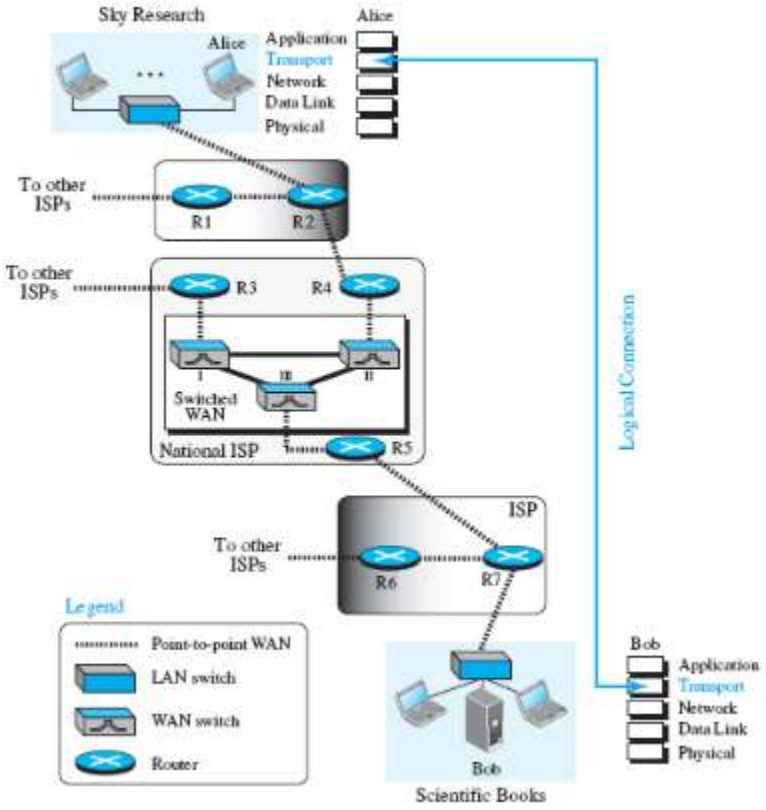
CHAPTER 9
TRANSPORT LAYER:

9.1 TRANSPORT-LAYER SERVICES

The transport layer is located between the network layer and the application layer. The transport layer is responsible for providing services to the application layer; it receives services from the network layer. Figure 9.1 shows the communication between Alice and Bob at the transport layer.

Figure 9.1 *Logical connection at the transport layer*



9.1.1 Process-to-Process Communication

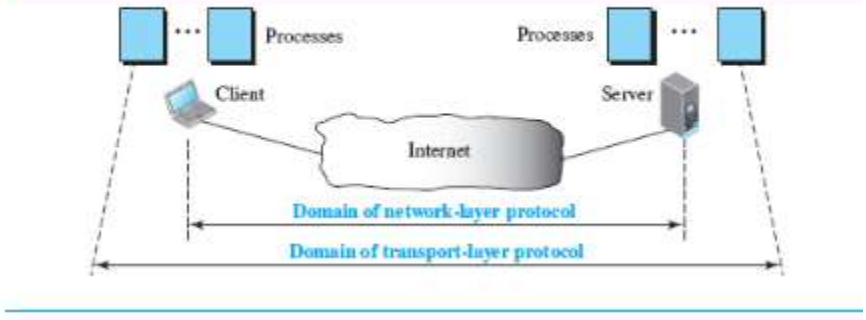
The first duty of a transport-layer protocol is to provide **process-to-process communication**. A process is an application-layer entity (running program) that uses the services of the transport layer. Before we discuss how process-to-process communication can be accomplished, we need to understand the difference between host-to-host communication and process-to-process communication.

The network layer is responsible for communication at the computer level (host-to-host communication). A network-layer protocol can deliver the message only to the destination computer. However, this is an incomplete delivery. The message still needs to be handed to the correct process. This is where a transport-layer protocol takes over. A transport-layer protocol is

responsible for delivery of the message to the appropriate process. Figure 9.2 shows the domains of a network layer and a transport layer.

Figure 9.2 *Network layer versus transport layer*

Figure 9.2 Network layer versus transport layer



9.1.2 Addressing: Port Numbers

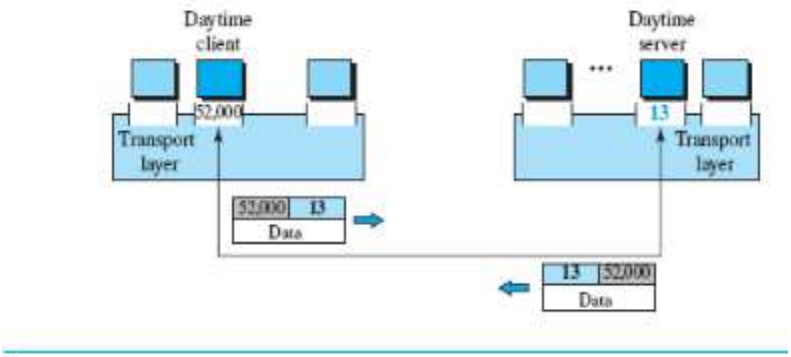
Although, there are a few ways to achieve process-to-process communication, the most common is through the **client/server paradigm** (see Chapter 10). A process on the local host, called a *client*, needs services from a process usually on the remote host, called a *server*.

However, operating systems today support both multiuser and multiprogramming environments. A remote computer can run several server programs at the same time, just as several local computers can run one or more client programs at the same time. For communication, we must define the local host, local process, remote host, and remote process. The local host and the remote host are defined using IP addresses (discussed in Chapter 7). To define the processes, we need second identifiers, called **port numbers**. In the TCP/IP protocol suite, the port numbers are integers between 0 and 65,535 (16 bits).

The client program defines itself with a port number, called the **ephemeral port number.** The word *ephemeral* means "short-lived" and is used because the life of a client is normally short. An ephemeral port number is recommended to be greater than 1023 for some client/server programs to work properly.

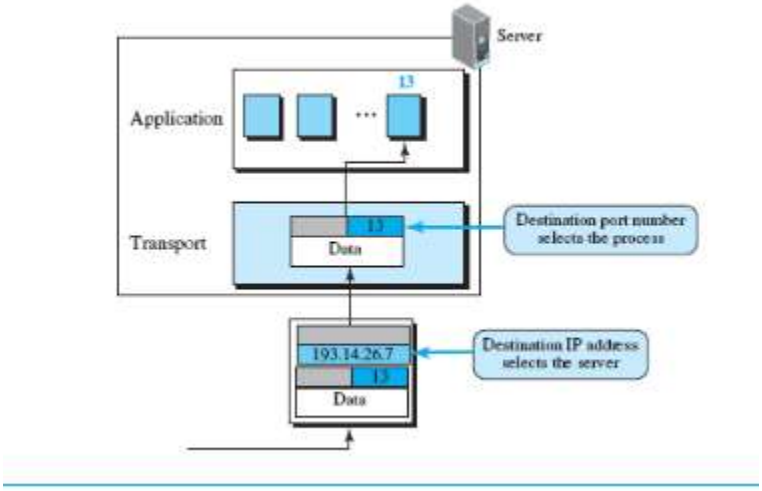
The server process must also define itself with a port number. This port number, however, cannot be chosen randomly. If the computer at the server site runs a server process and assigns a random number as the port number, the process at the client site that wants to access that server and use its services will not know the port number. TCP/IP has decided to use universal port numbers for servers; these are called **well-known port numbers.** Every client process knows the well-known port number of the corresponding server process. For example, while the daytime client process, an application program, can use an ephemeral (temporary) port number, 52,000, to identify itself, the daytime server process must use the well-known (permanent) port number 13. Figure 9.3 shows this concept.

Figure 9.3 *Port numbers*



It should be clear by now that the IP addresses and port numbers play different roles in selecting the final destination of data. The destination IP address defines the host among the different hosts in the world. After the host has been selected, the port number defines one of the processes on this particular host (see Figure 9.4).

Figure 9.4 *IP addresses versus port numbers*



ICANN Ranges

The Internet Corporation for Assigned Names and Numbers (ICANN) has divided the port numbers into three ranges: well-known, registered, and dynamic (or private), as shown in Figure 9.5.

Figure 9.5 *ICANN ranges*

Figure 9.5 *ICANN ranges*

- Well-known ports. The ports ranging from 0 to 1023 are assigned and controlled by ICANN. These are the well-known ports.
- Registered ports. The ports ranging from 1024 to 49,151 are not assigned or controlled by ICANN. They can only be registered with ICANN to prevent duplication.
- Dynamic ports. The ports ranging from 49,152 to 65,535 are neither controlled nor socket addresses.

A transport-layer protocol in the TCP suite needs both the IP address and the port number, at each end, to make a connection. The combination of an IP address and a port number is called a **socket address**. The client socket address defines the client process uniquely, just as the server socket address defines the server process uniquely (see Figure 9.6).

Figure 9.6 *Socket address*

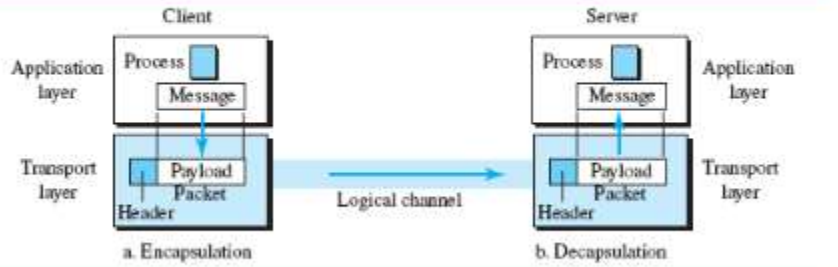
Figure 9.6 *Socket address*

To use the services of the transport layer in the Internet, we need a pair of socket addresses: the client socket address and the server socket address. These four pieces of information are part of the network-layer packet header and the transport-layer packet header. The first header contains the IP addresses; the second header contains the port numbers.

9.1.3 Encapsulation and Decapsulation

To send a message from one process to another, the transport-layer protocol encapsulates and decapsulates messages (Figure 9.7 on next page). Encapsulation happens at the sender site. When a process has a message to send, it passes the message to the transport layer along with a pair of socket addresses and some other pieces of information, which depend on the transport-layer protocol. The transport layer receives the data and adds the transport-layer header.

Figure 9.7 *Encapsulation and decapsulation*



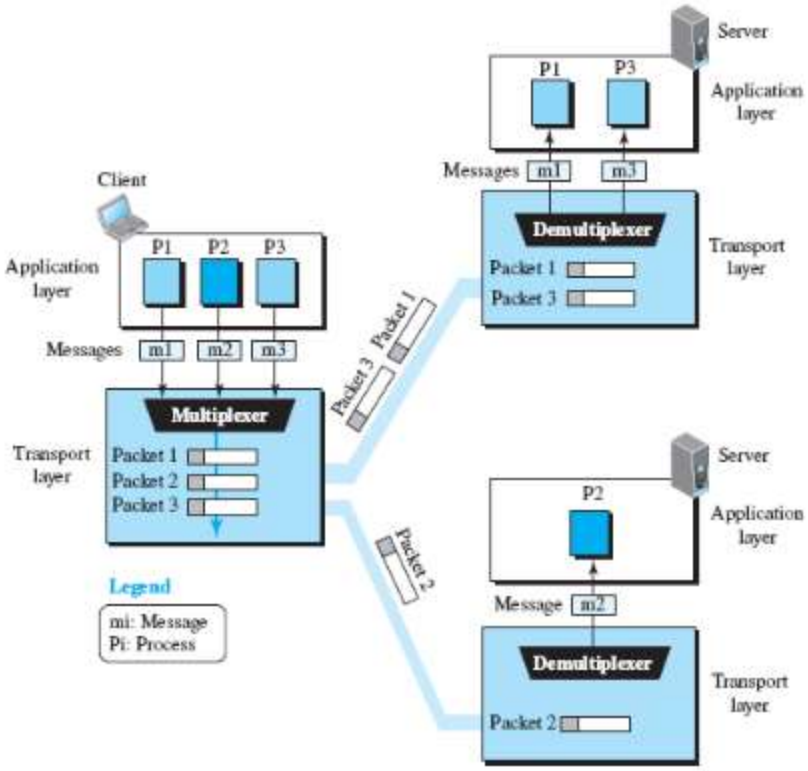
Decapsulation happens at the receiver site. When the message arrives at the destination transport layer, the header is dropped and the transport layer delivers the message to the process running at the application layer. The sender socket address is passed to the process in case it needs to respond to the message received.

9.1.4 Multiplexing and Demultiplexing

Whenever an entity accepts items from more than one source, this is referred to as **multiplexing** (many to one); whenever an entity delivers items to more than one source, this is referred to as **demultiplexing** (one to many). The transport layer at the source performs multiplexing; the transport layer at the destination performs demultiplexing (Figure 9.8).

Figure 9.8 shows communication between a client and two servers. Three client processes are running at the client site: P1, P2, and P3. The processes P1 and P3 need to send requests to the corresponding server process running in a server. The client process P2 needs to send a request to the corresponding server process running at another server. The transport layer at the client site accepts three messages from the three processes and creates three packets. It acts as a *multiplexer*. Packets 1 and 3 use the same logical channel to reach the transport layer of the first server. When they arrive at the server, the transport layer does the job of a *demultiplexer* and distributes the messages to two different processes. The transport layer at the second server receives packet 2 and delivers it to the corresponding process. Note that we still have demultiplexing although there is only one message.

Figure 9.8 *Multiplexing and demultiplexing*



9.1.5 Flow Control

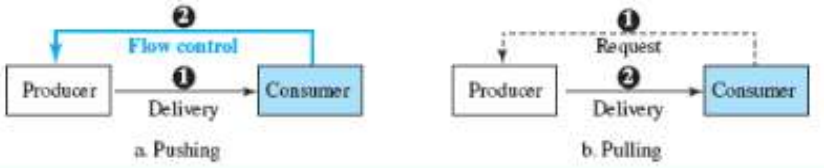
Whenever an entity produces items and another entity consumes them, there should be a balance between production and consumption rates. If the items are produced faster than they can be consumed, the consumer can be overwhelmed and may need to discard some items. If the items are produced more slowly than they can be consumed, the consumer must wait, and the system becomes less efficient. Flow control is related to the first issue. We need to prevent loss of the data items at the consumer site.

Pushing or Pulling

Delivery of items from a producer to a consumer can occur in one of two ways: *pushing* or *pulling*. If the sender delivers items whenever they are produced—without a prior request from the consumer—the delivery is referred to as *pushing*. If the producer delivers the items after the consumer has requested them, the delivery is referred to as *pulling*. Figure 9.9 shows these two types of deliveries.

Figure 9.9 *Pushing or pulling*

Figure 9.9 *Pushing or pulling*



When the producer *pushes* the items, the consumer may be overwhelmed and there is a need for flow control, in the opposite direction, to prevent the items being discarded. In other words, the consumer needs to warn the producer to stop the delivery and to inform it when it is ready again to receive the items. When the consumer pulls the items, it requests them when it is ready. In this case, there is no need for flow control.

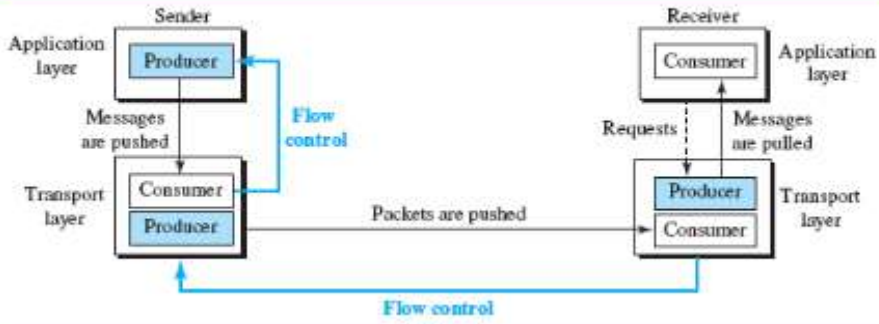
Handling Flow Control

In communication at the transport layer, we are dealing with four entities: sender process, sender transport layer, receiver transport layer, and receiver process. The sending process at the application layer is only a producer. It produces message chunks and pushes them to the transport layer. The sending transport layer has a double role: It is both a consumer and a producer. It consumes the messages pushed by the producer. It encapsulates the messages in packets and pushes them to the receiving transport layer. The receiving transport layer also has a double role: It is the consumer for the packets received from the sender and the producer that decapsulates the messages and delivers them to the application layer. The last delivery, however, is normally a pulling delivery; the transport layer waits until the application-layer process asks for messages.

Figure 9.10 shows that we need at least two cases of flow control: from the sending transport layer to the sending application layer and from the receiving transport layer to the sending transport layer.

Figure 9.10 *Flow control at the transport layer*

Figure 9.10 *Flow control at the transport layer*



Buffers

Although flow control can be implemented in several ways, one of the solutions is normally to use two *buffers*: one at the sending transport layer and the other at the receiving transport layer. A buffer is a set of memory locations that can hold packets at the sender and receiver. The flow-control communication can occur by sending signals from the consumer to the producer.

When the buffer of the sending transport layer is full, it informs the application layer to stop passing chunks of messages; when there are some vacancies, it informs the application layer that it can pass message chunks again.

When the buffer of the receiving transport layer is full, it informs the sending transport layer to stop sending packets. When there are some vacancies, it informs the sending transport layer that it can send packets again.

Example 9.1

The preceding discussion requires that consumers communicate with the producers on two occasions: when the buffer is full and when there are vacancies. If the two parties use a buffer with only one slot, the communication can be easier. Assume that each transport layer uses one single memory location to hold a packet. When this single slot in the sending transport layer is empty, the sending transport layer sends a note to the application layer to send its next chunk; when this single slot in the receiving transport layer is empty, it sends an acknowledgment to the sending transport layer to send its next packet. However, this type of flow control, using a single-slot buffer at the sender and the receiver, is inefficient.

9.1.6 Error Control

In the Internet, because the underlying network layer (IP) is unreliable, we need to make the transport layer reliable if the application requires reliability. Reliability can be achieved to add error-control services to the transport layer. Error control at the transport layer is responsible for

1. **Detecting and discarding corrupted packets**
2. **Keeping track of lost and discarded packets and resending them**
3. **Recognizing duplicate packets and discarding them**
4. **Buffering out-of-order packets until the missing packets arrive**

Error control, unlike flow control, involves only the sending and receiving transport layers. We are assuming that the message chunks exchanged between the application and transport layers are error-free. Figure 9.11 shows the error control between the sending and receiving transport layers. As with the case of flow control, the receiving transport layer manages error control, most of the time, by informing the sending transport layer about the problems.

Figure 9.11 *Error control at the transport layer*

Figure 9.11 *Error control at the transport layer*

9.1.8 Congestion Control

An important issue in a packet-switched network, such as the Internet, is **congestion**. Congestion in a network may occur if the *load* on the network—the number of packets sent to the network—is greater than the *capacity* of the network—the number of packets a network can handle. **Congestion control** refers to the mechanisms and techniques that control the congestion and keep the load below the capacity.

We may ask why there is congestion in a network. Congestion happens in any system that involves waiting. For example, congestion can happen on a freeway because any abnormality in the flow, such as that caused by an accident during rush hour, can create a blockage.

Congestion in a network or internetwork occurs because routers and switches have queues—buffers that hold the packets before and after processing. A router, for example, has an input queue and an output queue for each interface. If a router cannot process the packets at the same rate at which they arrive, the queues become overloaded and congestion occurs. Congestion at the transport layer is actually the result of congestion at the network layer, which manifests itself at the transport layer. We discuss congestion control at the network layer and its causes in Chapter 7. Later in Section 9.3.1, we show how TCP, assuming that there is no congestion control at the network layer, implements its own congestion-control mechanism.

9.2 TRANSPORT-LAYER PROTOCOLS

After discussing the general principle behind the transport layer in Section 9.1, we concentrate on the transport protocols in the Internet in this section. Figure 9.17 shows the position of **these three protocols in the TCP/IP protocol suite: UDP, TCP, and SCTP.**

9.2.1 Services

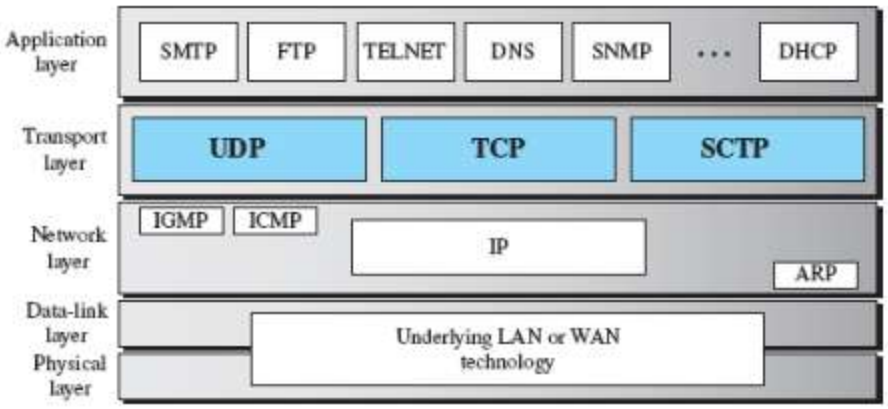
Each protocol provides a different type of service and should be used appropriately.

UDP

UDP is an unreliable connectionless transport-layer protocol used for its simplicity and efficiency in applications where error control can be provided by the application-layer process.

Figure 9.17 *Position of transport-layer protocols in the TCP/IP protocol suite*

Figure 9.17 *Position of transport-layer protocols in the TCP/IP protocol suite*



TCP

TCP is a reliable connection-oriented protocol that can be used in any application where reliability is important.

SCTP

SCTP is a new transport-layer protocol that combines the features of UDP and TCP.

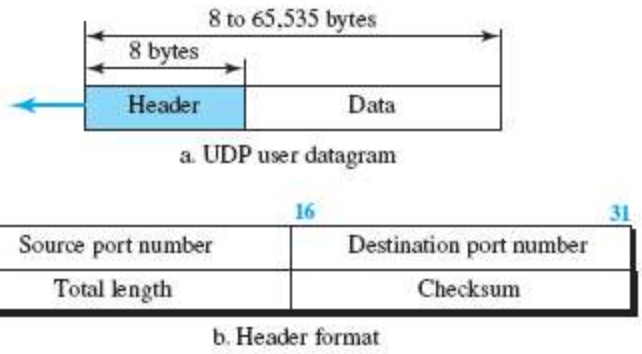
9.3 USER DATAGRAM PROTOCOL (UDP)

The **User Datagram Protocol (UDP)** is a **connectionless, unreliable transport protocol**. It does not add anything to the services of IP except for providing process-to-process communication instead of host-to-host communication. If UDP is so powerless, why would a process want to use it? With the disadvantages come some advantages. UDP is a very simple protocol using a minimum of overhead. If a process wants to send a small message and does not care much about reliability, it can use UDP. Sending a small message using UDP takes much less interaction between the sender and receiver than using TCP.

UDP packets, called **user datagrams**, have a fixed-size header of 8 bytes made up of four fields, each of 2 bytes (16 bits). Figure 9.18 shows the format of a user datagram. The first two fields define the source and destination port numbers, respectively. The third field defines the total length of the user datagram, header plus data. The 16 bits can define a total length of 0 to 65,535 bytes. However, the total length needs to be less because a UDP user datagram is stored in an IP datagram with the total length of 65,535 bytes. The last field can carry the optional checksum (explained soon).

Figure 9.18 *User datagram packet format*

Figure 9.18 *User datagram packet format*



9.3.1 UDP Services

Earlier we discussed the general services provided by a transport-layer protocol. In this section, we discuss what portions of those general services are provided by UDP.

Process-to-Process Communication

UDP provides process-to-process communication using socket addresses, which are a combination of IP addresses and port numbers.

Connectionless Services

As mentioned previously, UDP provides a *connectionless service*. This means that each user datagram sent by UDP is an independent datagram. There is no relationship between the different user datagrams even if they are coming from the same source process and going to the same destination program. The user datagrams are not numbered. Also, unlike TCP, there is no connection establishment and no connection termination. This means that each user datagram can travel on a different path.

One of the ramifications of being connectionless is that the process that uses UDP cannot send a stream of data to UDP and expect UDP to chop them into different, related user datagrams. Instead each request must be small enough to fit into one user datagram. Only those processes sending short messages, messages less than 65,507 bytes (65,535 minus 8 bytes for the UDP header and minus 20 bytes for the IP header), can use UDP.

Flow Control

UDP is a very simple protocol. There is no *flow control,* and hence no window mechanism. The receiver may overflow with incoming messages. The lack of flow control means that the process using UDP should provide for this service, if needed.

Error Control

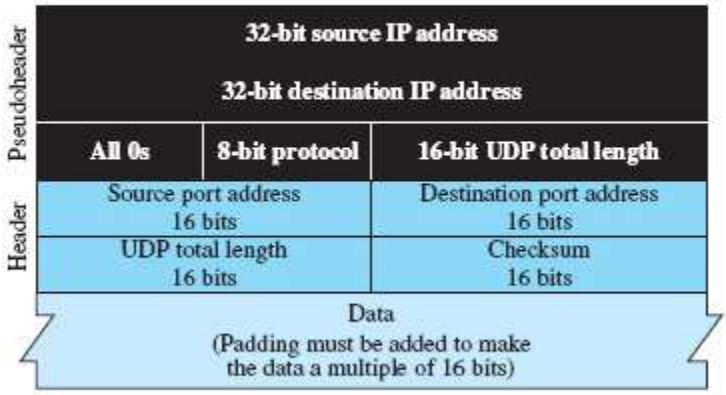
There is no *error-control* mechanism in UDP except for the checksum. This means that the sender does not know if a message has been lost or duplicated. When the receiver detects an error through

the checksum, the user datagram is silently discarded. The lack of error control means that the process using UDP should provide for this service, if needed.

Checksum

We discuss checksum and its calculation in Appendix F. UDP checksum calculation includes three sections: a pseudoheader, the UDP header, and the data coming from the application layer. The *pseudoheader* is the part of the header of the IP packet (discussed in Chapter 8) in which the user datagram is to be encapsulated with some fields filled with 0s (see Figure 9.19).

Figure 9.19 *Pseudoheader for checksum calculation*



If the checksum does not include the pseudoheader, a user datagram may arrive safe and sound. However, if the IP header is corrupted, it may be delivered to the wrong host.

The protocol field is added to ensure that the packet belongs to UDP, and not to TCP. This means that if a process can use either UDP or TCP, the destination port number can be the same. The value of the protocol field for UDP is 17. If this value is changed during transmission, the checksum calculation at the receiver will detect it and UDP drops the packet. It is not delivered to the wrong protocol.

Congestion Control

Because UDP is a connectionless protocol, it does not provide congestion control. UDP assumes that the packets sent are small and sporadic and cannot create congestion in the network. This assumption may or may not be true today, when UDP is used for interactive real-time transfer of audio and video.

Encapsulation and Decapsulation

To send a message from one process to another, UDP encapsulates and decapsulates messages.

Queuing

We have talked about ports without discussing the actual implementation of them. In UDP, **queues are associated with ports.**

At the client site, when a process starts, it requests a port number from the operating system. Some implementations create both an incoming and an outgoing queue associated with each process. Other implementations create only an incoming queue associated with each process.

Multiplexing and Demultiplexing

In a host running a TCP/IP protocol suite, there is only one UDP but possibly several processes that may want to use the services of UDP. To handle this situation, UDP multiplexes and demultiplexes.

Comparison between UDP and Generic Simple Protocol

We can compare UDP with the connectionless simple protocol we discussed earlier. The only difference is that UDP provides an optional checksum to detect corrupted packets at the receiver site. If the checksum is added to the packet, the receiving UDP can check the packet and discard the packet if it is corrupted. No feedback, however, is sent to the sender.

UDP is an example of the connectionless simple protocol we discussed earlier with the exception of an optional checksum added to packets for error detection.

9.3.2 UDP Applications

Although UDP meets almost none of the criteria we mentioned in Section 9.3.1 for a reliable transport-layer protocol, UDP is preferable for some applications. The reason is that some services may have some side effects that are either unacceptable or not preferable. An application designer sometimes needs to compromise to get the optimum. For example, in our daily life, we all know that a one-day delivery of a package by a carrier is more expensive than a three-day delivery. Although time and cost are both desirable features in delivery of a parcel, they are in conflict with each other. We need to choose the optimum.

Example 9.4

A client/server application such as **DNS** (see Chapter 10) uses the services of UDP because a client needs to send a short request to a server and to receive a quick response from it. The request and response can each fit in one user datagram. Because only one message is exchanged in each direction, the connectionless feature is not an issue; the client or server does not worry that messages are delivered out of order.

Example 9.5

A client/server application such as **SMTP, which is used in electronic mail, cannot use the services of UDP** because a user can send a **long e-mail message**, which may include multimedia (images, audio, or video). If the application uses UDP and the message does not fit in one single user datagram, the message must be split by the application into different user datagrams. Here the

connectionless service may create problems. The user datagrams may arrive and be delivered to the receiver application out of order. The receiver application may not be able to reorder the pieces. This means the connectionless service has a disadvantage for an application program that sends long messages. In SMTP, when one sends a message, one does not expect to receive a response quickly (sometimes no response is required). This means that the extra delay inherent in connection-oriented service is not crucial for SMTP.

Example 9.6

Assume we are **downloading a very large text file from the Internet**. We definitely need to use a transport layer that provides reliable service. We don't want part of the file to be missing or corrupted when we open the file. The delay created between the deliveries of the parts is not an overriding concern for us; we wait until the whole file is composed before looking at it. In this case, UDP is not a suitable transport layer.

Example 9.7

Assume we are using a **real-time interactive application, such as Skype**. Audio and video are divided into frames and sent one after another. If the transport layer is supposed to resend a corrupted or lost frame, the synchronizing of the whole transmission may be lost. The viewer suddenly sees a blank screen and needs to wait until the second transmission arrives. This is not tolerable. However, if each small part of the screen is sent using one single user datagram, the receiving UDP can easily ignore the corrupted or lost packet and deliver the rest to the application program. That part of the screen is blank for a very short period of time, which most viewers do not even notice.

Typical Applications

The following shows some typical applications that can benefit more from the services of UDP than from those of TCP.

- UDP is suitable for a process that requires simple request-response communication with little concern for flow and error control. It is not usually used for a process such as FTP that needs to send bulk data (see Chapter 10).
- UDP is a suitable transport protocol for **multicasting.** Multicasting capability is embedded in the UDP software but not in the TCP software.
- UDP is used for management processes such as SNMP (see Chapter 12).
- UDP is used for some route updating protocols such as Routing Information Protocol (RIP) (see Chapter 8).
- UDP is normally used for interactive real-time applications that cannot tolerate uneven delay between sections of a received message (See Chapter 10).

9.4 TRANSMISSION CONTROL PROTOCOL

Transmission Control Protocol (TCP) is a connection-oriented, reliable protocol. TCP explicitly defines connection establishment, data transfer, and connection teardown phases to provide a connection-oriented service. TCP uses a combination of the **Go-Back-*N* (GBN)** and **Selective-Repeat (SR) protocols** to provide reliability. To achieve this goal, TCP uses checksum (for error detection), retransmission of lost or corrupted packets, cumulative and selective acknowledgments, and timers. In this section, we first discuss the services provided by TCP; we then discuss the TCP features in more detail. TCP is the most common transport-layer protocol in the Internet.

9.4.1 TCP Services

Before discussing TCP in detail, let us explain the services offered by TCP to the processes at the application layer.

Process-to-Process Communication

As with UDP, TCP provides process-to-process communication using port numbers. We have already given some of the port numbers used by TCP in Table 9.1 in Section 9.2.2.

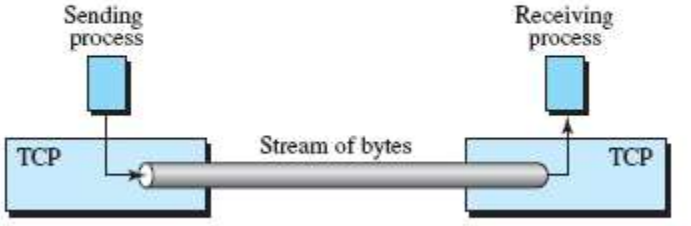
Stream Delivery Service

TCP, unlike UDP, is a stream-oriented protocol. In UDP, a process sends messages with predefined boundaries to UDP for delivery. UDP adds its own header to each of these messages and delivers it to IP for transmission. Each message from the process is called a *user datagram*, and becomes, eventually, one IP datagram. Neither IP nor UDP recognizes any relationship between the datagrams.

TCP, on the other hand, allows the sending process to deliver data as a stream of bytes and allows the receiving process to obtain data as a stream of bytes. TCP creates an environment in which the two processes seem to be connected by an imaginary "tube" that carries their bytes across the Internet. This imaginary environment is depicted in Figure 9.20. The sending process produces (writes to) the stream, and the receiving process consumes (reads from) it.

Figure 9.20 *Stream delivery*

Figure 9.20 *Stream delivery*



Sending and Receiving Buffers

Because the sending and the receiving processes may not necessarily write or read data at the same rate, TCP needs buffers for storage. There are two buffers, the sending buffer and the receiving buffer, one for each direction. One way to implement a buffer is to use a circular array of 1-byte locations as shown in Figure 9.21. For simplicity, we have shown two buffers of 20 bytes each; normally the buffers are hundreds or thousands of bytes, depending on the implementation. We also show the buffers as the same size, which is not always the case.

Figure 9.21 *Sending and receiving buffers*

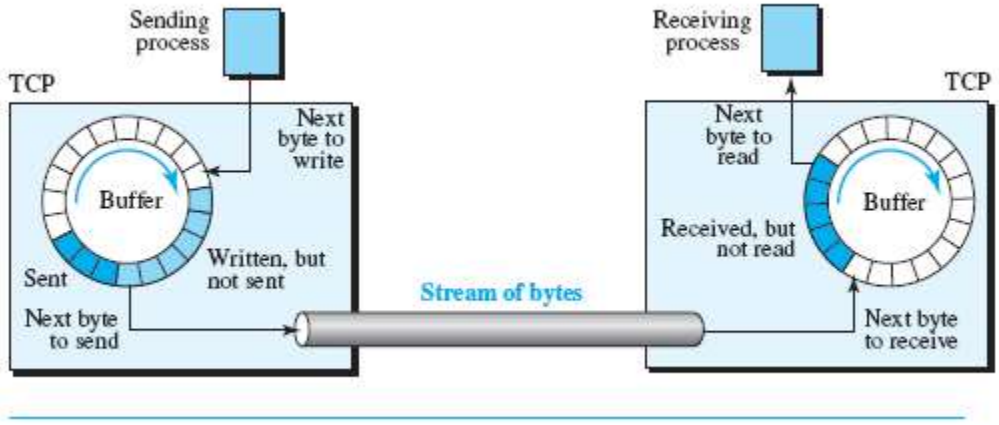


Figure 9.21 shows the movement of the data in one direction. At the sender, the buffer has three types of chambers. The white section contains empty chambers that can be filled by the sending process (producer). The colored area holds bytes that have been sent but not yet acknowledged. The TCP sender keeps these bytes in the buffer until it receives an acknowledgment.

The shaded area contains bytes to be sent by the sending TCP. However, TCP may be able to send only part of this shaded section. This could be due to the slowness of the receiving process or congestion in the network. Also note that, after the bytes in the colored chambers are acknowledged, the chambers are recycled and available for use by the sending process. This is why we show a circular buffer.

The operation of the buffer at the receiver is simpler. The circular buffer is divided into two areas (shown as white and colored in Figure 9.21). The white area contains empty chambers to be filled by bytes received from the network. The colored sections contain received bytes that can be read by the receiving process. When a byte is read by the receiving process, the chamber is recycled and added to the pool of empty chambers.

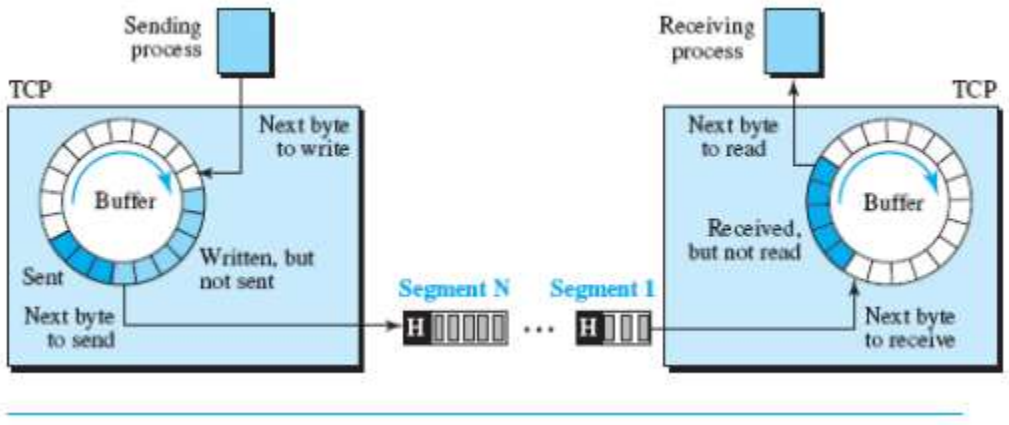
Segments

Although buffering handles the disparity between the speed of the producing and consuming processes, we need one more step before we can send data. The network layer, as a service provider for TCP, needs to send data in packets, not as a stream of bytes. At the transport layer, TCP groups a number of bytes together into a packet called **a *segment***. TCP adds a header to each segment (for control purposes) and delivers the segment to the network layer for transmission. The segments are encapsulated in an IP datagram and transmitted. This entire operation is transparent to the

receiving process. Segments may be received out of order, lost, or corrupted and re-sent. All these are handled by the TCP receiver with the receiving application process unaware of the TCP's activities. Figure 9.22 shows how segments are created from the bytes in the buffers.

Note that segments are not necessarily all the same size. In Figure 9.22, for simplicity, we show one segment carrying 3 bytes and the other carrying 5 bytes. In reality, segments carry hundreds, if not thousands, of bytes.

Figure 9.22 *TCP segments*



Full-Duplex Communication

TCP offers *full-duplex service*, where data can flow in both directions at the same time. Each TCP endpoint then has its own sending and receiving buffer, and segments move in both directions.

Multiplexing and Demultiplexing

Like UDP, TCP performs multiplexing at the sender and demultiplexing at the receiver. However, because TCP is a connection-oriented protocol, a connection needs to be established for each pair of processes.

Connection-Oriented Service

TCP, unlike UDP, is a connection-oriented protocol. When a process at site A wants to send to and receive data from another process at site B, the following three phases occur:

1. **The two TCP's establish a logical connection between them.**
2. **Data are exchanged in both directions.**
3. **The connection is terminated.**

Note that this is a logical connection, not a physical connection. The TCP segment is encapsulated in an IP datagram and can be sent out of order, or lost, or corrupted, and then re-sent. Each may be routed over a different path to reach the destination. There is no physical connection. TCP creates a stream-oriented environment in which it accepts the responsibility of delivering the bytes in order to the other site.

Reliable Service

TCP is a reliable transport protocol. It uses an acknowledgment mechanism to check the safe and sound arrival of data. We will discuss this feature further in the section on error control (Section 9.4.8).

9.4.2 TCP Features

To provide the services mentioned in Section 9.4.1, TCP has several features that are briefly summarized in this section and discussed later in detail.

Numbering System

Although the TCP software keeps track of the segments being transmitted or received, there is no field for a segment number value in the segment header. Instead, there are two fields, called the *sequence number* and *acknowledgment number*, respectively.

Byte Number

TCP numbers all data bytes (octets) that are transmitted in a connection. Numbering is independent in each direction. When TCP receives bytes of data from a process, TCP stores them in the sending buffer and numbers them. The numbering does not necessarily start from 0. Instead, TCP chooses an arbitrary number between 0 and $2^{32} - 1$ for the number of the first byte. For example, if the number happens to be 1057 and the total data to be sent are 6000 bytes, the bytes are numbered from 1057 to 7056. We will see that byte numbering is used for flow and error control.

The bytes of data being transferred in each connection are numbered by TCP. The numbering starts with an arbitrarily generated number.

Sequence Number

After the bytes have been numbered, TCP assigns a sequence number to each segment that is being sent. The sequence number, in each direction, is defined as follows:

1. **The sequence number of the first segment is the initial sequence number (ISN), which is a random number.**
2. **The sequence number of any other segment is the sequence number of the previous segment plus the number of bytes (real or imaginary) carried by the previous segment.**

Example 9.8

Suppose a TCP connection is transferring a file of 5000 bytes. The first byte is numbered 10,001. What are the sequence numbers for each segment if data are sent in five segments, each carrying 1000 bytes?

Solution

The following shows the sequence number for each segment:

Segment 1	→	Sequence number:	10,001	**Range:**	10,001	to	11,000
Segment 2	→	Sequence number:	11,001	**Range:**	11,001	to	12,000
Segment 3	→	Sequence number:	12,001	**Range:**	12,001	to	13,000
Segment 4	→	Sequence number:	13,001	**Range:**	13,001	to	14,000
Segment 5	→	Sequence number:	14,001	**Range:**	14,001	to	15,000

The value in the sequence number field of a segment defines the number assigned to the first data byte contained in that segment.

However, some segments, when carrying only control information, need a sequence number to allow an acknowledgment from the receiver. These segments are used for connection establishment, termination, or abortion. Each of these segments consume one sequence number as though it carries 1 byte, but there are no actual data. We will elaborate on this issue when we discuss connections.

Acknowledgment Number

As we discussed previously, communication in TCP is full duplex; when a connection is established, both parties can send and receive data at the same time. Each party numbers the bytes, usually with a different starting byte number. The sequence number in each direction shows the number of the first byte carried by the segment. Each party also uses an acknowledgment number to confirm the bytes it has received. However, **the acknowledgment number defines the number of the next byte that the party expects to receive.** In addition, the acknowledgment number is cumulative, which means that the party takes the number of the last byte that it has received, safe and sound, adds 1 to it, and announces this sum as the acknowledgment number. The term *cumulative* here means that if a party uses 5643 as an acknowledgment number, it has received all bytes from the beginning up to 5642. Note that this does not mean that the party has received 5642 bytes, because the first byte number does not have to be 0.

The value of the acknowledgment field in a segment defines the number of the next byte a party expects to receive. The acknowledgment number is cumulative.

9.4.3 Segment

Before discussing TCP in more detail, let us discuss the TCP packets themselves. A packet in TCP is called a **segment**.

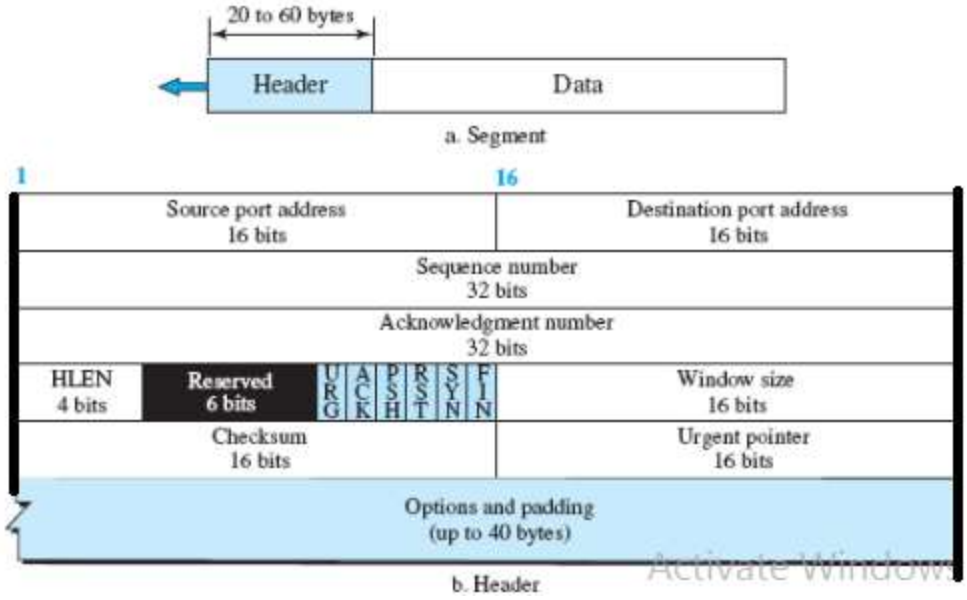
Format

The format of a segment is shown in Figure 9.23. The segment consists of a header of 20 to 60 bytes, followed by data from the application program. The header is 20 bytes if there are no options and up to 60 bytes if it contains options. We will discuss some of the header fields in this section. The meaning and purpose of these will become clearer as we proceed through the section.

- **Source port address.** This is a 16-bit field that defines the port number of the application program in the host that is sending the segment.
- **Destination port address.** This is a 16-bit field that defines the port number of the application program in the host that is receiving the segment.

Figure 9.23 *TCP segment format*

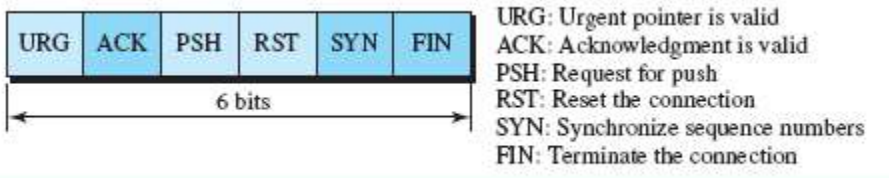
Figure 9.23 TCP segment format



- **Sequence number.** This 32-bit field defines the number assigned to the first byte of data contained in this segment. As we said before, TCP is a stream transport protocol. To ensure connectivity, each byte to be transmitted is numbered. The sequence number tells the destination which byte in this sequence is the first byte in the segment. During connection establishment (discussed in Section 9.4.4) each party uses a random number generator to create an **Initial Sequence Number (ISN)**, which is usually different in each direction.
- **Acknowledgment number.** This 32-bit field defines the byte number that the receiver of the segment is expecting to receive from the other party. If the receiver of the segment has successfully received byte number *x* from the other party, it returns *x* + 1 as the acknowledgment number. Acknowledgment and data can be piggybacked together.
- **Header length.** This 4-bit field indicates the number of 4-byte words in the TCP header. The length of the header can be between 20 and 60 bytes. Therefore, the value of this field is always between 5 (5 × 4 = 20) and 15 (15 × 4 = 60).

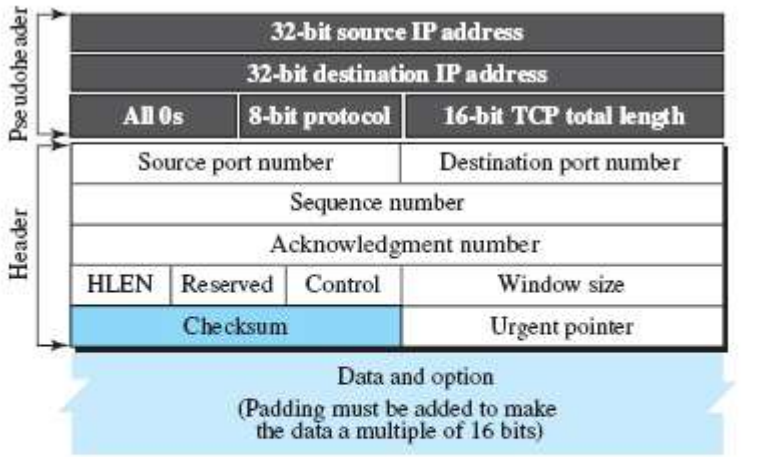
- **Control.** This field defines six different control bits or flags, as shown in Figure 9.24. One or more of these bits can be set at a time. These bits enable flow control, connection establishment and termination, connection abortion, and the mode of data transfer in TCP. A brief description of each bit is shown in Figure 9.24. We will discuss them further when we study the detailed operation of TCP.

Figure 9.24 *Control field*



- **Window size.** This field defines the window size of the sending TCP in bytes. Note that the length of this field is 16 bits, which means that the maximum size of the window is 65,535 bytes. This value is normally referred to as the receiving window (*rwnd*) and is determined by the receiver. The sender must obey the dictation of the receiver in this case.
- **Checksum.** This 16-bit field contains the checksum. The calculation of the checksum for TCP follows the same procedure as the one described for UDP. However, the use of the checksum in the UDP datagram is optional, whereas the use of the checksum for TCP is mandatory. The same pseudoheader, serving the same purpose, is added to the segment. For the TCP pseudoheader, the value for the protocol field is 6. See Figure 9.25.

Figure 9.25 *Pseudoheader added to the TCP datagram*



The use of the checksum in TCP is mandatory.

- **Urgent pointer.** This 16-bit field, which is valid only if the urgent flag is set, is used when the segment contains urgent data. It defines a value that must be added to the sequence number to obtain the number of the last urgent byte in the data section of the segment.
- **Options.** There can be up to 40 bytes of optional information in the TCP header. We will discuss some of the options used in the TCP header later in this section.

Encapsulation

A TCP segment encapsulates the data received from the application layer. The TCP segment is encapsulated in an IP datagram, which in turn is encapsulated in a frame at the data-link layer.

9.4.4 A TCP Connection

TCP is connection-oriented. As discussed before, a connection-oriented transport protocol establishes a logical path between the source and destination. All the segments belonging to a message are then sent over this logical path. Using a single logical pathway for the entire message facilitates the acknowledgment process as well as retransmission of damaged or lost frames. You may wonder how TCP, which uses the services of IP, a connectionless protocol, can be connection-oriented. The point is that a TCP connection is logical, not physical. TCP operates at a higher level. TCP uses the services of IP to deliver individual segments to the receiver, but it controls the connection itself. If a segment is lost or corrupted, it is retransmitted. Unlike TCP, IP is unaware of this retransmission. If a segment arrives out of order, TCP holds it until the missing segments arrive; IP is unaware of this reordering.

In TCP, connection-oriented transmission requires three phases: ***connection establishment, data transfer, and connection termination*.**

Connection Establishment

TCP transmits data in full-duplex mode. When two TCPs in two machines are connected, they are able to send segments to each other simultaneously. This implies that each party must initialize communication and get approval from the other party before any data are transferred.

Three-Way Handshaking

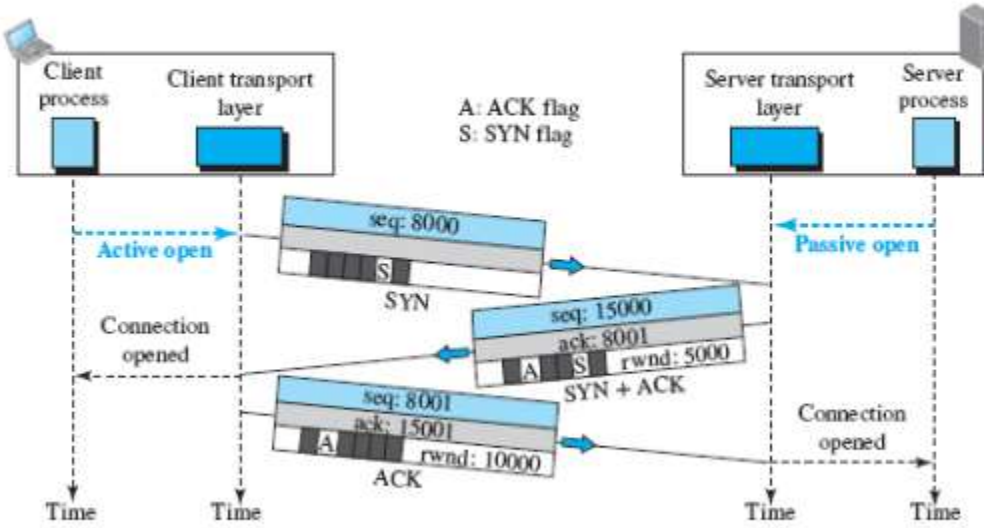
The connection establishment in TCP is called **three-way handshaking**. In our example, an application program, called the client, wants to make a connection with another application program, called the server, using TCP as the transport-layer protocol.

The process starts with the server. The server program tells its TCP that it is ready to accept a connection. This request is called a *passive open*. Although the server TCP is ready to accept a connection from any machine in the world, it cannot make the connection itself.

The client program issues a request for an *active open*. A client that wishes to connect to an open server tells its TCP to connect to a particular server. TCP can now start the three-way handshaking process, as shown in Figure 9.26.

To show the process we use time lines. Each segment has values for all its header fields and perhaps for some of its option fields too. However, we show only the few fields necessary to understand each phase. We show the sequence number, the acknowledgment number, the control flags (only those that are set), and window size if relevant. The three steps in this phase are as follows:

Figure 9.26 *Connection establishment using three-way handshaking*



1. **The client sends the first segment, a SYN segment, in which only the SYN flag is set. This segment is for synchronization of sequence numbers. The client in our example chooses a random number as the first sequence number and sends this number to the server. This sequence number is called the initial sequence number (ISN). Note that this segment does not contain an acknowledgment number. It does not define the window size either; a window size definition makes sense only when a segment includes an acknowledgment. Note that the SYN segment is a control segment and carries no data. However, it consumes one sequence number because it needs to be acknowledged. We can say that the SYN segment carries 1 imaginary byte.**

A SYN segment cannot carry data, but it consumes one sequence number.

2. **The server sends the second segment, a SYN + ACK segment with two flag bits set as SYN and ACK. This segment has a dual purpose. First, it is a SYN segment for communication in the other direction. The server uses this segment to initialize a sequence number for numbering the bytes sent from the server to the client. The server also acknowledges the receipt of the SYN segment from the client by setting the ACK flag and displaying the next sequence number it expects to receive from the client. Because it contains an acknowledgment, it also needs to define the receive window size, *rwnd* (to be used by the client), as we will see in the flow-control section. Because this segment is playing the role of a SYN segment, it needs to be acknowledged. It, therefore, consumes one sequence number.**

A SYN + ACK segment cannot carry data, but it does consume one sequence number.

3. **The client sends the third segment. This is just an ACK segment. It acknowledges the receipt of the second segment with the ACK flag and acknowledgment number field. Note that the ACK segment does not consume any sequence numbers if it does not carry data, but some implementations allow this third segment in the connection phase to carry the first chunk of data from the client. In this case, the segment consumes as many sequence numbers as the number of data bytes.**

If an ACK segment does not carry any data, it does not consume any sequence numbers.

SYN Flooding Attack

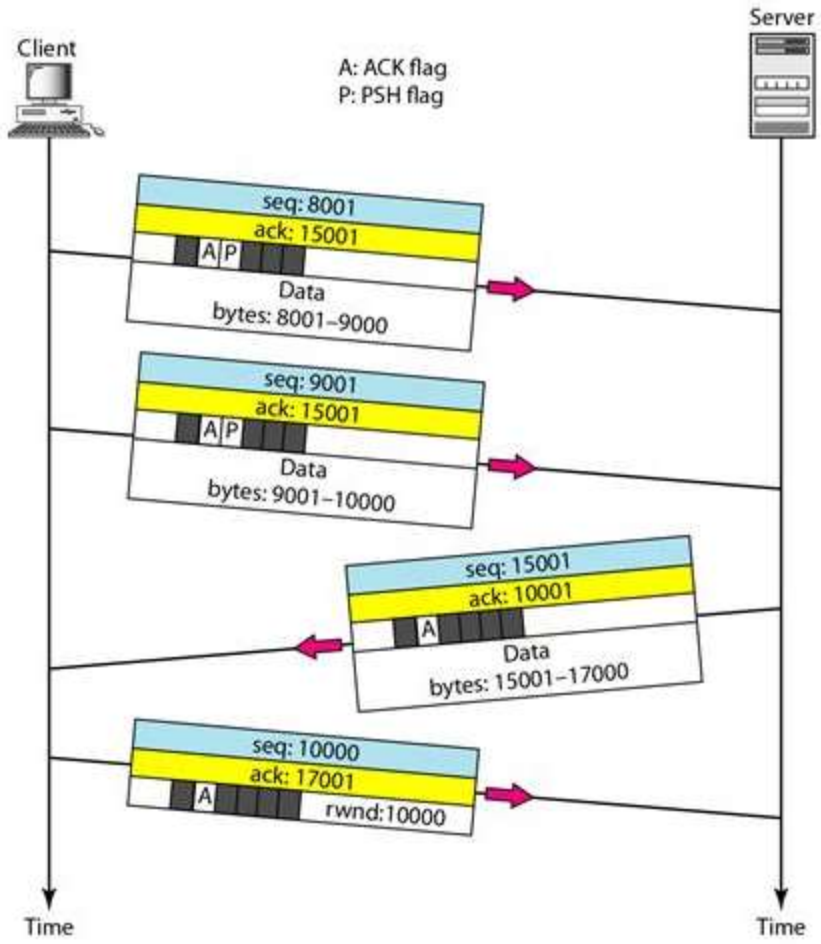
The connection establishment procedure in TCP is susceptible to a serious security problem called a **SYN flooding attack**. This happens when one or more malicious attackers send a large number of SYN segments to a server pretending that each of them is coming from a different client by faking the source IP addresses in the datagrams. The server, assuming that the clients are issuing an active open, allocates the necessary resources, such as creating transfer control block (TCB) tables and setting timers. The TCP server then sends the SYN + ACK segments to the fake clients, which are lost. When the server waits for the third leg of the handshaking process, however, resources are allocated without being used. If, during this short period of time, the number of SYN segments is large, the server eventually runs out of resources and may be unable to accept connection requests from valid clients. This SYN flooding attack belongs to a group of security attacks known as a **denial of service attack**, in which an attacker monopolizes a system with so many service requests that the system overloads and denies service to valid requests.

Some implementations of TCP have strategies to alleviate the effect of a SYN attack. Some have imposed a limit of connection requests during a specified period of time. Others try to filter out datagrams coming from unwanted source addresses. One recent strategy is to postpone resource allocation until the server can verify that the connection request is coming from a valid IP address, by using what is called a **cookie**. SCTP, the new transport-layer protocol that we discuss in Section 9.5 uses this strategy.

Data Transfer

After a connection is established, bidirectional data transfer can take place. The client and server can send data and acknowledgments in both directions. We will study the rules of acknowledgment later in the chapter; for the moment, it is enough to know that data traveling in the same direction as an acknowledgment are carried on the same segment. The acknowledgment is piggybacked with the data. Figure 9.27 shows an example.

Figure 9.27 *Data transfer*



In this example, after a connection is established, the client sends 2000 bytes of data in two segments. The server then sends 2000 bytes in one segment. The client sends one more segment. The first three segments carry both data and acknowledgment, but the last segment carries only an acknowledgment because there is no more data to be sent. Note the values of the sequence and acknowledgment numbers. The data segments sent by the client have the PSH (push) flag set so that the server TCP knows to deliver data to the server process as soon as they are received. We discuss the use of this flag in more detail later. The segment from the server, on the other hand, does not set the push flag. Most TCP implementations have the option to set or not to set this flag.

Pushing Data

We saw that the sending TCP uses a buffer to store the stream of data coming from the sending application program. The sending TCP can select the segment size. The receiving TCP also buffers the data when they arrive and delivers them to the application program when the application program is ready or when it is convenient for the receiving TCP. This type of flexibility increases the efficiency of TCP.

However, there are occasions in which the application program has no need for this flexibility. For example, consider an application program that communicates interactively with another application program on the other end. The application program on one site wants to send a chunk of data to the application at the other site and receive an immediate response. Delayed transmission and delayed delivery of data may not be acceptable by the application program.

TCP can handle such a situation. The application program at the sender can request a *push* operation. This means that the sending TCP must not wait for the window to be filled. It must create a segment and send it immediately. The sending TCP must also set the push bit (PSH) to let the receiving TCP know that the segment includes data that must be delivered to the receiving application program as soon as possible and not to wait for more data to come. This means to change the byte-oriented TCP to a chunk-oriented TCP, but TCP can choose whether or not to use this feature.

Urgent Data

TCP is a stream-oriented protocol. This means that the data are presented from the application program to TCP as a stream of bytes. Each byte of data has a position in the stream. However, there are occasions in which an application program needs to send *urgent* bytes, some bytes that need to be treated in a special way by the application at the other end. The solution is to send a segment with the URG bit set. The sending application program tells the sending TCP that the piece of data is urgent. The sending TCP creates a segment and inserts the urgent data at the beginning of the segment. The rest of the segment can contain normal data from the buffer. The urgent pointer field in the header defines the end of the urgent data (the last byte of urgent data). For example, if the segment sequence number is 15,000 and the value of the urgent pointer is 200, the first byte of urgent data is the byte 15,000 and the last byte is the byte 15,200. The rest of the bytes in the segment (if present) are nonurgent.

It is important to mention that TCP's urgent mode is neither a priority service nor an out-of-band data service as some people think. Rather, TCP urgent mode is a service by which the application program at the sender side marks some portion of the byte stream as needing special treatment by the application program at the receiver side. The receiving TCP delivers bytes (urgent or nonurgent) to the application program in order, but inform the application program about the beginning and end of urgent data. It is left to the application program to decide what to do with the urgent data.

Connection Termination

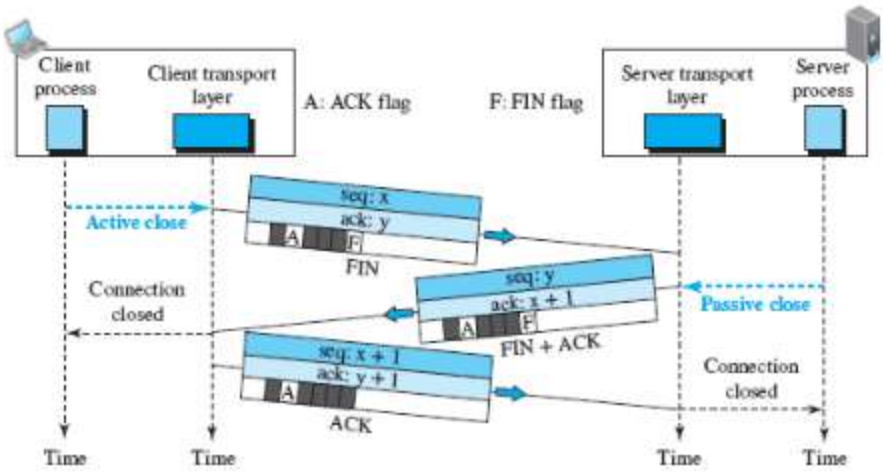
Either of the two parties involved in exchanging data (client or server) can close the connection, although it is usually initiated by the client. Most implementations today allow two options for connection termination: three-way handshaking and four-way handshaking with a half-close option.

Three-Way Handshaking

Most implementations today allow *three-way handshaking* for connection termination, as shown in Figure 9.28.

Figure 9.28 *Connection termination using three-way handshaking*

Figure 9.28 *Connection termination using three-way handshaking*



1. **In this situation, the client TCP, after receiving a close command from the client process, sends the first segment, a FIN segment in which the FIN flag is set. Note that a FIN segment can include the last chunk of data sent by the client or it can be just a control segment as shown in Figure 9.28. If it is only a control segment, it consumes only one sequence number because it needs to be acknowledged.**

The FIN segment consumes one sequence number if it does not carry data.

2. **The server TCP, after receiving the FIN segment, informs its process of the situation and sends the second segment, a FIN + ACK segment, to confirm the receipt of the FIN segment from the client and at the same time to announce the closing of the connection in the other direction. This segment can also contain the last chunk of data from the server. If it does not carry data, it consumes only one sequence number because it needs to be acknowledged.**

3. **The client TCP sends the last segment, an ACK segment, to confirm the receipt of the FIN segment from the TCP server. This segment contains the acknowledgment number, which is one plus the sequence number received in the FIN segment from the server. This segment cannot carry data and consumes no sequence numbers.**

The FIN + ACK segment consumes only one sequence number if it does not carry data.

9.5 SCTP

Stream Control Transmission Protocol (SCTP) is a new transport-layer protocol designed to combine some features of UDP and TCP in an effort to create a better protocol for multimedia communication.

9.5.1 SCTP Services

Before discussing the operation of SCTP, we explain the services offered by SCTP to the application-layer processes.

Process-to-Process Communication

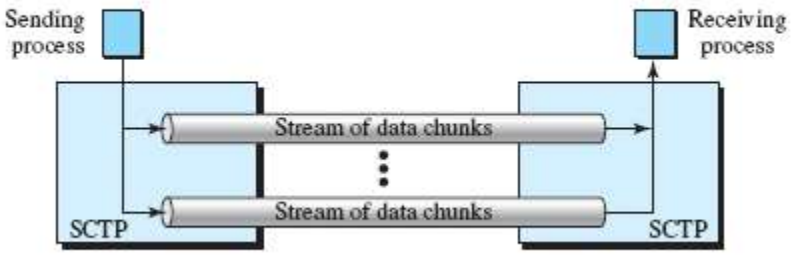
SCTP, like UDP or TCP, provides process-to-process communication.

Multiple Streams

We learned that TCP is a stream-oriented protocol. Each connection between a TCP client and a TCP server involves one single stream. The problem with this approach is that a loss at any point in the stream **blocks** the delivery of the rest of the data. This can be acceptable when we are transferring text; it is not when we are sending real-time data such as audio or video. SCTP allows **multistream service** in each connection, which is called **association** in SCTP terminology. If one of the streams is blocked, the other streams can still deliver their data. Figure 9.54 shows the idea of multiple-stream delivery.

Figure 9.54 *Multiple-stream concept*

Figure 9.54 *Multiple-stream concept*



Multihoming

A TCP connection involves one source and one destination IP address. This means that even if the sender or receiver is a multihomed host (connected to more than one physical address with multiple IP addresses), only one of these IP addresses per end can be utilized during the connection. An SCTP association, on the other hand, supports **multihoming service**. The sending and receiving host can define multiple IP addresses in each end for an association. In this fault-tolerant approach, when one path fails, another interface can be used for data delivery without interruption. This fault-tolerant feature is very helpful when we are sending and receiving a real-time payload such as Internet telephony. Figure 9.55 shows the idea of multihoming.

Figure 9.55 *Multihoming concept*

Figure 9.55 *Multihoming concept*

In Figure 9.55, the client is connected to two local networks with two IP addresses. The server is also connected to two networks with two IP addresses. The client and the server can make an association using four different pairs of IP addresses. However, note that in the current implementations of SCTP, only one pair of IP addresses can be chosen for normal communication; the alternative is used if the main choice fails. In other words, at present, SCTP does not allow load sharing between different paths.

Full-Duplex Communication

Like TCP, SCTP offers full-duplex service, where data can flow in both directions at the same time. Each SCTP then has a sending and receiving buffer, and packets are sent in both directions.

Connection-Oriented Service

Like TCP, SCTP is a connection-oriented protocol. However, in SCTP, a connection is called an *association*.

Reliable Service

SCTP, like TCP, is a reliable transport protocol. It uses an acknowledgment mechanism to check the safe and sound arrival of data. We will discuss this feature further in Section 9.5.6 on error control.

9.5.2 SCTP Features

The following shows the general features of SCTP.

Transmission Sequence Number (TSN)

The unit of data in SCTP is a data chunk, which may or may not have a one-to-one relationship with the message coming from the process because of fragmentation (discussed later in Section 9.5.4). Data transfer in SCTP is controlled by numbering the data chunks. SCTP uses a **transmission sequence number (TSN)** to number the data chunks. In other words, the TSN in

SCTP plays the analogous role as the sequence number in TCP. TSNs are 32 bits long and randomly initialized between 0 and $2^{32} - 1$. Each data chunk must carry the corresponding TSN in its header.

Stream Identifier (SI)

In SCTP, there may be several streams in each association. Each stream in SCTP needs to be identified using a **stream identifier (SI)**. Each data chunk must carry the SI in its header so that when it arrives at the destination, it can be properly placed in its stream. The SI is a 16-bit number starting from 0.

Stream Sequence Number (SSN)

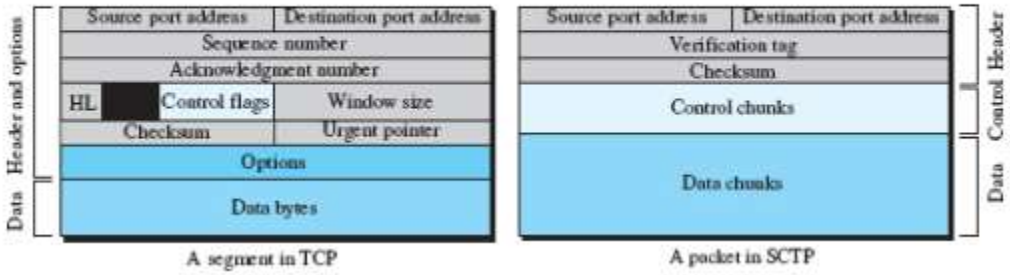
When a data chunk arrives at the destination SCTP, it is delivered to the appropriate stream and in the proper order. This means that, in addition to an SI, SCTP defines each data chunk in each stream with a **stream sequence number (SSN)**.

Packets

In TCP, a segment carries data and control information. Data are carried as a collection of bytes; control information is defined by six control flags in the header. The design of SCTP is totally different: Data are carried as **data chunks**, and control information is carried as **control chunks**. Several control chunks and data chunks can be packed together in a **packet.** A packet in SCTP plays the same role as a segment in TCP. Figure 9.56 compares a segment in TCP and a packet in SCTP. We will discuss the format of the SCTP packet in Section 9.5.4.

Figure 9.56 *Comparison between a TCP segment and an SCTP packet*

Figure 9.56 *Comparison between a TCP segment and an SCTP packet*



In SCTP, we have data chunks, streams, and packets. An association may send many packets, a packet may contain several chunks, and chunks may belong to different streams.

To make the definitions of these terms clear, let us suppose that process A needs to send 11 messages to process B in three streams. The first four messages are in the first stream, the second three messages are in the second stream, and the last four messages are in the third stream. Although a message, if long, can be carried by several data chunks, we assume that each message fits into one data chunk. Therefore, we have 11 data chunks in three streams.

The application process delivers 11 messages to SCTP, where each message is earmarked for the appropriate stream. Although the process could deliver one message from the first stream and

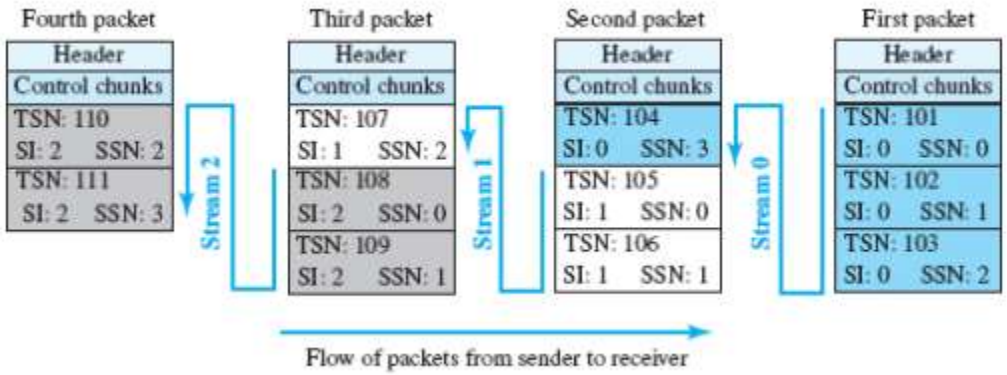
then another from the second, we assume that it delivers all messages belonging to the first stream first, all messages belonging to the second stream next, and finally, all messages belonging to the last stream.

We also assume that the network allows only three data chunks per packet, which means that we need four packets, as shown in Figure 9.57.

Data chunks in stream 0 are carried in the first packet and part of the second packet; those in stream 1 are carried in the second and third packet; those in stream 2 are carried in the third and fourth packet.

Note that each data chunk needs three identifiers: TSN, SI, and SSN. TSN is a cumulative number and is used, as we will see later, for flow control and error control. SI defines the stream to which the chunk belongs. SSN defines the chunk's order in a particular stream. In our example, SSN starts from 0 for each stream.

Figure 9.57 *Packets, data chunks, and streams*



Acknowledgment Number

TCP acknowledgment numbers are **byte-oriented** and refer to the sequence numbers. SCTP acknowledgment numbers are chunk-oriented. They refer to the TSN. A second difference between TCP and SCTP acknowledgments is the control information. Recall that this information is part of the segment header in TCP. To acknowledge segments that carry only control information, TCP uses a sequence number and acknowledgment number (for example, a SYN segment needs to be acknowledged by an ACK segment). In SCTP, however, the control information is carried by control chunks, which do not need a TSN. These control chunks are acknowledged by another control chunk of the appropriate type (some need no acknowledgment). For example, an INIT control chunk is acknowledged by an INIT-ACK chunk. There is no need for a sequence number or an acknowledgment number.

CHAPTER 10

APPLICATION LAYER

10.1 INTRODUCTION

The application layer provides services to the user. Communication is provided using a logical connection, which means that the two application layers assume that there is an imaginary direct connection through which they can send and receive messages. Figure 10.1 shows the idea behind this logical connection.

Figure 10.1 *Logical connection at the application layer*

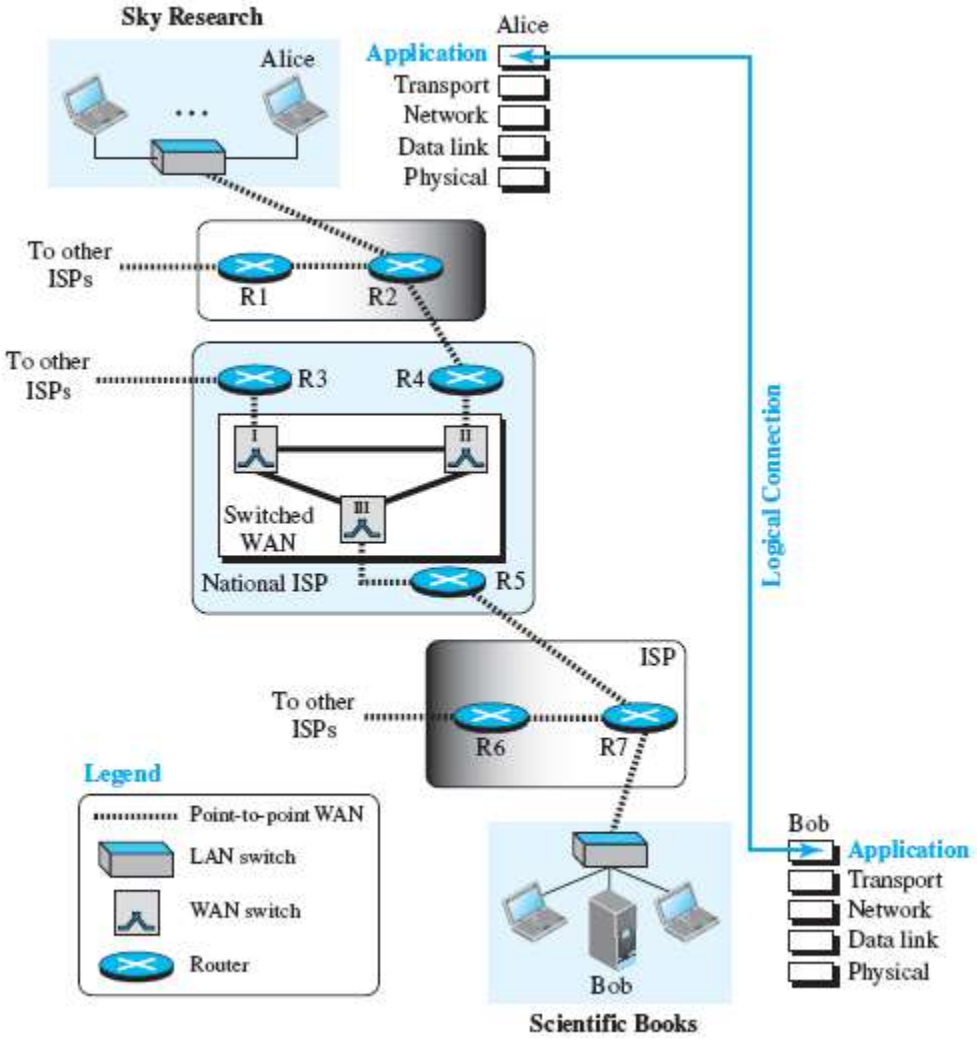


Figure 10.1 shows a scenario in which a scientist, Alice, working at a research company, Sky Research, needs to order a book related to her research from an online bookseller, Scientific Books, whose host is named Bob. A logical connection takes place between the application layer at Alice's computer at Sky Research and the application layer of Bob's computer server at Scientific Books. The communication at the application layer is logical, not physical. Alice and Bob assume that there is a two-way logical channel between them through which they can send and receive messages. The actual communication, however, takes place through several devices (Alice, R2, R4, R5, R7, and Bob) and several physical channels as shown in Figure 10.1.

10.1.1 Providing Services

All communication networks that started before the Internet were designed to provide services to network users. Most of these networks, however, were originally designed to provide one specific service. For example, the telephone network was originally designed to provide voice service: to allow people all over the world to talk to each other. This telephone network, however, was later used for some other services, such as facsimile (fax), which was enabled by users adding some extra hardware at both ends.

The Internet was originally designed for the same purpose: to provide service to users around the world. The layered architecture of the TCP/IP protocol suite, however, makes the Internet more flexible than other communication networks such as the postal or telephone networks. Each layer in the suite was originally made up of one or more protocols, but new protocols can be added or some protocols can be removed or replaced by the Internet authorities. However, if a protocol is added to each layer, it should be designed in such a way that it uses the service provided by one of the protocols at the lower layer. If a protocol is removed from a layer, care should be taken to change the protocol at the next higher layer that supposedly uses the service of the removed protocol.

The application layer, however, is somehow different from other layers in that it is the highest layer in the suite. The protocols in this layer do not provide services to any other protocol in the suite; they only receive services from the protocols in the transport layer. This means that protocols can be removed from this layer easily. New protocols can be also added to this layer as long as the new protocol can use the service provided by one of the transport-layer protocols.

Because the application layer is the only layer that provides services to the Internet user, the flexibility of the application layer, as just described, allows new application protocols to be easily added to the Internet, which has been occurring during the lifetime of the Internet. When the Internet was created, only a few application protocols were available to users; today we cannot give a number for these protocols because new ones are being added constantly.

Standard and Nonstandard Protocols

To provide a smooth operation of the Internet, the protocols used in the first four layers of the TCP/IP protocol suite need to be standardized and documented. They normally become part of the package that is included in operating systems such as Windows or UNIX. To be flexible, however, the application-layer protocols can be both standard and nonstandard.

Standard Application-Layer Protocols

There are several application-layer protocols that have been standardized and documented by the Internet authority, and we are using them in our daily interaction with the Internet. Each standard

protocol is a pair of computer programs that interact with the user and the transport layer to provide a specific service to the user. We will discuss some of these standard applications in this chapter. In the case of these application protocols, we should know what type of services they provide, how they work, the options that we can use with these applications, and so on. The study of these protocols enables a network manager to easily solve the problems that may occur when using these protocols. The deep understanding of how these protocols work will also give us some ideas about how to create new nonstandard protocols.

Nonstandard Application-Layer Protocols

A programmer can create a nonstandard application-layer program if she can write two programs that provide service to the user by interacting with the transport layer. In this chapter, we show how we can write these types of programs. The creation of a nonstandard (proprietary) protocol that does not even need the approval of the Internet authorities if privately used, has made the Internet so popular in the world. A private company can create a new customized application protocol to communicate with all its offices around the world using the service provided by the first four layers of the TCP/IP protocol suite without using any of the standard application programs. What is needed is to write programs, in one of the computer languages that use the available services provided by the transport-layer protocols.

10.1.2 Application-Layer Paradigms

It should be clear that to use the Internet we need two application programs to interact with each other: one running on a computer somewhere in the world, the other running on another computer somewhere else in the world. The two programs need to send messages to each other through the Internet infrastructure. However, we have not discussed what the relationship should be between these programs. Should both application programs be able to request services and provide services, or should the application programs just do one or the other? Two paradigms have been developed during the lifetime of the Internet to answer this question: the *client/server paradigm* and the *peer-to-peer paradigm.*

Traditional Paradigm: Client/Server

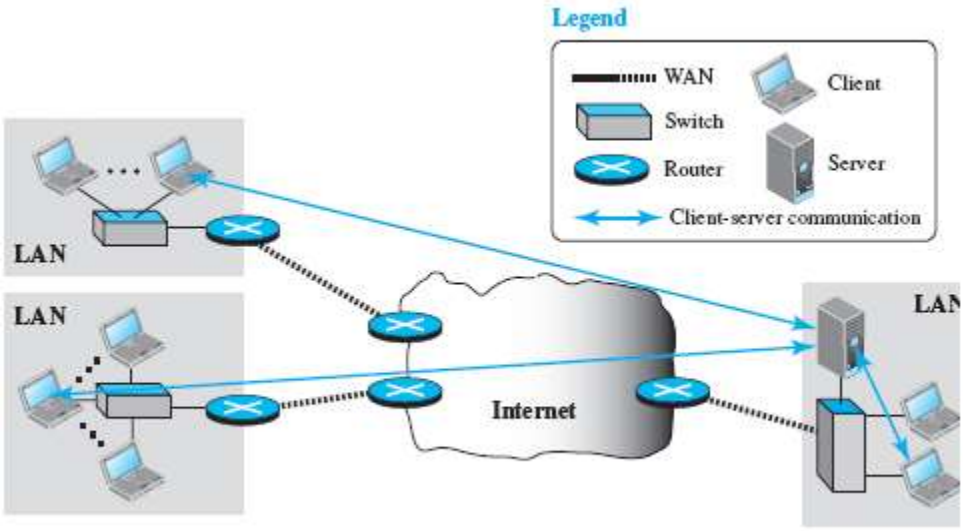
The traditional paradigm is called the **client/server paradigm**. It was the most popular paradigm until a few years ago. In this paradigm, the service provider is an application program, called the server process; it runs continuously, waiting for another application program, called the client process, to make a connection through the Internet and ask for service. There are normally some server processes that can provide a specific type of service, but there are many clients that request service from any of these server processes. The server process must be running all the time; the client process is started when the client needs to receive service.

The client/server paradigm is similar to some available services out of the territory of the Internet. For example, a telephone directory center in any area can be thought of as a server; a subscriber that calls and asks for a specific telephone number can be thought of as a client. The directory center must be ready and available all the time; the subscriber can call the center for a short period when the service is needed.

Although the communication in the client/server paradigm is between two application programs, the role of each program is totally different. In other words, we cannot run a client

program as a server program, or vice versa. In this chapter, when we talk about client/server programming in this paradigm, we show that we always need to write two application programs for each type of service. Figure 10.2 shows an example of a client/server communication in which three clients communicate with one server to receive the services provided by this server.

Figure 10.2 *Example of a client/server paradigm*



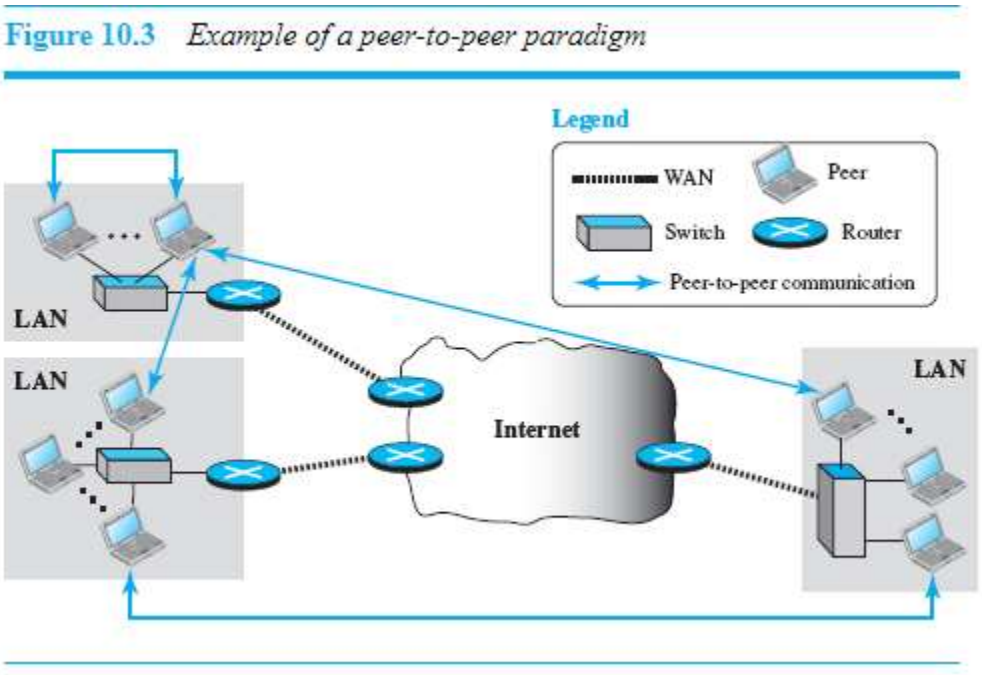
One problem with this paradigm is that the concentration of the communication load is on the shoulder of the server, which means the server should be a powerful computer. Even a powerful computer may become overwhelmed if a large number of clients try to connect to the server at the same time. Another problem is that there should be a service provider willing to accept the cost and create a powerful server for a specific service, which means the service must always return some type of income for the server to encourage such an arrangement.

Several traditional services still use this paradigm, including the **World Wide Web (WWW)** and its vehicle Hypertext Transfer Protocol (HTTP), file transfer protocol (FTP), secure shell (SSH), and e-mail. We discuss some of these protocols and applications later in the chapter.

New Paradigm: Peer-to-Peer

A new paradigm, called the **peer-to-peer (P2P)** paradigm has emerged to respond to the needs of some new applications. In this paradigm, there is no need for a server process to be running all the time and waiting for the client processes to connect. The responsibility is shared between peers. A computer connected to the Internet can provide service at one time and receive service at another time. A computer can even provide and receive services at the same time. Figure 10.3 shows an example of communication in this paradigm.

Figure 10.3 *Example of a peer-to-peer paradigm*



One of the areas that really fits in this paradigm is the Internet telephony. Communication by phone is indeed a peer-to-peer activity; no party needs to be running forever waiting for the other party to call. Another area in which the peer-to-peer paradigm can be used is when some computers connected to the Internet have something to share with each other. For example, if an Internet user has a file available to share with other Internet users, there is no need for the file holder to become a server and run a server process all the time waiting for other users to connect and retrieve the file.

Although the peer-to-peer paradigm has proven to be easily scalable and cost- effective in eliminating the need for expensive servers to be running and maintained all the time, there are also some challenges. The main challenge has been security; it is more difficult to create secure communication between distributed services than between those controlled by some dedicated servers. The other challenge is applicability; it appears that not all applications can use this new paradigm. For example, not many Internet users are ready to become involved, if one day the Web can be implemented as a peer-to-peer service.

There are some new applications, such as BitTorrent, Skype, IPTV, and Internet telephony, that use this paradigm. We will discuss some of these applications later in this chapter.

Mixed Paradigm

An application may choose to use a mixture of the two paradigms by combining the advantages of both. For example, a light-load client/server communication can be used to find the address of the peer that can offer a service. When the address of the peer is found, the actual service can be received from the peer by using the peer-to-peer paradigm.

10.2 CLIENT/SERVER PARADIGM

In a client/server paradigm, communication at the application layer is between two running application programs called **processes**: a *client* and a *server*. A client is a running program that initializes the communication by sending a request; a server is another application program that waits for a request from a client. The server handles the request received from a client, prepares a result, and sends the result back to the client. This definition of a server implies that a server must be running when a request from a client arrives, but the client program needs to be run only when it is needed. This means that if we have two computers connected to each other somewhere, we can run a client process on one of them and the server on the other. However, we need to be careful that the server program is started before we start running the client program. In other words, the lifetime of a server is infinite: It should be started and run forever, waiting for the clients. The lifetime of a client is finite: It normally sends a finite number of requests to the corresponding server, receives the responses, and stops.

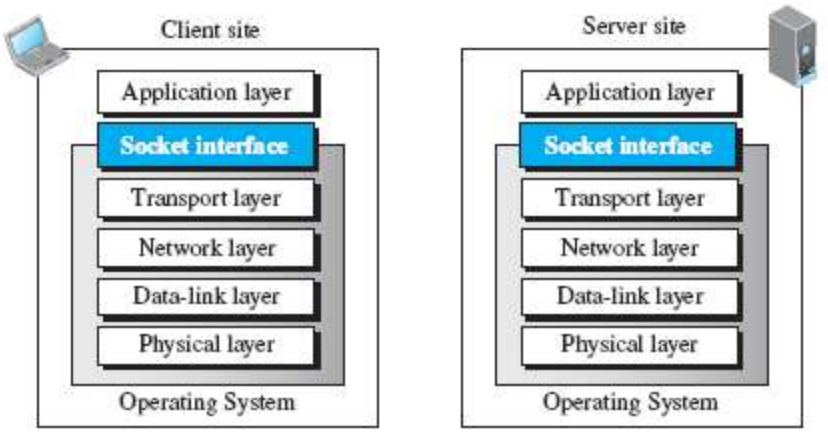
10.2.1 Application Programming Interface

How can a client process communicate with a server process? A computer program is normally written in a computer language with a predefined set of instructions that tells the computer what to do. A computer language has a set of instructions for mathematical operations, a set of instructions for string manipulation, a set of instructions for input/output access, and so on. If we need a process to be able to communicate with another process, we need a new set of instructions to tell the lowest four layers of the TCP/IP protocol suite to open the connection, send and receive data from the other end, and close the connection. A set of instructions of this kind is normally referred to as the **Application Programming Interface (API)**. An interface in programming is a set of instructions between two entities. In this case, one of the entities is the process at the application layer and the other is the *operating system* that encapsulates the first four layers of the TCP/IP protocol suite. In other words, a computer manufacturer needs to build the first four layers of the suite in the operating system and include an API. In this way, the processes running at the application layer are able to communicate with the operating system when sending and receiving messages through the Internet. Several APIs have been designed for communication. Three of them are common: **socket interface**, **transport layer interface (TLI)**, and **STREAM**. In this section, we briefly discuss only *socket interface*, the most common one, to give a general idea of network communication at the application layer.

Socket interface started in the early 1980s at UC Berkeley as part of a UNIX environment. The socket interface is a set of instructions that provide communication between the application layer and the operating system, as shown in Figure 10.4. It is a set of instructions that can be used by a process to communicate with another process.

Figure 10.4 *Position of the socket interface*

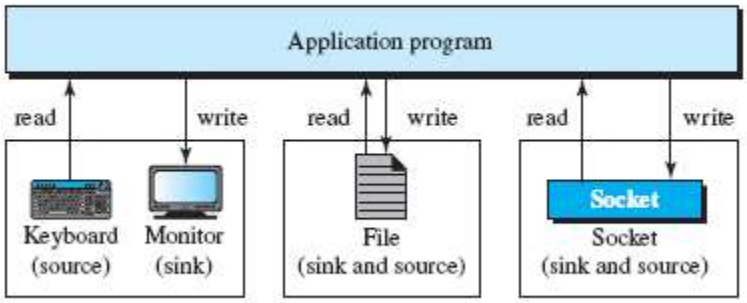
Figure 10.4 *Position of the socket interface*



The idea of sockets allows us to use the set of all instructions already designed in a programming language for other sources and sinks. For example, in most computer languages, like C, C++, or Java, we have several instructions that can read and write data to other sources and sinks such as a keyboard (a source), a monitor (a sink), or a file (source and sink). We can use the same instructions to read from or write to sockets. In other words, we are adding only new sources and sinks to the programming language without changing the way we send data or receive data. Figure 10.5 shows the idea and compares the sockets with other sources and sinks.

Figure 10.5 *Sockets used the same way as other sources and sinks*

Figure 10.5 *Sockets used the same way as other sources and sinks*



Sockets

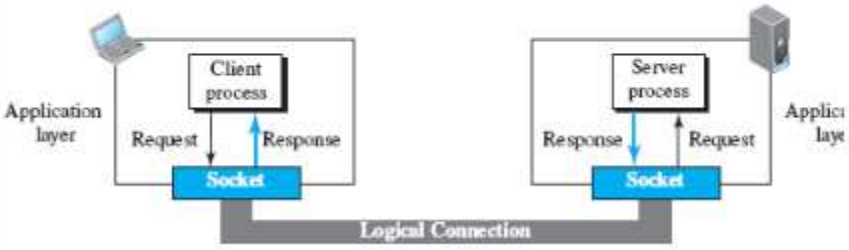
Although a socket is supposed to behave like a terminal or a file, it is not a physical entity like them; it is an abstraction. It is a *data structure* that is created and used by the application program.

We can say that, as far as the application layer is concerned, communication between a client process and server process is communication between two sockets, created at two ends, as shown in Figure 10.6. The client thinks that the socket is the entity that receives the request and gives the

response; the server thinks that the socket is the one that has a request and needs the response. If we create two sockets, one at each end, and define the source and destination addresses correctly, we can use the available instructions to send and receive data. The rest is the responsibility of the operating system and the embedded TCP/IP protocol.

Figure 10.6 *Use of sockets in process-to-process communication*

Figure 10.6 *Use of sockets in process-to-process communication*



Socket Addresses

The interaction between a client and a server is two-way communication. In a two-way communication, we need a pair of addresses: local (sender) and remote (receiver). The local address in one direction is the remote address in the other direction, and vice versa. Because communication in the client/server paradigm is between two sockets, we need a pair of **socket addresses** for communication: a local socket address and a remote socket address. However, we need to define a socket address in terms of identifiers used in the TCP/IP protocol suite.

A socket address should first define the computer on which a client or a server is running. As we discuss in Chapter 7, a computer in the Internet is uniquely defined by its IP address, a 32-bit integer in the current Internet version. However, several client or server processes may be running at the same time on a computer, which means that we need another identifier to define the specific client or server involved in the communication. As we discuss in Chapter 9, an application program can be defined by a port number, a 16-bit integer. This means that a socket address should be a combination of an IP address and a port number as shown in Figure 10.7.

Figure 10.7 *A socket address*

Because a socket defines the end-point of the communication, we can say that a socket is identified by a pair of socket addresses, a local and a remote.

Example 10.1

We can find a two-level address in telephone communication. A telephone number can define an organization, and an extension can define a specific connection in that organization. The telephone number in this case is similar to the IP address, which defines the whole organization; the extension is similar to the port number, which defines the particular connection.

10.3 STANDARD APPLICATIONS

During the lifetime of the Internet, several client/server application programs have been developed. We do not have to redefine them, but we need to understand what they do. For each application, we also need to know the options available to us. The study of these applications and the ways they provide different services can help us to create customized applications in the future.

We have selected six standard application programs in this section. We start with HTTP and the World Wide Web because they are used by almost all Internet users. We then introduce file transfer and electronic mail applications which have high traffic loads on the Internet. Next, we explain remote login and how it can be achieved using the TELNET and SSH protocols. Finally, we discuss DNS, which is used by all application programs to map the application-layer identifier to the corresponding host IP address.

Some other applications, such as Dynamic Host Configuration Protocol (DHCP) or Simple Network Management Protocol (SNMP), will be discussed in Chapter 12.

10.3.1 World Wide Web and HTTP

In this section, we first introduce the World Wide Web (abbreviated as the WWW or Web). We then discuss the Hypertext Transfer Protocol (HTTP), the most common client/server application program used in relation to the Web.

World Wide Web

The idea of the Web was first proposed by Tim Berners-Lee in 1989 at *CERN*, the European Organization for Nuclear Research,[†] to allow several researchers at different locations throughout Europe to access each others' research. The commercial Web started in the early 1990s.

The Web today is a repository of information in which the documents, called *web pages,* are distributed all over the world and related documents are linked together. The popularity and growth of the Web can be related to two terms in the above statement: *distributed* and *linked*. Distribution allows for the growth of the Web. Each web server in the world can add a new web page to the repository and announce it to all Internet users without overloading a few servers. Linking allows one web page to refer to another web page stored in another server somewhere else in the world. The linking of web pages was achieved using a concept called *hypertext*, which was introduced many years before the advent of the Internet. The idea was to use a machine that automatically retrieved another document stored in the system when a link to it appeared in the document. The Web implemented this idea electronically: to allow the linked document to be retrieved when the link was clicked by the user. Today, the term **hypertext**, coined to mean linked text documents, has been changed to **hypermedia**, to show that a web page can be a text document, an image, an audio file, or a video file.

The purpose of the Web has gone beyond the simple retrieving of linked documents. Today, the Web is used to provide electronic shopping and gaming. One can use the Web to listen to radio programs or view television programs whenever one desires without being forced to listen to or view these programs at the time when they are broadcast.

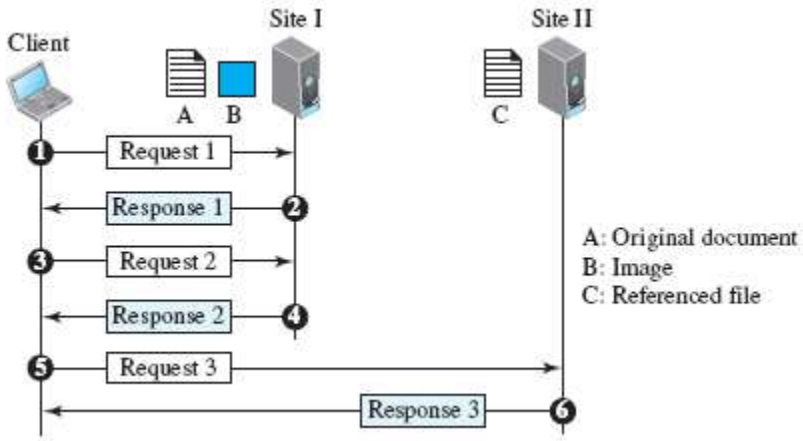
Architecture

The WWW today is a distributed client/server service, in which a client using a browser can access a service using a server. However, the service provided is distributed over many locations called *sites.* Each site holds one or more documents, referred to as web pages. Each **web page**, however, can contain some links to other web pages in the same or other sites. In other words, a web page can be simple or composite. A simple web page has no links to other web pages; a composite web page has one or more links to other web pages. Each web page is a file with a name and address.

Example 10.2

Assume we need to retrieve a scientific document that contains one reference to another text file and one reference to a large image. Figure 10.8 shows the situation.

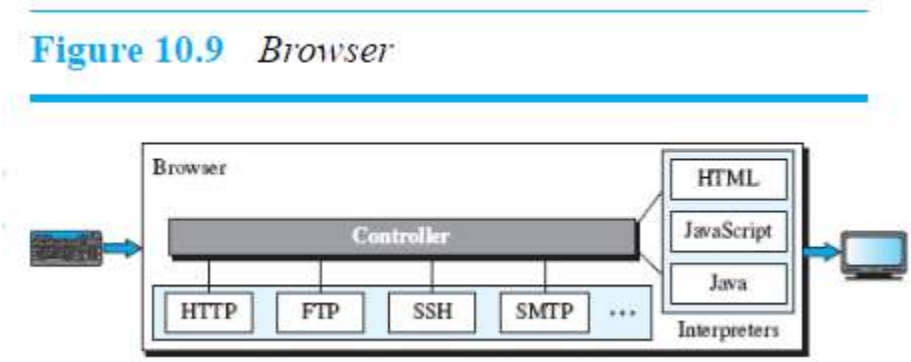
Figure 10.8 *Example 10.2*



The main document and the image are stored in two separate files in the same site (file A and file B); the referenced text file is stored in another site (file C). Because we are dealing with three different files, we need three transactions if we want to see the whole document. The first transaction (request/response) retrieves a copy of the main document (file A), which has references (pointers) to the second and third files. When a copy of the main document is retrieved and browsed, the user can click on the reference to the image to invoke the second transaction and retrieve a copy of the image (file B). If the user needs to see the contents of the referenced text file, she can click on its reference (pointer) invoking the third transaction and retrieving a copy of file C. Note that although files A and B both are stored in site I, they are independent files with different names and addresses. Two transactions are needed to retrieve them. A very important point we need to remember is that files A, B, and C in this example are independent web pages, each with independent names and addresses. Although references to file B or C are included in file A, it does not mean that each of these files cannot be retrieved independently. A second user can retrieve file B with one transaction. A third user can retrieve file C with one transaction.

Web Client (Browser) A variety of vendors offer commercial **browsers** that interpret and display a web page, and all of them use nearly the same architecture. Each browser usually consists of three parts: a controller, client protocols, and interpreters (see Figure 10.9).

Figure 10.9 *Browser*



The controller receives input from the keyboard or the mouse and uses the client programs to access the document. After the document has been accessed, the controller uses one of the interpreters to display the document on the screen. The client protocol can be one of the protocols described later, such as HTTP or FTP. The interpreter can be HTML, Java, or JavaScript, depending on the type of document. Some commercial browsers include Internet Explorer and Firefox.

Web Server The web page is stored at the server. Each time a request arrives, the corresponding document is sent to the client. To improve efficiency, servers normally store requested files in a cache in memory; memory is faster to access than a disk. A server can also become more efficient through multithreading or multiprocessing. In this case, a server can answer more than one request at a time.

Uniform Resource Locator (URL)

A web page, as a file, needs to have a unique identifier to distinguish it from other web pages. To define a web page, we need three identifiers: *host*, *port*, and *path*. However, before defining the web page, we need to tell the browser what client/server application we want to use, which is called the *protocol*. This means we need four identifiers to define the web page. The first is the type of vehicle to be used to fetch the web page; the last three make up the combination that defines the destination object (web page).

- **Protocol.** The first identifier is the abbreviation for the client/server program that we need to access the web page. Although most of the time the protocol

is Hypertext Transfer Protocol (HTTP), which we will discuss shortly, we can also use other protocols such as File Transfer Protocol (FTP).

- **Host.** The host identifier can be the IP address of the server or the unique name given to the server. IP addresses can be defined in dotted-decimal notations, as described in Chapter 7 (such as 64.23.56.17); the name is normally the domain name that uniquely defines the host, such as *forouzan.com*, which we discuss with the Domain Name System (DNS) later in Section 10.3.6.
- **Port.** The port, a 16-bit integer, is normally predefined for the client/server application. For example, if HTTP is used for accessing the web page, the well-known port number is 80. However, if a different port is used, the number can be explicitly given.

- **Path.** The path identifies the location and the name of the file in the underlying operating system. The format of this identifier normally depends on the operating system. In UNIX, a path is a set of directory names followed by the
    - file name, all separated by a slash. For example, */top/next/last/myfile* is a path that uniquely defines a file named *myfile*, stored in the directory *last*, which itself is part of the directory *next*, which itself is under the directory *top*. In other words, the path lists the directories from the top to the bottom, followed by the file name.

To combine these four pieces together, the **uniform resource locator (URL)** has been designed; it uses three different separators between the four pieces as shown below:

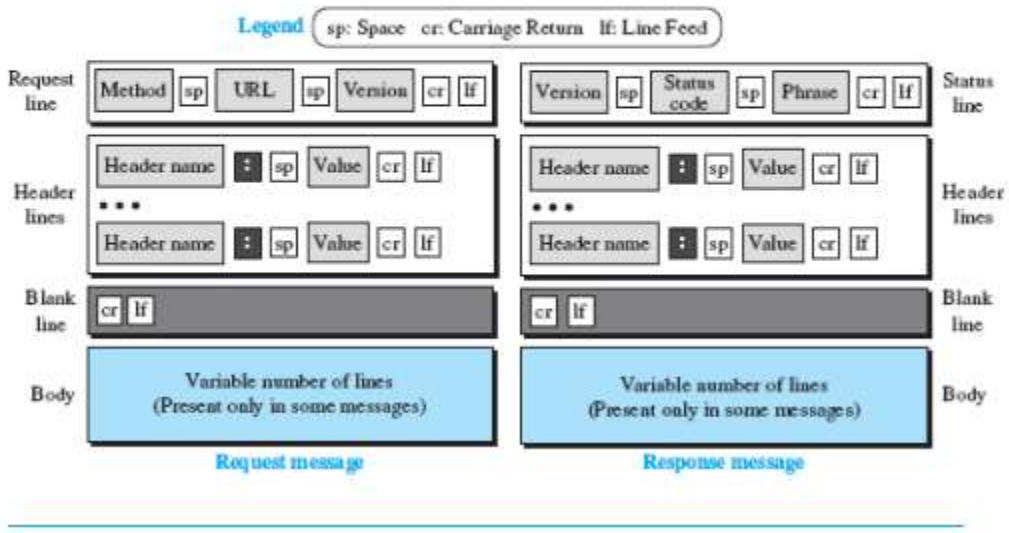
protocol://host/path	**Used most of the time**
protocol://host:port/path	**Used when port number is needed**

Hypertext Transfer Protocol (HTTP)

The **Hypertext Transfer Protocol (HTTP)** is a protocol that is used to define how the client/server programs can be written to retrieve web pages from the

Web. An HTTP client sends a request; an HTTP server returns a response. The server uses the port number 80; the client uses a temporary port number. HTTP uses the services of TCP, which, as discussed before, is a connection-oriented and reliable protocol. This means that, before any transaction between the client and the server can take place, a connection needs to be established between them.

Figure 10.12 *Formats of the request and response messages*



Request Message As we stated, the first line in a request message is called a request line. There are three fields in this line separated by one space and terminated by two characters (carriage return and line feed) as shown in Figure 10.12. The fields are called *method*, URL, and *version*.

The method field defines the request types. In version 1.1 of HTTP, several methods are defined, as shown in Table 10.1. Most of the time, the client uses the GET method to send a request. In this case, the body of the message is empty. The HEAD method is used when the client needs only some information about the web page from the server, such as the last time it was modified. It can also be used to test the validity of a URL. The response message in this case has only the header section; the body section is empty. The PUT method is the inverse of the GET method; it allows the client to post a new web page on the server (if permitted). The POST method is similar to the PUT method, but it is used to send some information to the server to be added to the web page or to modify the web page. The TRACE method is used

for debugging; the client asks the server to echo back the request to check whether the server is getting the requests. The DELETE method allows the client to delete a web page on the server if the client has permission to do so. The CONNECT method was originally created as a reserve method; it may be used by proxy servers. Finally, the OPTIONS method allows the client to ask about the properties of a web page.

Table 10.1 *Methods*

Method	*Action*
GET	Requests a document from the server
HEAD	Requests information about a document but not the document itself
PUT	Sends a document from the client to the server
POST	Sends some information from the client to the server
TRACE	Echoes the incoming request
DELETE	Removes the web page
CONNECT	Reserved
OPTIONS	Inquires about available options

The second field, URL, was discussed earlier in this section. It defines the address and name of the corresponding web page. The third field, version, gives the version of the protocol; the most current version of HTTP is 1.1.

After the request line, we can have zero or more *request header* lines. Each header line sends additional information from the client to the server. For example, the client can request that the document be sent in a special format. Each header line has a header name, a colon, a space, and a header value (see Figure 10.12). Table 10.2 shows some header names commonly used in a request.

Response Message The format of the response message is also shown in Figure 10.12. A response message consists of a status line, header lines, a blank line, and sometimes a body. The first line in a response message is called the *status line*. There are three fields in this line separated by spaces and terminated by a carriage return and line feed. The first field defines the version of HTTP, currently 1.1. The status code field defines the status of the request. It consists of three digits. Whereas the codes in the 100 range are only informational, the codes in the 200 range indicate a successful request. The codes in the 300 range redirect the client to another URL, and the codes in the 400 range indicate an error at the client site. Finally, the codes in the 500 range indicate an error at the server site. The status phrase explains the status code in text form.

After the status line, we can have zero or more *response header* lines. Each header line sends additional information from the server to the client. For example, the sender can send extra information about the document. Each header line has a header name, a colon, a space, and a header value. We will show some header lines in the examples at the end of this section. Table 10.3 shows some header names commonly used in a response message.

Table 10.3 *Response header names*

Header	*Description*
Date	Shows the current date
Upgrade	Specifies the preferred communication protocol

Server	Gives information about the server
Set-Cookie	The server asks the client to save a cookie
Content-Encoding	Specifies the encoding scheme
Content-Language	Specifies the language
Content-Length	Shows the length of the document
Content-Type	Specifies the media type
Location	To ask the client to send the request to another site
Accept-Ranges	The server will accept the requested byte-ranges
Last-modified	Gives the date and time of the last change

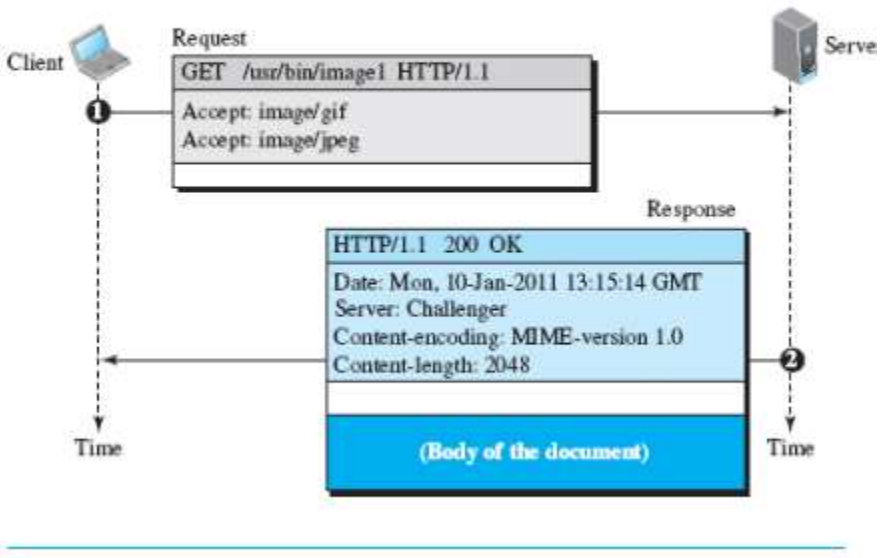
The body contains the document to be sent from the server to the client. The body is present unless the response is an error message.

Example 10.6

This example retrieves a document (see Figure 10.13). We use the GET method to retrieve an image with the path /usr/bin/image1. The request line shows the method (GET), the URL, and the HTTP version (1.1). The header

has two lines that show that the client can accept images in the GIF or JPEG format. The request does not have a body. The response message contains the status line and four lines of header. The header lines define the date, server, content encoding (MIME version, which will be described in Section 10.3.3 on electronic mail), and length of the document. The body of the document follows the header.

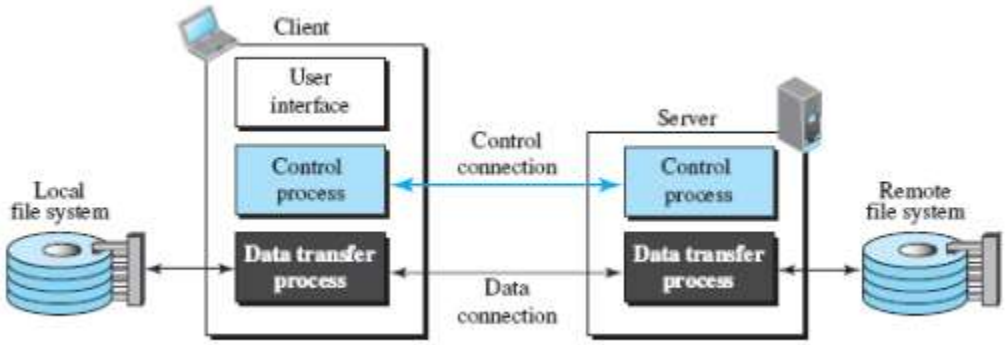
Figure 10.13 *Example 10.6*



10.3.2 FTP

File Transfer Protocol (FTP) is the standard protocol provided by TCP/IP for copying a file from one host to another. Although transferring files from one system to another seems simple and traightforward, some problems must be dealt with first. For example, two systems may use different file name conventions, have different ways to represent data, and have different directory structures. All these problems have been solved by FTP in a very simple and elegant approach. Although we can transfer files using HTTP, FTP is a better choice to transfer large files or to transfer files using different formats. Figure 10.17 shows the basic model of FTP. The client has three components: user interface, client control process, and the client data transfer process. The server has two components: server control process and server data transfer process. The control connection is made between the control processes. The data connection is made between the data transfer processes.

Figure 10.17 *FTP basic model*



The separation of commands and data transfer makes FTP more efficient. The control connection uses very simple rules of communication. We need to transfer only a line of command or a line of response at a time. The data connection, on the other hand, needs more complex rules due to the variety of data types transferred.

Lifetimes of Two Connections

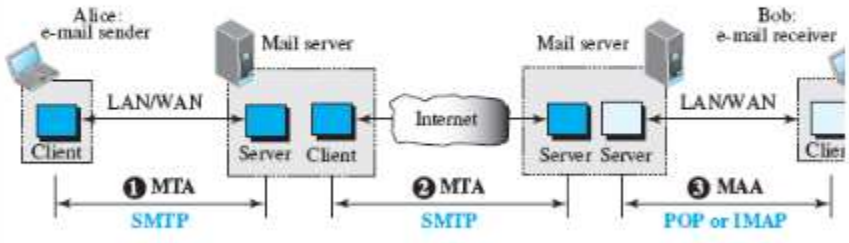
The two connections in FTP have different lifetimes. The control connection remains connected during the entire interactive FTP session. The data connection is opened and then closed for each file transfer activity. It opens each time commands that involve transferring files are used, and it closes when the file is transferred. In other words, when a user starts an FTP session, the control connection opens. While the control connection is open, the data connection can be opened and closed multiple times if several files are transferred. FTP uses two well-known TCP ports: Port 21 is used for the control connection, and port 20 is used for the data connection.

Simple Mail Transfer Protocol (SMTP)

Based on the common scenario (Figure 10.19), we can say that e-mail is one of those applications that needs three uses of client/server paradigms to accomplish its task. It is important that we distinguish these three when we are dealing with e-mail. Figure 10.22 shows these three client/server applications. We refer to the first and the second as message transfer agents (MTAs) and the third as a message access agent (MAA).

Figure 10.22 *Protocols used in electronic mail*

Figure 10.22 *Protocols used in electronic mail*



The formal protocol that defines the MTA client and server in the Internet is called **Simple Mail Transfer Protocol (SMTP)**. SMTP is used two times, between the sender and the sender's mail server and between the two mail servers. As we will see shortly, another protocol is needed between the mail server and the receiver. SMTP simply defines how commands and responses must be sent back and forth.

10.3.6 Domain Name System (DNS)

The last client/server application program we discuss has been designed to help other application programs. To identify an entity, TCP/IP protocols use the IP address, which uniquely identifies the connection of a host to the Internet. However, people prefer to use names instead of numeric addresses. Therefore, the Internet needs to have a directory system that can map a name to an address. This is analogous to the telephone network. A telephone network is designed to use telephone numbers, not names. People can either keep a private file to map a name to the corresponding telephone number or can call the telephone directory to do so.

Because the Internet is so huge today, a central directory system cannot hold all the mapping. In addition, if the central computer fails, the whole communication network will collapse. A better solution is to distribute the information among many computers in the world. In this method, the host that needs mapping can contact the closest computer holding the needed information. This method is used by the **Domain Name System (DNS)**.

Figure 10.35 shows how TCP/IP uses a DNS client and a DNS server to map a name to an address. A user wants to use a file transfer client to access the corresponding file transfer server running on a remote host. The user knows

only the file transfer server name, such as *afilesource.com*. However, the TCP/IP suite needs the IP address of the file transfer server to make the connection. The following six steps map the host name to an IP address:

1. **The user passes the host name to the file transfer client.**
2. **The file transfer client passes the host name to the DNS client.**
3. **Each computer, after being booted, knows the address of one DNS server. The DNS client sends a message to a DNS server with a query that gives the file transfer server name using the known IP address of the DNS server.**
4. **The DNS server responds with the IP address of the desired file transfer server.**
5. **The DNS client passes the IP address to the file transfer server.**
6. **The file transfer client now uses the received IP address to access the file transfer server.**

Note that the purpose of accessing the Internet is to make a connection between the file transfer client and server, but before this can happen, another connection needs to be made between the DNS client and DNS server. In other words, we need at least two connections in this case. The first is for mapping the name to an IP address; the second is for transferring files.

Figure 10.35 *Purpose of DNS*