3

Internet Applications And Network Programming

3.1 Introduction

The Internet offers users a rich diversity of services that include web browsing, email, text messaging, and video teleconferences. Surprisingly, none of the services is part of the underlying communication infrastructure. Instead, the Internet provides a general purpose communication mechanism on which all services are built, and individual services are supplied by application programs that run on computers attached to the Internet. In fact, it is possible to devise entirely new services without changing the Internet.

This chapter covers two key concepts that explain Internet applications. First, the chapter describes the conceptual paradigm that applications follow when they communicate over the Internet. Second, the chapter presents the details of the *socket Application Programming Interface* (socket API) that Internet applications use.

The chapter demonstrates that one does not need to understand the details of data communication or network protocols to write innovative applications — once a programmer masters a few basic concepts, it is possible to construct applications that communicate over the Internet. The next chapter continues the discussion by examining example Internet applications such as email.

Although programmers can get started easily, and it is possible to create Internet applications without knowing how networks operate, understanding network protocols and technologies allows a programmer to write efficient and reliable code that enables

applications to scale across many sites. Later parts of the text provide the necessary information by explaining data communications and protocols used to form the Internet.

3.2 Two Basic Internet Communication Paradigms

The Internet supports two basic communication paradigms: a *stream* paradigm and a *message* paradigm. Figure 3.1 summarizes the differences.

Stream Paradigm	Message Paradigm
Connection-oriented	Connectionless
1-to-1 communication	Many-to-many communication
Sequence of individual bytes	Sequence of individual messages
Arbitrary length transfer	Each message limited to 64 Kbytes
Used by most applications	Used for multimedia applications
Built on TCP protocol	Built on UDP protocol

Figure 3.1 The two paradigms that Internet applications use.

3.2.1 Stream Transport In The Internet

The term *stream* denotes a paradigm in which a sequence of bytes flows from one application program to another. In fact, the Internet's mechanism arranges two streams between a pair of communicating applications, one in each direction. For example, a browser uses the stream service to communicate with a web server: the browser sends a request and the web server responds by sending the page. The network accepts input from either application, and delivers the data to the other application.

The stream mechanism transfers a sequence of bytes without attaching meaning to the bytes and without inserting boundaries. In particular, a sending application can choose to generate one byte at a time, or can generate blocks of bytes. The network chooses the number of bytes to deliver at any time. That is, the network can choose to combine smaller blocks into one large block or can divide a large block into smaller blocks. The point is:

Although it delivers all bytes in sequence, the stream paradigm does not guarantee that the chunks of bytes passed to a receiving application correspond to the chunks of bytes transferred by the sending application.

3.2.2 Message Transport In The Internet

The alternative Internet communication mechanism follows a *message paradigm* in which the network accepts and delivers messages. Each message delivered to a receiver corresponds to a message that was transmitted by a sender; the network never delivers part of a message, nor does it join multiple messages together. Thus, if a sender places exactly n bytes in an outgoing message, the receiver will find exactly n bytes in the incoming message.

The message paradigm allows unicast, multicast, or broadcast delivery. That is, a message can be sent from an application on one computer directly to an application on another, the message can be broadcast to all computers on a given network, or the message can be multicast to some of the computers on a network. Furthermore, applications on many computers can send messages to a given application. Thus, the message paradigm can provide 1-to-1, 1-to-many, or many-to-1 communication.

Surprisingly, the message service does not make any guarantees about the order in which messages are delivered or whether a given message will arrive. The service permits messages to be:

- Lost (i.e., never delivered)
- Duplicated (more than one copy arrives)
- Delivered out-of-order

A programmer who uses the message paradigm must insure that the application operates correctly, even if packets are lost or reordered†. Because most applications require delivery guarantees, programmers tend to use the stream service except in special situations, such as video, where multicast is needed and the application provides support to handle packet reordering and loss. Thus, we will focus on the stream paradigm.

3.3 Connection-oriented Communication

The Internet stream service is *connection-oriented*, which means the service operates analogous to a telephone call: before they can communicate, two applications must request that a *connection* be created. Once it has been established, the connection allows the applications to send data in either direction. Finally, when they finish communicating, the applications request that the connection be terminated. Algorithm 3.1 summarizes the interaction.

†Later chapters explain why such errors can occur.

Algorithm 3.1

Purpose:

Interaction over a connection-oriented mechanism

Method:

A pair of applications requests a connection
The pair uses the connection to exchange data
The pair requests that the connection be terminated

Algorithm 3.1 Communication over a connection-oriented mechanism.

3.4 The Client-Server Model Of Interaction

The first step in Algorithm 3.1 raises a question: how can a pair of applications that run on two independent computers coordinate to guarantee that they request a connection at the same time? The answer lies in a form of interaction known as the *client-server model*. One application, known as a *server*, starts first and awaits contact. The other application, known as a *client*, start second and initiates the connection. Figure 3.2 summarizes the interaction.

Server Application	Client Application
Starts first	Starts second
Does not need to know which client will contact it	Must know which server to contact
Waits passively and arbitrarily long for contact from a client	Initiates a contact whenever communication is needed
Communicates with a client by both sending and receiving data	Communicates with a server by sending and receiving data
Stays running after servicing one client, and waits for another	May terminate after interacting with a server

Figure 3.2 A summary of the client-server model.

Subsequent sections describe how specific services use the client-server model. For now, it is sufficient to understand:

Although it provides basic communication, the Internet does not initiate contact with, or accept contact from, a remote computer; application programs known as a clients and severs handle all services.

3.5 Characteristics Of Clients And Servers

Although minor variations exist, most instances of client-server interaction have the same general characteristics. In general, client software:

- Is an arbitrary application program that becomes a client temporarily when remote access is needed, but also performs other computation
- Is invoked directly by a user, and executes only for one session
- Runs locally on a user's personal computer
- Actively initiates contact with a server
- Can access multiple services as needed, but usually contacts one remote server at a time
- Does not require especially powerful computer hardware

In contrast, server software:

- Is a special-purpose, privileged program dedicated to providing one service that can handle multiple remote clients at the same time
- Is invoked automatically when a system boots, and continues to execute through many sessions
- Runs on a large, powerful computer
- Waits passively for contact from arbitrary remote clients
- Accepts contact from arbitrary clients, but offers a single service
- Requires powerful hardware and a sophisticated operating system

3.6 Server Programs And Server-Class Computers

Confusion sometimes arises over the term *server*. Formally, the term refers to a program that waits passively for communication, and not to the computer on which it executes. However, when a computer is dedicated to running one or more server programs, the computer itself is sometimes called a *server*. Hardware vendors contribute to the confusion because they classify computers that have fast CPUs, large memories, and powerful operating systems as *server* machines. Figure 3.3 illustrates the definitions.

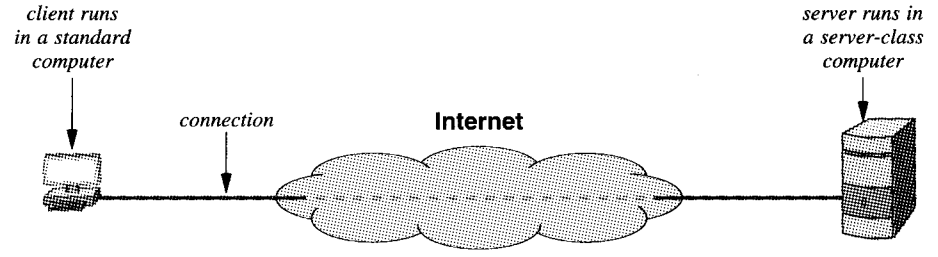

Figure 3.3 Illustration of a client and server.

3.7 Requests, Responses, And Direction Of Data Flow

The terms *client* and *server* arise from which side initiates contact. Once contact has been established, two-way communication is possible (i.e., data can flow from a client to a server or from a server to a client). Typically, a client sends a request to a server, and the server returns a response to the client. In some cases, a client sends a series of requests and the server issues a series of responses (e.g., a database client might allow a user to look up more than one item at a time). The concept can be summarized:

Information can flow in either or both directions between a client and server. Although many services arrange for the client to send one or more requests and the server to return responses, other interactions are possible.

3.8 Multiple Clients And Multiple Servers

A client or server consists of an application program, and a computer can run multiple applications at the same time. As a consequence, a computer can run:

- A single client
- A single server
- Multiple copies of a client that contact a given server
- Multiple clients that each contact a particular server
- Multiple servers, each for a particular service

Allowing a computer to operate multiple clients is useful because services can be accessed simultaneously. For example, a user can have three windows open simultaneously running three applications: one that retrieves and displays email, another that connects to a chat service, and a third running a web browser. Each application is a client

that contacts a particular server independent of the others. In fact, the technology allows a user to have two copies of a single application open, each contacting a server (e.g., two copies of a web browser).

Allowing a given computer to operate multiple servers is useful because the hardware can be shared. In addition, a single computer has lower system administration overhead than multiple computer systems. More important, experience has shown that the demand for a server is often sporadic — a server can remain idle for long periods of time. An idle server does not use the CPU while waiting for a request to arrive. Thus, if demand for services is low, consolidating servers on a single computer can dramatically reduce cost without significantly reducing performance. To summarize:

A single, powerful computer can offer multiple services at the same time; a separate server program is needed for each service.

3.9 Server Identification And Demultiplexing

How does a client identify a server? The Internet protocols divide identification into two pieces:

- An identifier for the computer on which a server runs
- An identifier for a particular service on the computer

Identifying A Computer. Each computer in the Internet is assigned a unique 32-bit identifier known as an *Internet Protocol address* (*IP address*)†. When it contacts a server, a client must specify the server's IP address. To make server identification easy for humans, each computer is also assigned a name, and the Domain Name System described in Chapter 4 is used to translate a name into an address. Thus, a user specifies a name such as *www.cisco.com* rather than an integer address.

Identifying A Service. Each service available in the Internet is assigned a unique 16-bit identifier known as a *protocol port number* (often abbreviated *port number*). For example, email is assigned port number 25, and the web is assigned port number 80. When a server begins execution, it registers with its local system by specifying the port number for the service it offers. When a client contacts a remote server to request service, the request contains a port number. Thus, when a request arrives at a server, software on the server uses the port number in the request to determine which application on the server computer should handle the request.

Figure 3.4 summarizes the discussion by listing the basic steps a client and server take to communicate.

†Chapter 21 explains Internet addresses in detail.

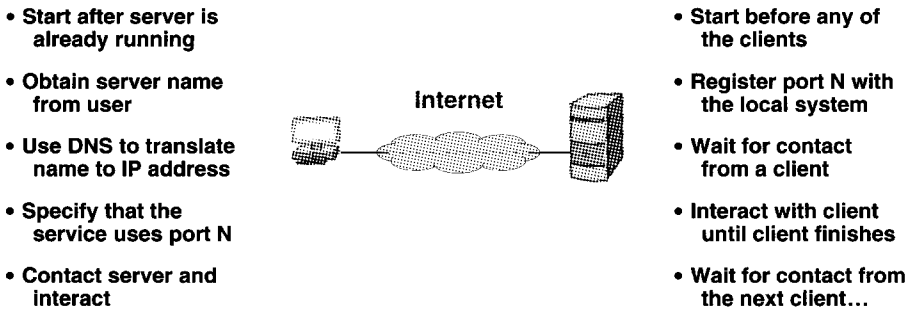

Figure 3.4 The conceptual steps a client and server take to communicate.

3.10 Concurrent Servers

The steps in Figure 3.4 imply that a server handles one client at a time. Although a *serial* approach works in a few trivial cases, most servers are *concurrent*. That is, a server uses more than one *thread of control*†, to handle multiple clients at the same time.

To understand why simultaneous service is important, consider what happens if a client downloads a movie from a server. If a server handles one request at a time, all clients must wait while the server transfers the movie. In contrast, a concurrent server does not force a client to wait. Thus, if a second client arrives and requests a short download (e.g., a single song), the second request will start immediately, and may finish before the movie transfer completes.

The details of concurrent execution depend on the operating system being used, but the idea is straightforward: concurrent server code is divided into two pieces, a main program (thread) and a handler. The main thread merely accepts contact from a client, and creates a thread of control for the client. Each thread of control interacts with a single client, and runs the handler code. After handling one client, the thread terminates. Meanwhile, the main thread keeps the server alive — after creating a thread to handle a request, the main thread waits for another request to arrive.

Note that if N clients are simultaneously using a concurrent server, $N+1$ threads will be running: the main thread is waiting for additional requests, and N threads are each interacting with a single client. We can summarize:

A concurrent server uses threads of execution to handle requests from multiple clients at the same time.

†Some operating systems use the term *thread of execution* or *process* to denote a thread of control.

3.11 Circular Dependencies Among Servers

Technically, any program that contacts another is acting as a client, and any program that accepts contact from another is acting as a server. In practice, the distinction blurs because a server for one service can act as a client for another. For example, before it can fill in a web page, a web server may need to become a client of a database. A server may also become the client of a security service (e.g., to verify that a client is allowed to access the service).

Of course, programmers must be careful to avoid circular dependencies among servers. For example, consider what can happen if a server for service X_1 becomes a client of service X_2, which becomes a client of service X_3, which becomes a client of X_1. The chain of requests can continue indefinitely until all three servers exhaust resources. The potential for circularity is especially high when services are designed independently because no single programmer controls all servers.

3.12 Peer-To-Peer Interactions

If a single server provides a given service, the network connection between the server and the Internet can become a bottleneck. Figure 3.5 illustrates the architecture.

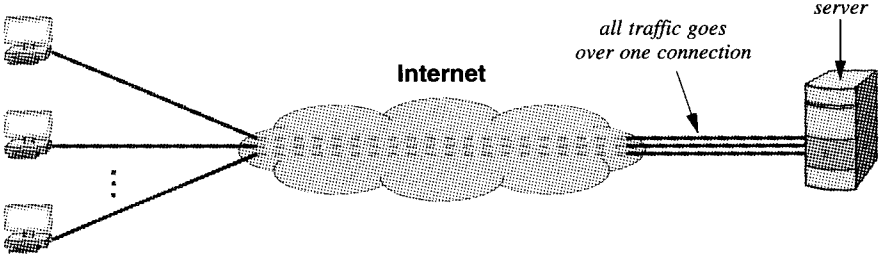

Figure 3.5 The traffic bottleneck in a design that uses a single server.

The question arises, "can Internet services be provided without creating a central bottleneck?" One way to avoid a bottleneck forms the basis of file sharing applications. Known as a *peer-to-peer* (*p2p*) architecture, the scheme avoids placing data on a central server. Conceptually, data is distributed equally among a set of N servers, and each client request is sent to the appropriate server. Because a given server only provides $1/N$ of the data, the amount of traffic between a server and the Internet is $1/N$ as much as in the single-server architecture. Thus, server software can run on the same computers as clients. Figure 3.6 illustrates the architecture.

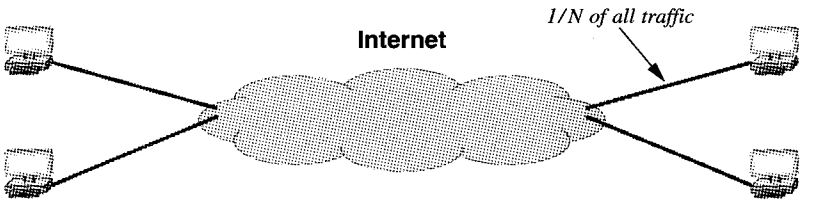

Figure 3.6 Interaction in a peer-to-peer system.

3.13 Network Programming And The Socket API

The interface an application uses to specify communication is known as an *Application Program Interface* (*API*)†. Although the exact details of an API depend on the operating system, one particular API has emerged as the de facto standard for software that communicates over the Internet. Known as the *socket API*, and commonly abbreviated *sockets*, the API is available for many operating systems, such as Microsoft's Windows systems as well as various UNIX systems, including Linux. The point is:

The socket API is a de facto standard for Internet communication.

3.14 Sockets, Descriptors, And Network I/O

Because it was originally developed as part of the UNIX operating system, the socket API is integrated with I/O. In particular, when an application creates a *socket* to use for Internet communication, the operating system returns a small integer *descriptor* that identifies the socket. The application then passes the descriptor as an argument when it calls functions to perform an operation on the socket (e.g., to transfer data across the network or to receive incoming data).

In many operating systems, socket descriptors are integrated with other I/O descriptors. As a result, an application can use the *read* and *write* operations for socket I/O or I/O to a file. To summarize:

When an application creates a socket, the operating system returns a small integer descriptor that the application uses to reference the socket.

†Appendix 1 contains a simplified API (with only seven functions) and example code that demonstrates how such an API can be used to create Internet applications, including a working web server.