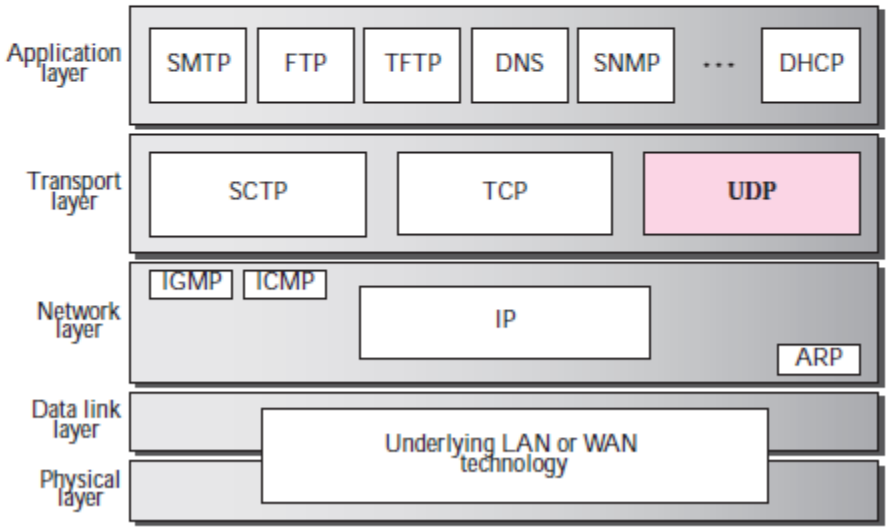
User Datagram Protocol (UDP)

14.1 INTRODUCTION

Figure 14.1 shows the relationship of the **User Datagram Protocol** (UDP) to the other protocols and layers of the TCP/IP protocol suite: UDP is located between the application layer and the IP layer, and serves as the intermediary between the application programs and the network operations.

Figure 14.1 *Position of UDP in the TCP/IP protocol suite*



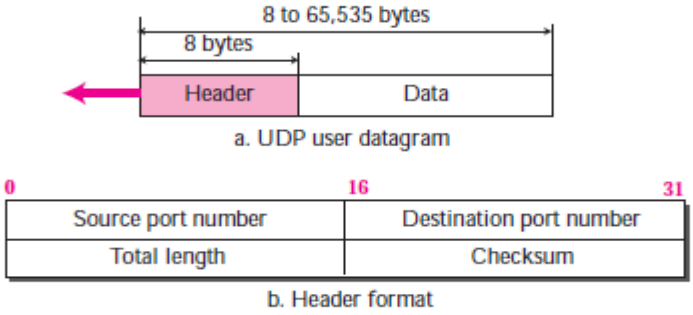
UDP is a **connectionless, unreliable transport protocol**. It does not add anything to the services of IP except for providing process-to-process communication instead of host-to-host communication.

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.

14.2 USER DATAGRAM

UDP packets, called **user datagrams**, have a fixed-size header of 8 bytes. Figure 14.2 shows the format of a user datagram. The fields are as follows:

Figure 14.2 *User datagram format*



- **Source port number.** This is the port number used by the process running on the source host. It is 16 bits long, which means that the port number can range from 0 to 65,535. If the source host is the client (a client sending a request), the port number, in most cases, is an ephemeral port number requested by the process and chosen by the UDP software running on the source host. If the source host is the server (a server sending a response), the port number, in most cases, is a well-known port number.

- **Destination port number.** This is the port number used by the process running on the destination host. It is also 16 bits long. If the destination host is the server (a client sending a request), the port number, in most cases, is a well-known port number. If the destination host is the client (a server sending a response), the port number, in most cases, is an ephemeral port number. In this case, the server copies the ephemeral port number it has received in the request packet.
- **Length.** This is a 16-bit field that 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 much less because a UDP user datagram is stored in an IP datagram with the total length of 65,535 bytes. The length field in a UDP user datagram is actually not necessary. A user datagram is encapsulated in an IP datagram. There is a field in the IP datagram that defines the total length. There is another field in the IP datagram that defines the length of the header. So if we subtract the value of the second field from the first, we can deduce the length of the UDP datagram that is encapsulated in an IP datagram.

  UDP length = IP length − IP header's length

  However, the designers of the UDP protocol felt that it was more efficient for the destination UDP to calculate the length of the data from the information provided in the UDP user datagram rather than ask the IP software to supply this information. We should remember that when the IP software delivers the UDP user datagram to the UDP layer, it has already dropped the IP header.
- **Checksum.** This field is used to detect errors over the entire user datagram (header plus data). The checksum is discussed in the next section.

14.3 UDP SERVICES

We discussed the general services provided by a transport layer protocol in Chapter 13. 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 discussed in Chapter 13 using sockets, a combination of IP addresses and port numbers. Several port numbers used by UDP are shown in Table 14.1.

Table 14.1 *Well-known Ports used with UDP*

Port	*Protocol*	*Description*
7	Echo	Echoes a received datagram back to the sender
9	Discard	Discards any datagram that is received
11	Users	Active users
13	Daytime	Returns the date and the time
17	Quote	Returns a quote of the day
19	Chargen	Returns a string of characters
53	Domain	Domain Name Service (DNS)
67	Bootps	Server port to download bootstrap information
68	Bootpc	Client port to download bootstrap information
69	TFTP	Trivial File Transfer Protocol
111	RPC	Remote Procedure Call
123	NTP	Network Time Protocol
161	SNMP	Simple Network Management Protocol
162	SNMP	Simple Network Management Protocol (trap)

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, there is no connection establishment and no connection termination as is the case for TCP. 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

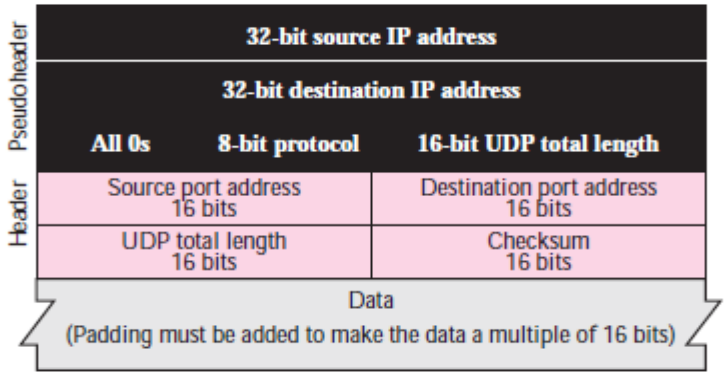
There is no *error control* mechanism in UDP except for the checksum. This means that the sender does not know if a message has been lost or duplicated. When the receiver detects an error through the checksum, the user datagram is silently discarded. The lack of error control means that the process using UDP should provide for this service if needed.

Checksum

We have already talked about the concept of the *checksum* and the way it is calculated for IP in Chapter 7. UDP checksum calculation is different from the one for IP. Here the checksum 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 in which the user datagram is to be encapsulated with some fields filled with 0s (see Figure 14.3).

Figure 14.3 *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. We will see later 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.

Note the similarities between the pseudoheader fields and the last 12 bytes of the IP header.

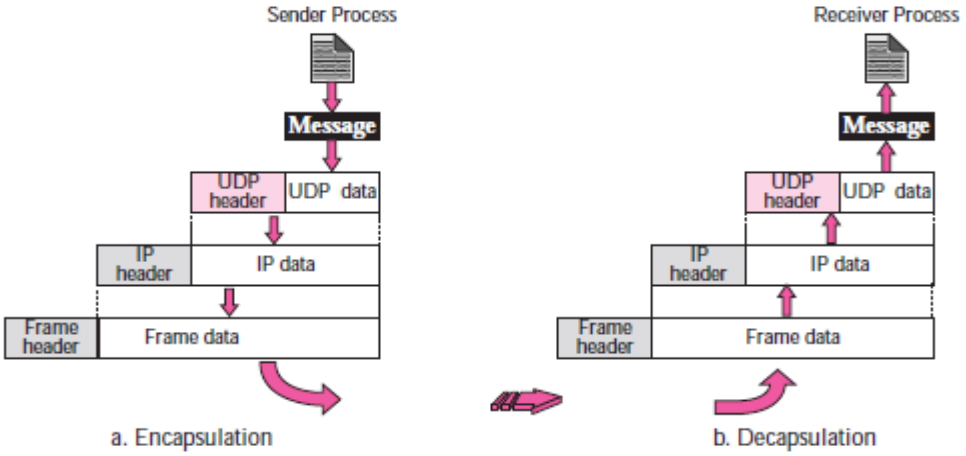
Congestion Control

Since 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 real-time transfer of audio and video.

Encapsulation and Decapsulation

To send a message from one process to another, the UDP protocol encapsulates and decapsulates messages (see Figure 14.5).

Figure 14.5 *Encapsulation and decapsulation*



Encapsulation

When a process has a message to send through UDP, it passes the message to UDP along with a pair of socket addresses and the length of data. UDP receives the data and adds the UDP header. UDP then passes the user datagram to IP with the socket addresses. IP adds its own header, using the value 17 in the protocol field, indicating that the data has come from the UDP protocol. The IP datagram is then passed to the data link layer. The data link layer receives the IP datagram, adds its own header (and possibly a trailer), and passes it to the physical layer. The physical layer encodes the bits into electrical or optical signals and sends it to the remote machine.

Decapsulation

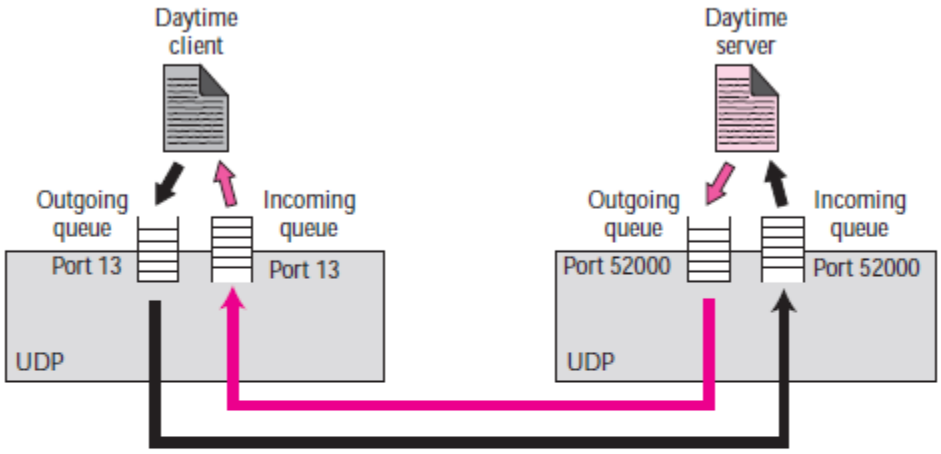
When the message arrives at the destination host, the physical layer decodes the signals into bits and passes it to the data link layer. The data link layer uses the header (and the trailer) to check the data. If there is no error, the header and trailer are dropped and the datagram is passed to IP. The IP software does its own checking. If there is no error, the header is dropped and the user datagram is passed to UDP with the sender and receiver IP addresses. UDP uses the checksum to check the entire user datagram. If there is no error, the header is dropped and the application data along with the sender socket address is passed to the process. The sender socket address is passed to the process in case it needs to respond to the message received.

Queuing

We have talked about ports without discussing the actual implementation of them. In UDP, queues are associated with ports (see Figure 14.6).

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.

Figure 14.6 *Queues in UDP*



Note that even if a process wants to communicate with multiple processes, it obtains only one port number and eventually one outgoing and one incoming **queue**. The queues opened by the client are, in most cases, identified by ephemeral port numbers. The queues function as long as the process is running. When the process terminates, the queues are destroyed.

The client process can send messages to the outgoing queue by using the source port number specified in the request. UDP removes the messages one by one, and, after adding the UDP header, delivers them to IP. An outgoing queue can overflow. If this happens, the operating system can ask the client process to wait before sending any more messages.

When a message arrives for a client, UDP checks to see if an incoming queue has been created for the port number specified in the destination port number field of the user datagram. If there is such a queue, UDP sends the received user datagram to the end of the queue. If there is no such queue, UDP discards the user datagram and asks the ICMP protocol to send a *port unreachable* message to the server. All of the incoming messages for one particular client program, whether coming from the same or a different server, are sent to the same queue. An incoming queue can overflow. If this happens, UDP drops the user datagram and asks for a port unreachable message to be sent to the server.

At the server site, the mechanism of creating queues is different. In its simplest form, a server asks for incoming and outgoing queues using its well-known port when it starts running. The queues remain open as long as the server is running.

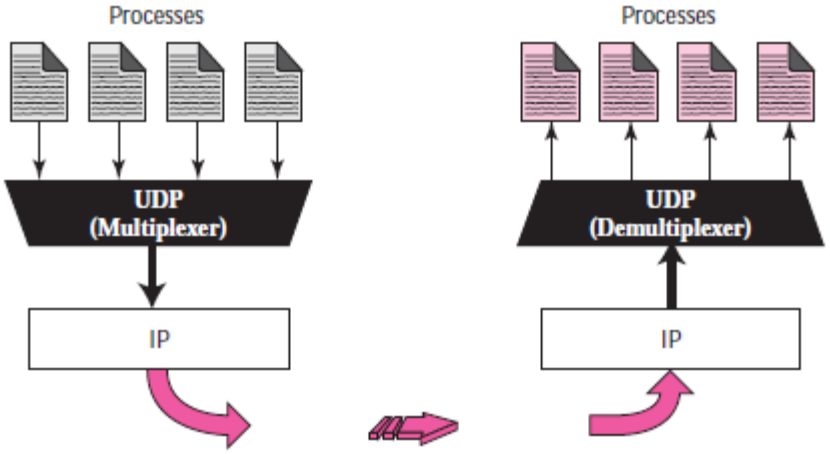
When a message arrives for a server, UDP checks to see if an incoming queue has been created for the port number specified in the destination port number field of the user datagram. If there is such a queue, UDP sends the received user datagram to the end of the queue. If there is no such queue, UDP discards the user datagram and asks the ICMP protocol to send a port unreachable message to the client. All of the incoming messages for one particular server, whether coming from the same or a different client, are sent to the same queue. An incoming queue can overflow. If this happens, UDP drops the user datagram and asks for a port unreachable message to be sent to the client.

When a server wants to respond to a client, it sends messages to the outgoing queue using the source port number specified in the request. UDP removes the messages one by one, and, after adding the UDP header, delivers them to IP. An outgoing queue can overflow. If this happens, the operating system asks the server to wait before sending any more messages.

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 (see Figure 14.7).

Figure 14.7 *Multiplexing and demultiplexing*



Multiplexing

At the sender site, there may be several processes that need to send user datagrams. However, there is only one UDP. This is a many-to-one relationship and requires multiplexing. UDP accepts messages from different processes, differentiated by their assigned port numbers. After adding the header, UDP passes the user datagram to IP.

Demultiplexing

At the receiver site, there is only one UDP. However, we may have many processes that can receive user datagrams. This is a one-to-many relationship and requires demultiplexing. UDP receives user datagrams from IP. After error checking and dropping of the header, UDP delivers each message to the appropriate process based on the port numbers.

14.4 UDP APPLICATIONS

Although UDP meets almost none of the criteria we mentioned in Chapter 13 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 needs sometimes 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.

In this section, we first discuss some features of UDP that may need to be considered when one designs an application program and then show some typical applications.

UDP Features

We briefly discuss some features of UDP and their advantages and disadvantages.

Connectionless Service

As we mentioned previously, UDP is a connectionless protocol. Each UDP packet is independent from other packets sent by the same application program. This feature can be considered as an advantage or disadvantage depending on the application requirement. It is an advantage if, for example, a client application needs to send a short request to a server and to receive a short response. If the request and response can each fit in one single user datagram, a connectionless service may be preferable. The overhead to establish and close a connection may be significant in this case. In the connection-oriented service, to achieve the above goal, at least 9 packets are exchanged between the client and the server; in connectionless service only two packets are exchanged. The connectionless service provides less delay; the connection-oriented service creates more delay. If delay is an important issue for the application, the connectionless service is preferred.

Lack of Congestion Control

UDP does not provide congestion control. However, UDP does not create additional traffic in an error-prone network. TCP may resend a packet several times and thus contribute to the creation of congestion or worsen a congested situation. Therefore, in some cases, lack of error control in UDP can be considered an advantage when congestion is a big issue.

Lack of Error Control

UDP does not provide error control; it provides an unreliable service. Most applications expect reliable service from a transport-layer protocol. Although a reliable service is desirable, it may have some side effects that are not acceptable to some applications. When a transport layer provides reliable services, if a part of the message is lost or corrupted, it needs to be resent. This means that the receiving transport layer cannot deliver that part to the application immediately; there is an uneven delay between different parts of the message delivered to the application layer. Some applications by nature do not even notice these uneven delays, but for some they are very crucial.

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 21).
- UDP is suitable for a process with internal flow and error-control mechanisms. For example, the Trivial File Transfer Protocol (TFTP) (see Chapter 21) process includes flow and error control. It can easily use UDP.
- 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 24).
- UDP is used for some route updating protocols such as Routing Information Protocol (RIP) (see Chapter 11).
- UDP is normally used for real-time applications that cannot tolerate uneven delay between sections of a received message.

14.5 UDP PACKAGE

To show how UDP handles the sending and receiving of UDP packets, we present a simple version of the UDP package.

We can say that the UDP package involves five components: a control-block table, input queues, a control-block module, an input module, and an output module. Figure 14.8 shows these five components and their interactions.

Control-Block Table

In our package, UDP has a control-block table to keep track of the open ports. Each entry in this table has a minimum of four fields: the state, which can be FREE or IN-USE, the process ID, the port number, and the corresponding queue number.

Input Queues

Our UDP package uses a set of input queues, one for each process. In this design, we do not use output queues.

Control-Block Module

The control-block module (Table 14.2) is responsible for the management of the control-block table. When a process starts, it asks for a port number from the operating system. The operating system assigns well-known port numbers to servers and ephemeral port numbers to clients. The process passes the process ID and the port number to the control-block module to create an entry in the table for the process. The module

does not create the queues. The field for queue number has a value of zero. Note that we have not included a strategy to deal with a table that is full.

Input Module

The input module (Table 14.3) receives a user datagram from the IP. It searches the control-block table to find an entry having the same port number as this user datagram. If the entry is found, the module uses the information in the entry to enqueue the data. If the entry is not found, it generates an ICMP message.

Figure 14.8 *UDP design*

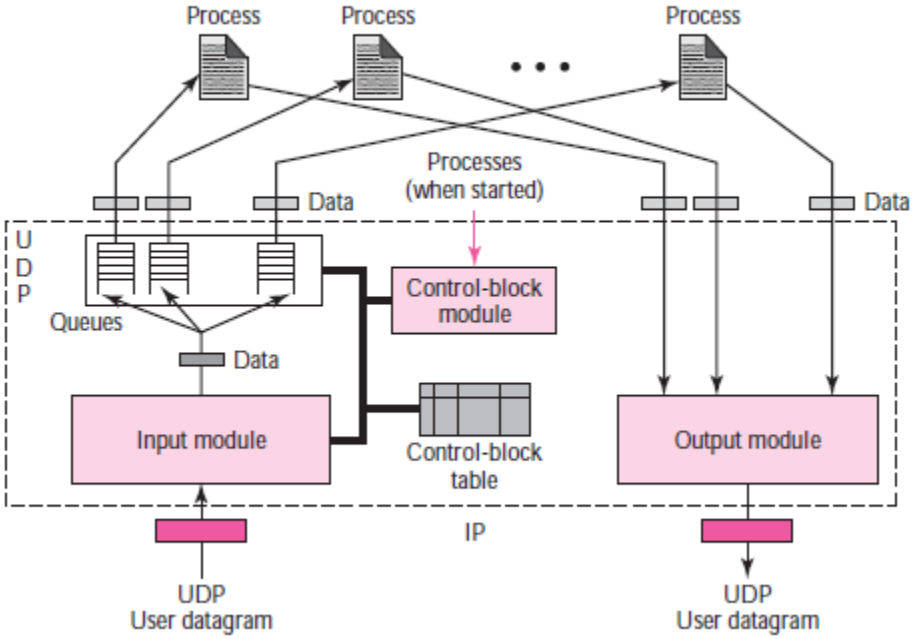


Table 14.2 *Control Block Module*

```
1  UDP_Control_Block_Module (process ID, port number)
2  {
3       Search the table for a FREE entry.
4       if (not found)
5            Delete one entry using a predefined strategy.
6       Create a new entry with the state IN-USE
7       Enter the process ID and the port number.
8       Return.
9  } // End module
```

Table 14.3 *Input Module*

```
UDP_INPUT_Module (user_datagram)
{
     Look for the entry in the control_block table
     if (found)
     {
          Check to see if a queue is allocated
          If (queue is not allocated)
               allocate a queue
          else
               enqueue the data
     } //end if
     else
     {
          Ask ICMP to send an "unreachable port" message
          Discard the user datagram
     } //end else

     Return.
} // end module
```

Output Module

The output module (Table 14.4) is responsible for creating and sending user datagrams.

Table 14.4 *Output Module*

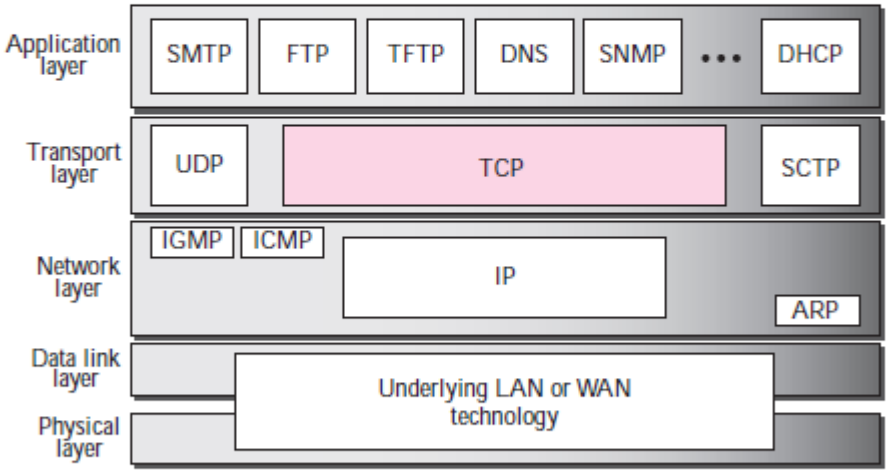
```
UDP_OUTPUT_MODULE (Data)
{
     Create a user datagram
     Send the user datagram
     Return.
}
```

Transmission Control Protocol (TCP)

15.1 TCP SERVICES

Figure 15.1 shows the relationship of TCP to the other protocols in the TCP/IP protocol suite. TCP lies between the application layer and the network layer, and serves as the intermediary between the application programs and the network operations.

Figure 15.1 *TCP/IP protocol suite*



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 (see Chapter 13). Table 15.1 lists some well-known port numbers used by TCP.

Table 15.1 *Well-known Ports used by TCP*

Port	*Protocol*	*Description*
7	Echo	Echoes a received datagram back to the sender
9	Discard	Discards any datagram that is received
11	Users	Active users
13	Daytime	Returns the date and the time
17	Quote	Returns a quote of the day

Table 15.1 *Well-known Ports used by TCP (continued)*

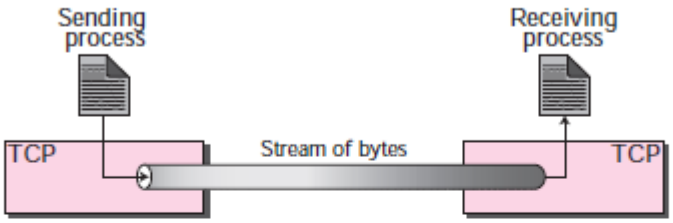
Port	*Protocol*	*Description*
19	Chargen	Returns a string of characters
20 and 21	FTP	File Transfer Protocol (Data and Control)
23	TELNET	Terminal Network
25	SMTP	Simple Mail Transfer Protocol
53	DNS	Domain Name Server
67	BOOTP	Bootstrap Protocol
79	Finger	Finger
80	HTTP	Hypertext Transfer Protocol

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 15.2. The sending process produces (writes to) the stream of bytes and the receiving process consumes (reads from) them.

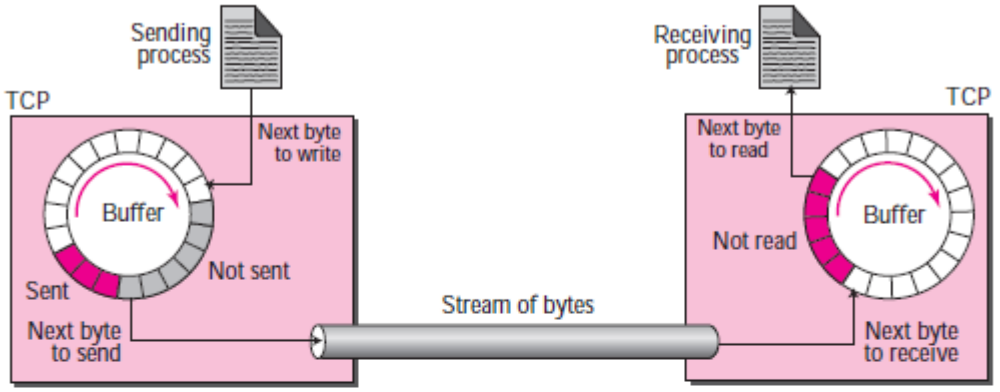
Figure 15.2 *Stream delivery*



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. We will see later that these buffers are also necessary for flow- and error-control mechanisms used by TCP. One way to implement a buffer is to use a circular array of 1-byte locations as shown in Figure 15.3. 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 15.3 *Sending and receiving buffers*



The figure 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, as we will see later in this chapter, 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.

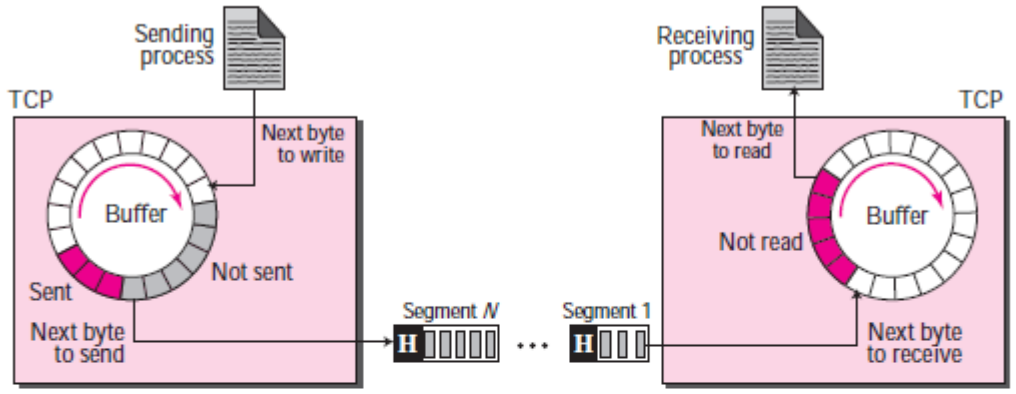
The operation of the buffer at the receiver is simpler. The circular buffer is divided into two areas (shown as white and colored). The white area contains empty chambers to be filled by bytes received from the network. The colored sections contain received bytes that can be read by the receiving process. 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 IP 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 IP layer for transmission. The segments are encapsulated in an IP datagram and transmitted. This entire operation is transparent to the receiving process. Later we will see that segments may be received out of order, lost, or corrupted and resent. All of these are handled by the TCP sender with the receiving application process unaware of TCP's activities. Figure 15.4 shows how segments are created from the bytes in the buffers.

Note that segments are not necessarily all the same size. In the figure, 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 15.4 *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, since TCP is a connection-oriented protocol, a connection needs to be established for each pair of processes. This will be more clear when we discuss the client/server paradigm in Chapter 17.

Connection-Oriented Service

TCP, unlike UDP, is a connection-oriented protocol. As shown in Chapter 13, 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 TCPs establish a virtual connection between them.
2. Data are exchanged in both directions.
3. The connection is terminated.

Note that this is a virtual 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 resent. 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.

15.2 TCP FEATURES

To provide the services mentioned in the previous section, 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 the *acknowledgment number*. These two fields refer to a byte number and not a segment number.

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 1,057 and the total data to be sent is 6,000 bytes, the bytes are numbered from 1,057 to 7,056. 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 for each segment is the number of the first byte of data carried in that segment.

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

When a segment carries a combination of data and control information (piggybacking), it uses a sequence number. If a segment does not carry user data, it does not logically define a sequence number. The field is there, but the value is not valid. 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 one 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 5,643 as an acknowledgment number, it has received all bytes from the beginning up to 5,642. Note that this does not mean that the party has received 5,642 bytes because the first byte number does not have to start from 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.

Flow Control

TCP, unlike UDP, provides flow control. The sending TCP controls how much data can be accepted from the sending process; the receiving TCP controls how much data can to be sent by the sending TCP (See Chapter 13). This is done to prevent the receiver from being overwhelmed with data. The numbering system allows TCP to use a byte-oriented flow control, as we discuss later in the chapter.

Error Control

To provide reliable service, TCP implements an error control mechanism. Although error control considers a segment as the unit of data for error detection (loss or corrupted segments), error control is byte-oriented, as we will see later.

Congestion Control

TCP, unlike UDP, takes into account congestion in the network. The amount of data sent by a sender is not only controlled by the receiver (flow control), but is also determined by the level of congestion, if any, in the network.

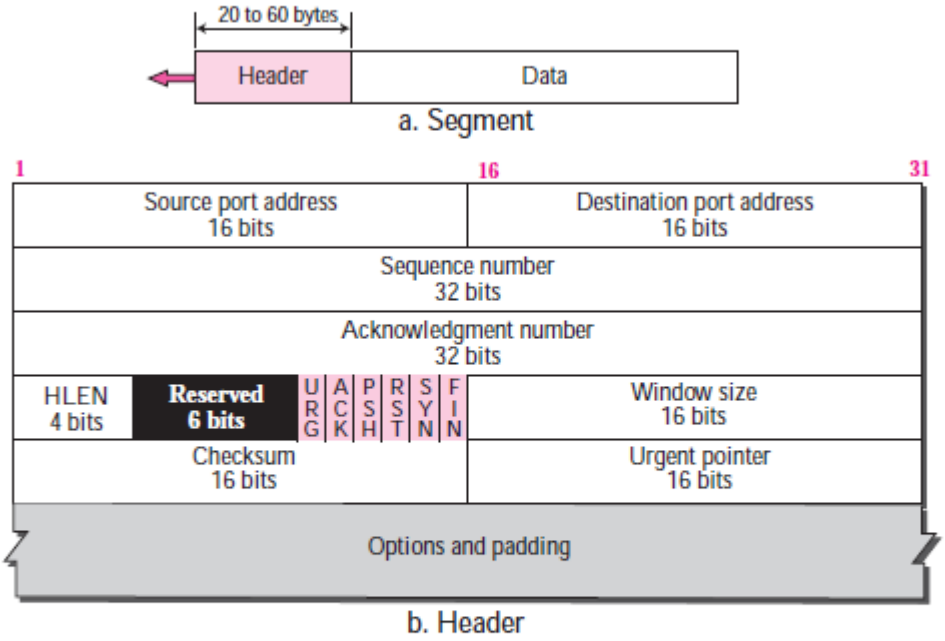
15.3 SEGMENT

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 15.5. 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 chapter.

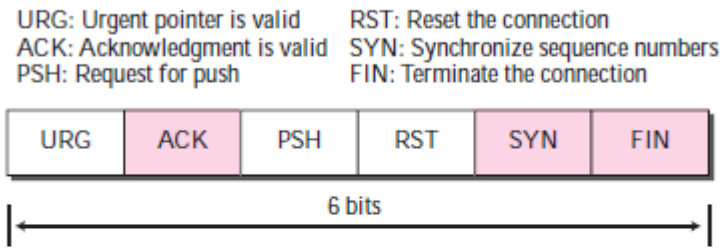
Figure 15.5 *TCP segment format*



- **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. This serves the same purpose as the source port address in the UDP header discussed in Chapter 14.

- **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. This serves the same purpose as the destination port address in the UDP header discussed in Chapter 14.
- **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 later) 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 ($5 \times 4 = 20$) and 15 ($15 \times 4 = 60$).
- **Reserved.** This is a 6-bit field reserved for future use.
- **Control.** This field defines 6 different control bits or flags as shown in Figure 15.6. 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 the figure. We will discuss them further when we study the detailed operation of TCP later in the chapter.

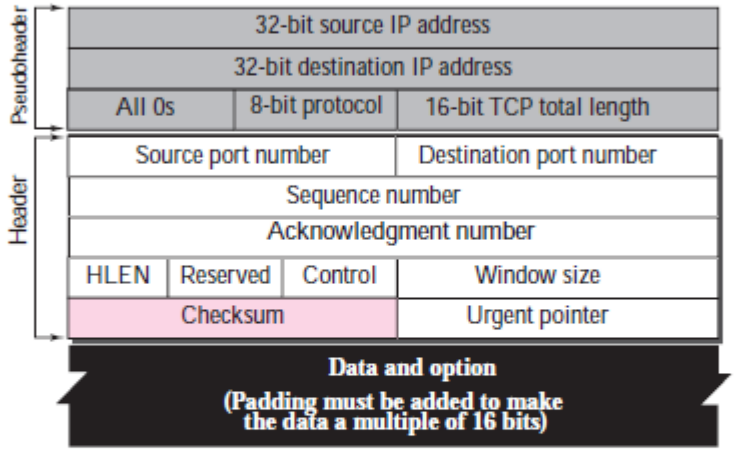
Figure 15.6 *Control field*



- **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 in Chapter 14.

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 15.7.

Figure 15.7 *Pseudoheader added to the TCP datagram*



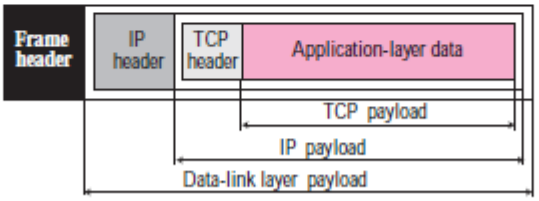
The use of the checksum in TCP is mandatory.

- **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. This will be discussed later in this chapter.
- **Options.** There can be up to 40 bytes of optional information in the TCP header. We will discuss the different options currently used in the TCP header later in the chapter.

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 as shown in Figure 15.8.

Figure 15.8 *Encapsulation*



15.4 A TCP CONNECTION

TCP is connection-oriented. As discussed in Chapter 13, a connection-oriented transport protocol establishes a virtual path between the source and destination. All of the segments belonging to a message are then sent over this virtual path. Using a single virtual 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 virtual, 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.

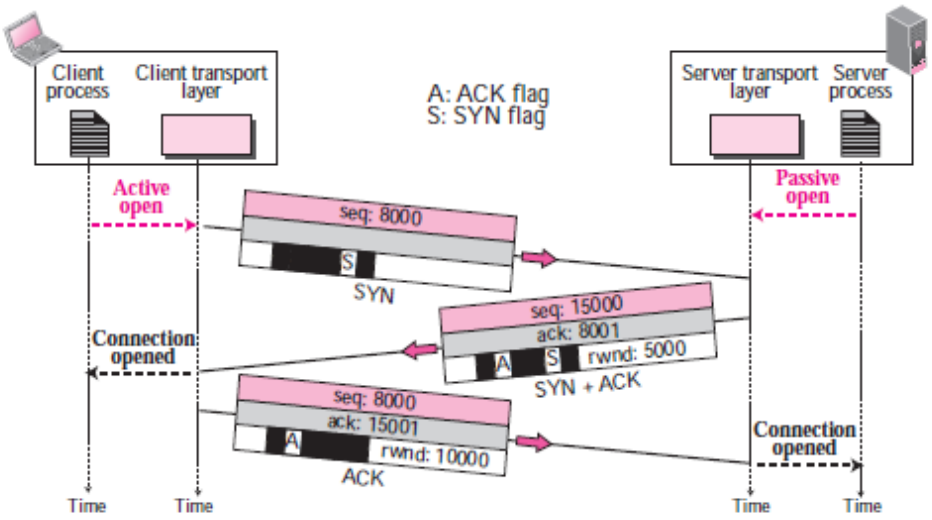
The process starts with the server. The server program tells its TCP that it is ready to accept a connection. This request is called a *passive open*. Although the server TCP is ready to accept a connection from any machine in the world, it cannot make the connection itself.

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 15.9.

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.

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. The segment can also include some

Figure 15.9 *Connection establishment using three-way handshaking*



options that we discuss later in the chapter. Note that the SYN segment is a control segment and carries no data. However, it consumes one sequence number. When the data transfer starts, the ISN is incremented by 1. We can say that the SYN segment carries no real data, but we can think of it as containing one 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: 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.

A SYN + ACK segment cannot carry data, but 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 sequence number in this segment is the same as the one in the SYN segment; the ACK segment does not consume any sequence numbers. The client must also define the server window size. Some implementations allow this third segment in the connection phase to carry the first chunk of data from the

client. In this case, the third segment must have a new sequence number showing the byte number of the first byte in the data. In general, the third segment usually does not carry data and consumes no sequence numbers.

An ACK segment, if carrying no data, consumes no sequence number.

Simultaneous Open

A rare situation may occur when both processes issue an active open. In this case, both TCPs transmit a SYN + ACK segment to each other and one single connection is established between them. We will show this case when we discuss the transition diagram in the next section.

SYN Flooding Attack

The connection establishment procedure in TCP is susceptible to a serious security problem called **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 (explained later in the chapter) 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.

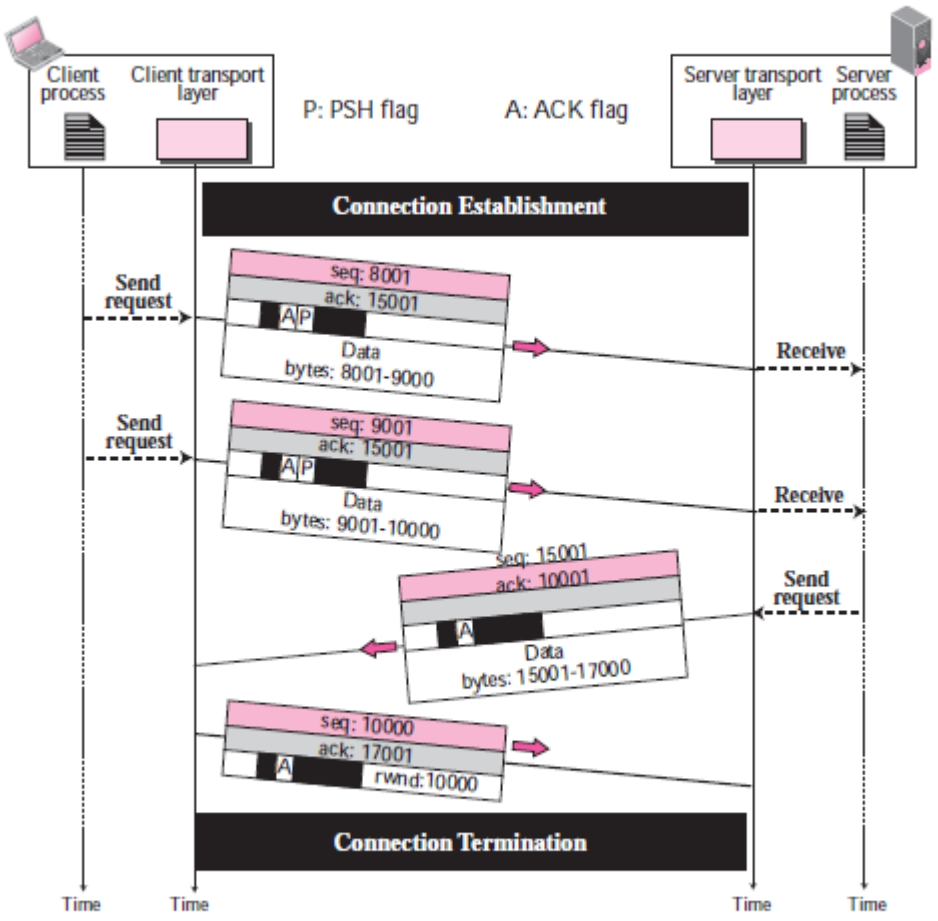
Some implementations of TCP have strategies to alleviate the effect of a SYN attack. Some have imposed a limit of connection requests during a specified period of time. Others try to filter out datagrams coming from unwanted source addresses. One recent strategy is to postpone resource allocation until 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 the next chapter, uses this strategy.

Data Transfer

After 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 15.10 shows an example.

In this example, after a connection is established, the client sends 2,000 bytes of data in two segments. The server then sends 2,000 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 tries to deliver data to the server process as soon as they are received. We discuss the use of this

Figure 15.10 *Data transfer*



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 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 keystroke 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.

Although the push operation can be requested by the application program, most current TCP implementations ignore such requests. TCP can choose whether or not to use this feature.

Urgent Data

TCP is a stream-oriented protocol. This means that the data is 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).

When the receiving TCP receives a segment with the URG bit set, it informs the receiving application of the situation. How this is done, depends on the operation system. It is then to the discretion of the receiving program to take an action.

It is important to mention that TCP's urgent data is neither a priority service nor an expedited data service. 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.

Thus, signaling the presence of urgent data and marking its position in the data stream are the only aspects that distinguish the delivery of urgent data from the delivery of all other TCP data. For all other purposes, urgent data is treated identically to the rest of the TCP byte stream. The application program at the receiver site must read every byte of data exactly in the order it was submitted regardless of whether or not urgent mode is used. The standard TCP, as implemented, does not ever deliver any data out of order.

Connection Termination

Any 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.

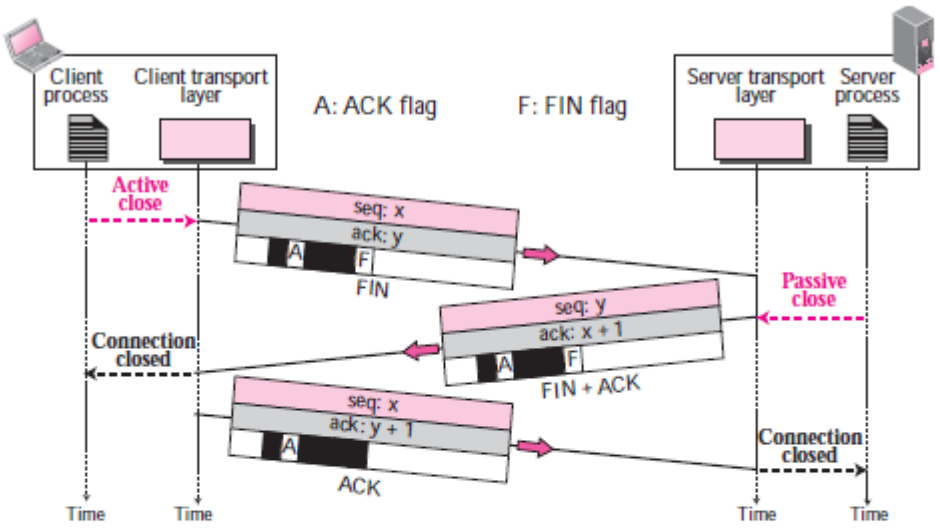
Three-Way Handshaking

Most implementations today allow *three-way handshaking* for connection termination as shown in Figure 15.11.

1. In a common 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 the figure. If it is only a control segment, it consumes only one sequence number.

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

Figure 15.11 *Connection termination using three-way handshaking*



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.
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.

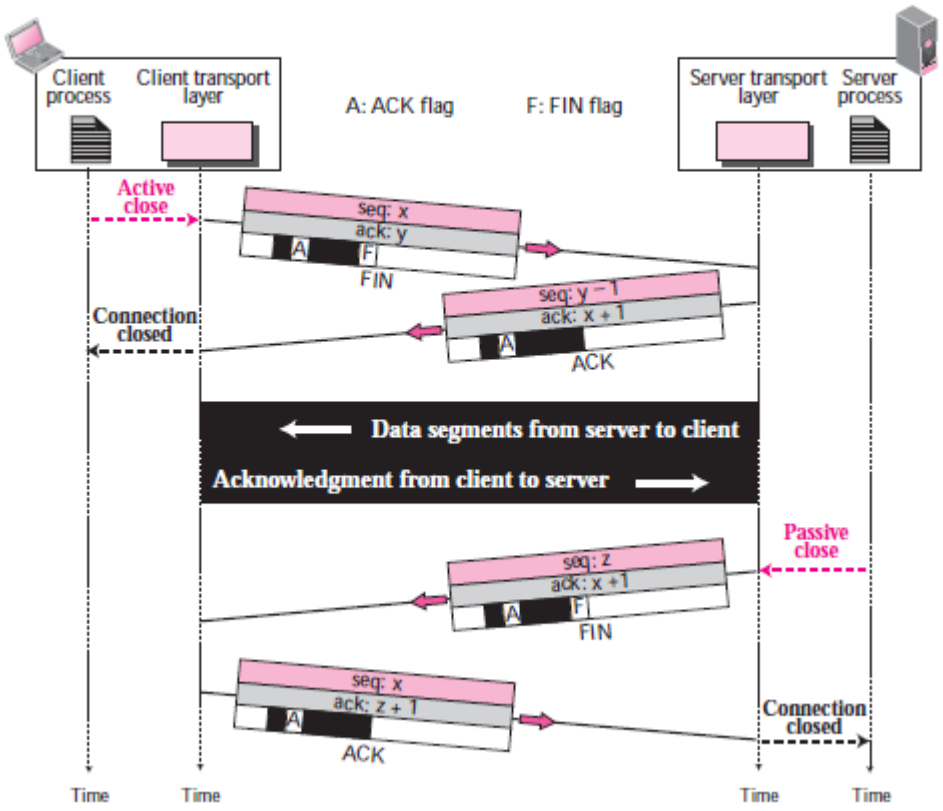
Half-Close

In TCP, one end can stop sending data while still receiving data. This is called a **half-close.** Either the server or the client can issue a half-close request. It can occur when the server needs all the data before processing can begin. A good example is sorting. When the client sends data to the server to be sorted, the server needs to receive all the data before sorting can start. This means the client, after sending all data, can close the connection in the client-to-server direction. However, the server-to-client direction must remain open to return the sorted data. The server, after receiving the data, still needs time for sorting; its outbound direction must remain open.

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

Figure 15.12 shows an example of a half-close. The data transfer from the client to the server stops. The client half-closes the connection by sending a FIN segment. The server accepts the half-close by sending the ACK segment. The server, however, can still send data. When the server has sent all of the processed data, it sends a FIN segment, which is acknowledged by an ACK from the client.

Figure 15.12 *Half-close*



After half closing the connection, data can travel from the server to the client and acknowledgments can travel from the client to the server. The client cannot send any more data to the server. Note the sequence numbers we have used. The second segment (ACK) consumes no sequence number. Although the client has received sequence number $y - 1$ and is expecting y, the server sequence number is still $y - 1$. When the connection finally closes, the sequence number of the last ACK segment is still x, because no sequence numbers are consumed during data transfer in that direction.

Connection Reset

TCP at one end may deny a connection request, may abort an existing connection, or may terminate an idle connection. All of these are done with the RST (reset) flag.

Denying a Connection

Suppose the TCP on one side has requested a connection to a nonexistent port. The TCP on the other side may send a segment with its RST bit set to deny the request. We will show an example of this case in the next section.

Aborting a Connection

One TCP may want to abort an existing connection due to an abnormal situation. It can send an RST segment to close the connection. We also show an example of this case in the next section.

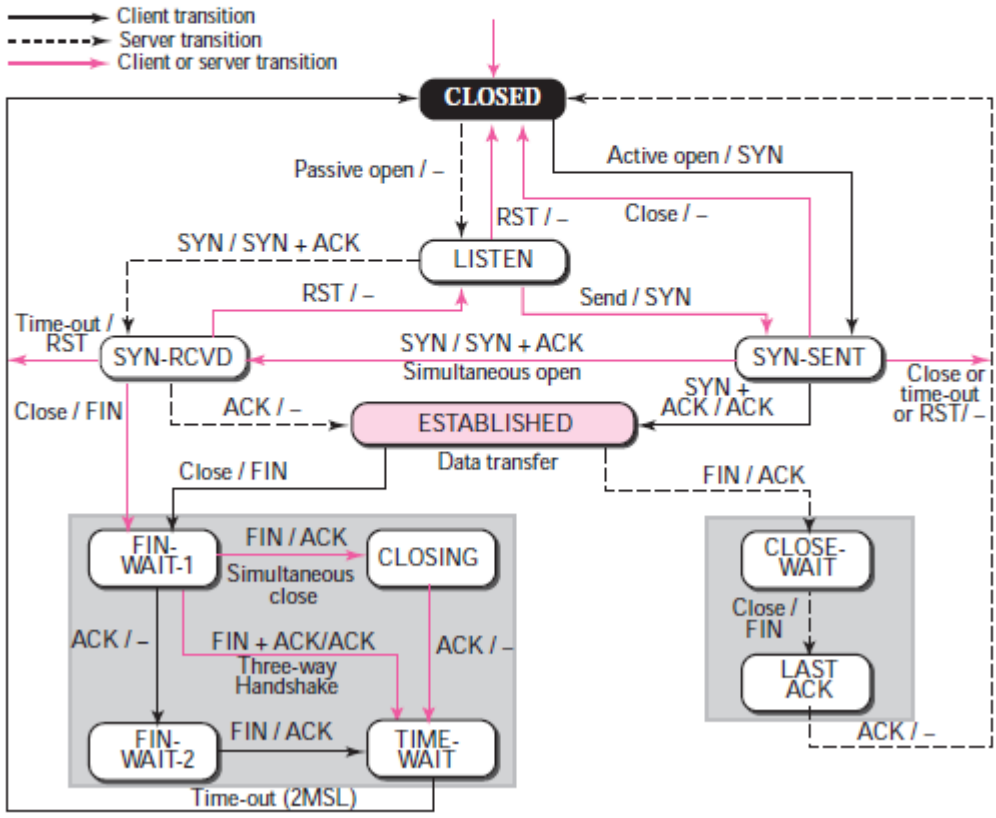
Terminating an Idle Connection

The TCP on one side may discover that the TCP on the other side has been idle for a long time. It may send an RST segment to end the connection. The process is the same as aborting a connection.

15.5 STATE TRANSITION DIAGRAM

To keep track of all the different events happening during connection establishment, connection termination, and data transfer, TCP is specified as the finite state machine shown in Figure 15.13.

Figure 15.13 *State transition diagram*



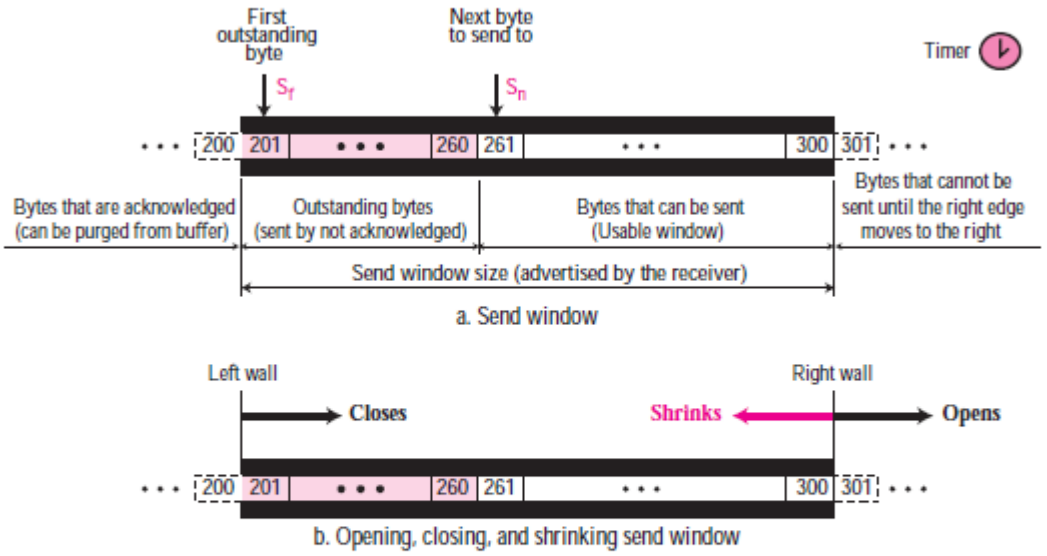
15.6 WINDOWS IN TCP

Before discussing data transfer in TCP and the issues such as flow, error, and congestion control, we describe the windows used in TCP. TCP uses two windows (send window and receive window) for each direction of data transfer, which means four windows for a bidirectional communication. However, to make the discussion simple, we make an unrealistic assumption that communication is only unidirectional (say from client to server); the bidirectional communication can be inferred using two unidirectional communications with piggybacking.

Send Window

Figure 15.22 shows an example of a send window. The window we have used is of size 100 bytes (normally thousands of bytes), but later we see that the send window size is dictated by the receiver (flow control) and the congestion in the underlying network (congestion control). The figure shows how a send window *opens*, *closes*, or *shrinks*.

Figure 15.22 *Send window in TCP*



The send window in TCP is similar to one used with the Selective Repeat protocol (Chapter 13), but with some differences:

1. One difference is the nature of entities related to the window. The window in SR numbers pockets, but the window in the TCP numbers bytes. Although actual

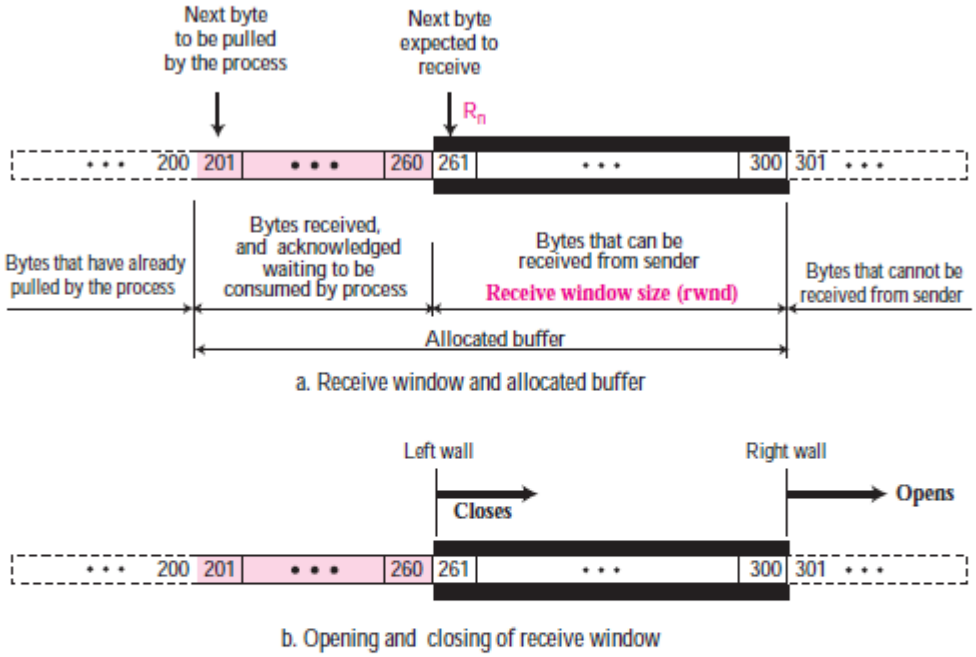
transmission in TCP occurs segment by segment, the variables that control the window are expressed in bytes.

2. The second difference is that, in some implementations, TCP can store data received from the process and send them later, but we assume that the sending TCP is capable of sending segments of data as soon as it receives them from its process.
3. Another difference is the number of timers. The theoretical Selective Repeat protocol may use several timers for each packet sent, but the TCP protocol uses only one timer. We later explain the use of this timer in error control.

Receive Window

Figure 15.23 shows an example of a receive window. The window we have used is of size 100 bytes (normally thousands of bytes). The figure also shows how the receive window opens and closes; in practice, the window should never shrink.

Figure 15.23 *Receive window in TCP*



There are two differences between the receive window in TCP and the one we used for SR in Chapter 13.

1. The first difference is that TCP allows the receiving process to pull data at its own pace. This means that part of the allocated buffer at the receiver may be occupied by bytes that have been received and acknowledged, but are waiting to be pulled by the receiving process. The receive window size is then always smaller or equal to the buffer size, as shown in the above figure. The receiver window size determines the number of bytes that the receive window can accept from the sender before being

overwhelmed (flow control). In other words, the receive window size, normally called *rwnd*, can be determined as:

rwnd = buffer size – number of waiting bytes to be pulled

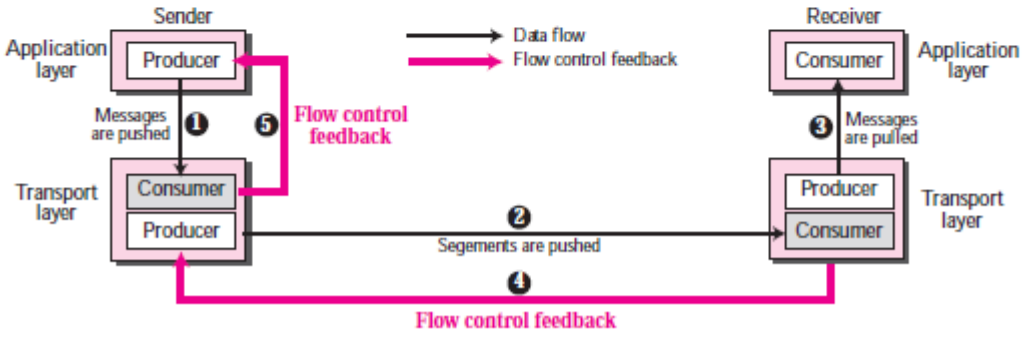
2. The second difference is the way acknowledgments are used in the TCP protocol. Remember that an acknowledgement in SR is selective, defining the uncorrupted packets that have been received. The major acknowledgment mechanism in TCP is a cumulative acknowledgment announcing the next expected byte to receive (in this way TCP looks like GBN discussed in Chapter 13). The new versions of TCP, however, uses both cumulative and selective acknowledgements as we will discuss later in the option section.

15.7 FLOW CONTROL

As discussed in Chapter 13, *flow control* balances the rate a producer creates data with the rate a consumer can use the data. TCP separates flow control from error control. In this section we discuss flow control, ignoring error control. We temporarily assume that the logical channel between the sending and receiving TCP is error-free.

Figure 15.24 shows unidirectional data transfer between a sender and a receiver; bidirectional data transfer can be deduced from unidirectional one as discussed in Chapter 13.

Figure 15.24 *Data flow and flow control feedbacks in TCP*



The figure shows that data travel from the sending process down to the sending TCP, from the sending TCP to the receiving TCP, and from receiving TCP up to the receiving process (paths 1, 2, and 3). Flow control feedbacks, however, are traveling from the receiving TCP to the sending TCP and from the sending TCP up to the sending process (paths 4 and 5). Most implementations of TCP do not provide flow control feedback from the receiving process to the receiving TCP; they let the receiving process pull data from the receiving TCP whenever it is ready to do so. In other words, the receiving TCP controls the sending TCP; the sending TCP controls the sending process.

Flow control feedback from the sending TCP to the sending process (path 5) is achieved through simple rejection of data by sending TCP when its window is full. This means that our discussion of flow control concentrates on the feedback sent from the receiving TCP to the sending TCP (path 4).

Opening and Closing Windows

To achieve flow control, TCP forces the sender and the receiver to adjust their window sizes, although the size of the buffer for both parties is fixed when the connection is established. The receive window closes (moves its left wall to the right) when more bytes arrive from the sender; it opens (moves its right wal l to the right) when more bytes are pulled by the process. We assume that it does not shrink (the right wall does not move to the left).

The opening, closing, and shrinking of the send window is controlled by the receiver. The send window closes (moves its left wall to the right) when a new acknowledgement allows it to do so. The send window opens (its right wall moves to the right) when the receive window size (rwnd) advertised by the receiver allows it to do so. The send window shrinks on occasion. We assume that this situation does not occur.

Shrinking of Windows

As we said before, the receive window cannot shrink. But the send window can shrink if the receiver defines a value for rwnd that results in shrinking the window. Some implementations do not allow the shrinking of the send window. The limitation does not allow the right wall of the send window to move to the left. In other words, the receiver needs to keep the following relationship between the last and new acknowledgment and the last and new rwnd values to prevent the shrinking of the send window:

$$\textbf{new ackNo} + \textbf{new rwnd} \geq \textbf{last ackNo} + \textbf{last rwnd}$$

The left side of the inequality represents the new position of the right wall with respect to the sequence number space; the right side shows the old position of the right wall. The relationship shows that the right wall should not move to the left. The inequality is a mandate for the receiver to check its advertisment. However, note that the inequality is valid only if $S_f < S_n$; we need to remember that all calculations are in modulo 2^{32}.

15.8 ERROR CONTROL

TCP is a reliable transport layer protocol. This means that an application program that delivers a stream of data to TCP relies on TCP to deliver the entire stream to the application program on the other end in order, without error, and without any part lost or duplicated.

TCP provides reliability using error control. Error control includes mechanisms for detecting and resending corrupted segments, resending lost segments, storing out-of-order segments until missing segments arrive, and detecting and discarding duplicated segments. Error control in TCP is achieved through the use of three simple tools: checksum, acknowledgment, and time-out.

Checksum

Each segment includes a checksum field, which is used to check for a corrupted segment. If a segment is corrupted as deleted by an invalid checksum, the segment is discarded by the destination TCP and is considered as lost. TCP uses a 16-bit checksum that is mandatory in every segment. We discussed how to calculate checksums earlier in the chapter.

Acknowledgment

TCP uses acknowledgments to confirm the receipt of data segments. Control segments that carry no data, but consume a sequence number, are also acknowledged. ACK segments are never acknowledged.

ACK segments do not consume sequence numbers and are not acknowledged.

Acknowledgment Type

In the past, TCP used only one type of acknowledgment: cumulative acknowledgment. Today, some TCP implementations also use selective acknowledgment.

Cumulative Acknowledgment (ACK) TCP was originally designed to acknowledge receipt of segments cumulatively. The receiver advertises the next byte it expects to receive, ignoring all segments received and stored out of order. This is sometimes referred to as positive cumulative acknowledgment or ACK. The word "positive" indicates that no feedback is provided for discarded, lost, or duplicate segments. The 32-bit ACK field in the TCP header is used for cumulative acknowledgments and its value is valid only when the ACK flag bit is set to 1.

15.9 CONGESTION CONTROL

We briefly discussed congestion control in Chapter 13. Congestion control in TCP is based on both open-loop and closed-loop mechanisms. TCP uses a congestion window and a congestion policy that avoid congestion and detect and alleviate congestion after it has occurred.

Congestion Window

Previously, we talked about flow control and tried to discuss solutions when the receiver is overwhelmed with data. We said that the sender window size is determined by the available buffer space in the receiver (rwnd). In other words, we assumed that it is only the receiver that can dictate to the sender the size of the sender's window. We totally ignored another entity here, the network. If the network cannot deliver the data as fast as it is created by the sender, it must tell the sender to slow down. In other words, in addition to the receiver, the network is a second entity that determines the size of the sender's window.

The sender has two pieces of information: the receiver-advertised window size and the congestion window size. The actual size of the window is the minimum of these two.

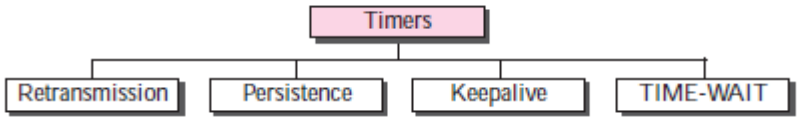
Actual window size = minimum (rwnd, cwnd)

We show shortly how the size of the congestion window (cwnd) is determined.

15.10 TCP TIMERS

To perform its operation smoothly, most TCP implementations use at least four timers as shown in Figure 15.38.

Figure 15.38 *TCP timers*



Retransmission Timer

To retransmit lost segments, TCP employs one retransmission timer (for the whole connection period) that handles the retransmission time-out (RTO), the waiting time for an acknowledgment of a segment. We can define the following rules for the retransmission timer:

1. When TCP sends the segment in front of the sending queue, it starts the timer.
2. When the timer expires, TCP resends the first segment in front of the queue, and restarts the timer.
3. When a segment (or segments) are cumulatively acknowledged, the segment (or segments) are purged from the queue.
4. If the queue is empty, TCP stops the timer; otherwise, TCP restarts the timer.

Round-Trip Time (RTT)

To calculate the retransmission time-out (RTO), we first need to calculate the **round-trip time (RTT)**. However, calculating RTT in TCP is an involved process that we explain step by step with some examples.

Measured RTT We need to find how long it takes to send a segment and receive an acknowledgment for it. This is the measured RTT. We need to remember that the segments and their acknowledgments do not have a one-to-one relationship; several segments may be acknowledged together. The measured round-trip time for a segment is the time required for the segment to reach the destination and be acknowledged, although the acknowledgment may include other segments. Note that in TCP, only one RTT measurement can be in progress at any time. This means that if an RTT measurement is started, no other measurement starts until the value of this RTT is finalized. We use the notation RTT_M to stand for measured RTT.

In TCP, there can be only one RTT measurement in progress at any time.

Smoothed RTT The measured RTT, RTT_M, is likely to change for each round trip. The fluctuation is so high in today's Internet that a single measurement alone cannot

Keepalive Timer

A **keepalive timer** is used in some implementations to prevent a long idle connection between two TCPs. Suppose that a client opens a TCP connection to a server, transfers some data, and becomes silent. Perhaps the client has crashed. In this case, the connection remains open forever.

To remedy this situation, most implementations equip a server with a keepalive timer. Each time the server hears from a client, it resets this timer. The time-out is usually 2 hours. If the server does not hear from the client after 2 hours, it sends a probe segment. If there is no response after 10 probes, each of which is 75 s apart, it assumes that the client is down and terminates the connection.

TIME-WAIT Timer

The TIME-WAIT (2MSL) timer is used during connection termination. We discussed the reasons for this timer in Section 15.5 (State Transition Diagram).

15.11 OPTIONS

The TCP header can have up to 40 bytes of optional information. Options convey additional information to the destination or align other options. We can define two categories of options: 1-byte options and multiple-byte options. The first category contains two types of options: end of option list and no operation. The second category, in most implementations, contains five types of options: maximum segment size, window scale factor, timestamp, SACK-permitted, and SACK (see Figure 15.41).

End of Option (EOP)

The **end-of-option (EOP) option** is a 1-byte option used for padding at the end of the option section. It can only be used as the last option. Only one occurrence of this option is allowed. After this option, the receiver looks for the payload data. Figure 15.42 shows an example. A 3-byte option is used after the header; the data section follows this option. One EOP option is inserted to align the data with the boundary of the next word.

Figure 15.41 *Options*

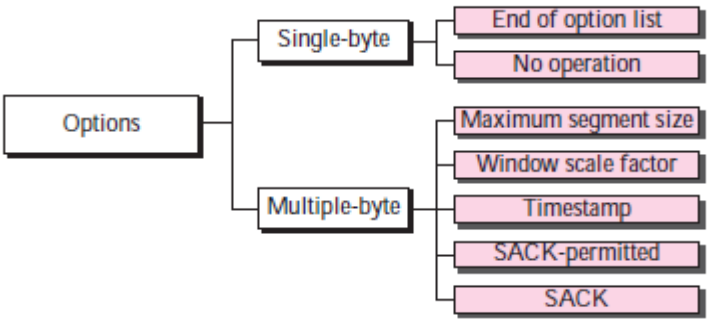
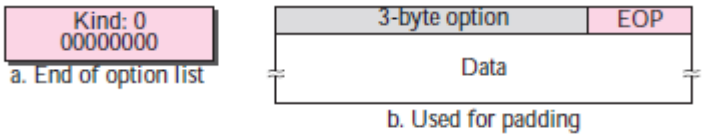


Figure 15.42 *End-of-option*



EOP can be used only once.

The EOP option imparts two pieces of information to the destination:

1. There are no more options in the header.
2. Data from the application program starts at the beginning of the next 32-bit word.

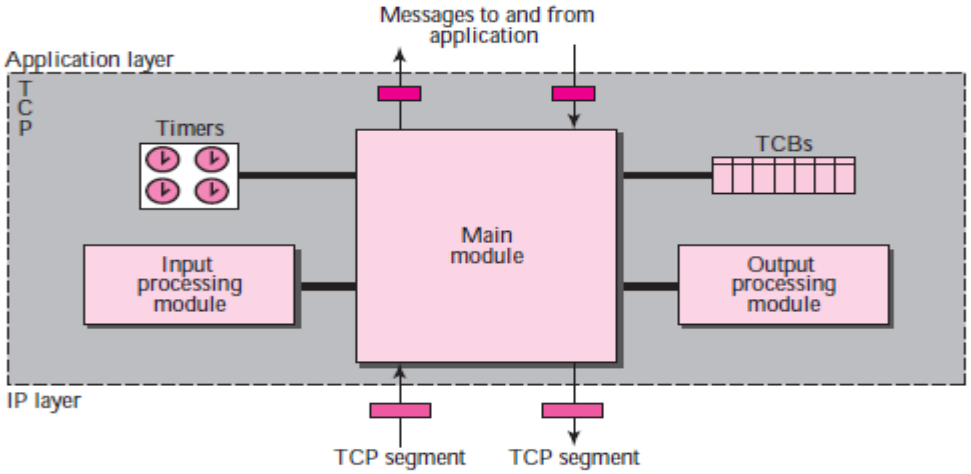
15.12 TCP PACKAGE

TCP is a complex protocol. It is a stream-service, connection-oriented protocol with an involved state transition diagram. It uses flow and error control. It is so complex that actual code involves tens of thousands of lines.

In this section, we present a simplified, bare-bones TCP package. Our purpose is to show how we can simulate the heart of TCP, as represented by the state transition diagram.

The package involves tables called transmission control blocks, a set of timers, and three software modules: a main module, an input processing module, and an output processing module. Figure 15.52 shows these five components and their interactions.

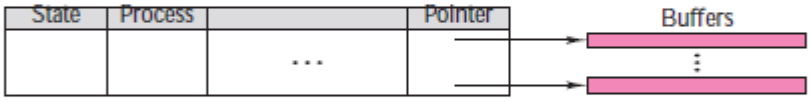
Figure 15.52 *TCP package*



Transmission Control Blocks (TCBs)

TCP is a connection-oriented transport protocol. A connection may be open for a long period of time. To control the connection, TCP uses a structure to hold information about each connection. This is called a *transmission control block* (TCB). Because at any time there can be several connections, TCP keeps an array of TCBs in the form of a table. The table is usually referred to as the TCB (see Figure 15.53).

Figure 15.53 *TCBs*



Many fields can be included in each TCB. We mention only the most common ones here.

- **State.** This field defines the state of the connection according to the state transition diagram.
- **Process.** This field defines the process using this connection at this machine as a client or a server.

- **Local IP address.** This field defines the IP address of the local machine used by this connection.
- **Local port number.** This field defines the local port number used by this connection.
- **Remote IP address.** This field defines the IP address of the remote machine used by this connection.
- **Remote port number.** This field defines the remote port number used by this connection.
- **Interface.** This field defines the local interface.
- **Local window.** This field, which can comprise several subfields, holds information about the window at the local TCP.
- **Remote window.** This field, which can comprise several subfields, holds information about the window at the remote TCP.
- **Sending sequence number.** This field holds the sending sequence number.
- **Receiving sequence number.** This field holds the receiving sequence number.
- **Sending ACK number.** This field holds the value of the ACK number sent.
- **Round-trip time.** Several fields may be used to hold information about the RTT.
- **Time-out values.** Several fields can be used to hold the different time-out values such as the retransmission time-out, persistence time-out, keepalive time-out, and so on.
- **Buffer size.** This field defines the size of the buffer at the local TCP.
- **Buffer pointer.** This field is a pointer to the buffer where the received data are kept until they are read by the application.

Timers

We have previously discussed the several timers TCP needs to keep track of its operations.

Main Module

The main module (Table 15.3) is invoked by an arriving TCP segment, a time-out event, or a message from an application program. This is a very complicated module because the action to be taken depends on the current state of the TCP. Several approaches have been used to implement the state transition diagram including using a process for each state, using a table (two-dimensional array), and so on. To keep our discussion simple, we use cases to handle the state. We have 11 states; we use 11 different cases. Each state is implemented as defined in the state transition diagram. The ESTABLISHED state needs further explanation. When TCP is in this state and data or an acknowledgment segment arrives, another module, the input processing module, is called to handle the situation. Also, when TCP is in this state and a "send data" message is issued by an application program, another module, the output processing module, is called to handle the situation.

Input Processing Module

In our design, the input processing module handles all the details needed to process data or an acknowledgment received when TCP is in the ESTABLISHED state. This module sends an ACK if needed, takes care of the window size announcement, does error checking, and so on. The details of this module are not needed for an introductory textbook.

Output Processing Module

In our design, the output processing module handles all the details needed to send out data received from application program when TCP is in the ESTABLISHED state. This module handles retransmission time-outs, persistent time-outs, and so on. One of the ways to implement this module is to use a small transition diagram to handle different output conditions. Again, the details of this module are not needed for an introductory textbook.