with the issue of scale, the DNS uses a large number of name servers, organized in a hierarchical fashion and distributed around the world. No one name server has all of the mappings for all of the hosts in the Internet. Instead, the mappings are distributed across the name servers. To a first approximation, there are three types of name servers: local name servers, root name servers, and authoritative name servers. These name servers, again to a first approximation, interact with each other and with the querying host as follows:

- Local name servers. Each ISP--such as a university, an academic department, an employee's company, or a residential ISP--has a local name server (also called a default name server). When a host issues a DNS query message, the message is first sent to the host's local name server. The IP address of the local name server is typically configured by hand in a host. (On a Windows 95/98 machine, you can find the IP address of the local name server used by your PC by opening the Control Panel, and then selecting "Network," then selecting an installed TCP/IP component, and then selecting the DNS configuration folder tab.) The local name server is typically "close to" the client; in the case of an institutional ISP, it may be on the same LAN as the client host; for a residential ISP, the name server is typically separated from the client host by no more than a few routers. If a host requests a translation for another host that is part of the same local ISP, then the local name server will be able to immediately provide the requested IP address. For example, when the host surf.eurecom.fr requests the IP address for baie.eurecom.fr, the local name server at Eurecom will be able to provide the requested IP address without contacting any other name servers.

- Root name servers. In the Internet there are a dozen or so root name servers, most of which are currently located in North America. A February 1998 map of the root servers is shown in Figure 2.18. When a local name server cannot immediately satisfy a query from a host (because it does not have a record for the hostname being requested), the local name server behaves as a DNS client and queries one of the root name servers. If the root name server has a record for the hostname, it sends a DNS reply message to the local name server, and the local name server then sends a DNS reply to the querying host. But the root name server may not have a record for the hostname. Instead, the rootname server knows the IP address of an "authoritative name server" that has the mapping for that particular hostname.

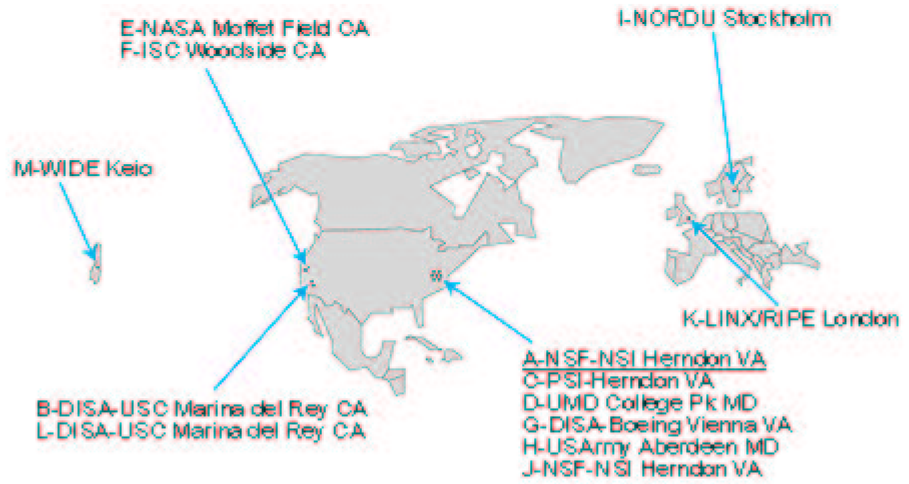

Figure 2.18: A February 1998 map of the DNS root servers. Data from the WIA alliance Web site (http://www.wia.org/)

- Authoritative name servers. Every host is registered with an authoritative name server. Typically, the authoritative name server for a host is a name server in the host's local ISP. (Actually, each host is required to have at least two authoritative name servers, in case of failures.) By definition, a name server is authoritative for a host if it always has a DNS record that translates the host's hostname to that host's IP address. When an authoritative name server is queried by a root server, the authoritative name server responds with a DNS reply that contains the requested mapping. The root server then forwards the mapping to the local name server, which in turn forwards the mapping to the requesting host. Many name servers act as both local and authoritative name servers.

Let's take a look at a simple example. Suppose the host surf.eurecom.fr desires the IP address of gaia.cs.umass.edu. Also suppose that Eurecom's local name server is called dns.eurecom.fr and that an authoritative name server for gaia.cs.umass.edu is called dns.umass.edu. As shown in Figure 2.19, the host surf.eurecom.fr first sends a DNS query message to its local name server, dns.eurecom.fr. The query message contains the hostname to be translated, namely, gaia.cs.umass.edu. The local name server forwards the query message to a root name server. The root name server forwards the query message to the name server that is authoritative for all the hosts in the domain umass.edu, for example, to dns.umass.edu. The authoritative name server then sends the desired mapping to the querying host, via the root name server and the local name server. Note that in this example, in order to obtain the mapping for one hostname, six DNS messages were sent: three query messages and three reply messages.

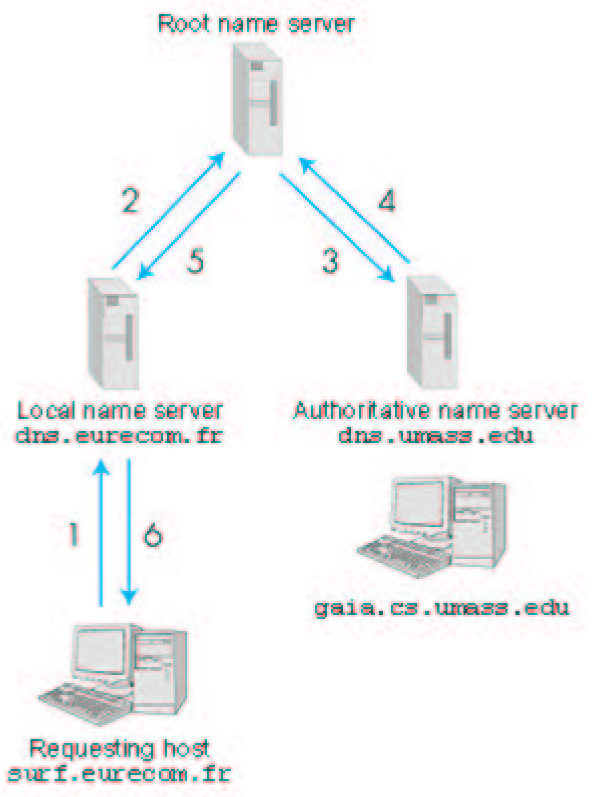

Figure 2.19: Recursive queries to obtain the mapping for gaia.cs.umass.edu

Our discussion up to this point has assumed that the root name server knows the IP address of an authoritative name server for *every* hostname. This assumption may be incorrect. For a given hostname, the root name server may only know the IP address of an intermediate name server that in turn knows the IP address of an authoritative name server for the hostname. To illustrate this, consider once again the above example with the host surf.eurecom.fr querying for the IP address of gaia.cs.umass.edu. Suppose now that the University of Massachusetts has a name server for the university, called dns.umass.edu. Also suppose that each of the departments at the University of Massachusetts has its own name server, and that each departmental name server is authoritative for all the hosts in the department. As shown in Figure 2.20, when the root name server receives a query for a host with hostname ending with umass.edu, it forwards the query to the name server dns.umass.edu. This name server forwards all queries with hostnames ending with .cs.umass.edu to the name server dns.cs.umass.edu, which is authoritative for all hostnames ending with .cs.umass.edu. The authoritative name server sends the desired mapping to the intermediate name server, dns. umass.edu, which forwards the mapping to the root name server, which forwards the mapping to the local name server, dns.eurecom.fr, which forwards the mapping to the requesting host! In this example, eight DNS messages are sent. Actually, even more DNS messages can be sent in order to translate a single hostname--there can be two or more intermediate name servers in the chain between the root name server and the authoritative name server!

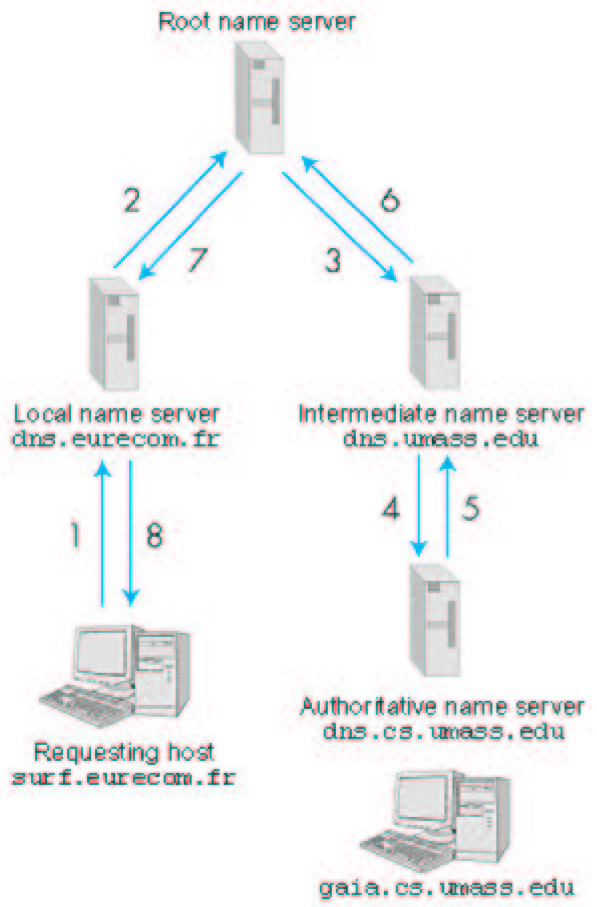

Figure 2.20: Recursive queries with an intermediate name server between the root and authoritative name servers

The examples up to this point assumed that all queries are so-called recursive queries. When a host or name server A makes a recursive query to a name server B, then name server B obtains the requested mapping *on behalf* of A and then forwards the mapping to A. The DNS protocol also allows for iterative queries at any step in the chain between requesting host and authoritative name server. When a name server A makes an iterative query to name server B, if name server B does not have the requested mapping, it immediately sends a DNS reply to A that contains the IP address of the next name server in the chain, say, name server C. Name server A then sends a query directly to name server C.

In the sequence of queries that are required to translate a hostname, some of the queries can be iterative and others recursive. Such a combination of recursive and iterative queries is illustrated in Figure 2.21. Typically, all queries in the query chain are recursive except for the query from the local name server to the root name server, which is iterative. (Because root servers handle huge volumes of queries, it is preferable to use the less burdensome iterative queries for root servers.)

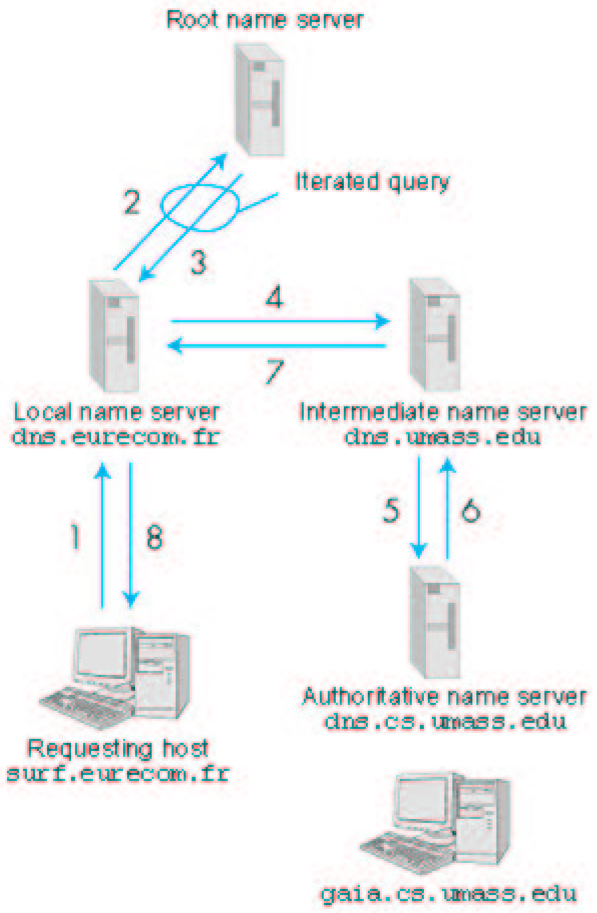

Figure 2.21: A query chain with recursive and iterative queries

Our discussion this far has not touched on one important feature of the DNS: DNS caching. In reality, DNS extensively exploits caching in order to improve the delay performance and to reduce the number of DNS messages in the network. The idea is very simple. When a name server receives a DNS mapping for some hostname, it caches the mapping in local memory (disk or RAM) while passing the message along the name server chain. Given a cached hostname/IPaddress translation pair, if another query arrives to the name server for the same hostname, the name server can provide the desired IP address, even if it is not authoritative for the hostname. In order to deal with ephemeral hosts, a cached record is discarded after a period of time (often set to two days). As an example, suppose that surf. eurecom.fr queries the DNS for the IP address for the hostname cnn.com. Furthermore, suppose that a few hours later, another Eurecom host, say, baie. eurecom.fr, also queries DNS with the same hostname. Because of caching, the local name server at Eurecom will be able to immediately return the IP address of cnn.com to the requesting host without having to query name servers on another continent. Any name server may cache DNS mappings.

2.5.3: DNS Records

The name servers that together implement the DNS distributed database, store resource records (RR) for the hostname to IP address mappings. Each DNS reply message carries one or more resource records. In this and the following subsection, we provide a brief overview of DNS resource records and messages; more details can be found in [Abitz 1993] or in the DNS RFCs [RFC 1034; RFC 1035].

A resource record is a four-tuple that contains the following fields:

(Name, Value, Type, TTL)

TTL is the time to live of the resource record; it determines the time at which a resource should be removed from a cache. In the example records given below, we will ignore the TTL field. The meaning of Name and Value depend on Type:

- If Type=A, then Name is a hostname and Value is the IP address for the hostname. Thus, a Type A record provides the standard hostname to IP address mapping. As an example, (relay1.bar.foo.com, 145.37.93.126, A) is a Type A record.

- If Type=NS, then Name is a domain (such as foo.com) and Value is the hostname of an authoritative name server that knows how to obtain the IP addresses for hosts in the domain. This record is used to route DNS queries further along in the query chain. As an example, (foo.com, dns.foo.com, NS) is a Type NS record.

- If Type=CNAME, then Value is a canonical hostname for the alias hostname Name. This record can provide querying hosts the canonical name for a hostname. As an example, (foo.com, relay1.bar.foo.com, CNAME) is a CNAME record.

- If Type=MX, then Value is a hostname of a mail server that has an alias hostname Name. As an example, (foo.com. mail.bar.foo.com, MX) is an MX record. MX records allow the hostnames of mail servers to have simple aliases.

If a name server is authoritative for a particular hostname, then the name server will contain a Type A record for the hostname. (Even if the name server is not authoritative, it may contain a Type A record in its cache.) If a server is not authoritative for a hostname, then the server will contain a Type NS record for the domain that includes the hostname; it will also contain a Type A record that provides the IP address of the name server in the Value field of the NS record. As an example, suppose a root server is not authoritative for the host gaia.cs.umass.edu. Then the root server will contain a record for a domain that includes the host cs.umass.edu, for example, (umass.edu, dns.umass.edu, NS). The root server would also contain a Type A record, which maps the name server dns.umass.edu to an IP address, for example, (dns.umass.edu, 128.119.40.111, A).

2.5.4: DNS Messages

Earlier in this section we alluded to DNS query and reply messages. These are the only two kinds of DNS messages. Furthermore, both request and reply messages have the same format, as shown in Figure 2.22.

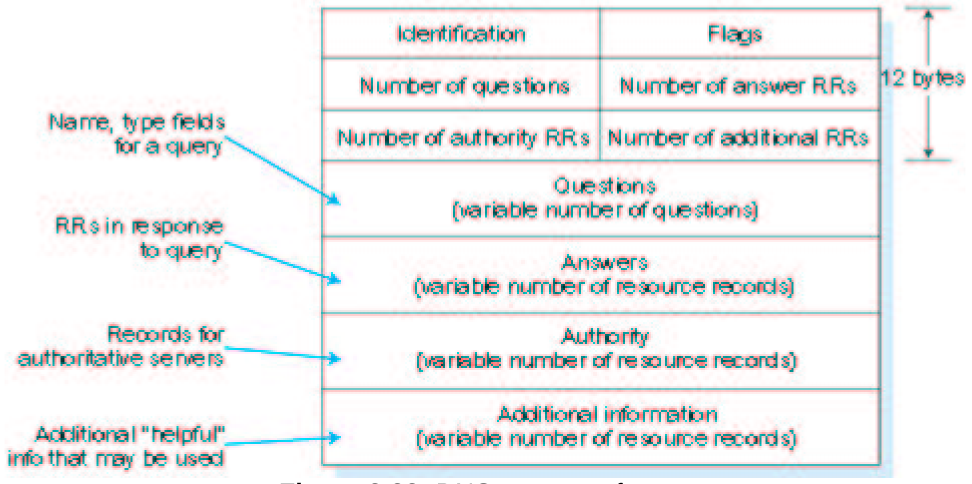
Identification
Flags
Number of questions
Number of answer RRs
Number of authority RRs
Number of additional RRs
12 bytes
Name, type fields for a query
Questions (variable number of questions)
RRs in response to query
Answers (variable number of resource records)
Records for authoritative servers
Authority (variable number of resource records)
Additional "helpful" info that may be used
Additional information (variable number of resource records)

Figure 2.22: DNS message format

The semantics of the various fields in a DNS message is as follows:

- The first 12 bytes is the *header section,* which has a number of fields. The first field is a 16-bit number that identifies the query. This identifier is copied into the reply message to a query, allowing the client to match received replies with sent queries. There are a number of flags in the flag field. A one-bit query/reply flag indicates whether the message is a query (0) or a reply (1). A one-bit authoritative flag is set in a reply message when a name server is an authoritative server for a queried name. A one-bit recursion-desired flag is set when a client (host or name server) desires that the name server perform recursion when it doesn't have the record. A one-bit recursion-available field is set in a reply if the name server supports recursion. In the header, there are also four "number of" fields. These fields indicate the number of occurrences of the four types of "data" sections that follow the header.

- The *question section* contains information about the query that is being made. This section includes (1) a name field that contains the name that is being queried, and (2) a type field that indicates the type of question being asked about the name (for example, a host address associated with a name--type A, or the mail server for a name--type MX).

- In a reply from a name server, the *answer section* contains the resource records for the name that was originally queried. Recall that in each resource record there is the Type (for example, A, NS, CSNAME, and MX), the Value, and the TTL. A reply can return multiple RRs in the answer, since a hostname can have multiple IP addresses (for example, for replicated Web servers, as discussed earlier in this section).

- The *authority section* contains records of other authoritative servers.
- The *additional section* contains other "helpful" records. For example, the answer field in a reply to an MX query will contain the hostname of a mail server associated with the alias name Name. The additional section will contain a Type A record providing the IP address for the canonical hostname of the mail server.

The discussion above has focused on how data is retrieved from the DNS database. You might be wondering how data gets into the database in the first place? Until recently, the contents of each DNS server was configured statically, for example, from a configuration file created by a system manager. More recently, an UPDATE option has been added to the DNS protocol to allow data to be dynamically added or deleted from the database via DNS messages. RFC 2136 specifies DNS dynamic updates. DNSNet provides a nice collection of documents pertaining to DNS [DNSNet 1999]. The Internet Software Consortium provides many resources for BIND, a popular public-domain name server for Unix machines [BIND 2000].

Online Book

Interactive Programs for Exploring DNS

There are at least three client programs available for exploring the contents of name servers in the Internet. The most widely available program is **nslookup**; two other programs, which are a little more powerful than nslookup, are **dig** and **host**. Lucky for us, several institutions and individuals have made these client programs available through Web. browsers.

We stongly encourage you to get your hands dirty and play with these programs. They can give significant insight into how DNS works. All of these programs mimic DNS clients. They send a DNS query message to a name server (which can often be supplied by the user), and they receive a corresponding DNS response. They then extract information (e.g., IP addresses, whether the response is authoritative, etc.) and present the information to the user.

nslookup

Some of the nslookup sites provide only the basic nslookup service, i.e., they allow you to enter a hostname and they return an IP address. Visit some of the nslookup sights below and try entering hostnames for popular hosts (such as cnn.com or www.microsoft.com) as well as hostnames for the less popular hosts. You will see that the popular hostnames typically return numerous IP addresses, because the site is replicated in numerous servers. (See the discussion in Section 2.5 on DNS rotation.) Some of the nslookup sites also return the hostname and IP address of the name server that provides the information. Also, some of the nslookup sites indicate whether the result is non-authoritative (i.e., obtained from a cache).

- http://www.infobear.com/nslookup-form.cgi

Some of the nslookup sites allow the user to supply more information. For example, the user can request to receive the canonical hostname and IP address for a mail server. And the user can also indicate the name server at which it wants the chain of queries to begin.

- http://ipalloc.utah.edu/HTML_Docs/NSLookup.html

dig and host
The programs dig and host allow the user to further refine the query by indicating, for example, whether the query should be recursive or interative. There are currently not as many Web sites that provide the dig and host service. But there are a few:

- http://www.toetag.com/cgi-bin/host
- http://www.netliner.com/dig.html

Online Book

2.6: Socket Programming with TCP

This and the subsequent sections provide an introduction to network application development. Recall from Section 2.1 that the core of a network application consists of a pair of programs--a client program and a server program. When these two programs are executed, a client and server process are created, and these two processes communicate with each other by reading from and writing to sockets. When creating a network application, the developer’s main task is to write the code for both the client and server programs.

There are two sorts of client/server applications. One sort is a client/server application that is an *implementation* of a protocol standard defined in an RFC. For such an implementation, the client and server programs must conform to the rules dictated by the RFC. For example, the client program could be an implementation of the FTP client, defined in RFC 959, and the server program could be an implementation of the FTP server, also defined in RFC 959. If one developer writes code for the client program and an independent developer writes code for the server program, and both developers carefully follow the rules of the RFC, then the two programs will be able to interoperate. Indeed, most of today's network applications involve communication between client and server programs that have been created by independent developers. (For example, a Netscape browser communicating with an Apache Web server, or an FTP client on a PC uploading a file to a Unix FTP server.) When a client or server program implements a protocol defined in an RFC, it should use the port number associated with the protocol. (Port numbers were briefly discussed in Section 2.1. They will be covered in more detail in the next chapter.)

The other sort of client/server application is a *proprietary* client/server application. In this case the client and server programs do not necessarily conform to any existing RFC. A single developer (or development team) creates both the client and server programs, and the developer has complete control over what goes in the code. But because the code does not implement a public-domain protocol, other independent developers will not be able to develop code that interoperates with the application. When developing a proprietary application, the developer must be careful not to use one of the well-known port numbers defined in the RFCs.

In this and the next section, we will examine the key issues in developing a proprietary client/server application. During the development phase, one of the first decisions the developer must make is whether the application is to run over TCP or over UDP. Recall that TCP is connection-oriented and provides a *reliable byte- stream channel* through which data flows between two end systems. UDP is connectionless and sends *independent packets* of data from one end system to the other, without any guarantees about delivery.

In this section we develop a simple client application that runs over TCP; in the subsequent section, we develop a simple client application that runs over UDP. We present these simple TCP and UDP applications in Java. We could have written the code in C or C++, but we opted for Java for several reasons. First, the applications are more neatly and cleanly written in Java; with Java there are fewer lines of code, and each line can be explained to the novice programmer without much difficulty. Second, client/server programming in Java is becoming increasingly popular, and may even become the norm in upcoming years. Java is platform-independent, it has exception mechanisms for robust handling of common problems that occur during I/O and networking operations, and its threading facilities provide a way to easily implement powerful servers. But there is no need to be frightened if you are not familiar with Java. You should be able to follow the code if you have experience programming in another language.

For readers who are interested in client/server programming in C, there are several good references available [Stevens 1997; Frost 1994; Kurose 1996].

2.6.1: Socket Programming with TCP

Recall from Section 2.1 that processes running on different machines communicate with each other by sending messages into sockets. We said that each process was analogous to a house and the process's socket is analogous to a door. As shown in Figure 2.23, the socket is the door between the application process and TCP. The application developer has control of everything on the application-layer side of the socket; however, it has little control of the transport-layer side. (At the very most, the application developer has the ability to fix a few TCP parameters, such as maximum buffer size and maximum segment sizes.)

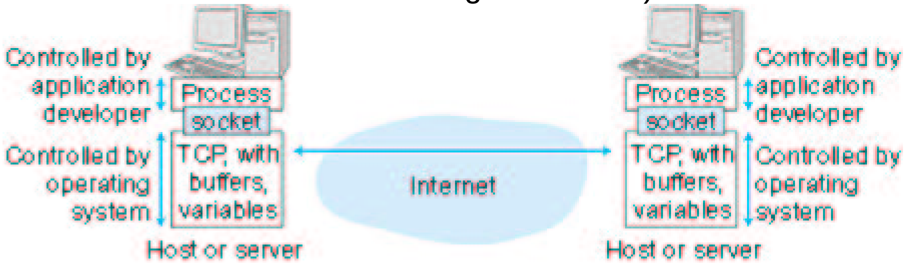
Controlled by application developer
Process
socket
Controlled by operating system
TCP, with buffers, variables
Internet
Host or server
Process
socket
Controlled by application developer
TCP, with buffers, variables
Controlled by operating system
Host or server

Figure 2.23: Process communicating through TCP sockets

Now let's take a little closer look at the interaction of the client and server programs. The client has the job of initiating contact with the server. In order for the server to be able to react to the client's initial contact, the server has to be ready. This implies two things. First, the server program cannot be dormant; it must be running as a process before the client attempts to initiate contact. Second, the server program must have some sort of door (that is, socket) that welcomes some initial contact from a client running on an arbitrary machine. Using our house/door analogy for a process/socket, we will sometimes refer to the client's initial contact as "knocking on the door."

With the server process running, the client process can initiate a TCP connection to the server. This is done in the client program by creating a socket object. When the client creates its socket object, it specifies the address of the server process, namely, the IP address of the server and the port number of the process. Upon creation of the socket object, TCP in the client initiates a three-way handshake and establishes a TCP connection with the server. The three-way handshake is completely transparent to the client and server programs.

During the three-way handshake, the client process knocks on the welcoming door of the server process. When the server "hears" the knocking, it creates a new door (that is, a new socket) that is dedicated to that particular client. In our example below, the welcoming door is a

ServerSocket object that we call the welcomeSocket. When a client knocks on this door, the program invokes welcomeSocket's accept() method, which creates a new door for the client. At the end of the handshaking phase, a TCP connection exists between the client's socket and the server's new socket. Henceforth, we refer to the new socket as the server's connection socket.

From the application's perspective, the TCP connection is a direct virtual pipe between the client's socket and the server's connection socket. The client process can send arbitrary bytes into its socket; TCP guarantees that the server process will receive (through the connection socket) each byte in the order sent. Furthermore, just as people can go in and out the same door, the client process can also receive bytes from its socket and the server process can also send bytes into its connection socket. This is illustrated in Figure 2.24.

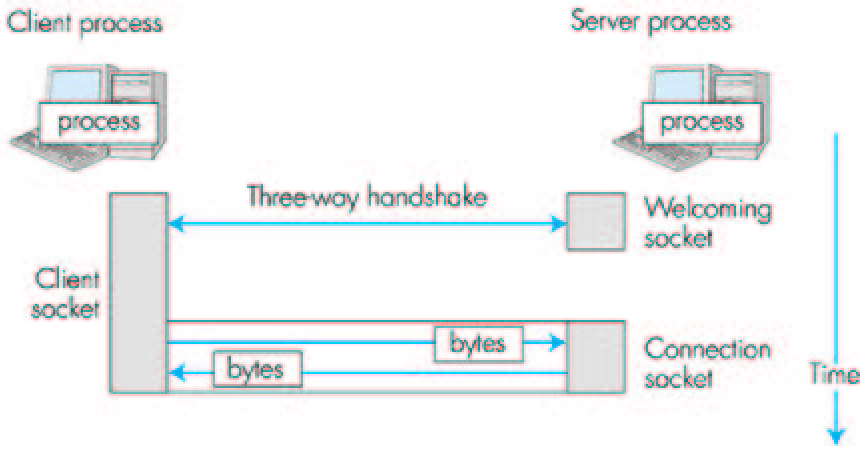

Figure 2.24: Client socket, welcoming socket, and connection socket

Because sockets play a central role in client/server applications, client/server application development is also referred to as socket programming. Before providing our example client/server application, it is useful to discuss the notion of a stream. A stream is a sequence of characters that flow into or out of a process. Each stream is either an input stream for the process or an output stream for the process. If the stream is an input stream, then it is attached to some input source for the process, such as standard input (the keyboard) or a socket into which data flow from the Internet. If the stream is an output stream, then it is attached to some output source for the process, such as standard output (the monitor) or a socket out of which data flow into the Internet.

2.6.2: An Example Client/Server Application in Java

We will use the following simple client/server application to demonstrate socket programming for both TCP and UDP:

A client reads a line from its standard input (keyboard) and sends the line out its socket to the server.

The server reads a line from its connection socket.

The server converts the line to uppercase.

The server sends the modified line out its connection socket to the client. The client reads the modified line from its socket and prints the line on its standard output (monitor).
Let us begin with the case of a client and server communicating over a connection- oriented (TCP) transport service. Figure 2.25 illustrates the main socket-related activity of the client and server.

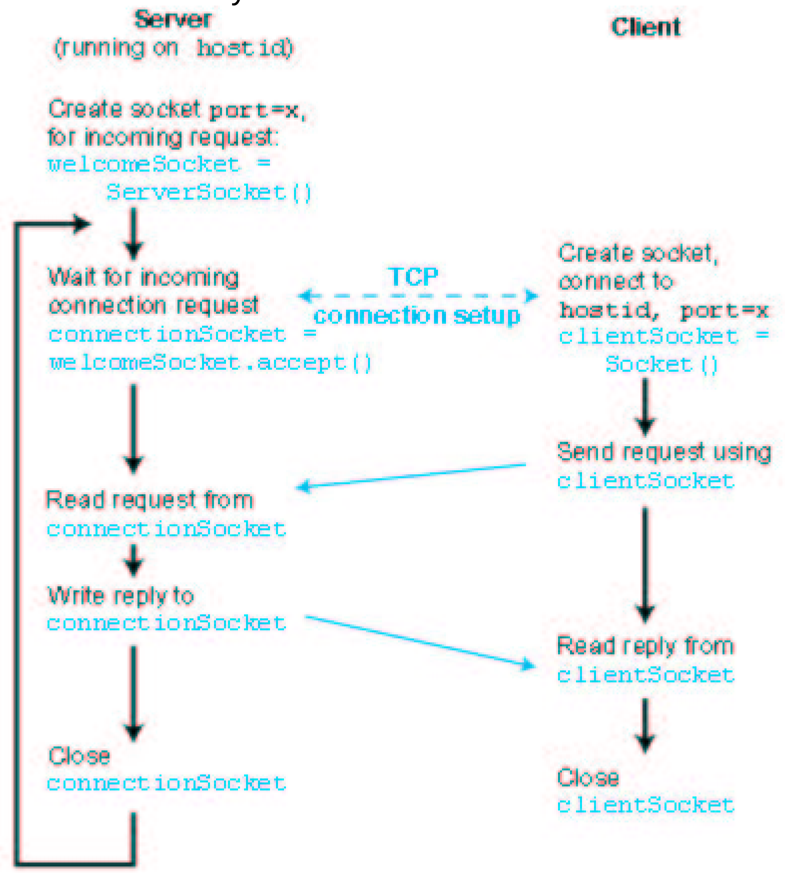
Server
(running on hostid)
Client
Create socket port=x, for incoming request:
welcomeSocket = ServerSocket()
Wait for incoming connection request
connectionSocket = welcomeSocket.accept()
TCP connection setup
Create socket, connect to hostid, port=x
clientSocket = Socket()
Send request using clientSocket
Read request from connectionSocket
Write reply to connectionSocket
Read reply from clientSocket
Close connectionSocket
Close clientSocket

Figure 2.25: The client/server application, using connection-oriented transport services

Next we provide the client/server program pair for a TCP implementation of the application. We provide a detailed, line-by-line analysis after each program. The client program is called TCPClient.java, and the server program is called TCPServer.java. In order to emphasize the key issues, we intentionally provide code that is to the point but not bullet proof. "Good code" would certainly have a few more auxiliary lines.
Once the two programs are compiled on their respective hosts, the server program is first executed at the server, which creates a process at the server. As discussed above, the server process waits to be contacted by a client process. When the client program is executed, a process is created at the client, and this process contacts the server and establishes a TCP connection with it. The user at the client may then "use" the application to send a line and then receive a capitalized version of the line.

TCPClient.java

Here is the code for the client side of the application:

```
import java.io.*;
import java.net.*;
class TCPClient {
 public static void main(String argv[]) throws Exception
 {
  String sentence;
  String modifiedSentence;
  BufferedReader inFromUser =
   new BufferedReader(
     new InputStreamReader(System.in));
  Socket clientSocket = new Socket("hostname", 6789);
  DataOutputStream outToServer =
   new DataOutputStream(
     clientSocket.getOutputStream());
  BufferedReader inFromServer =
   new BufferedReader(new InputStreamReader(
     clientSocket.getInputStream()));
  sentence = inFromUser.readLine();
  outToServer.writeBytes(sentence + '\n');
  modifiedSentence = inFromServer.readLine();
  System.out.println("FROM SERVER: " +
          modifiedSentence);
  clientSocket.close();
 }
}
```

The program TCPClient creates three streams and one socket, as shown in Figure 2.26.

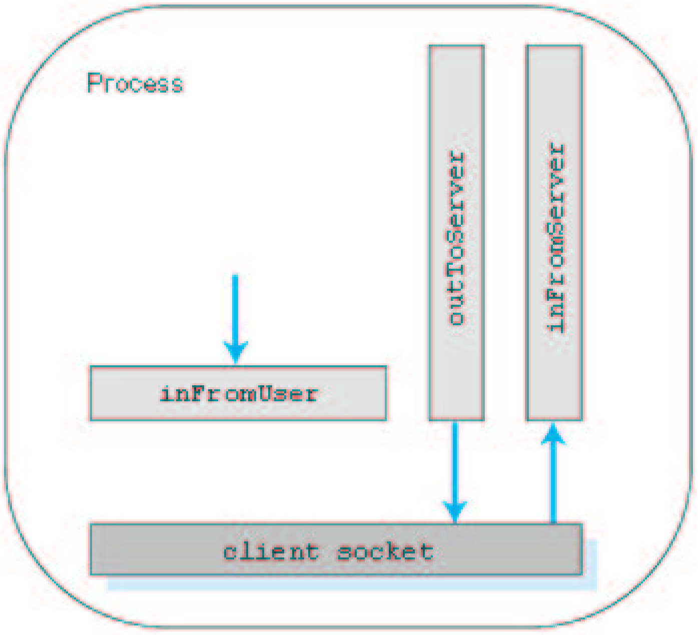

Figure 2.26: TCPClient has three streams and one socket

The socket is called clientSocket. The stream inFromUser is an input stream to the program; it is attached to the standard input (that is, the keyboard). When the user types characters on the keyboard, the characters flow into the stream inFromUser. The stream inFromServer is another input stream to the program; it is attached to the socket. Characters that arrive from the network flow into the stream inFromServer. Finally, the stream outToServer is an output stream from the program; it is also attached to the socket. Characters that the client sends to the network flow into the stream outToServer.

Let's now take a look at the various lines in the code.

```
import java.io.*;
import java.net.*;
```

java.io and java.net are java packages. The java.io package contains classes for input and output streams. In particular, the java.io package contains the BufferedReader and DataOutputStream classes, classes that the program uses to create the three streams previously illustrated. The java.net package provides classes for network support. In particular, it contains the Socket and ServerSocket classes. The clientSocket object of this program is derived from the Socket class.

```
class TCPClient {
  public static void main(String argv[]) throws Exception
    {......}
}
```

So far, what we've seen is standard stuff that you see at the beginning of most Java code. The first line is the beginning of a class definition block. The keyword class begins the class definition for the class named TCPClient.

A class contains variables and methods. The variables and methods of the class are embraced by the curly brackets that begin and end the class definition block. The class TCPClient has no class variables and exactly one method, the main() method. Methods are similar to the functions or procedures in languages such as C; the main method in the Java language is similar to the main function in C and C++. When the Java interpreter executes an application (by being invoked upon the application's controlling class), it starts by calling the class's main method. The main method then calls all the other methods required to run the application. For this introduction into socket programming in Java, you may ignore the keywords public, static, void, main, and throws Exceptions (although you must include them in the code).

```
String sentence;
String modifiedSentence;
```

These above two lines declare objects of type String. The object sentence is the string typed by the user and sent to the server. The object modifiedSentence is the string obtained from the server and sent to the user's standard output.

The above line creates the stream object inFromUser of type Buffered Reader. The input stream is initialized with System.in, which attaches the stream to the standard input. The command allows the client to read text from its keyboard.

```
Socket clientSocket = new Socket("hostname", 6789);
```

The above line creates the object clientSocket of type Socket. It also initiates the TCP connection between client and server. The string "hostname" must be replaced with the host name of the server (for example, "fling. seas.upenn.edu"). Before the TCP connection is actually initiated, the client performs a DNS look-up on the hostname to obtain the host's IP address. The number 6789 is the port number. You can use a different port number; but you must make sure that you use the same port number at the server side of the application. As discussed earlier, the host's IP address along with the application's port number identifies the server process.

```
DataOutputStream outToServer =
  new DataOutputStream(clientSocket.getOutputStream());
BufferedReader inFromServer =
  new BufferedReader(new inputStreamReader(
              clientSocket.getInputStream()));
```

The above two lines create stream objects that are attached to the socket. The outToServer stream provides the process output to the socket. The inFromServer stream provides the process input from the socket (see Figure 2.26).

```
sentence = inFromUser.readLine();
```

The above line places a line typed by the user into the string sentence. The string sentence continues to gather characters until the user ends the line by typing a carriage return. The line passes from standard input through the stream inFromUser into the string sentence.

```
outToServer.writeBytes(sentence + '\n');
```

The above line sends the string sentence augmented with a carriage return into the outToServer stream. The augmented sentence flows through the client's socket and into the TCP pipe. The client then waits to receive characters from the server.

```
modifiedSentence = inFromServer.readLine();
```

When characters arrive from the server, they flow through the stream inFromServer and get placed into the string modifiedSentence. Characters continue to accumulate in modifiedSentence until the line ends with a carriage return character.

```
System.out.println("FROM SERVER " + modifiedSentence);
```

The above line prints to the monitor the string modifiedSentence returned by the server.

```
clientSocket.close();
```

This last line closes the socket and, hence, closes the TCP connection between the client and the server. It causes TCP in the client to send a TCP message to TCP in the server (see Section 3.5).

TCPServer.java

Now let's take a look at the server program.

```
import java.io.*;
import java.net.*;
class TCPServer {
  public static void main(String argv[]) throws Exception
    {
      String clientSentence;
      String capitalizedSentence;
      ServerSocket welcomeSocket = new Server Socket
     (6789);

      while(true) {
        Socket connectionSocket = welcomeSocket.
        accept();
        BufferedReader inFromClient =
          new BufferedReader(new InputStreamReader(
            connectionSocket.getInputStream()));
        DataOutputStream outToClient =
          new DataOutputStream(
            connectionSocket.getOutputStream());
        clientSentence = inFromClient.readLine();
        capitalizedSentence =
            clientSentence.toUpperCase() + '\n';
            outToClient.writeBytes(capitalizedSentence);
         }
      }
}
```

TCPServer has many similarities with TCPClient. Let’s now take a look at the lines in TCPServer.java. We will not comment on the lines that are identical or similar to commands in TCPClient.java.

The first line in TCPServer that is substantially different from what we saw in TCPClient is:

```
ServerSocket welcomeSocket = new ServerSocket(6789);
```

That line creates the object welcomeSocket, which is of type ServerSocket. The WelcomeSocket, as discussed above, is a sort of door that waits for a knock from some client. The port number 6789 identifies the process at the server. The next line is:

```
Socket connectionSocket = welcomeSocket.accept();
```

This line creates a new socket, called connectionSocket, when some client knocks on welcomeSocket. TCP then establishes a direct virtual pipe between clientSocket at the client and connectionSocket at the server. The client and server can then send bytes to each other over the pipe, and all bytes sent arrive at the other side in order. With connectionSocket established, the server can continue to listen for other requests from other clients for the application using welcomeSocket. (This version of the program doesn’t actually listen for more connection requests, but it can be modified with threads to do so.) The program then creates several stream objects, analogous to the stream objects created in clientSocket. Now consider:

```
capitalizedSentence = clientSentence.toUpperCase() + '\n';
```

This command is the heart of the application. It takes the line sent by the client, capitalizes it, and adds a carriage return. It uses the method toUpperCase(). All the other commands in the program are peripheral; they are used for communication with the client.

To test the program pair, you install and compile TCPClient.java in one host and TCPServer.java in another host. Be sure to include the proper host name of the server in TCPClient.java. You then execute TCPServer.class, the compiled server program, in the server. This creates a process in the server that idles until it is contacted by some client. Then you execute TCPClient.class, the compiled client program, in the client. This creates a process in the client and establishes a TCP connection between the client and server processes. Finally, to use the application, you type a sentence followed by a carriage return.

To develop your own client/server application, you can begin by slightly modifying the programs. For example, instead of converting all the letters to uppercase, the server can count the number of times the letter "s" appears and return this number.

2.7: Socket Programming with UDP

We learned in the previous section that when two processes communicate over TCP, from the perspective of the processes it is as if there is a pipe between the two processes. This pipe remains in place until one of the two processes closes it. When one of the processes wants to send some bytes to the other process, it simply inserts the bytes into the pipe. The sending process does not have to attach a destination address to the bytes because the pipe is logically connected to the destination. Furthermore, the pipe provides a reliable byte stream channel--the sequence of bytes received by the receiving process is exactly the sequence of bytes that the sender inserted into the pipe.

UDP also allows two (or more) processes running on different hosts to communicate. However, UDP differs from TCP in many fundamental ways. First, UDP is a connectionless service--there isn’t an initial handshaking phase during which a pipe is established between the two processes. Because UDP doesn’t have a pipe, when a process wants to send a batch of bytes to another process, the sending process must attach the destination process’s address to the batch of bytes. And this must be done for each batch of bytes the sending process sends. Thus, UDP is similar to a taxi service--each time a group of people get in a taxi, the group has to inform the driver of the destination address. As with TCP, the destination address is a tuple consisting of the IP address of the destination host and the port number of the destination process. We shall refer to the batch of information bytes along with the IP destination address and port number as the "packet."

After having created a packet, the sending process pushes the packet into the network through a socket. Continuing with our taxi analogy, at the other side of the socket, there is a taxi waiting for the packet. The taxi then drives the packet in the direction of the packet’s destination address. However, the taxi does not guarantee that it will eventually get the datagram to its ultimate destination; the taxi could break down. In other terms, *UDP provides an unreliable transport service to its communication processes*--it makes no guarantees that a datagram will reach its ultimate destination.

In this section we will illustrate UDP client-server programming by redeveloping the same application of the previous section, but this time over UDP. We shall also see that the Java code for UDP is different from the TCP code in many important ways. In particular, we shall see that there is (1) no initial handshaking between the two processes and therefore no need for a welcoming socket, (2) no streams are attached to the sockets, (3) the sending hosts create packets by attaching the IP destination address and port number to each batch of bytes it sends, and (4) the receiving process must unravel each received packet to obtain the packet's information bytes. Recall once again our simple application:

A client reads a line from its standard input (keyboard) and sends the line out its socket to the server.

The server reads a line from its socket.

The server converts the line to uppercase.

The server sends the modified line out its socket to the client.

The client reads the modified line through its socket and prints the line on its standard output (monitor).

Figure 2.27 highlights the main socket-related activity of the client and server that communicate over a connectionless (UDP) transport service.

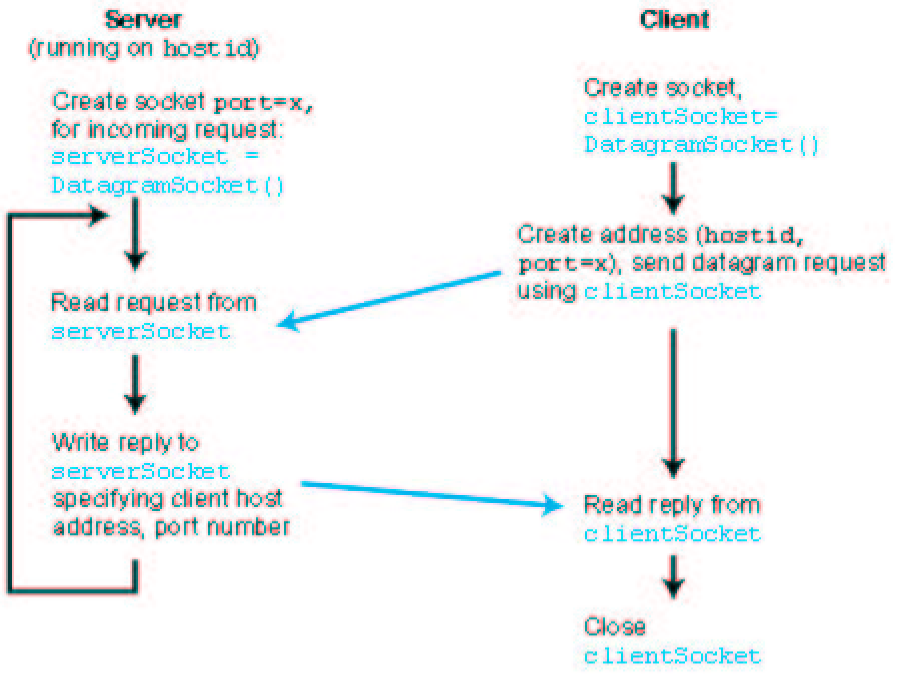
Server
(running on hostid)
Create socket port=x, for incoming request:
serverSocket = DatagramSocket()
Read request from serverSocket
Write reply to serverSocket specifying client host address, port number
Client
Create socket, clientSocket= DatagramSocket()
Create address (hostid, port=x), send datagram request using clientSocket
Read reply from clientSocket
Close clientSocket

Figure 2.27: The client/server application, using connectionless transport services

UDPClient.java

Here is the code for the client side of the application:

```
import java.io.*;
import java.net.*;
class UDPClient {
  public static void main(String args[]) throws Exception
  {
    BufferedReader inFromUser =
      new BufferedReader(new InputStreamReader
              (System.in));
    DatagramSocket clientSocket = new DatagramSocket();
    InetAddress IPAddress =
              InetAddress.getByName("hostname");
    byte[] sendData = new byte[];
    byte[] receiveData = new byte[];
    String sentence = inFromUser.readLine();
    sendData = sentence.getBytes();
    DatagramPacket sendPacket =
      new DatagramPacket(sendData, sendData.length,
              IPAddress, 9876);
    clientSocket.send(sendPacket);
    DatagramPacket receivePacket =
      new DatagramPacket(receiveData,
              receiveData.length);
    clientSocket.receive(receivePacket);
    String modifiedSentence =
      new String(receivePacket.getData());
    System.out.println("FROM SERVER:" +
              modifiedSentence);
    clientSocket.close();
  }
}
```

The program UDPClient.java constructs one stream and one socket, as shown in Figure 2.28. The socket is called clientSocket, and it is of type DatagramSocket. Note that UDP uses a different kind of socket than TCP at the client. In particular, with UDP our client uses a DatagramSocket whereas with TCP our client used a Socket. The stream inFromUser is an input stream to the program; it is attached to the standard input, that is, to the keyboard. We had an equivalent stream in our TCP version of the program. When the user types characters on the keyboard, the characters flow into the stream inFromUser. But in contrast with TCP, there are no streams (input or output) attached to the socket. Instead of feeding bytes to the stream attached to a Socket object, UDP will push individual packets through the DatagramSocket object.

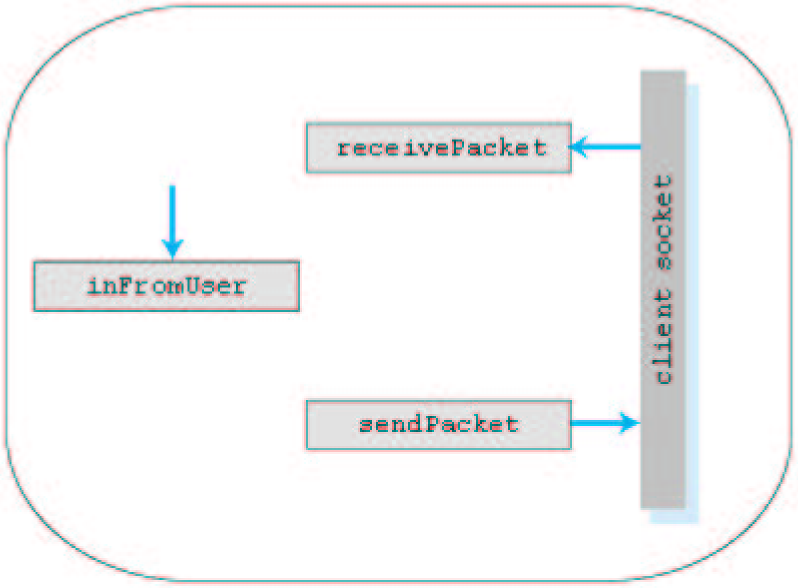

Figure 2.28: UDPClient.java has one stream and one socket

Let's now take a look at the lines in the code that differ significantly from TCPClient.java.

DatagramSocket clientSocket = new DatagramSocket();

The previous line creates the object clientSocket of type DatagramSocket. In contrast with TCPClient.java, this line does not initiate a TCP connection. In particular, the client host does not contact the server host upon execution of this line. For this reason, the constructor DatagramSocket() does not take the server hostname or port number as arguments. Using our door/pipe analogy, the execution of the above line creates a door for the client process but does not create a pipe between the two processes.

InetAddress IPAddress = InetAddress.getByName("hostname");

In order to send bytes to a destination process, we shall need to obtain the address of the process. Part of this address is the IP address of the destination host. The above line invokes a DNS look-up that translates the hostname (in this example, supplied in the code by the developer) to an IP address. DNS was also invoked by the TCP version of the client, although it was done there implicitly rather than explicitly. The method getByName() takes as an argument the hostname of the server and returns the IP address of this same server. It places this address in the object IPAddress of type InetAddress.

byte[] sendData = new byte[];

byte[] receiveData = new byte[];

The byte arrays sendData and receiveData will hold the data the client sends and receives, respectively.

sendData = sentence.getBytes();

The above line essentially performs a type conversion. It takes the string sentence and renames it as sendData, which is an array of bytes.

```
DatagramPacket sendPacket =
  new DatagramPacket(sendData, sendData.length,
                  IPAddress, 9876);
```

This line constructs the packet, sendPacket, that the client will pop into the network through its socket. This packet includes that data that is contained in the packet, sendData, the length of this data, the IP address of the server, and the port number of the application (which we have set to 9876). Note that sendPacket is of type DatagramPacket.

```
clientSocket.send(sendPacket);
```

In the above line, the method send() of the object clientSocket takes the packet just constructed and pops it into the network through clientSocket. Once again, note that UDP sends the line of characters in a manner very different from TCP. TCP simply inserted the line into a stream, which had a logical direct connection to the server; UDP creates a packet that includes the address of the server. After sending the packet, the client then waits to receive a packet from the server.

```
DatagramPacket receivePacket =
  new DatagramPacket(receiveData, receiveData.length);
```

In the above line, while waiting for the packet from the server, the client creates a place holder for the packet, receivePacket, an object of type DatagramPacket.

```
clientSocket.receive(receivePacket);
```

The client idles until it receives a packet; when it does receive a packet, it puts the packet in receivePacket.

```
String modifiedSentence =
  new String(receivePacket.getData());
```

The above line extracts the data from receivePacket and performs a type conversion, converting an array of bytes into the string modifiedSentence.

```
System.out.println("FROM SERVER:" + modifiedSentence);
```

This line, which is also present in TCPClient, prints out the string modifiedSentence at the client's monitor.

```
clientSocket.close();
```

This last line closes the socket. Because UDP is connectionless, this line does not cause the client to send a transport-layer message to the server (in contrast with TCPClient).

UDPServer.java

Let's now take a look at the server side of the application:

```
import java.io.*;
import java.net.*;
class UDPServer {
  public static void main(String args[]) throws Exception
    {
      DatagramSocket serverSocket = new
```

```
                  DatagramSocket(9876);
      byte[] receiveData = new byte[];
      byte[] sendData = new byte[];
      while(true)
        {
          DatagramPacket receivePacket =
            new DatagramPacket(receiveData,
                  receiveData.length);
          serverSocket.receive(receivePacket);
          String sentence = new String(
                  receivePacket.getData());
          InetAddress IPAddress =
                  receivePacket.getAddress();
          int port = receivePacket.getPort();
          String capitalizedSentence =
                  sentence.toUpperCase();
          sendData = capitalizedSentence.
                  getBytes();
          DatagramPacket sendPacket =
          new DatagramPacket(sendData,
                sendData.length,
                  IPAddress, port);
          serverSocket.send(sendPacket);
        }
    }
}
```

The program UDPServer.java constructs one socket, as shown in Figure 2.29. The socket is called serverSocket. It is an object of type DatagramSocket, as was the socket in the client side of the application. Once again, no streams are attached to the socket.

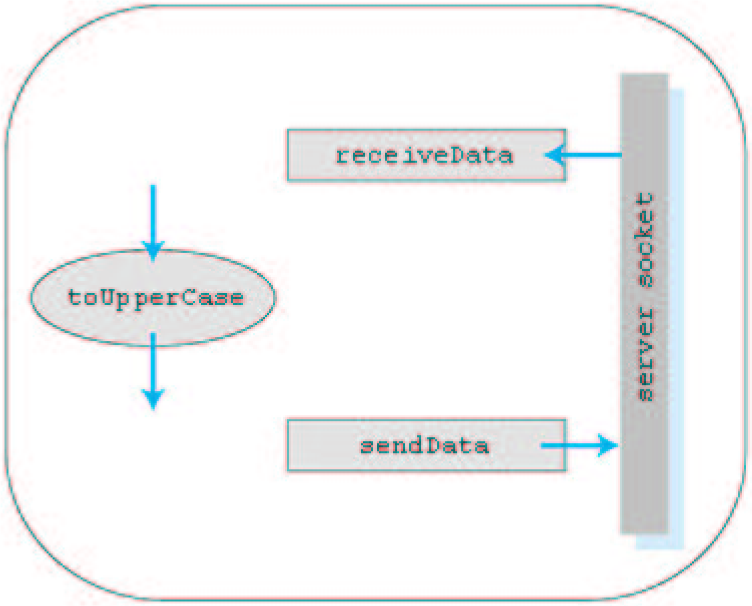

Figure 2.29: UDPServer.java has one socket

Let's now take a look at the lines in the code that differ from TCPServer. java.

```
DatagramSocket serverSocket = new DatagramSocket(9876);
```

The above line constructs the DatagramSocket serverSocket at port 9876. All data sent and received will pass through this socket. Because UDP is connectionless, we do not have to spawn a new socket and continue to listen for new connection requests, as done in TCPServer.java. If multiple clients access this application, they will all send their packets into this single door, serverSocket.

```
String sentence = new String(receivePacket.getData());
InetAddress IPAddress = receivePacket.getAddress();
int port = receivePacket.getPort();
```

The above three lines unravel the packet that arrives from the client. The first of the three lines extracts the data from the packet and places the data in the String sentence; it has an analogous line in UDPClient. The second line extracts the IP address; the third line extracts the *client port number,* which is chosen by the client and is different from the server port number 9876. (We will discuss client port numbers in some detail in the next chapter.) It is necessary for the server to obtain the address (IP address and port number) of the client, so that it can send the capitalized sentence back to the client.

That completes our analysis of the UDP program pair. To test the application, you install and compile UDPClient.java in one host and UDPServer.java in another host. (Be sure to include the proper hostname of the server in UDPClient.java.) Then execute the two programs on their respective hosts. Unlike with TCP, you can first execute the client side and then the server side. This is because, when you execute the client side, the client process does not attempt to initiate a connection with the server.

Once you have executed the client and server programs, you may use the application by typing a line at the client.

Online Book

2.8: Building a Simple Web Server

Now that we have studied HTTP in some detail and have learned how to write client/server applications in Java, let us combine this new-found knowledge and build a simple Web server in Java. We will see that the task is remarkably easy.

2.8.1: Web Server Functions

Our goal is to build a server that does the following:

- Handles only one HTTP request
- Accepts and parses the HTTP request
- Gets the requested file from the server's file system
- Creates an HTTP response message consisting of the requested file preceded by header lines
- Sends the response directly to the client

Let's try to make the code as simple as possible in order to shed insight on the networking concerns. The code that we present will be far from bullet proof! For example, let's not worry about handling exceptions. And let's assume that the client requests an object that is in the server's file system.

WebServer.java

Here is the code for a simple Web server:

```
import java.io.*;
import java.net.*;
import java.util.*;
class WebServer {
  public static void main(String argv[]) throws Exception
  {
    String requestMessageLine;
    String fileName;
    ServerSocket listenSocket = new ServerSocket(6789);
    Socket connectionSocket = listenSocket.accept();
    BufferedReader inFromClient =
      new BufferedReader(new InputStreamReader(
          connectionSocket.getInputStream()));
      DataOutputStream outToClient =
        new DataOutputStream(
            connectionSocket.getOutputStream());
      requestMessageLine = inFromClient.readLine();
      StringTokenizer tokenizedLine =
        new StringTokenizer(requestMessageLine);
      if (tokenizedLine.nextToken().equals("GET")){
        fileName = tokenizedLine.nextToken();
        if (fileName.startsWith("/") == true )
            fileName = fileName.substring(1);
        File file = new File(fileName);
        int numOfBytes = (int) file.length();
        FileInputStream inFile = new FileInputStream (
            fileName);
        byte[] fileInBytes = new byte[];
        inFile.read(fileInBytes);
        outToClient.writeBytes(
            "HTTP/1.0 200 Document Follows\r\n");
        if (fileName.endsWith(".jpg"))
          outToClient.writeBytes("Content-Type:
              image/jpeg\r\n");
        if (fileName.endsWith(".gif"))
          outToClient.writeBytes("Content-Type:
              image/gif\r\n");
        outToClient.writeBytes("Content-Length: " +
            numOfBytes + "\r\n");
        outToClient.writeBytes("\r\n");
        outToClient.write(fileInBytes, 0, numOfBytes);
        connectionSocket.close();
        }
      else System.out.println("Bad Request Message");
      }
}
```

Let us now take a look at the code. The first half of the program is almost identical to TCPServer.java. As with TCPServer.java, we import the java.io and java.net packages. In addition to these two packages we also import the java.util package, which contains the StringTokenizer class, which is used for parsing HTTP request messages. Looking now at the lines within the class WebServer, we define two string objects:

```
String requestMessageLine;
String fileName;
```

The object requestMessageLine is a string that will contain the first line in the

HTTP request message. The object fileName is a string that will contain the file name of the requested file. The next set of commands is identical to the correspond-ing set of commands in TCPServer.java.

```
ServerSocket listenSocket = new ServerSocket(6789);
Socket connectionSocket = listenSocket.accept();
BufferedReader inFromClient =
  new BufferedReader(new InputStreamReader
        (connectionSocket.getInputStream()));
  DataOutputStream outToClient =
    new DataOutputStream(connectionSocket.
        getOutputStream());
```

Two socket-like objects are created. The first of these objects is listenSocket, which is of type ServerSocket. The object listenSocket is created by the server program before receiving a request for a TCP connection from a client. It listens at port 6789 and waits for a request from some client to establish a TCP connection. When a request for a connection arrives, the accept() method of listenSocket creates a new object, connectionSocket, of type Socket. Next two streams are created: the BufferedReader inFromClient and the DataOutputStream outToClient. The HTTP request message comes from the network, through connectionSocket and into inFromClient; the HTTP response message goes into outToClient, through connectionSocket and into the network. The remaining portion of the code differs significantly from TCPServer.java.

```
requestMessageLine = inFromClient.readLine();
```

The above command reads the first line of the HTTP request message. This line is supposed to be of the form:

```
GET file_name HTTP/1.0
```

Our server must now parse the line to extract the filename.

```
StringTokenizer tokenizedLine =
        new StringTokenizer(requestMessageLine);
if (tokenizedLine.nextToken().equals("GET")){
  fileName = tokenizedLine.nextToken();
  if (fileName.startsWith("/") == true )
    fileName = fileName.substring(1);
```

The above commands parse the first line of the request message to obtain the requested filename. The object tokenizedLine can be thought of as the original request line with each of the "words" GET, file_name, and HTTP/1.0 placed in a separate placeholder called a token. The server knows from the HTTP RFC that the file name for the requested file is contained in the token that follows the token containing "GET." This file name is put in a string called fileName. The purpose of the last if statement in the above code is to remove the backslash that may precede the filename.

```
FileInputStream inFile = new FileInputStream (fileName);
```

The above command attaches a stream, inFile, to the file fileName.

```
byte[] fileInBytes = new byte[];
inFile.read(fileInBytes);
```

These commands determine the size of the file and construct an array of bytes of that size. The name of the array is fileInBytes. The last command reads from the stream inFile to the byte array fileInBytes. The program must convert to bytes because the output stream outToClient may only be fed with bytes.

Now we are ready to construct the HTTP response message. To this end we must first send the HTTP response header lines into the DataOutputStream outToClient:

```
outToClient.writeBytes("HTTP/1.0 200 Document
        Follows\r\n");
if (fileName.endsWith(".jpg"))
        outToClient.writeBytes("Content-Type:
        image/jpeg\r\n");
if (fileName.endsWith(".gif"))
        outToClient.writeBytes("Content-Type:
        image/gif\r\n");
outToClient.writeBytes("Content-Length: " + numOfBytes +
       "\r\n");
outToClient.writeBytes("\r\n");
```

The above set of commands are particularly interesting. These commands prepare the header lines for the HTTP response message and send the header lines to the TCP send buffer. The first command sends the mandatory status line: HTTP/1.0 200 Document Follows, followed by a carriage return and a line feed. The next two command lines prepare a single content-type header line. If the server is to transfer a GIF image, then the server prepares the header line Content-Type: image/gif. If, on the other hand, the server is to transfer a JPEG image, then the server prepares the header line Content-Type: image/jpeg. (In this simple Web server, no content line is sent if the object is neither a GIF nor a JPEG image.) The server then prepares and sends a content-length header line and a mandatory blank line to precede the object itself that is to be sent. We now must send the file FileName into the DataOutputStream outToClient.

We can now send the requested file:

```
outToClient.write(fileInBytes, 0, numOfBytes);
```

The above command sends the requested file, fileInBytes, to the TCP send buffer. TCP will concatenate the file, fileInBytes, to the header lines just created, segment the concatenation if necessary, and send the TCP segments to the client.

```
connectionSocket.close();
```

After serving one request for one file, the server performs some

housekeeping by closing the socket connectionSocket.
To test this Web server, install it on a host. Also put some files in the host. Then use a browser running on any machine to request a file from the server. When you request a file, you will need to use the port number that you include in the server code (for example, 6789). So if your server is located at somehost.somewhere.edu, the file is somefile.html, and the port number is 6789, then the browser should request the following:

```
http://somehost.somewhere.edu:6789/somefile.html
```

Online Book

2.9: Summary

In this chapter we've studied both the conceptual and the implementation aspects of network applications. We've learned about the ubiquitous client/server paradigm adopted by Internet applications and seen its use in the HTTP, FTP, SMTP, POP3, and DNS protocols. We've studied these important application-level protocols, their associated applications (the Web, file transfer, e-mail, and the Domain Name System) in some detail. We've examined how the socket API can be used to build network applications, and we've walked through not only the use of sockets over connection-oriented (TCP) and connectionless (UDP) end-to-end transport services, but also built a simple Web server using this API. The first step in our top-down journey "down" the layered network architecture is complete.

At the very beginning of this book, in Section 1.2, we gave a rather vague, bare- bones definition of a protocol as defining "the format and the order of messages exchanged between two communicating entities, as well as the actions taken on the transmission and/or receipt of a message or other event." The material in this chapter, and in particular our detailed study of the HTTP, FTP, SMTP, POP3, and DNS protocols, has now added considerable substance to this definition. Protocols are a key concept in networking; our study of applications protocols has now given us the opportunity to develop a more intuitive feel for what protocols are all about.

In Section 2.1 we described the service models that TCP and UDP offer to applications that invoke them. We took an even closer look at these service models when we developed simple applications that run over TCP and UDP in Sections 2.6-2.7. However, we have said little about *how* TCP and UDP provide these service models. For example, we have said very little about how TCP provides a reliable data transfer service to its applications. In the next chapter we shall take a careful look at not only the *what,* but also the *how* and *why,* of transport protocols.

Armed with a knowledge about Internet application structure and application-level protocols, we're now ready to head further down the protocol stack and examine the transport layer in Chapter 3.

© 2000-2001 by Addison Wesley Longman
A division of Pearson Education