[CHAPTER] 13

Networking

13.1 Introduction

Looking over the technological innovations of the past century, one could argue that the development of computer networks was one of the most significant and most important. Thirty-five years ago, when networks first appeared, only a few highly technical users had access to the technology. Today, networks are a common sight in schools, government buildings, offices, and libraries. Even private homes have local area computer networks connected via high-speed links to the Internet. If you look at the advertising on television and radio, or even the placards on the sides of buses, you will see ample evidence of networking's impact on the mainstream world.

The idea of a computer network was first proposed in 1962, when J. C. R. Licklider of the Massachusetts Institute of Technology (MIT) wrote a series of memos entitled "A Galactic Network," in which he described a globally interconnected set of computers. That same year, Leonard Kleinrock, working on his PhD at MIT, developed the theory of **packet switching** that forms the technical basis for the modern computer network.

In a packet-switched network, messages are broken up into pieces called **packets**. The individual packets are sent separately to their destination, where the original message is then reassembled. The primary advantages of a packet-switched network are that users can share a common communications channel, and if one part of a message is lost or damaged during transmission, only that part of the message needs to be resent, not the entire message.

Near the end of the 1960s, the U.S. Defense Advanced Research Projects Agency (DARPA) initiated a research program to develop the technologies required to interconnect packet-switched networks. The U.S. Department of Defense was interested in building a communication system that could withstand a nuclear attack. Unlike the telephone network, which cannot function if one of its switching centers is damaged, a packet-switched network can automatically route packets around damaged nodes, allowing communication to continue. The ARPANET, developed as a result of DARPA funding, was brought online in October 1969 and connected computers at four major organizations—UCLA, Stanford Research Institute, University of California-Santa Barbara, and the University of Utah. The first packets on the ARPANET were sent from UCLA to Stanford on October 29, 1969. (Rumor has it that the system crashed after the first three letters of the message were sent.) In addition to developing the hardware, designers were also creating protocols that allowed computers to communicate across linked packet-switched networks. The protocols developed over the course of this research became known as the **TCP/IP protocol** suite, named after its two most important components.[1]

Nothing was user friendly or easy about the early Internet. The network was designed for and used by computer engineers, physicists, and mathematicians. Little documentation or support was available. In 1973, the only network applications were e-mail, Telnet, and FTP. E-mail provided a way for users to electronically exchange messages, Telnet allowed users to log on to a

[1] Much of the information in the following pages is taken from "A Brief History of the Internet," a 1997 article by B. Leinter et al., at *www.isoc.org/internet-history/*.

remote system and use it as if it were local, and FTP (File Transfer Protocol) provided a mechanism to transfer code and data files between computers. During this time, the ARPANET had grown to 23 hosts that connected universities and government research centers around the country. Because the primary funding for this work was coming from the military, only government installations or groups doing research for the Department of Defense were allowed to connect.

Due to restricted access to the ARPANET, other networking projects were started to provide a way for nonmilitary organizations to connect their computers to a network. In 1980, the BITNET and CSNET networks were started. BITNET (Because It's Time Network) provided mail services to IBM mainframes, and CSNET (Computer Science Network) was funded by the National Science Foundation (NSF) to provide networking services to university, industry, and government computer science research groups. BITNET and CSNET were just two of the dozen or so packet-switched networks designed and built in the early 1980s. Most of them were very specialized and intended for use by a restricted group of people. There was little effort to standardize the protocols and little motivation to interconnect these separate networks.

VINTON G. CERF AND ROBERT E. KAHN

Vin Cerf and Bob Kahn are American computer scientists who are widely considered the "founding fathers" of the Internet. (However, many others contributed to its development, including Leonard Kleinrock, Larry Roberts, and John Postel.)

Cerf was a graduate student at UCLA under Professor Kleinrock, and helped to design the ARPANET. After receiving his PhD in 1972, he moved to Stanford as a professor to work in the fledgling field of packet-switched computer networks. He met Kahn, who was at DARPA directing the ARPANET project. The two of them began discussing the problem of the appearance of packet-switched networks using different protocols and different standards. The problem was similar to that of the railroads in the 19th century. Each country adopted its own national standards for rail gauge, which led to incompatible rail systems that could not be connected.

Cerf and Kahn worked on this problem from 1972 to 1974, and their groundbreaking 1974 paper, "A Protocol for Packet Network Interconnection," is considered the technical foundation for the design of the Internet. In 1976, Cerf joined Kahn at DARPA, and together they refined and standardized the protocols TCP/IP. The Internet officially adopted these as its universal standard in 1983.

continued

In 1997, President Bill Clinton awarded Cerf and Kahn the National Medal of Technology for their contributions to network development. In 2004, they were given the ACM A.M. Turing Award, the most prestigious award in computer science, for "pioneering work on internetworking including...the Internet's basic communications protocols...and for inspired leadership in networking." Today, Kahn is CEO of the Corporation for National Research Initiatives, a nonprofit organization for research and development of the national information infrastructure. Cerf currently works for Google, holding the joint titles of vice president and "chief Internet evangelist."

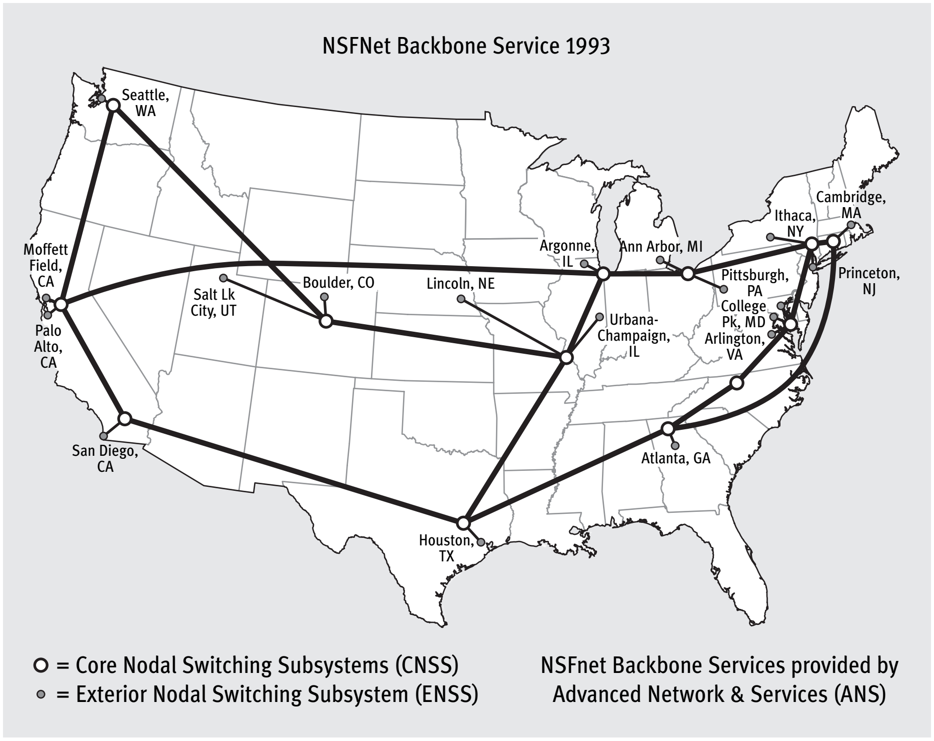
In 1986, NSF announced a program to develop a national communication infrastructure that would interconnect packet-switched networks and support the needs of the general academic and research community. NSF decided to use the same networking technology developed by DARPA. This meant that the new network would use the organizational infrastructure already in place in the ARPANET, and its primary protocols would be TCP/IP. The result was a major new communication service called NSFNet, which served as the primary conduit for all network traffic. NSFNet provided a cross-country 56-Kbps network that formed the core of the Internet. NSF maintained its sponsorship of NSFNet for nearly 10 years, establishing rules for noncommercial, government, and research use. By 1988, a 1.5-Mbps network had been established that connected the six original NSF supercomputer centers plus seven additional research sites. NSFNet now connected 217 networks, and network traffic began to double approximately every seven months.

During the mid-1980s, inexpensive minicomputers and personal computers became widely available. This combination of inexpensive desktop machines and powerful network servers fueled the rapid growth of the Internet. Companies could now purchase and maintain their own computing systems and were interested in connecting to the Internet. However, because funding came primarily from the government, Internet use was limited to research, education, and government applications. Commercial use of the Internet was prohibited.

In 1990, the ARPANET was officially dissolved, and the responsibility for the Internet passed to NSFNet. The network continued to spread among research and academic institutions throughout the United States, including connections to networks in Canada and Europe. As the network grew, so did the pressure to allow Internet access to private businesses. In 1991, NSFNet modified its acceptable use policy to allow commercial use. With the introduction of for-profit traffic, the growth of NSFNet over the next two years was explosive. Figure 13-1 shows the structure of NSFNet in mid-1993. By 1994, traffic on the network surpassed 10 trillion bytes per month. At its peak, NSFNet connected more than 4000 institutions and 50,000 networks across the United States, Canada, and Europe.

However, NSF is in the business of funding basic scientific research, not ongoing commercial ventures. Therefore, NSFNet was officially dissolved in 1995, and administration of the Internet was turned over to the private sector.

Since that time, the Internet has grown from a secretive Cold War technology to a general purpose, user-oriented and user-friendly environment that has profoundly changed the world in which we live. The phrase "the network is the computer" (coined by John Gage, a founder of Sun Microsystems) epitomizes the fundamental importance of networking technology. Networks are one of the least visible and least apparent parts of a modern computing system, yet they provide an indispensable resource. Networks are the conduits used for sharing and communicating data. Almost every computer program in use today uses a network in some way. For example, the word processor that I am using to write this text has a feature that allows me to access a collection of clip art that resides on remote machines attached to the Internet. Collaborative games, productivity tools, and Web browsers are all examples of programs that rely on a network to function properly. Organizations of all sizes depend on enterprise software that consists of distributed servers and databases. Knowledge of networking and the ability to develop software that effectively uses networks have become basic skills that every programmer must have.



(Source: Copyright Merit Network, Inc.)

[FIGURE 13-1] NSFNet

The programming languages of the 1960s, 1970s, and 1980s often included no direct support for networking. Instead, programmers had to rely on the communication services of the underlying operating system. This made network programming hard to implement. Java, developed in the mid-1990s, appeared when it was becoming obvious how important networking would be to software development. From the very beginning, Java included library support for networking and data communications in a way that was easy to understand.

This chapter introduces the basics of networking, explains the protocols that make network communication possible, and shows how to use the classes in the `java.net` package to develop programs that communicate across a network. This chapter only scratches the surface of this huge topic. However, after reading this material, you will understand some basic principles of networking and be able to develop Java programs that use a TCP/IP network.

13.2 Networking with TCP/IP

13.2.1 Protocols

The best place to start a discussion about computer networking is to ask: What is a network? If you look up the term **computer network** in a technical dictionary, you are likely to find a definition like this:

> *A computer network is a set of computers using common protocols to communicate over connecting transmission media.*

Although most of this definition makes sense, it still does not intuitively explain what a computer network really is. Some parts of this definition are easy to understand. For example, the phrase "set of computers" implies that a network has more than one computer. The phrase "connecting transmission media" indicates that the machines are connected via wires, fiber-optic cables, or radio waves, and they can exchange messages across this media. However, the phrase "common protocols" may need a little more explanation.

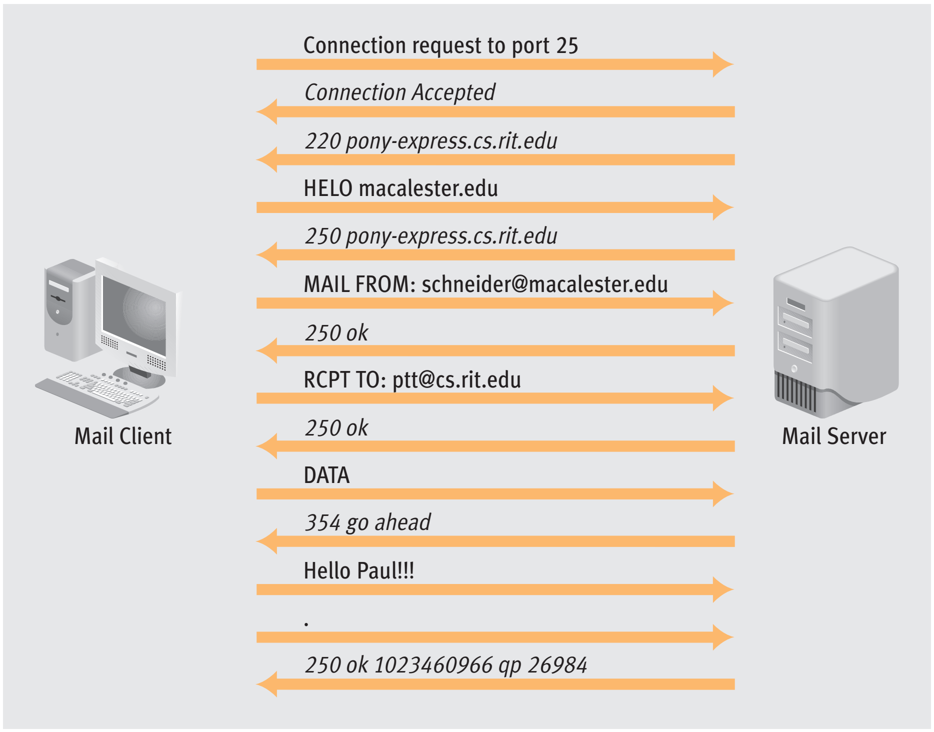
If we look up the term **protocol**, we are likely to find a definition like this:

> *A protocol is a formal description of message formats and the rules two or more machines follow to exchange messages.*

Consider, for example, how you call a friend on the telephone. You pick up the receiver and listen for a dial tone. After hearing the tone, you enter your friend's number on the keypad. If you hear a busy signal, you know that you cannot establish a connection, and hang up. If you hear a ring, you wait until someone picks up the phone on the other end and says

"Hello." This sequence of events is the "telephone protocol" used to make a phone call. The entities on both sides of the connection must understand and use the same protocol, which allows communication to proceed in an orderly way. For example, what would happen in our telephone protocol if the receiver did not say "Hello," but just picked up the receiver and said nothing? The caller could become confused and hang up.

A network protocol functions in the same way. It defines the messages that can be sent, the possible replies to these messages, and the actions that should occur upon receipt of a particular message. The only real difference between a computer protocol and a human protocol is that the computer protocol is much more detailed. It must address every possible sequence of messages, and it must define what actions to take in each situation. For example, Figure 13-2 shows the sequence of messages exchanged between two computers when sending electronic mail using the Simple Mail Transport Protocol (SMTP).


Connection request to port 25
Connection Accepted
220 pony-express.cs.rit.edu
HELO macalester.edu
250 pony-express.cs.rit.edu
MAIL FROM: schneider@macalester.edu
250 ok
RCPT TO: ptt@cs.rit.edu
250 ok
DATA
354 go ahead
Hello Paul!!!
.
250 ok 1023460966 qp 26984
Mail Client
Mail Server

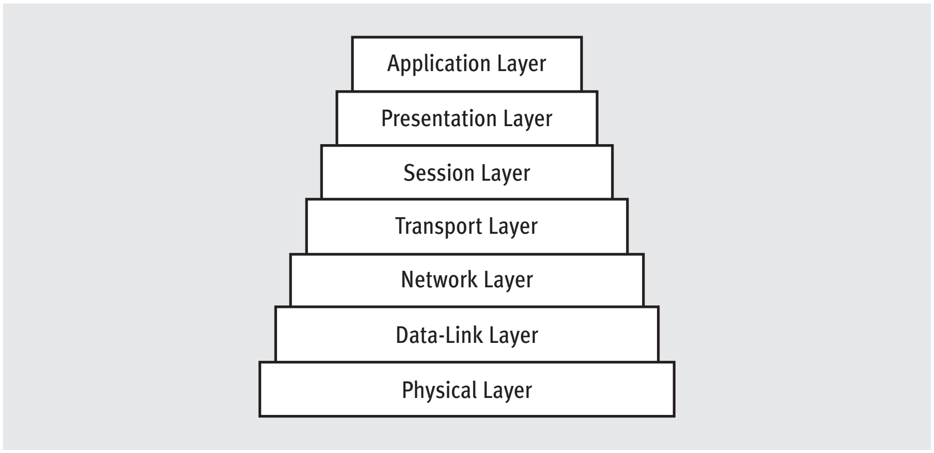
[FIGURE 13-2] Mail delivery protocol (SMTP) in action

Returning to the definition of a computer network, the phrase "uses common protocols" means that the sending and receiving computers are speaking the same language and following the same set of procedures. Thus, a computer network consists of a set of interconnected computers that use a common language to exchange information.

In the world of computer networks, standardization is extremely important. Without standardization, it would be impossible to build a network that allowed different computers, made by different manufacturers and running different operating systems, to communicate and exchange information. All official standards in the Internet community are published as a **Request for Comments** document, commonly referred to as an RFC. Every RFC is assigned a unique number and serves as a standard for a particular aspect of Internet computing. An RFC contains the detailed technical information a programmer requires to use or write a network protocol. This chapter includes references to the appropriate RFCs so you can obtain detailed information about the protocols we discuss. RFCs are available via e-mail, FTP, or the Web. One of the best sites for RFCs is *www.rfc-editor.org*.

13.2.2 The OSI Model

To help manage the complexity of a computer network, models have been developed to describe the functions the software must perform. The best-known model is the International Organization for Standardization (ISO) Open Systems Interconnection (OSI) reference model, which is shown in Figure 13-3.

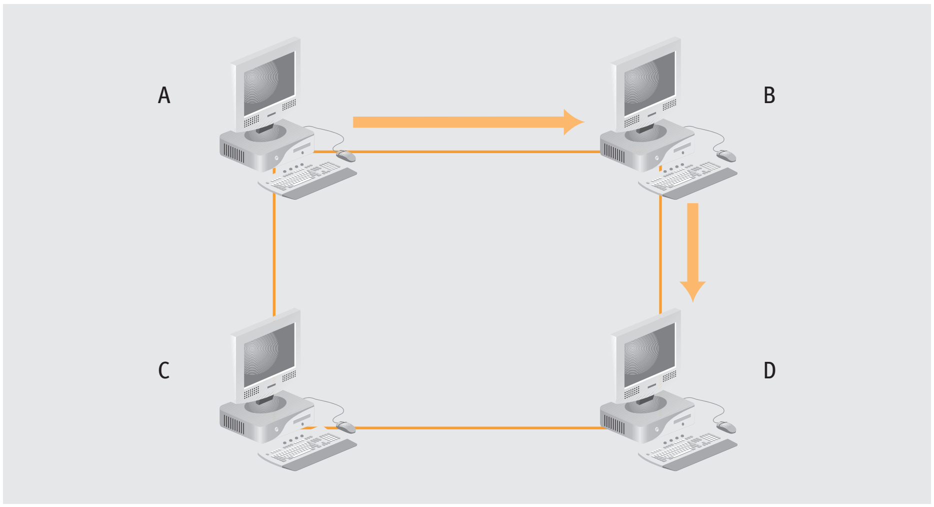


[FIGURE 13-3] The ISO Open Systems Interconnection model

The OSI model consists of seven layers; each performs a specific, well-defined function within the network. The bottom layer of the OSI model is the Physical layer. The **Physical layer** provides a raw, unreliable channel across which data can be sent in the form of binary digits, or bits, from one machine to another. You can think of the Physical layer as a bit pipe and the bits that it carries as ping-pong balls. The sending machine drops a ping-pong ball into the pipe, and the ball rolls down the pipe until it comes out the other end. Because this is an unreliable channel, the ball sometimes disappears and never comes out.

The next layer in the OSI model is the **Data Link layer**, which provides a reliable point-to-point service. It uses the services of the Physical layer to transmit messages from one machine to another. The Data Link layer includes error-checking information in the message so it can detect whether information is garbled or lost during transmission and retransmit it if necessary. You can think of the Data Link layer as a message pipe in which you send a sequence of ping-pong balls that represent a single message. You are guaranteed that every ball dropped into the pipe comes out correctly at the other end and in exactly the same order they were inserted. An important aspect of the Data Link layer is that it uses a point-to-point connection, which means it only guarantees the delivery of data between two machines that are directly connected by a physical link.

The **Network layer** is built on top of the Data Link layer and provides end-to-end delivery within the network. In other words, the Network layer makes it possible for two machines that are not directly connected to exchange data. The Network layer deals with the issue of addressing and mechanisms to identify the final destination. It then must determine how to route a message through the network so it arrives at its intended destination. Given the network depicted in Figure 13-4, the Network layer is responsible for delivering messages from machine A to machine D and for deciding if the message should be sent along the route A → B → D or the route A → C → D. The Network layer uses the services of the Data Link layer to transmit the message from one machine to another until the message arrives at its final destination. In other words, the Network layer is responsible for getting the message from A to D; the Data Link layer is responsible for getting the message first from A to B and then from B to D.



[FIGURE 13-4] End-to-end communication in the Network layer

The Network layer, like the Physical layer, does not guarantee that data will be delivered correctly. Even though the Network layer uses the reliable services of the Data Link layer, data might be lost as it passes through a node or it might cycle forever ($A \rightarrow B \rightarrow A \rightarrow B$ and so on). The services provided by the Network layer are similar to those provided by the Physical layer; both layers provide an unreliable delivery service. The difference is that the Network layer attempts to deliver messages between machines that may not be directly connected to each other, whereas the Physical layer delivers bits between machines that are directly connected. The Network layer deals with end-to-end delivery, and the Physical and Data Link layers deal with point-to-point delivery.

The **Internet Protocol** (IP) implements the services of the Network layer within the Internet. IP is truly the workhorse of the Internet—almost every message sent within the Internet is handled by IP. IP provides an unreliable, connectionless, datagram-based delivery service, which means that any message sent using IP must be broken down into units called **datagrams**. When a datagram is sent across the network, IP makes every effort to deliver it, but cannot guarantee it will be delivered correctly. Every datagram is treated separately, with no notion of ordering. Thus, datagrams sent to a common destination may not arrive in the same order they were sent. RFC 791 is the official specification of the Internet Protocol.

Although routing is an important IP function, you do not need to understand it to write a Java program. Instead, you can think of the Network layer as a black box into which you drop your messages; most of the time, they magically appear at their intended destination.

Every machine on the Internet is assigned a unique IP address. An IP address is **32 bits** long[2] and is organized in a way that simplifies the routing process. Normally, IP addresses are written in dotted decimal notation, which consists of four decimal numbers separated by a period. Each number in the address represents one of the four bytes in the IP address. An example of a dotted decimal address is 129.21.38.169.

IP identifies machines using these 32-bit addresses. However, people prefer to identify computers using symbolic names. For example, most people in the Computer Science Department at RIT would identify the department Web server as *www.cs.rit.edu* instead of by its IP address, 129.21.30.99. The **Domain Name System** (DNS), one of the services of the Internet, allows users to look up the IP address associated with a particular symbolic name. DNS does for computer names and IP addresses what the telephone book does for people's names and telephone numbers. A DNS server accepts client queries that contain a machine name, and it responds by sending the IP address associated with the name, assuming that the name is known to DNS.

[2] The number of available 32-bit IP addresses is running out. The new version of IP, IPv6, has 128-bit addresses. This provides an addressing space large enough to assign 1024 addresses to every square meter of the earth's surface. The designers of IPv6 were determined not to run out this time.

GEOGRAPHY LESSON

The Internet is growing at an extraordinarily fast rate. In 1986, 2300 hosts were connected to the network. Ten years later, that number had increased to 9.4 million. By 2006, there were 440 million host computers on the network. In the last 20 years the Internet has increased in size by a factor of 200,000, a growth rate unmatched by virtually any other technology.

Not only is the number of machines on the Internet impressive, so is its spread. The Internet is a truly global phenomenon, affecting the way people work, shop, and communicate throughout the world, not simply the United States or Western Europe. Consider that, while the United Nations has 192 member states, the DNS of the Internet includes entries for 247 countries, territories, and possessions. The DNS includes standardized symbolic domain names for such exotic locales as Bouvet Island (.bv), Comoros (.km), Djibouti (.dj), Wallis and Futuna (wf), Reunion (.re), Niue (.nu), Kiribati (.ki), Svalbard and Jan Mayen Islands (.sj), and even the continent of Antarctica (.aq), which, surprisingly, contains more than 7000 host computers in its domain.

The largest domains, by host count, are .net and .com, with 186 million and 77 million hosts, respectively. The largest non-U.S. domain is .jp (Japan), with 28 million hosts, followed by .it (Italy) with 13 million. The smallest nonempty DNS domain is .yt—the tiny French island of Mayotte off the coast of Madagascar in Eastern Africa. As of mid-2006, it contained exactly one computer!

Source: Internet Systems Consortium, "ISC Internet Domain Survey", www.isc.org/index.pl.

The next layer in the OSI model of Figure 13-3 is the **Transport layer**. Its primary function is to deliver data received from the Network layer to its ultimate destination. A typical computing system runs several network applications. The Network layer knows how to get a message from one machine to another, but not how to get that message to the specific application that is waiting for the information. That is the job of the Transport layer.

The Transport layer's function is similar to that of a mailroom in a large organization. For example, when the post office receives a letter addressed to:

Prof. Paul Tymann
Computer Science Department
Rochester Institute of Technology (RIT)
102 Lomb Memorial Drive
Rochester, New York 14623-5680

it does not deliver the letter directly to Prof. Tymann; instead, the letter goes to the central mail handling facility at the university. The staff in the mail facility sort all letters sent to RIT and deliver them to the intended recipient (in this case, Prof. Tymann). In a computer network, the Network layer performs the function of the post office, in that it delivers a message to the destination machine. The Transport layer performs the function of the central mail facility, which sorts the arriving messages and delivers them to their intended application.

The Transport layer usually provides two different types of service: connection-oriented and connectionless. A **connection-oriented transport protocol** provides a reliable, end-to-end, stream-based transfer of data. This means that when you send data, you are guaranteed that it will be correctly delivered to the machine on the other end of the connection. Furthermore, you are guaranteed that the information will be received in the exact same order that it was sent. With a connection-oriented transfer, a connection must be established before data can be sent; once the data transfer is complete, the connection must be terminated.

A **connectionless transport protocol**, as the name implies, does not require a connection to be established before communication can begin. A connectionless protocol provides an unreliable datagram-based transfer: there is no guarantee that data will ever be delivered, nor any guarantee about the order in which the datagrams are received.

The services provided by connection-oriented and connectionless transport protocols are similar to the services offered by the telephone system and the post office, respectively. The telephone system provides a connection-oriented transfer service for conversations. Before you start transmitting a stream of words to someone over the phone, you must establish a connection by dialing the person's number. Once the connection has been established, you speak into the phone, and your words are delivered to the person on the other end of the connection in the same order that you speak them. When you finish, you terminate the connection by hanging up.

The post office, on the other hand, provides a connectionless transfer service for letters. To send a letter, you place it in an envelope (a datagram), put the recipient's address and correct postage on the envelope, and then drop it into a mailbox for delivery. About 99.9 percent of the time, the letter is successfully delivered. However, there is a small chance that your letter will be damaged or lost. Furthermore, the post office makes no guarantee about the order in which letters are delivered. If you mail 10 letters to the same address, they will almost certainly not be delivered to the recipient in the same order they were sent. Even if you mail only one letter per week to the same person, there is still no guarantee they will arrive in order.

Within the Internet, the **Transmission Control Protocol** (TCP) and the **User Datagram Protocol** (UDP) are the two primary protocols used by the Transport layer. TCP is a connection-oriented protocol, and it provides a reliable, stream-oriented delivery service. UDP, on the other hand, provides an unreliable, datagram-based delivery service. The official specification for TCP is in RFC 793, and UDP is described in RFC 768.

Both TCP and UDP use the concept of a **port** to identify the ultimate destination of a message (like Prof. Tymann in our earlier example). A port is similar to the apartment number in an address. The apartment is identified by the single address of the building that houses the apartments (such as 123 North Main Street). This is like the single IP address of a machine that runs a number of applications. The apartment number (say, Apartment 3c) identifies a specific apartment in the building. This more specific number is like the TCP port number that identifies the application to which data should be delivered. Any application that uses either TCP or UDP must first obtain a port assignment from the operating system before it can communicate on the network. A TCP or UDP port is nothing more than an unsigned 16-bit integer.

To send a message to an application using either TCP or UDP, you must know the IP address of the machine on which the application runs and the port number assigned to that application. When a client application, such as a Web browser, requires a port number, it does not matter what number it is assigned, as long as the number is not being used by another application on the same machine. A server is a different story. If a server is assigned a random port number, how does a client that wants to communicate with the server learn what port number has been assigned? This is like trying to telephone someone you just met. How do you obtain their phone number?

The most common way to assign port numbers to services is to use the concept of **well-known ports**. The idea is a simple one: A list of port numbers used by important services on the Internet is compiled and published in RFC 1700. When a client wants to communicate with a specific service on a remote machine, it looks up the well-known port for that service and sends the message to the specified port. Table 13-1 lists a few of the well-known ports defined in RFC 1700.

For example, let's say we start a browser and want to view the home page of Macalester College. Our browser is given a random port number by the operating system. It then attempts to establish a connection to the machine *www.macalester.edu* on port 80, the well-known port for a Web server, as shown in Table 13-1. On most computing systems, the port numbers between 0 and 4096 are reserved for system services, so it is impossible for a user process to request a port in that range.

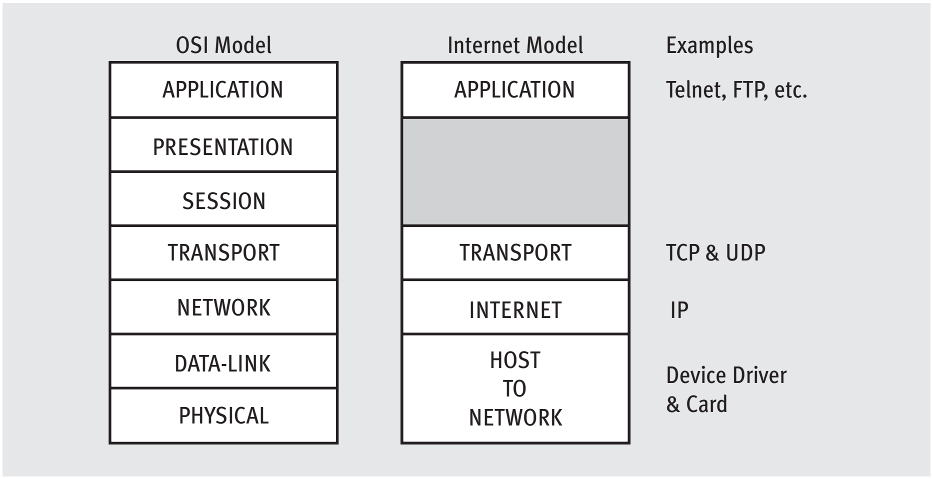
PORT	SERVICE	DESCRIPTION
7	Echo	Echoes back whatever message is sent
13	Day/time	Returns a string that gives the current time on the remote machine
19	Chargen	Upon connection, the remote host sends back a stream of random characters until the connection is closed
20	ftp-data	Used by the FTP service to transfer data
21	ftp-control	The port that an FTP client connects to initiate a file transfer
23	Telnet	The port where the Telnet server listens for remote terminal connections
25	SMTP	The port where the mail delivery agents listen for incoming mail
80	WWW	The port where a Web server will listen for HTTP requests

[TABLE 13-1] Examples of well-known ports

The next two layers in the OSI model are the Session and Presentation layers. The **Session layer** provides enhanced services that are useful in some applications. The **Presentation layer** handles how data is represented in the network. This layer might be responsible for converting data to a network standard form before it is transmitted or for encrypting data before it is sent. However, these two layers are not implemented in the Internet protocols. An application that requires Session or Presentation layer services must provide these services itself.

The final layer in the OSI model is the **Application layer**. It provides the interface that applications use to access the network. For example, the classes within the `java.net` package provide the application programming interface (API) that we use to access the network.

Although the ISO/OSI reference model is commonly used to describe the various software layers that constitute a network, few networks actually implement the model exactly as shown in Figure 13-3. OSI is a conceptual model, not an implementation model; it is a way of thinking about services that must be provided, not a road map for how to implement these services. Most software used to implement the Internet protocols is organized into four layers instead of seven. Figure 13-5 illustrates how the seven layers of the OSI model map into the four layers of the Internet model. It also lists the major protocols used in each layer in the Internet model. When developing software in Java using the `java.net` package, our primary concerns are the Transport layer, TCP, and UDP. The next section discusses the Java classes you can use to access the resources on a network.



[FIGURE 13-5] OSI and Internet networking models

13.3 Network Communication in Java

The classes in the `java.net` package provide a variety of networking services that a Java program can use. These classes can be divided into three categories. The first category, the **socket classes**, provides access to the Transport layer of the network (see Table 13-2). These classes provide direct access to the fundamental building blocks of communication in the Internet, such as datagrams, TCP, and UDP. The design of these classes is based in part on a package called **Berkeley Sockets**, a network API introduced in the Berkeley UNIX Software Distribution during the early 1980s. In this API, network communication is modeled as taking place between two endpoints called **sockets**. An application plugs into the network using a socket in the same way that a toaster plugs into an electrical socket to obtain power.

CLASS	DESCRIPTION
`DatagramPacket`	Represents a datagram, the unit of transfer used by UDP
`DatagramSocket`	Represents a socket for UDP (unreliable connectionless transport)
`InetAddress`	Represents an IP address; provides the capability to look up an IP address for a given host name
`ServerSocket`	Represents a socket for TCP (reliable stream-oriented delivery); usually used by a server because it allows you to wait and listen for incoming requests
`Socket`	Represents a socket for TCP; a connection is established when a socket is created

[TABLE 13-2] Socket-level classes in `java.net`

The second category, the **URL classes**, consists of classes that participate in HTTP, a popular Web protocol. The URL classes can be used to communicate directly with a Web server (see Table 13-3), and can provide higher-level access to network services. These classes use both the socket classes and their knowledge of specific protocols (for example, HTTP) to allow a program to deal rather easily with a Web server. For example, these classes enable the transfer of an entire Web page. Although you could perform the same function using the lower-level socket classes, you would have to write the code to implement the protocols that accomplish the transfer. The URL classes simplify the process of writing programs that use the Web.

CLASS	DESCRIPTION
`HttpURLConnection`	A subclass of `URLConnection` that implements HTTP
`JarURLConnection`	A subclass of `URLConnection` that can download and extract information from a Java Archive (JAR) file
`URL`	Represents a Uniform Resource Locator (URL)
`URLClassLoader`	A class that can load classes and resources from a list of URLs
`URLConnection`	An abstract class that represents a connection between an application and a Web server; the `openConnection()` method of this class returns a subclass that implements the protocol specified in the URL

[TABLE 13-3] URL classes in `java.net`

The third category of classes in `java.net` consists of a variety of utility classes and others that implement basic security mechanisms on the Internet. Although these classes are important, we do not discuss them in this chapter. Instead, we focus on the socket and URL classes.

The next section discusses the socket classes and illustrates how you can use them to write programs that use TCP and UDP.

13.4 The Socket Classes

13.4.1 Representing Addresses in Java

The `InetAddress` class in the `java.net` package creates an object that represents an IP address, and it provides the ability to convert symbolic machine names to numerical IP addresses. An instance of the `InetAddress` class consists of an IP address and possibly the symbolic name that corresponds to that address.

The `InetAddress` class does not have a public constructor. Instances of an `InetAddress` object are obtained using one of the static class methods listed in Table 13-4. All of these methods throw an `UnknownHostException` if the parameters do not specify a valid address. The `getLocalHost()` and `getByName()` methods are the most common ways to obtain instances of this class.

SIGNATURE	DESCRIPTION
`InetAddress[] getAllByName(String host);`	Creates `InetAddress` objects for the known IP addresses for the given name
`InetAddress getByAddress(byte[] addr);`	Creates an `InetAddress` given an IP address
`InetAddress getByAddress(String host, byte[] addr);`	Creates an `InetAddress` for the given name and IP address without checking the validity of the name and address
`InetAddress getByName( String host );`	Creates an `InetAddress` for the given host
`InetAddress getLocalHost();`	Creates an `InetAddress` for the local host
`String toString();`	Returns a string representation of this address; includes the IP address in dotted decimal notation and the symbolic name associated with the address

[TABLE 13-4] Methods to create `InetAddress` objects

It is easy to use the `InetAddress` class to obtain IP addressing information. The program in Figure 13-6 uses the `getLocalHost()` method to obtain the IP address for the machine on which the program runs. The `InetAddress` class overrides the `toString()` method to print the name associated with the address along with the IP address in dotted decimal notation. Note the use of the try block to handle the situation in which the IP address of the local host could not be found. This might happen, for example, if TCP/IP has not been configured on this system.

```
import java.net.*

/**
 * Print the IP address of the local host using the
 * getLocalHost() method from the InetAddress class
 */
public class HostInfo {
    public static void main( String args[] ) {

        // Must be executed in a try block because getLocalHost()
        // might throw a NoSuchHostException

        try {

            // Attempt to print the local address
```

continued

```
            System.out.println( "Local address:  " +
                                InetAddress.getLocalHost() );
        }
        catch ( UnknownHostException e ) {

            // Will be thrown if the local address cannot be
            // determined. This might happen, for example, if
            // the machine has not been assigned an IP address.

            System.out.println( "Local address unknown" );
        }
    }

} // HostInfo
```

[FIGURE 13-6] Determining the IP address of the local host

The program in Figure 13-7 is a slight modification of the one in Figure 13-6. The revised version prints the IP address information for any symbolic machine name specified on the command line. The program uses the `getByName()` method for name resolution. The `getByName()` method uses the services of the DNS (discussed earlier in this chapter) to convert symbolic names to numerical IP addresses. Again, note the use of the try block to handle exceptions caused by machine names that cannot be resolved.

Now that we know how to represent an IP address, we look at the `Socket` and `ServerSocket` classes in the next section.

```
import java.net.*;

/**
 * Print the IP address for each machine name on the
 * command line.
 */
public class Resolver {
    public static void main( String args[] ) {

        // Iterate over the command-line arguments

        for ( int i = 0; i < args.length; i++ ) {

            // Must be executed in a try block because getByName()
            // might throw a NoSuchHostException

            try {
```

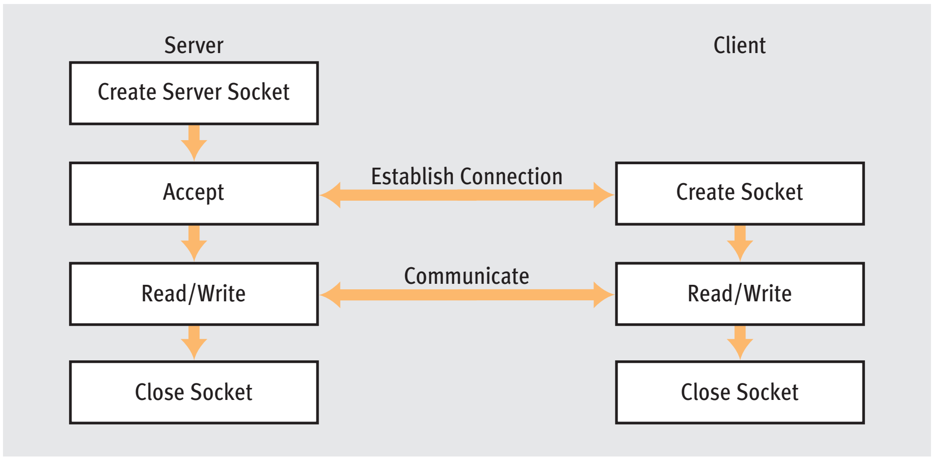
continued

```
                System.out.print( args[ i ] + ":  " );

                // Attempt to print the IP address of the
                // current host

                System.out.println(
                    InetAddress.getByName( args[ i ] ) );
            }
            catch ( UnknownHostException e ) {

                // Will be thrown if the current machine name
                // is not known

                System.out.println( "Unknown host" );
            }
        }
    }
} // Resolver
```

[FIGURE 13-7] Determining the address of an arbitrary host

13.4.2 Reliable Communication

The Socket and ServerSocket classes provide reliable stream-oriented delivery using TCP. The first step in a TCP communication is to establish a connection between the two machines that want to exchange messages. One machine actively establishes the connection (meaning it places the phone call), while the other machine takes a passive role and waits for a connection request (it answers the phone). The term **client** denotes the machine that actively opens the connection, while the term **server** denotes the machine that takes the passive role. The steps that the client and server take to establish a connection are illustrated in Figure 13-8.



[FIGURE 13-8] Connection-oriented communication

The words that describe how to establish a connection between machines may sound foreign to you, but the process should be familiar. Consider the events that occur when you decide to call a friend on the telephone. You pick up the phone and dial your friend's number, taking an active role in establishing the connection. Your friend must be near the phone and willing to answer it; when your friend hears the phone ring, he or she answers the phone, and the connection is established. You took the active role in establishing the connection, whereas your friend took a more passive role. You were the client and your friend was the server.

In Java, a server uses an instance of the `ServerSocket` class to obtain a port number to which clients can connect. Two parameters can be passed to the constructor of a `ServerSocket`. The first is the port number to which the server wants to connect this socket. Normally, a server specifies the number of its well-known port (see Table 13-1) when creating a `ServerSocket`. However, if this parameter is set to 0, the socket will be connected to any available port number on the system. The second parameter sets the maximum size of the queue that holds connection requests to a `ServerSocket`. If a `ServerSocket` receives a connection request while it is busy, the request is placed in the queue. When the server finishes and indicates that it is willing to accept another connection, the requests in the server queue are processed one at a time on a first-come, first-served basis. If the `ServerSocket` receives a connection request and the queue is full, the connection is refused. (You may have encountered this situation when trying to log on to a popular Web site during a busy time.)

A server indicates its willingness to accept a connection request by invoking the `accept()` method on a `ServerSocket`. The `accept()` method checks the pending request queue for a connection request. If there is a request in the queue, the request at the front of the queue is removed, the connection is established, and `accept()` returns a reference to a `Socket` that communicates with the client on the other end of the connection. If there are no pending connection requests in the queue, `accept()` will wait indefinitely until one arrives. You can set a time-out that limits the length of time the `accept()` method waits for a connection request.

Table 13-5 lists some of the methods provided by the `ServerSocket` class.

METHOD	DESCRIPTION
`Socket accept();`	Waits for a connection to be established; returns a socket that can be used to communicate with the client
`void close();`	Terminates the connection and releases all system resources associated with the socket
`InetAddress getInetAddress();`	Returns the IP address to which the socket is bound
`int getLocalPort();`	Returns the port on which the socket is accepting connections
`void setSoTimeout(int timeout);`	Determines the length of time in milliseconds that the `accept()` method waits for a connection request; otherwise, it waits indefinitely
`String toString();`	Returns a string representation of the socket

[TABLE 13-5] `ServerSocket` methods

On the other side of the TCP line, a client establishes a connection with a server using an instance of the `Socket` class. When the client creates a `Socket`, it specifies the host and port number of the server to which it wants to connect. The constructor for the `Socket` class uses this information to contact the server and take the necessary steps to establish a connection. Once the `Socket` has been created, it can be used to communicate with the server.

An often overlooked step when developing network programs is closing the connection when it is no longer needed. It is important to terminate a connection, whether it is active or not, because a connection uses resources within the operating system. If connections are not closed, the network resources of a computing system are slowly consumed until the system is no longer usable and may have to be restarted. To close the connection, simply invoke the `close()` method on the `Socket` being used for communication. Note that, for the connection to be completely closed, the client and server sides of the connection must each close their own socket.

Once a connection has been established, instances of the `Socket` class serve as the endpoints of the connection. Because TCP provides reliable stream-oriented transport, standard Java streams are used to read information from and write information to the network using a `Socket`. After a `Socket` has been created, a program uses the `getInputStream()` and getOutputStream() methods of the Socket class to obtain references to the streams associated with the connection. The next step is to wrap the appropriate streams around the socket. Once that is done, you can read and write to a Socket in the same way you read or write to a file. Table 13-6 lists some of the methods provided by the Socket class.

METHOD	DESCRIPTION
`void close();`	Terminates the connection and releases all system resources associated with the socket
`InetAddress getInetAddress();`	Returns the IP address of the host to which the socket is connected
`InputStream getInputStream();`	Returns an `InputStream` that can be used to send data across the network
`InetAddress getLocalAddress();`	Returns the IP address to which the socket is bound
`int getLocalPort();`	Returns the port assigned to the socket
`OutputStream getOutputStream();`	Returns an `OutputStream` that can be used to read data from the network
`int getPort();`	Returns the port on the remote machine to which this socket is connected
`String toString();`	Returns a string representation of the connection with which the socket is associated; the string includes the names, IP addresses, and ports used by both hosts

[TABLE 13-6] Socket methods

To illustrate how to use the Socket and ServerSocket classes, we will write Java programs that implement the TCP day/time service. This service allows a client to obtain the current date and time on a remote host (the time returned is the local time as recorded by the host, not universal time). The protocol for this service is straightforward. Table 13-1 specifies that a day/time server should accept connections on TCP port 13 for connections from a client. When a connection is established, the server sends a string to the client that contains the current date and time as recorded on the server. After sending the string, the server closes the connection.

A client program that wanted to use the day/time service would create a Socket to establish a TCP connection to port 13 on the host running the day/time server. Once the Socket is created, the client wraps a character-based reader around the InputStream associated with the Socket. The reader is used exactly as it was described in Chapter 11 to read the string sent by the server from the stream and print the result. The Java program in Figure 13-9 implements a day/time client.

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

/**
 * A simple client that queries the day/time service on a
 * remote computer to determine the date and time at
 * that location
 */
public class DayTimeClient {
    static int DAYTIME_PORT = 13;  // Well-known port for the
                                   // day/time server

    public static void main( String args[] ) {

        if ( args.length != 1 ) {
            System.err.println( "Usage: java DayTimeClient host" );
        }
        else {
            try {
                // Attempt to connect to the specified host
                Socket sock = new Socket( args[ 0 ], DAYTIME_PORT );

                // Wrap a stream around the socket so we
                // can read the reply
                BufferedReader in =
                    new BufferedReader(
                        new InputStreamReader(
                            sock.getInputStream() ) );

                // Read and print the reply sent by the server
                System.out.println( in.readLine() );

                // All done; close the streams and the socket
                in.close();
                sock.close();
            }
            catch ( UnknownHostException e ) {
                System.err.println( "DayTimeClient: no such host" );
            }
            catch ( IOException e ) {
                System.err.println( e.getMessage() );
            }
        }
    }
} // DayTimeClient
```

[FIGURE 13-9] A Java day/time client

The program in Figure 13-9 follows the steps outlined earlier in this section to establish a connection with a server. The creation of an instance of a `Socket` in the first line inside the try block actively establishes a connection with the server whose name is specified on the command line. Should something prevent the connection from being established (e.g., the host name is invalid, the server has crashed, or the server is refusing connections), an exception is thrown and the program terminates.[3] After the `Socket` has been created and the connection established, the program wraps an `InputStreamReader` around the socket's `InputStream` and a `BufferedReader` around the `InputStreamReader`. The `InputStreamReader` is required to convert the bytes being read from the `Socket` into characters. The program reads the response from the server by invoking the `readLine()` method on the `BufferedReader`. After the response has been printed, the program closes the socket and terminates.

When reading information from a socket using a stream, you might wonder how the program detects that there is no more information. A client detects that a server has closed its connection when it encounters an end of file (EOF) in the input stream. In other words, the act of closing a network connection is mapped into an EOF in a stream.

The remarkable thing about the program in Figure 13-9 is that the same streams we described in Chapter 10—the ones that read information from a file, a keyboard, or a serial port—are used here to read information from a network using TCP. Polymorphism makes this possible. The `InputStream` object returned by the `getInputStream()` method of the `Socket` class has had its `read()` method overridden so that instead of reading from a file, it uses TCP to read from the network.

Writing a Java day/time server is almost as easy as writing a day/time client. The only issue you need to address is that the Java day/time server you write cannot listen for incoming connections on port 13 because a day/time server may already be running on your machine; or, more likely, you will not have permission to use a port number in the range 0–4095. The Java day/time server in this chapter does not run on the well-known day/time port. Instead, it allows the operating system to select the port to which it will bind. The server prints the port it was assigned so that a modified client can learn this port number and access the server.

The first action the day/time server must take is to create a `ServerSocket` that accepts incoming requests. The server then enters an infinite loop, where it invokes the `accept()` method on the socket to express its willingness to accept a connection from a client. The `accept()` method blocks until a connection has been established and an instance of a `Socket` that can communicate with the client has been returned. The server wraps an output stream around the `Socket` and prints its message (the current date and time) to the stream. Once the message has been sent, the server closes the streams, and the socket and repeats the process. The code in Figure 13-10 is a Java implementation of a day/time server.

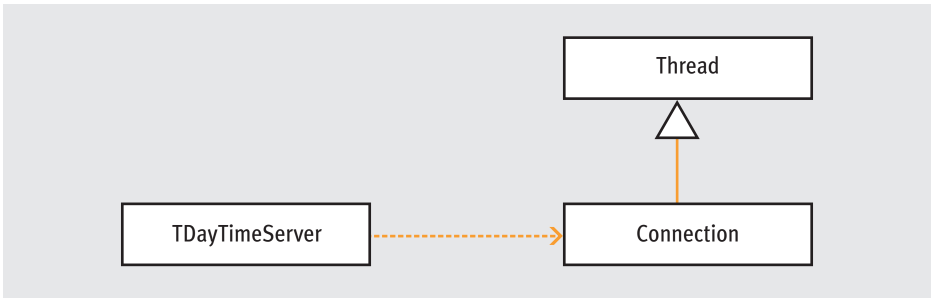
[3] For security reasons, many computer systems have disabled their day/time service. Thus, connection requests sent to port 13 may not be accepted.

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

/**
 * A Java implementation of a day/time server. This program
 * does not accept connections on the well-known port for
 * the day/time service.
 */
public class DayTimeServer {
    public static void main( String args[] ) {
        try {
            // Create the server socket that will be used
            // to accept incoming connections
                ServerSocket listen =
                    new ServerSocket( 0 ); // Bind to any port

            // Print the port so we can run a client
            // that will connect to the server
            System.out.println( "I am Listening on port:  " +
                                listen.getLocalPort() );

            // Process clients forever...
            while ( true ) {
                // Wait for a client to connect
                Socket client = listen.accept();

                // Create streams so a reply can be sent
                PrintWriter out =
                    new PrintWriter( client.getOutputStream(),
                                     true );

                // Print the current date
                out.println( new Date() );

                // That's it for this client
                out.close();
                client.close();
            }

        }
        catch( IOException e) {
            System.err.println( e.getMessage() );
        }
    }
}  // DayTimeServer
```

[FIGURE 13-10] A Java day/time server

Servers are rarely implemented using a single thread. Imagine for a moment a Web server that used a single thread to service client requests. Once one client established a connection, it would have exclusive use of the server until the connection was terminated. Any other client that wanted to access the server would have to wait until the current client was finished and the server was ready to accept a new connection. This might be a long wait. Instead, most modern network servers handle each client's request in a separate thread. The server waits for a client to establish a connection and then creates a separate thread to process the request using the threading concepts presented in Chapter 11. Once the client thread has been created and scheduled, the server is ready to accept a connection from another client.

It is quite easy to create a multithreaded server in Java. At the most basic level, a multithreaded server consists of only two classes: one implements the server itself and a second class is used to create the threads that handle individual requests from clients. The UML diagram in Figure 13-11 illustrates the design of a simple multithreaded server.



[FIGURE 13-11] UML design of a multithreaded server

The server of Figure 13-11 contains two classes: `TDayTimeServer` and `Connection`. The `Connection` class is a subclass of `Thread` and contains the code required to process a client request. `TDayTimeServer` is the primary server class; it accepts new connections and creates instances of the `Connection` class to handle the request.

Not all servers should be designed in this way. For example, the day/time server in Figure 13-10 should probably not be implemented using multiple threads. The tiny amount of time it takes the day/time server to process a request is probably less than that needed for the Java Virtual Machine (JVM) to create and schedule a new thread to service the request. However, a server that takes considerable time to process its requests, such as an FTP or Web server, should be implemented using threads.

To illustrate how easily you can write a multithreaded server in Java, we implement one by modifying the day/time server from Figure 13-10. The server consists of two classes: `Connection`, which processes a client request, and `TDayTimeServer`, which implements the server itself. The `Connection` class is shown in Figure 13-12.

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

/**
 * This class handles a client connection to the day/time
 * service in a separate thread
 */
public class Connection extends Thread {
    private Socket myClient;  // The client this thread services

    /**
     * Create a connection
     *
     * @param client the socket connected to the client
     */
    public Connection( Socket client ) {
        myClient = client;
    }

    /**
     * Send the current date and time to the client
     */
    public void run() {
        try {
            // Wrap streams around the socket so a reply can be sent
            PrintWriter out =
                new PrintWriter( myClient.getOutputStream(), true );

            // Print the current date
            out.println( new Date() );

            // That's it for this client
            out.close();
            myClient.close();

        }
        catch( IOException e) {
            System.err.println( e.getMessage() );
        }
    }
}  // Connection
```

[FIGURE 13-12] Thread to handle day/time requests

The code in Figure 13-12 is quite similar to the code in Figure 13-10. The only significant difference is that `Connection` extends `Thread`. The constructor for the `Connection` class takes as a parameter the `Socket` used to communicate with the client. The `run()` method contains the code to process the client's request. Once the client has been serviced, the streams and sockets are closed and the thread terminates.

The code that implements the threaded day/time server, `TDayTimeServer`, is shown in Figure 13-13.

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

/**
 * A multithread Java implementation of a day/time server.
 * This program does not accept connections on the well-known
 * port for the day/time service.
 */
public class TDayTimeServer {
    public static void main( String args[] ) {
        try {
            // Create the server socket that will be
            // used to accept incoming connections
            ServerSocket listen =
                new ServerSocket( 0 ); // Bind to any port

            // Print the port so we can run a client that
            // will connect to the server
            System.out.println( "Listening on port:  " +
                                listen.getLocalPort() );

            // Process clients forever...
            while ( true ) {
                // Wait for a client to connect
                Socket client = listen.accept();

                // Create and start the thread to handle the client
                Connection newThread = new Connection( client );
                newThread.start();
            }
        }
        catch( IOException e) {
            System.err.println( e.getMessage() );
        }
    }
}  // TDayTimeServer
```

[FIGURE 13-13] Threaded day/time server

The multithreaded day/time server invokes the `accept()` method on a `ServerSocket` to establish a connection with a client. Once the connection is established, an instance of the `Connection` class is created to process the request. Before returning to the top of the loop and waiting for another connection, the server invokes `start()` on the `Connection` to schedule execution of the thread that will process this request.

This section examined how the `ServerSocket` and `Socket` classes can provide reliable network communication in Java. The next section discusses how a datagram, the basic unit of transfer in UDP, is represented in Java.

13.4.3 Representing Datagrams

Every message sent using UDP is carried inside a datagram. You can think of a datagram as the envelope in which UDP messages are placed. When using the postal service, you place the letter you want to send inside an envelope that has the address of the intended recipient, your return address, and the proper amount of postage. To have UDP deliver a message, the message must be placed in a datagram, the UDP version of an envelope. In addition to the message, a datagram contains the IP address and port number of the destination and the IP address and port of the sending machine. The `DatagramPacket` class represents datagrams in Java; some of its methods are listed in Table 13-7.

METHOD	DESCRIPTION
`InetAddress getAddress();`	Returns the address of the machine to which the datagram is sent or the address of the machine that sent the datagram
`byte[] getData();`	Returns a reference to the byte array that contains the message within the datagram
`int getLength();`	Returns the length of the data within this datagram
`int getPort();`	Returns the port to which or from which the datagram is sent
`void setAddress();`	Sets the destination address of the datagram
`void setData();`	Sets the message that is sent in the datagram
`void setLength();`	Sets the length of this datagram
`void setPort();`	Sets the port to which this datagram is sent

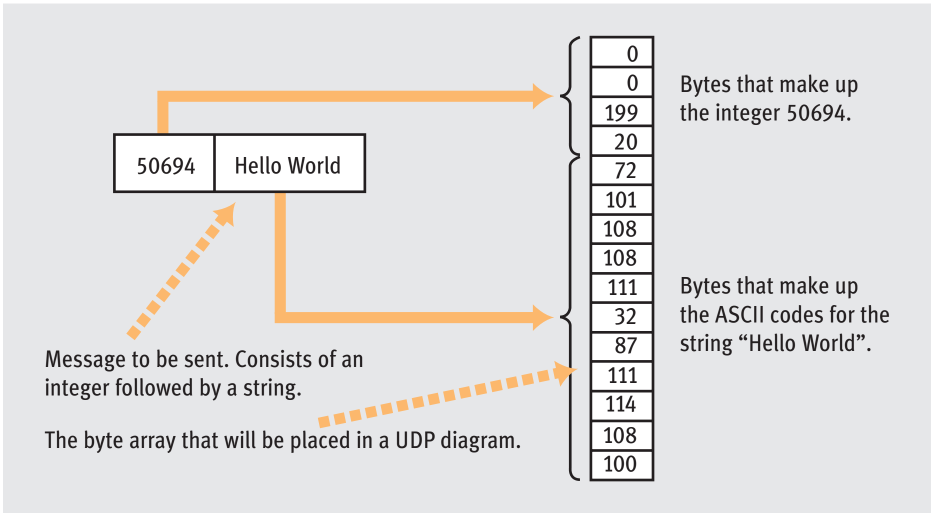
[TABLE 13-7] `DatagramPacket` methods

Although the datagram sent on the network contains the addressing information for both the source and destination machines, a `DatagramPacket` object only contains the addressing information for the *other* machine involved in the transfer. When a program creates a `DatagramPacket` to send to another machine, the IP address and port associated with the `DatagramPacket` object are that of the destination. When a program receives a `DatagramPacket`, the `getAddress()` and `getPort()` methods return the address and port from which the datagram was sent. The `setAddress()` and `setPort()` methods, however, only affect the address and port of the destination.

Datagrams transfer messages as an array of bytes, which means that any message sent using UDP must first be converted into bytes. Some messages, such as strings, are easy to convert because a string is nothing more than a sequence of character codes, which are usually represented as bytes. Converting messages containing other types of information, such as integer or floating-point values, is a little more difficult.

Perhaps the easiest way to deal with this issue is to use the `DataInputStream` and `DataOutputStream` classes in the `java.io` package. These classes provide methods to convert different types of information into a sequence of bytes. For example, the `writeInt()` method of `DataOutputStream` takes an integer value and converts it into the equivalent sequence of 4 bytes. The `DataInputStream` and `DataOutputStream` classes can be used in conjunction with the `ByteArrayInputStream` and `ByteArrayOutputStream` classes to read and write the bytes directly to or from a byte array. These four classes make it relatively easy to create byte-formatted messages.

As an example, assume that a program is using a UDP-based protocol whose messages consist of a single integer value followed by the 11-character string "Hello World" (see Figure 13-14).


50694
Hello World
0
0
199
20
72
101
108
108
111
32
87
111
114
108
100
Bytes that make up the integer 50694.
Bytes that make up the ASCII codes for the string "Hello World".
Message to be sent. Consists of an integer followed by a string.
The byte array that will be placed in a UDP diagram.

[FIGURE 13-14] Converting a message into a sequence of bytes

The code in Figure 13-15 shows how to use the `DataOutputStream` and `ByteArrayOutputStream` classes to convert the message of Figure 13-14 into a byte array for transmission across a network. The program uses the `writeInt()` and `writeBytes()` methods to place the data into the stream. The `DataOutputStream` converts the data into a sequence of bytes that is sent to a `ByteArrayOutputStream` to be written to a byte array. Invoking `toByteArray()` on the `ByteArrayOutputStream` returns a byte array containing the message as a sequence of bytes.

```
public static DatagramPacket makePak( int number, String message )
    throws IOException {

    DatagramPacket retVal = null;
    byte data[] = null;

    // Streams used to do the conversion
    ByteArrayOutputStream bytes = new ByteArrayOutputStream();
    DataOutputStream out = new DataOutputStream( bytes );

    // Write the message into the stream. Make sure to flush
    // the stream so that all the bytes are sent to the byte
    // array. Note that these methods may throw an exception.
    out.writeInt( number );
    out.writeBytes( message );
    out.flush();

    // Get the message in byte form
    data = bytes.toByteArray();

    // Close the streams
    out.close();
    bytes.close();

    // Create the datagram
    retVal = new DatagramPacket( data, data.length );

    return retVal;
}
```

[FIGURE 13-15] Using a `DataOutputStream` and a `ByteArrayOutputStream`

It is not difficult to convert the byte form of the message created in Figure 13-15 back to an `int` and a `String`. The easiest way to obtain the integer value from the message is to use a `ByteArrayInputStream` and a `DataInputStream` to extract the information from the data portion of the datagram (see Figure 13-16).

```
public static int getNumber( DatagramPacket packet )
    throws IOException {

    // Wrap streams around the data portion of the packet so
    // that the integer in the first part of the packet can
    // be extracted
    DataInputStream data =
        new DataInputStream(
            new ByteArrayInputStream( packet.getData() ) );

    // Read the number and return
    return data.readInt();
}
```

[FIGURE 13-16] Extracting an integer from a datagram

Because the characters that make up a string can be expressed in a variety of character codes, you must be careful to interpret the byte form of a string using the correct character encoding. Internally, Java uses Unicode to represent the characters in a string, whereas the characters in the transmitted message are represented using U.S. ASCII. Unfortunately, the `DataInputStream` class does not provide a method to read a sequence of U.S. ASCII codes and convert them to Unicode. However, the `String` class provides a constructor to do the necessary translation.

The program in Figure 13-17 extracts the string from the message using one of the constructors of the `String` class. The constructor in this program takes as parameters a byte array that contains the string in byte form, the length of the string, the index of the array where the first character code may be found, and the name of the character code that represents the characters in the string. The character code used in the message is ASCII, and is represented in Java by the string "US-ASCII".

```
public static String getString( DatagramPacket packet )
    throws IOException {

   // Extract the string using one of the constructors of
   // the string class.
   return new String( packet.getData(),       // Where the codes
                                              //   are stored
                      4,                      // Where the codes
                                              //   are in the array
                      packet.getLength() - 4, // Length of the
                                              //   string (less 4
```

continued

```
                                                  //   because of the
                                                  //   integer at the
                                                  //   front)
                        "US-ASCII" );             // Coding used
    }
```

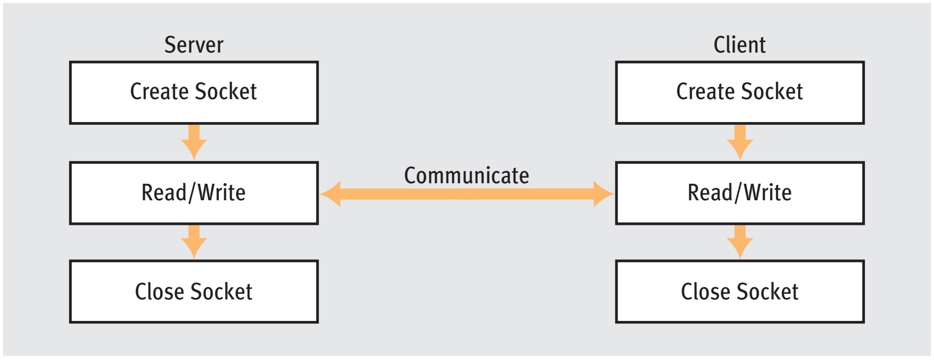
[FIGURE 13-17] Extracting an ASCII string from a datagram

As you can see, Java makes it fairly easy to place data into the datagrams you are sending and \to extract data from the datagrams you receive. (However, these techniques are tedious when compared to the ease of dealing with data in a TCP stream.) The next section discusses how the `DatagramSocket` class sends and receives datagrams.

13.4.4 Unreliable Communication

The `DatagramSocket` class in the `java.net` package allows UDP to send and receive packets over a network. UDP provides a connectionless, unreliable, datagram-based delivery system. Unlike TCP, UDP requires you to first convert your messages to byte form and place them into a datagram. Furthermore, you can never be certain that a datagram will reach its destination.

Because UDP is a connectionless protocol, you do not have to establish a connection before sending a datagram. As soon as a client has created a socket, that socket can transmit datagrams even if the server is not ready to receive them. Figure 13-18 illustrates the general steps required for communicating using UDP.



[FIGURE 13-18] Connectionless-oriented communication

Writing programs that communicate using UDP is more tedious than writing programs that communicate using TCP. Because TCP is reliable, a program that uses it does not have to worry about lost or damaged packets. Furthermore, because TCP is stream oriented, you can use standard Java input/output streams to send and receive data. When using UDP, a program has to worry about errors. Lost packets are typically handled in a UDP-based application using **time-outs**, which use a straightforward concept: when a client is sending packets via UDP, it keeps track of the average amount of time needed to deliver messages and for a reply to be received. After sending a message, the client starts a clock. If the reply does not come back within the expected time, the message times out, and the client knows that either the request or the reply message was lost. The original datagram can then be resent. This process is repeated until a correct copy of the message gets through.

You might wonder why anyone would bother to use UDP given the extra work required. The answer is that UDP is less complicated and therefore less time consuming than TCP. The amount of overhead associated with UDP, both in terms of the computer and the network, is small when compared to that of TCP. When you want to take full advantage of the speed of your network connection, UDP is the protocol of choice. For example, most network game applications use UDP to transmit game status between players because of its low overhead. Also, UDP is frequently used to implement audio and video streaming when speed is more important than an occasional missed packet—even when 3 to 5 percent of voice packets are lost in transmission, you can usually still understand the words of the speaker.

The `DatagramSocket` class is used to create a UDP communication endpoint in a Java program. Because UDP is connectionless, the addressing information is associated with each datagram, not with the socket, as is the case with TCP. A single instance of a `DatagramSocket` can communicate with several different applications, whereas a `Socket` can only communicate with the single server at the other end of the connection. There is no special datagram server socket class, so both a client and a server use the same `DatagramSocket` to communicate via UDP. Some of the methods of the `DatagramSocket` class are listed in Table 13-8.

METHOD	DESCRIPTION
`void close();`	Releases any system resources associated with this socket
`InetAddress getLocalAddress();`	Returns the IP address that this socket is using to receive datagrams
`int getLocalPort();`	Returns the port number to which this socket is bound; the value –1 is returned if the socket is not bound to a port
`int getSoTimeout();`	Gets the number of milliseconds a call to `receive()` will wait until throwing a time-out
`void receive(DatagramPacket p);`	Retrieves the next datagram sent to this socket; if a time-out has been set, this method will block until a datagram is received or the time-out period expires
`void send(DatagramPacket p);`	Sends a datagram
`void setSoTimeout(int timeout);`	Set the time-out for this socket in milliseconds; a value of zero disables time-outs

[TABLE 13-8] `DatagramSocket` methods

The Javadoc pages for the `DatagramSocket` class include a number of methods that seem to imply you can establish a "TCP-like" connection using a `DatagramSocket`. A connected `DatagramSocket` can only send and receive datagrams to or from the application whose address and port are specified when the `connect()` method is invoked. A connected `DatagramSocket` discards any packets that are sent to or received from a source that is not specified in the connection. However, a connected `DatagramSocket` does not provide any of the value-added services you might expect from a connection-oriented protocol, such as ordered delivery or guaranteed delivery.

You can specify several parameters when creating a `DatagramSocket`. When the socket is created, you can specify the port to which the socket is bound. Typically, a server uses this form of constructor to bind the `DatagramSocket` to a well-known port. The default constructor for the class binds the socket to any available port. The other parameters allow you to specify the address and port on the remote machine to which the `DatagramSocket` should be bound. Again, remember that a connected `DatagramSocket` only filters out datagrams from unwanted hosts. It does not provide reliable stream-oriented delivery.

The `send()` method is used to transmit a datagram on the network. The datagram must contain the IP address and port number of the application where it is being sent. Because UDP is unreliable, the `send()` method provides no information about the success or failure of the transmission. The `send()` method throws an `IOException` if an I/O error occurs. This exception indicates that the local machine encountered an error while placing the datagram on the network. The absence of an `IOException` does not indicate that the datagram was correctly received.

A program reads incoming datagrams by invoking the `receive()` method on a `DatagramSocket`. The `receive()` method blocks until a datagram has arrived at the socket. It is possible to enable time-outs by invoking the `setSoTimeout()` method prior to invoking `receive()`. If time-outs are enabled, `receive()` blocks until a datagram arrives or the time-out period expires, in which case it throws a `SocketTimeoutException`.

The datagram read by the `receive()` method is copied into the `DatagramPacket` object that is passed as a parameter. Recall from Section 13.4.3 that the state of a `DatagramPacket` includes a byte array that holds the data portion of the datagram. When reading datagrams using `receive()`, you must ensure that the `DatagramPacket` parameter has enough room to hold the incoming datagram. If there is not have enough room, only the portion of the datagram that fits into the `DatagramPacket` is read and stored. The remaining bytes are read the next time `receive()` is invoked. For many datagram-based protocols, you know the largest datagram the application will ever receive. In that case, you create a `DatagramPacket` object that is big enough to hold the largest datagram and pass it to `receive()`.

You can now appreciate the importance of the `getLength()` method associated with the `DatagramPacket`. The byte array that holds the datagram may be larger than the packet just read. The only way to determine the true length of the packet is not to use the length of the byte array parameter but to invoke the `getLength()` method.

To illustrate the use of the `DatagramPacket` and `DatagramSocket` classes, we will rewrite the day/time service from Section 13.4.2 so that it uses UDP instead of TCP to receive requests and send replies. It is common for machines to provide servers that use both TCP and UDP to communicate. A UDP-based day/time server (see Figure 13-19) creates a socket and then waits for a datagram to arrive. The arrival of the datagram causes the day/time server to obtain the current time, place the string that represents the day and time into a datagram, and return that datagram to the machine that sent the request. The day/time server ignores the contents of the request packet. It only uses the datagram to obtain the IP address and port where it should send the reply.

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

/**
 * A Java implementation of a UDP day/time server. This
 * program does not accept connections on the well-known
 * port for the day/time service.
 */
```

continued

```
public class UDPDayTimeServer {
    public static int MAX_PACKET_SIZE = 1024;

    public static void main( String args[] ) {
        // Create the server socket bound to any port
        DatagramSocket sock = null;
        try {
            sock = new DatagramSocket();
        }
        catch ( SocketException e ) {
            System.err.println(
                "UDPDayTimeServer:  unable to create socket" );
        }

        // Create the datagram that messages will be read into
        byte data[] = new byte[ MAX_PACKET_SIZE ];

        DatagramPacket packet =
            new DatagramPacket( data, MAX_PACKET_SIZE );

        // Print the port so we can run a client that will
        // connect to the server
        System.out.println( "Listening on port:  " +
                            sock.getLocalPort() );

        // Process clients forever...
        while ( true ) {
            byte now[] = null;  // Current time as bytes

            // Wait for a client to send a request
            try {
                // Need to reset the size back to the full size
                // The size is reset every time a packet is read
                packet.setLength( MAX_PACKET_SIZE );
                sock.receive( packet );

                // Got something. Put the current date and time
                // into the packet and send it back.
                now = new Date().toString().getBytes( "US-ASCII" );

                for ( int i = 0; i < now.length; i++ ) {
                    data[ i ] = now[ i ];
                }

                // Set the size of the packet
                packet.setLength( now.length );
```

continued

```
                // Send the reply
                sock.send( packet );
            }
            catch ( IOException e ) {
                // Ignore anything that goes wrong. This is
                // acceptable because UDP is an unreliable protocol
            }
        }
    }
}  // UDPDayTimeServer
```

[FIGURE 13-19] UDP day/time server

The program starts by creating a socket and a `DatagramPacket` that can hold the largest datagram a client might ever send. The server then enters a loop waiting for the arrival of a request. Upon receipt of a datagram, the server obtains the current day and time and places their character codes into the `DatagramPacket` it received. The datagram is then returned to the client. By reusing the `DatagramPacket` that holds the client's request, the addressing information for the reply is already stored in the state of the datagram.

Note that the server invoked the `setLength()` method two times within the program. The length of the byte array and the length of the packet are not necessarily the same. The `DatagramSocket` relies on the `getLength()` method, not the length of the byte array, to determine the size of the datagram to be sent. Setting the length of the `DatagramPacket` after storing the day and time ensures that only the portion of the byte array that actually contains data is sent to the client.

The program in Figure 13-20 provides an implementation of a UDP-based day/time client. The client in Figure 13-20 takes two command-line arguments: the name of the host where the day/time request should be sent and the port number assigned to the server on the remote machine.

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

/**
 * A simple client that queries the day/time service on
 * a remote computer to determine the date and time
 * at that location
 */
public class UDPDayTimeClient {
    public static final int MAX_PACKET_SIZE = 1024;
    public static final int TIMEOUT_PERIOD = 3000;  // 3 seconds
```

continued

```
public static void main( String args[] ) {
    if ( args.length != 2 ) {
        System.err.println(
            "Usage:  UDPDayTimeClient host port" );
    }
    else {
        try {
            // Create a socket to send and receive on
            DatagramSocket sock = new DatagramSocket();

            // Determine the IP address of the server
            InetAddress server =
                InetAddress.getByName( args[ 0 ] );

            // Get the port
            int port = Integer.parseInt( args[ 1 ] );

            // Assemble an empty packet to send
            DatagramPacket packet =
                new DatagramPacket( new byte[ MAX_PACKET_SIZE ],
                                    MAX_PACKET_SIZE,
                                    server,
                                    port );

            // The message from the server
            String serverTime = null;

            // Send an empty message to the server
            sock.send( packet );

            // Wait for a reply from the server, but only
            // for the specified time-out period
            sock.setSoTimeout( TIMEOUT_PERIOD );
            sock.receive( packet );

            // Convert the message to a string and print it.
            // The server will place a newline at the end of
            // the string. The -1 in the third parameter ensures
            // that this character is not copied to the string.
            serverTime = new String( packet.getData(),
                                     0,
                                     packet.getLength() - 1,
                                     "US-ASCII" );

            System.out.println( serverTime );
```

continued

```
                // All done; close the socket
                sock.close();
            }
            catch ( UnknownHostException e ) {
                System.err.println(
                    "UDPDayTimeClient:  no such host" );
            }
            catch ( SocketException e ) {
                System.err.println(
                    "UDPDayTimeClient:  can't create socket" );
            }
            catch ( SocketTimeoutException e ) {
                System.out.print( "No response from server" );
            }
            catch ( IOException e ) {
                System.err.println( e.getMessage() );
            }
        }
    }
} // UDPDayTimeClient
```

[FIGURE 13-20] UDP-based day/time client

It is possible that the client may send a request but never receive a reply. This is a UDP-based service, which means that the delivery of messages is not guaranteed. Either the request sent by the client or the reply sent by the server could be lost. To deal with this situation, the client sets a time-out to ensure that it does not wait indefinitely. Prior to invoking `receive()`, the client invokes `setSoTimeout()` to establish a time-out of 3 seconds (3000 milliseconds = 3 seconds). This means that the `receive()` method throws an exception if a reply is not received within 3 seconds. It is not necessary to reset the time-out after invoking `read()`. Once set, the time-out period remains the same until changed by a subsequent invocation of `setSoTimeout()`.

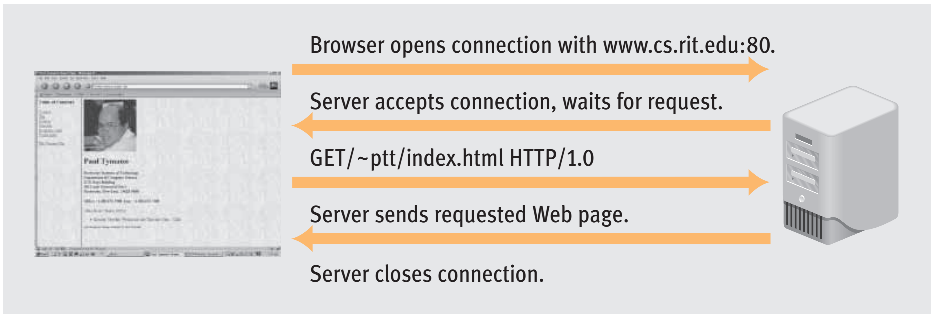
The datagram sent by the server contains the current time as a sequence of ASCII characters terminated by a line feed. The length of the datagram containing the response can be used to determine the number of characters in the reply (including the line feed). Because we do not want to include the line feed in the string version, the program uses `packet.getLength()` – 1 when converting the bytes in the reply into a `String`.

This section discussed the socket classes in the `java.net` package. These classes provide direct access to the transport protocols available within the Internet. The `ServerSocket` and `Socket` classes provide TCP-style delivery, whereas the `DatagramPacket` and `DatagramSocket` classes provide UDP-style delivery. The `InetAddress` class provides a representation of IP addresses and name resolution within a Java program.

13.5 The URL Classes

The URL classes in `java.net` provide high-level access to information stored on a Web server. In a typical transaction with a Web server, a client establishes a TCP connection and issues a request for a Web page using the Hypertext Transfer Protocol (HTTP). The server then returns the requested page, and the TCP connection is terminated (see Figure 13-21).

You could use the socket classes discussed in the previous sections to write the code necessary to implement the operations shown in Figure 13-21, but this work is already done by the URL classes. These classes not only provide a way to access a Web server, they provide classes to recognize and manipulate the different types of information you can download from a server.


Browser opens connection with www.cs.rit.edu:80.
Server accepts connection, waits for request.
GET/~ptt/index.html HTTP/1.0
Server sends requested Web page.
Server closes connection.

[FIGURE 13-21] Obtaining a Web page from a server with HTTP

The `HttpURLConnection` class can deal with the various forms found on many Web sites. The class provides methods that allow you to obtain information about a Web page and to fill in the fields on a form. The `JarURLConnection` class can manipulate **Java Archive Files (JAR)**. JAR files are similar to Zip files, in that they contain a compacted collection of files that are transported as a unit and then extracted at their final destination. Using the `JarURLConnection` class, you can determine what files are stored in an archive and extract them if you want. Finally, `URLClassLoader` loads class files so they may be executed by JVM.

The URL classes represent a resource on the network, and you can use them to obtain information directly from a Web server. The next section defines the term *URL* and illustrates how the URL classes access information stored on a Web server.

BLOGS

The "killer app" for networks of the 1970s was e-mail. In the 1980s, it was remote log-in and FTP. The appearance of the World Wide Web in the mid-1990s sent network usage skyrocketing. Many people feel that, along with e-commerce, the most important new network application of the 21st century will be the "blog," a contraction of the term *Web Log* coined by Peter Merholz in 1999.

A blog is a Web-based publication that consists of virtually any periodic articles that its writer(s) want to share with the general public. Sometimes, it is nothing more than a daily journal. More commonly, the articles are political, social, or cultural essays that reflect the opinions and biases of the blog author(s). Although some blogs are produced by a community of like-minded people who share responsibility for writing and posting articles, the majority are simply the thoughts and feelings of people with a computer and the necessary "blogware"—contact management software specifically created for editing, organizing, and publishing blogs. Recently, blogs have begun to focus on political issues of the day, containing the writers' thoughts on candidates, upcoming elections, scandals, current events, and government policies. Blogs have become an inexpensive way for people to instantly disseminate their own views and opinion.

Our history is filled with stories of individual crusaders who published fiery newsletters supporting or decrying some government policy. For example, Alexander Hamilton and James Madison wrote the *Federalist Papers* in support of the proposed U.S. constitution. *The Liberator* was a fervent antislavery newsletter published in Boston by William Lloyd Garrison, a Quaker abolitionist. However, these early crusaders could only reach limited audiences because of the cost of printing and the time required to manually distribute these newsletters to readers. (At the peak of its influence, *The Liberator* had a circulation of less than 3000.) The Web has changed all that. It costs virtually nothing to write and post your thoughts on a Web page, and if your ideas become widely discussed in other mass media, millions of readers might access and read your blog.

The ease of publishing personal opinions seems to have caught on with the general public, especially in the United States. The Web site Technorati (*www.technorati.com*), which searches blogs and measures their popularity, was tracking more than 51 million sites as of 2006.

13.5.1 Representing a URL

URLs (Uniform Resource Locators) are used to identify resources that are available on the Web. People often mistakenly think that a URL refers only to a file, but it can refer to many different types of network resources. A URL consists of two parts: a protocol identifier and a resource identifier. A colon and two slashes separate these two components. For example, in the URL *http://www.cs.rit.edu*, the protocol identifier is `http` and the resource name is `www.cs.rit.edu`.

The **protocol identifier** determines the type of protocol used to access the resource. For Web pages, the protocol identifier is almost always `http`. Another commonly used protocol identifier is `ftp`, which indicates that the File Transfer Protocol should obtain the resource (a remote file). When accessing the Web, the **resource identifier** specifies both the resource name and the path that must be followed to access it. For example, the URL *http://www.cs.rit.edu/~ptt/index.html* specifies that the file `index.html` can be found in the home directory of the user `ptt` on the machine `www.cs.rit.edu`.

A URL in a Java program is represented by an instance of the URL class. Several constructors are provided by this class, but the one you will use most often takes a single parameter, a string containing the URL. If the URL is not properly formatted or specifies an invalid protocol, it will throw a `MalformedURLException`. Some of the other methods in the URL class are described in Table 13-9.

METHOD	DESCRIPTION
`Object getContent();`	Returns the object referred to by this URL; if the Java system knows about the type of object being referred to, the object will be converted into the appropriate Java type
`String getFile();`	Returns the name of the file to which this URL refers
`String getPath();`	Returns the path to the file referred to by this URL
`int getPort();`	Returns the port associated with this URL
`String getProtocol();`	Returns the protocol identifier of this URL
`URLConnection openConnection();`	Returns a `URLConnection` object that can be used to access the resource identified by this URL
`InputStream openStream();`	Returns a stream connected to the URL; this can be used to obtain the data associated with this URL

[TABLE 13-9] URL methods

Many of the methods in Table 13-9 are simple accessor methods that return the various parts of the URL: the protocol, port, and resource name. However, some of the methods allow you to obtain the information associated with the resource identified by the URL. The next section discusses how the `openStream()` method can read the data associated with a URL.

13.5.2 Reading from a URL

Once you have created a URL, you can use it to read the data associated with the resource it identifies. The `openStream()` method of the URL class returns a stream that, when read, returns the bytes contained in the resource. If the URL refers to a Web page, the bytes read from the stream are the characters contained in the Web page. For example, the program in Figure 13-22 prints the contents of a Web page, the HTML codes that make up the page. The URL that identifies the Web page to be printed is obtained from the command line.

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

/**
 * Use the URL class to read the HTML associated with a Web
 * page. The Web page's URL is taken from the command line.
 */
public class ReadHTML {
    public static void main( String[] args ) {
        // Usage
        if ( args.length != 1 ) {
            System.err.println( "Usage:  java ReadHTML url" );
        }
        else {
            try {
                // Create the URL
                URL webPage = new URL( args[ 0 ] );

                // Connect the streams
                BufferedReader in =
                    new BufferedReader(
                        new InputStreamReader(
                            webPage.openStream() ) );

                // Used to store lines as they are read
                String line = null;
```

continued

```
                // Read and print the HTML
                while ( ( line = in.readLine() ) != null ) {
                    System.out.println( line );
                }

                // Close the streams
                in.close();
            }
            catch ( MalformedURLException e ) {
                System.err.println( "ReadHTML:  Invalid URL" );
            }
            catch ( IOException e ) {
                System.err.println( "ReadHTML: " + e.getMessage() );
            }
        }
    }
} // ReadHTML
```

[FIGURE 13-22] Using `openStream()` to print a Web page

The `openStream()` method returns the data stream associated with the URL; it does not allow you to initiate and control an HTTP dialog with the server. The `openConnection()` method, on the other hand, opens an HTTP connection with the server, and this connection can be used to communicate with a resource using HTML. Using the `openConnection()` method, for example, you can interact with Web forms or other GUI objects on the server.

This section discussed the URL classes that provide high-level access to resources on the Web identified by URLs. Using the classes in this section, it becomes relatively easy to write a Java program that can obtain different types of information stored on a Web server.

13.6 Security

In all modern computer applications, especially applications that use a network, security is a vital consideration. You have probably read many stories about poorly designed software with security flaws that allowed hackers to gain unauthorized access to a system. You may even have had the misfortune to experience such an attack firsthand. Security has always been a major concern for software developers, but the rapid growth of distributed applications has made it even more essential for all software designers to know its basic concepts. To familiarize you with issues you need to consider when developing software that uses a network, we conclude this chapter with an introduction to Java's basic security mechanisms.

Computer security, and network security in particular, is a broad area that includes topics such as encryption, authentication, digital signatures, and protocol design. Java provides support for some of these features in the `java.security` and `javax.crypto` packages. These packages provide classes that implement authentication mechanisms and the ability to encrypt and decrypt data. Thorough coverage of these topics is well beyond the scope of this text; you will study them in future courses in networking and security. This section focuses on the basic security mechanisms built into the Java Virtual Machine (JVM).

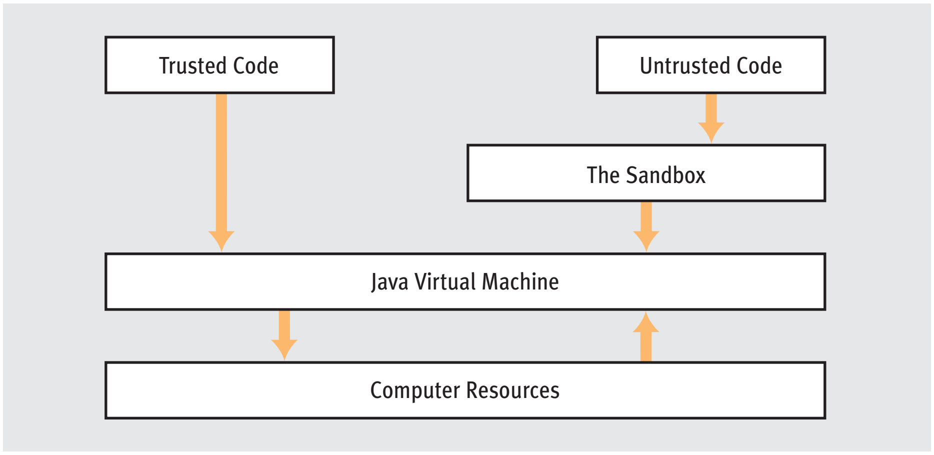
From its inception, Java was designed to dynamically load programs, in the form of applets, from a variety of sources. Java's ability to download and execute programs from anywhere is one of its strengths, but it is also inherently unsafe. Whether the program file is local or comes from the network via an untrusted source, giving it full access to all the system's resources could wreak havoc on a computer system. An applet, for example, could erase system files, generate forged e-mail, or launch attacks on other systems.

The JVM serves as a barrier to protect the underlying computer system from the execution of Java programs. The JVM does not allow a program to directly access any aspect of the computing system on which it runs. Any attempt by a Java program to access resources on the local system, whether to access memory or erase a file, must be done through JVM. This allows JVM to check any operations that a program wants to perform and prevent them if JVM deems them unsafe.

When JVM loads a class file, it goes through a process known as **bytecode verification**: JVM checks the bytecodes of the class file and verifies that they are valid. Bytecode verification is designed to prevent the JVM from executing class files that might make it crash or make it vulnerable to attack. Because every class file executed by the JVM goes through this verification process, you can be virtually certain that the bytecodes interpreted on your system have been generated by a trusted Java compiler and cannot bypass the JVM.

However, even programs that pass this verification process can cause damage. For example, given what you have learned in this text, you could easily write a Java program that deletes all files from a computer. **Access control** policies are the second line of defense against malicious code. These policies specify exactly what a Java program can and cannot do. The basic idea behind access control is that a user can explicitly enumerate what operations are permitted to a program. If the program attempts something that is not allowed, it is aborted and JVM throws an exception to indicate what happened.

In the basic Java security model, trusted code is allowed full access to the system, and untrusted code is forced to execute in a restricted environment called the **sandbox** (see Figure 13-23). The access control policies of the sandbox are established by an instance of the `java.lang.SecurityManager` class. A security manager provides methods that determine if a particular operation is allowed. For example, the `checkWrite()` method of the `SecurityManager` class determines if a program has permission to write to a file. If the `checkWrite()` method determines that the program cannot access a file, it throws a `SecurityException`; otherwise, it simply returns.



[FIGURE 13-23] The sandbox model

All of the classes in the Java API have been designed to invoke the appropriate methods in the `SecurityManager` class before executing potentially dangerous code. For example, when you use a `FileWriter` to write to a file, the `write()` method invokes the `checkWrite()` method of the current `SecurityManager` to determine if this operation is allowed. You do not have to worry about making these checks yourself. They are done for you in the classes you use to access system resources.

A **policy configuration file** specifies the operations that are permitted for code from specific sources. For a program to be allowed to perform an operation, such as read the contents of a file, it must be granted permission. A policy file consists of zero or more entries. Each entry specifies the permissions for one or more classes that form a domain. A **domain** is a group of classes that have the same set of permissions.

An entry in a policy configuration file starts with the keyword `grant`, which may be followed by the keywords `signedBy` and `codeBase`. The keyword `codeBase` is used to define a domain based on the source of the code. A **code base** is a URL that specifies the directory from which the classes are loaded. For example, the following URL:

```
    file:/c:/classes/ptt
```

refers to the files that were loaded from the directory

```
    c:/classes/ptt
```

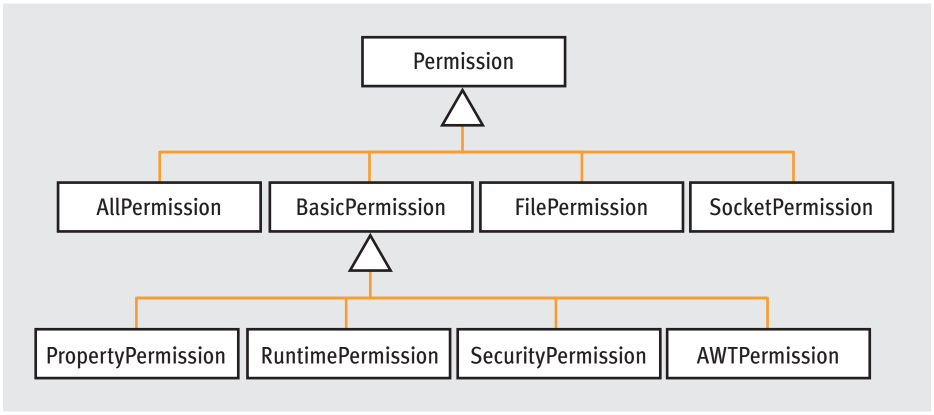
on the local machine. The URL that identifies the source of the classes in the domain follows the keyword `codeBase` in an entry. The keyword `signedBy` is used to define a domain based on the authority that signed the code. The **signer** of a class identifies the organization that digitally signed the class file. Java provides the ability to attach digital signatures to a class file so that the author of a class may be verified.

The keywords `codeBase` and `signedBy` are optional; their omission signifies any code base or any signer. Figure 13-24 contains a policy file that grants all permissions to all classes.

```
grant {
    permission java.security.AllPermission;
}
```

[FIGURE 13-24] A policy file that grants all permissions to all classes

The body of an entry lists the permissible operations of classes in the domain. The permissions in an entry are string representations of a Java `Permission` class. The Java `Permission` classes represent access to system resources and specify what is allowed for their specific resource. The top-level class in the class hierarchy is the abstract class `Permission`. New permissions are entered as subclasses of `Permission`. Like the exception classes, the subclasses of `Permission` are stored in the package whose permissions they describe. The UML diagram in Figure 13-25 shows the `Permission` class hierarchy.



[FIGURE 13-25] `Permission` class hierarchy

The policy file in Figure 13-25 uses the `AllPermission` class to specify that classes in the domain have access to all system resources. The entry in Figure 13-26 refers to all classes because both the keywords `codeBase` and `signedBy` have been omitted. The syntax the `Permission` classes use to specify allowable operations varies slightly from class to class. The Javadoc pages for each of the `Permission` classes specify the format used by each class. The policy file in Figure 13-26 illustrates how to use the `FilePermission` class.

```
grant codeBase "file:c:/classes/RoadMap/" {
   permission java.io.FilePermission "roads.txt", "read";
   permission java.io.FilePermission "cities.txt", "read";
};

grant codeBase "file:c:/classes/MapDatabase/" {
   permission java.io.FilePermission "roads.txt", "read,write";
};
```

[FIGURE 13-26] Using the `FilePermission` class in a policy file

The policy file in Figure 13-26 defines two domains. The first entry defines a domain that consists of code coming from c:/classes/RoadMap. The classes in this domain are allowed to read the files roads.txt and cities.txt. Any other file operations are not allowed. The second entry defines a domain whose classes have been given permission to both read from and write to the roads.txt file. Note that if a class is in more than one domain, the permissions from the domains are combined. For example, if you tried to add a third entry to Figure 13-26 that granted all permissions to all classes, the third entry would essentially override the other entries in the file.

When running an application using the JVM, you may use the command line to specify whether the application should be run with a security manager installed, and you can specify the policy file to use to define run-time permissions. The following command:

```
java -Djava.security.manager some-application
```

runs the program `some-application` using the sandbox model described earlier in this section. You can modify the default security policy by specifying a policy file in the command line, as follows:

```
java -Djava.security.Manager -Djava.security.policy-file
   some-application
```

where `policy-file` is the name of the policy file that defines the domains for `some-application` and the operations they are allowed to perform. Java provides the ability to define system-wide and user-specific policies. You may want to check with the administrator of the machine you use to find out what security policies, if any, are in effect.

13.7 Summary

This chapter completes our discussion of networking and the entire topic of modern software development. We hope that the book has made you aware of the many steps involved in the creation of a large, complex software system.

In Part I, Chapters 2 through 4, we introduced the requirements, specifications, and design phases you must successfully address before ever raising the issue of implementation. These phases specify the exact problem to be solved and the design of the software to solve the problem. The most widely used design methodology today is object-oriented design, and the most popular way to express it is with the Unified Modeling Language (UML). Software developers today must not only be familiar with implementation issues but with program specification and design as well.

In object-oriented design, we divide a problem into many smaller units called classes. We must decide on the data structures that represent the information in our classes and the algorithms that our methods will use to access and modify these structures. In Part II, Chapters 5 through 9, we introduced a taxonomy of data structures and showed that many of these data structures already exist as part of the Java Collections Framework. Today, it is more common to use libraries, such as the Java Collections Framework, than to design and implement data structures from scratch. Software developers today must be comfortable using the libraries provided by the language they use to write programs.

Finally, once you have selected your data structures and algorithms, you must implement them in Java. Of course, this phase will use all the basic programming concepts you learned in your first course—things like iteration, conditionals, declarations, scope, methods, and parameters. However, software implementation involves much more than these basic issues. Part III, Chapters 10 through 13, introduced some important programming techniques that are widely used in software development today. These techniques include exception handling, streams, threads, GUIs, network programming, and system security. Many of these topics were once considered "advanced subjects" that were not even introduced until much later in the computer science curriculum. However, their central importance today makes it essential to introduce them in the earliest courses, with future classes building on this base and providing more advanced material. Students must, from the very beginning, be comfortable writing simple, fault-tolerant, multithreaded, distributed software with a powerful visual interface.

To summarize, the text presented the following key points:

- Familiarity with requirement and specification documents
- Knowledge of object-oriented design and its representation in a formal notation such as UML
- The ability to implement an object-oriented design in a language such as Java that supports object-oriented programming
- Understanding the basic data structures in common use, including lists, stacks, queues, trees, sets, maps, and graphs
- The ability to use data structure resources provided by a package such as the Java Collections Framework
- The ability to use modern programming concepts such as exception handling, streams, threads, networking, and toolkits for building user interfaces

Now that is what modern software development is all about!

EXERCISES

1 This chapter briefly described the Session and Presentation layers. Use the Web to learn more about these layers.

2 Clearly, one advantage of building a network as a series of layers is that it makes the software easier to understand. Can you name some disadvantages to this approach?

3 Many people add an eighth layer to the OSI model. The layer is called the Media Access Control (MAC) layer, and it fits between the Data Link and Network layers. What services does the MAC layer provide?

4 Obtain a copy of RFC 768 that defines UDP. How many bytes are in the header of a UDP datagram? Describe what a checksum is and how it is computed in UDP.

5 What is the Dynamic Host Configuration Protocol (DHCP)? How is it used in a computing system?

6 Search the Web for information on the transport protocols used in a Novell network. How are the protocols in a Novell network similar to TCP/IP, and how do they differ? List the advantages of using each family of protocols.

7 An echo server "echoes" any packets it receives back to the sender. An echo server provides a way to test basic network connectivity. Write a UDP-based echo client (you may want to base your program on the UDP day/time client in Figure 13-20). Use your echo client to determine the average amount of time it takes to get a response from a variety of echo servers.

8 The chargen service on many UNIX machines can generate a stream of data to test the basic operation of a network. Like the day/time service, chargen is available for both UDP and TCP. The TCP chargen server accepts connections on port 19 and generates a constant stream of character data until the connection is terminated. The UDP-based version of the service waits for the arrival of a datagram on port 19 and replies by sending a datagram that contains a random number of characters. Write a TCP- and a UDP-based chargen client and test them on your system.

9 Write a TCP- and UDP-based chargen service. Test your servers using the clients you wrote in the previous exercise.

10 Obtain a copy of RFC 1288, which describes the finger service. After reading RFC 1288, write a finger client that can retrieve user data from arbitrary finger servers. Write your program so that it reads the query information from the command line.

11 UDP is an unreliable protocol. How unreliable do you think UDP is? Devise a way to measure how many packets are dropped in a UDP transfer. Write a set of programs in Java that implement your testing scheme and run them on your network. Be sure to run your tests on a pair of machines that are on the same network and then on a pair of machines on different networks. How do the error rates compare?

12 Earlier in this chapter, you learned that more overhead was associated with TCP than with UDP. Devise a way to measure the overall speed of a UDP and a TCP connection. For UDP, be sure to take into account that packets might be lost. Write a suite of programs that implements your testing strategy. What conclusions can you draw from your experiments?

13 Using the URL classes described in Section 13.5, write a program that prints the title of a Web page, given its URL. Write the program to read the URL from the command line.

14 Get a copy of the RFC that describes HTTP, the protocol used to transfer hypertext information. After studying HTTP, write a Java program that serves as a simple Web server. Once your Web server is completed, test it using one of the available Web browsers on your system.

15 A hacker could potentially misuse the Web server you wrote in the previous exercise to obtain unauthorized access to files on your computer. Design a Java security policy that will prevent such access. Run your Web server using your policy and attempt to misuse your server. Are you successful?

16 Trivial File Transfer Protocol (TFTP) uses UDP instead of TCP to transfer files. Obtain a copy of the RFC that defines TFTP and name some of its common uses. Why do you think it is designed to use UDP instead of TCP?

17 Write a TFTP server and a TFTP client and test them on your system. Be sure to define a security policy for your server to prevent unauthorized access.

18 The class `MultiCastSocket` in the `java.net` package allows multicasting. When a host sends a message to a multicast group, the message is delivered to all machines in the group. Write a multicast version of the echo program. What are the advantages and potential disadvantages of multicast?

CHALLENGE WORK EXERCISE

An important network application is remote access to a database via a **database server**. We use this application, for example, to retrieve airline or hotel reservations from a Web site. Typically, a client connects to a Web site and sends a special key value to the server, such as a reservation number. The server uses this key to locate the desired record in its database and returns either the desired information or an error message that the information could not be found.

Write a TCP-based network application that implements a database server for student records. Assume that we maintain the following information for each student registered at our school:

Student ID:	six-digit integer
Name:	string
Year in School:	integer in the range 1–4
Major:	string
Credits:	integer
GPA:	double

To make this task easier (and to focus on the networking concepts), assume that this information is stored in an array rather than in a file.

Write a TCP server class called `StudentServer` that listens for connection requests from clients. When a connection is established, the client sends it a six-digit ID number. The server accepts this value and searches for this ID number in the array. If the number is found, the server returns the entire student record to the requesting client. If the number is not found, the server returns an appropriate error message.

Next, write a TCP client class called `StudentClient`. The client's job is to connect to `StudentServer`, send it a student ID number, and wait for the response, which is displayed on the screen. The client should then terminate the connection.