CHAPTER 23

Process-la-Process Delivery:
UDp, TCp, and SCTP

We begin this chapter by giving the rationale for the existence of the transport layer-the need for process-to-process delivery. We discuss the issues arising from this type of delivery, and we discuss methods to handle them.

The Internet model has three protocols at the transport layer: UDP, TCP, and SCTP. First we discuss UDP, which is the simplest of the three. We see how we can use this very simple transport layer protocol that lacks some of the features of the other two.

We then discuss TCP, a complex transport layer protocol. We see how our previously presented concepts are applied to TCP. We postpone the discussion of congestion control and quality of service in TCP until Chapter 24 because these two topics apply to the data link layer and network layer as well.

We finally discuss SCTP, the new transport layer protocol that is designed for multihomed, multistream applications such as multimedia.

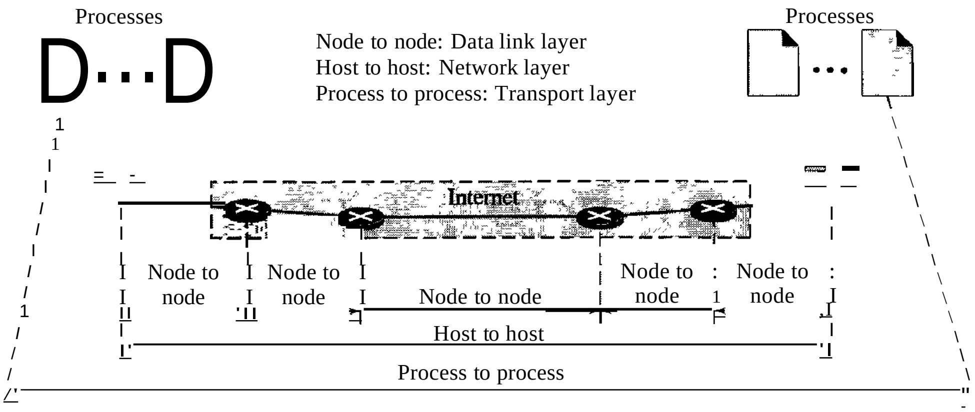
23.1 PROCESS-TO-PROCESS DELIVERY

The data link layer is responsible for delivery of frames between two neighboring nodes over a link. This is called *node-to-node delivery.* The network layer is responsible for delivery of datagrams between two hosts. This is called *host-to-host delivery.* Communication on the Internet is not defined as the exchange of data between two nodes or between two hosts. Real communication takes place between two processes (application programs). We need process-to-process delivery. However, at any moment, several processes may be running on the source host and several on the destination host. To complete the delivery, we need a mechanism to deliver data from one of these processes running on the source host to the corresponding process running on the destination host.

The transport layer is responsible for process-to-process delivery-the delivery of a packet, part of a message, from one process to another. Two processes communicate in a client/server relationship, as we will see later. Figure 23.1 shows these three types of deliveries and their domains.

The transport layer is responsible for process-to-process delivery.

Figure 23.1 *Types of data deliveries*



Client/Server Paradigm

Although there are several ways to achieve process-to-process communication, the most common one is through the client/server paradigm. A process on the local host, called a client, needs services from a process usually on the remote host, called a server.

Both processes (client and server) have the same name. For example, to get the day and time from a remote machine, we need a Daytime client process running on the local host and a Daytime server process running on a remote machine.

Operating systems today support both multiuser and multiprogramming environments. A remote computer can run several server programs at the same time, just as local computers can run one or more client programs at the same time. For communication, we must define the following:

1. Local host
2. Local process
3. Remote host
4. Remote process

Addressing

Whenever we need to deliver something to one specific destination among many, we need an address. At the data link layer, we need a MAC address to choose one node among several nodes if the connection is not point-to-point. A frame in the data link layer needs a destination MAC address for delivery and a source address for the next node's reply.

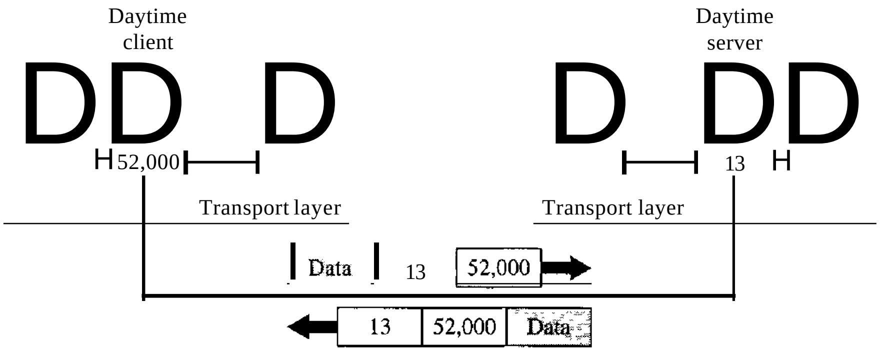
At the network layer, we need an IP address to choose one host among millions. A datagram in the network layer needs a destination IP address for delivery and a source IP address for the destination's reply.

At the transport layer, we need a transport layer address, called a port number, to choose among multiple processes running on the destination host. The destination port number is needed for delivery; the source port number is needed for the reply.

In the Internet model, the port numbers are 16-bit integers between 0 and 65,535. The client program defines itself with a port number, chosen randomly by the transport layer software running on the client host. This is the ephemeral port number.

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. Of course, one solution would be to send a special packet and request the port number of a specific server, but this requires more overhead. The Internet has decided to use universal port numbers for servers; these are called well-known port numbers. There are some exceptions to this rule; for example, there are clients that are assigned 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, discussed above, 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 23.2 shows this concept.

Figure 23.2 *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 23.3).

lANA Ranges

The lANA (Internet Assigned Number Authority) has divided the port numbers into three ranges: well known, registered, and dynamic (or private), as shown in Figure 23.4.

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

Figure 23.3 *IP addresses versus port numbers*

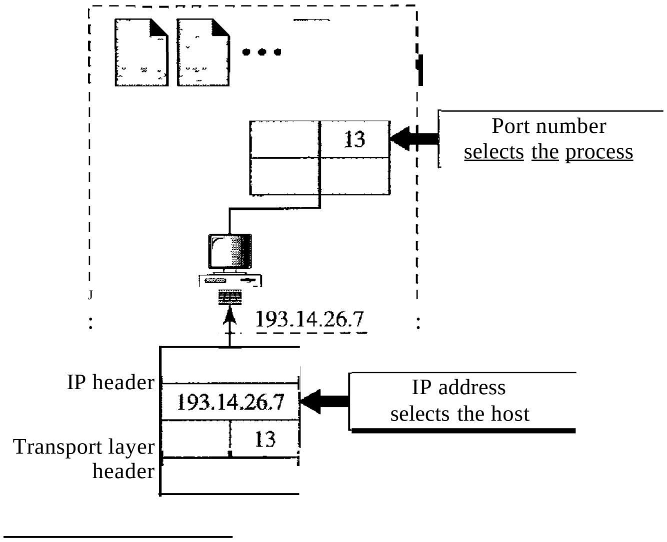
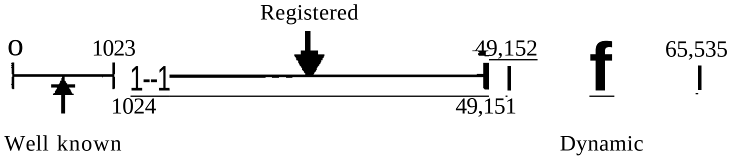


Figure 23.4 *IANA ranges*

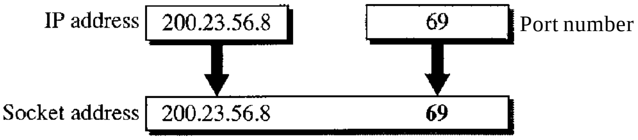


Socket Addresses

Process-to-process delivery needs two identifiers, 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 23.5).

A transport layer protocol needs a pair of socket addresses: the client socket address and the server socket address. These four pieces of information are part of the IP header and the transport layer protocol header. The IP header contains the IP addresses; the UDP or TCP header contains the port numbers.

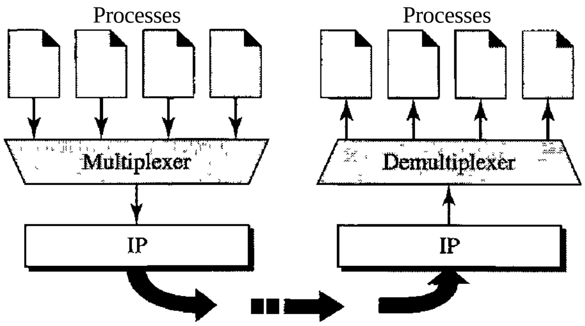
Figure 23.5 *Socket address*



Multiplexing and Demultiplexing

The addressing mechanism allows multiplexing and demultiplexing by the transport layer, as shown in Figure 23.6.

Figure 23.6 *Multiplexing and demultiplexing*



Multiplexing

At the sender site, there may be several processes that need to send packets. However, there is only one transport layer protocol at any time. This is a many-to-one relationship and requires multiplexing. The protocol accepts messages from different processes, differentiated by their assigned port numbers. After adding the header, the transport layer passes the packet to the network layer.

Demultiplexing

At the receiver site, the relationship is one-to-many and requires demultiplexing. The transport layer receives datagrams from the network layer. After error checking and dropping of the header, the transport layer delivers each message to the appropriate process based on the port number.

Connectionless Versus Connection-Oriented Service

A transport layer protocol can either be connectionless or connection-oriented.

Connectionless Service

In a connectionless service, the packets are sent from one party to another with no need for connection establishment or connection release. The packets are not numbered; they may be delayed or lost or may arrive out of sequence. There is no acknowledgment either. We will see shortly that one of the transport layer protocols in the Internet model, UDP, is connectionless.

Connection-Oriented Service

In a connection-oriented service, a connection is first established between the sender and the receiver. Data are transferred. At the end, the connection is released. We will see shortly that TCP and SCTP are connection-oriented protocols.

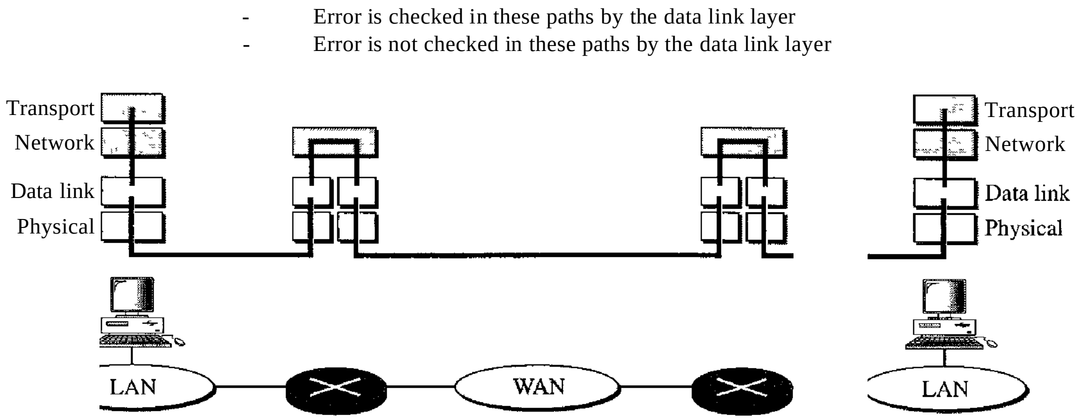
Reliable Versus Unreliable

The transport layer service can be reliable or unreliable. If the application layer program needs reliability, we use a reliable transport layer protocol by implementing flow and error control at the transport layer. This means a slower and more complex service. On the other hand, if the application program does not need reliability because it uses its own flow and error control mechanism or it needs fast service or the nature of the service does not demand flow and error control (real-time applications), then an unreliable protocol can be used.

In the Internet, there are three common different transport layer protocols, as we have already mentioned. UDP is connectionless and unreliable; TCP and SCTP are connection-oriented and reliable. These three can respond to the demands of the application layer programs.

One question often comes to the mind. If the data link layer is reliable and has flow and error control, do we need this at the transport layer, too? The answer is yes. Reliability at the data link layer is between two nodes; we need reliability between two ends. Because the network layer in the Internet is unreliable (best-effort delivery), we need to implement reliability at the transport layer. To understand that error control at the data link layer does not guarantee error control at the transport layer, let us look at Figure 23.7.

Figure 23.7 *Error control*

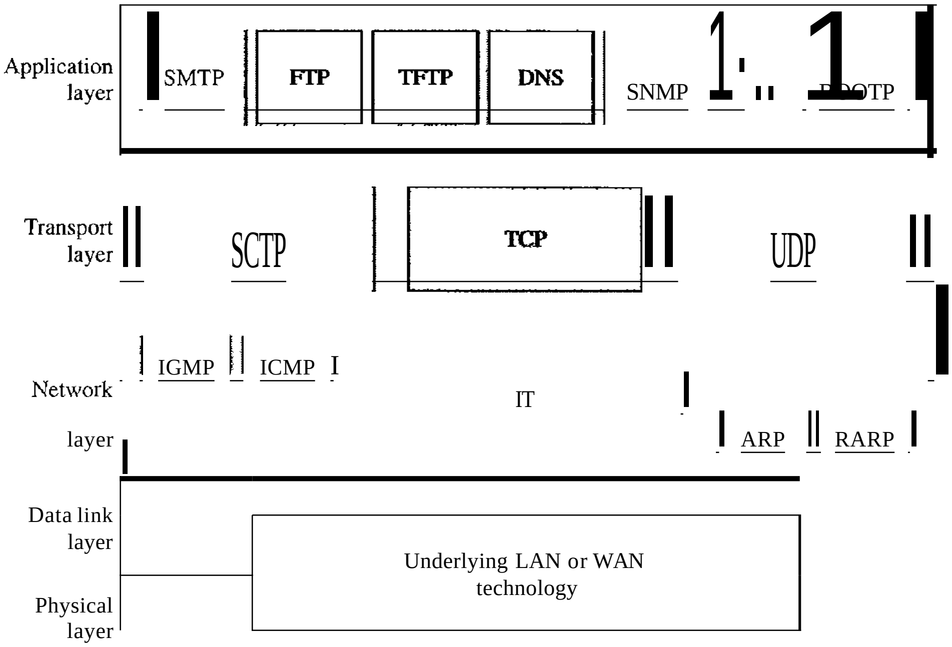


As we will see, flow and error control in TCP is implemented by the sliding window protocol, as discussed in Chapter 11. The window, however, is character-oriented, instead of frame-oriented.

Three Protocols

The original TCP/IP protocol suite specifies two protocols for the transport layer: UDP and TCP. We first focus on UDP, the simpler of the two, before discussing TCP. A new transport layer protocol, SCTP, has been designed, which we also discuss in this chapter. Figure 23.8 shows the position of these protocols in the TCP/IP protocol suite.

Figure 23.8 *Position of UDp, TCp, and SCTP in TCPIIP suite*



23.2 USER DATAGRAM PROTOCOL (UDP)

The User Datagram Protocol (UDP) is called a connectionless, unreliable transport protocol. It does not add anything to the services of IP except to provide process-to-process communication instead of host-to-host communication. Also, it performs very limited error checking.

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 by using UDP takes much less interaction between the sender and receiver than using TCP or SCTP.

Well-Known Ports for UDP

Table 23.1 shows some well-known port numbers used by UDP. Some port numbers can be used by both UDP and TCP. We discuss them when we talk about TCP later in the chapter.

Table 23.1 *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

Table 23.1 *Well-known ports used with UDP (continued)*

Port	*Protocol*	*Description*
13	Daytime	Returns the date and the time
17	Quote	Returns a quote of the day
19	Chargen	Returns a string of characters
53	Nameserver	Domain Name Service
67	BOOTPs	Server port to download bootstrap information
68	BOOTPc	Client port to download bootstrap information
69	TFTP	Trivial File Transfer Protocol
III	RPC	Remote Procedure Call
123	NTP	Network Time Protocol
161	SNMP	Simple Network Management Protocol
162	SNMP	Simple Network Management Protocol (trap)

Example 23.1

In UNIX, the well-known ports are stored in a file called fetcfservices. Each line in this file gives the name of the server and the well-known port number. We can use the *grep* utility to extract the line corresponding to the desired application. The following shows the port for FTP. Note that FTP can use port 21 with either UDP or TCP.

```
$grep     ftp   fetclservices
ftp        21ftcp
fip        211udp
```

SNMP uses two port numbers (161 and 162), each for a different purpose, as we will see in Chapter 28.

```
$grep       snmp fetclservices
snmp             161ftcp          #Simple Net  Mgmt Proto
snmp             1611udp          #Simple Net  Mgmt Proto
snmptrap         162/udp          #Traps for SNMP
```

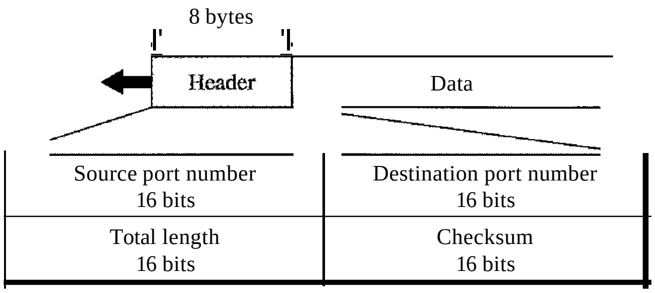
User Datagram

UDP packets, called **user datagrams**, have a fixed-size header of 8 bytes. Figure 23.9 shows the format of a user datagram.

The fields are as follows:

- **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.

Figure 23.9 *User datagram format*



- 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 a 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 a 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 next.

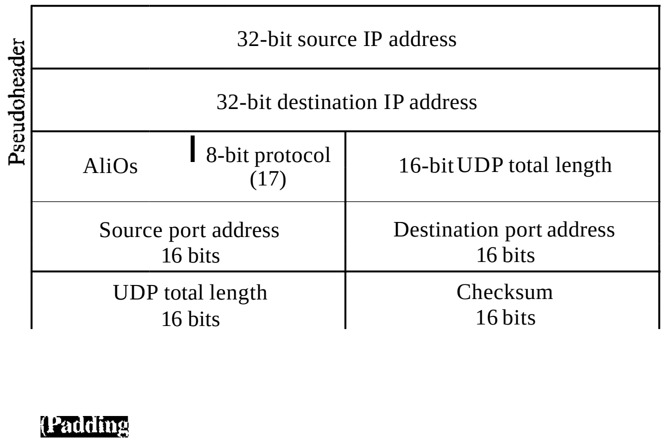
Checksum

We have already talked about the concept of the checksum and the way it is calculated in Chapter 10. We have also shown how to calculate the checksum for the IP and ICMP packet. We now show how this is done for UDP.

The UDP checksum calculation is different from the one for IP and ICMP. 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 Os (see Figure 23.10).

Figure 23.10 *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 other transport-layer protocols. 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.

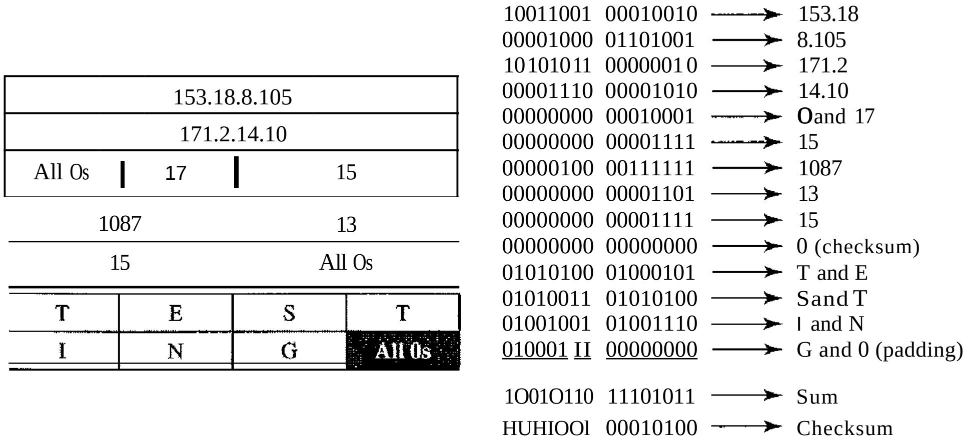
Example 23.2

Figure 23.11 shows the checksum calculation for a very small user datagram with only 7 bytes of data. Because the number of bytes of data is odd, padding is added for checksum calcUlation. The pseudoheader as well as the padding will be dropped when the user datagram is delivered to IP.

Optional Use of the Checksum

The calculation of the checksum and its inclusion in a user datagram are optional. If the checksum is not calculated, the field is filled with 1s. Note that a calculated checksum can never be allIs because this implies that the sum is all Os, which is impossible because it requires that the value of fields to be Os.

Figure 23.11 *Checksum calculation ofa simple UDP user datagram*



UDP Operation

UDP uses concepts common to the transport layer. These concepts will be discussed here briefly, and then expanded in the next section on the TCP protocol.

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 should use UDP.

Flow and Error Control

UDP is a very simple, unreliable transport protocol. There is no flow control and hence no window mechanism. The receiver may overflow with incoming messages.

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 flow control and error control means that the process using UDP should provide these mechanisms.

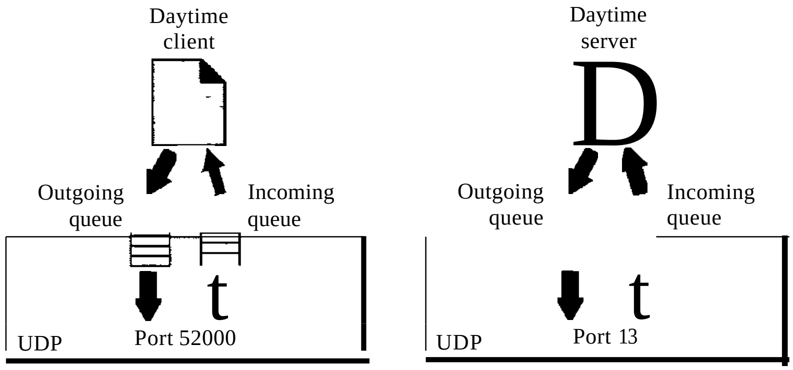
Encapsulation and Decapsulation

To send a message from one process to another, the UDP protocol encapsulates and decapsulates messages in an IP datagram.

Queuing

We have talked about ports without discussing the actual implementation of them. In UDP, queues are associated with ports (see Figure 23.12).

Figure 23.12 *Queues in UDP*



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.

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 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 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.

Use of UDP

The following lists some uses of the UDP protocol:

- 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 FrP that needs to send bulk data (see Chapter 26).
- UDP is suitable for a process with internal flow and error control mechanisms. For example, the Trivial File Transfer Protocol (TFTP) 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 28).
- UDP is used for some route updating protocols such as Routing Information Protocol (RIP) (see Chapter 22).

23.3 **TCP**

The second transport layer protocol we discuss in this chapter is called Transmission Control Protocol (TCP). TCP, like UDP, is a process-to-process (program-to-program) protocol. TCP, therefore, like UDP, uses port numbers. Unlike UDP, TCP is a connection-oriented protocol; it creates a virtual connection between two TCPs to send data. In addition, TCP uses flow and error control mechanisms at the transport level.

In brief, TCP is called a *connection-oriented, reliable* transport protocol. It adds connection-oriented and reliability features to the services of IP.

TCP Services

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

Process-to-Process Communication

Like UDP, TCP provides process-to-process communication using port numbers. Table 23.2 lists some well-known port numbers used by TCP.

Table 23.2 *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
19	Chargen	Returns a string of characters
20	FIP, Data	File Transfer Protocol (data connection)
21	FIP, Control	File Transfer Protocol (control connection)
23	TELNET	Tenninal Network
25	SMTP	Simple Mail Transfer Protocol
53	DNS	Domain Name Server
67	BOOTP	Bootstrap Protocol
79	Finger	Finger
80	HTTP	Hypertext Transfer Protocol
111	RPC	Remote Procedure Call

Stream Delivery Service

TCP, unlike UDP, is a stream-oriented protocol. In UDP, a process (an application program) sends messages, with predefined boundaries, to UDP for delivery. UDP adds its own header to each of these messages and delivers them 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 data across the Internet. This imaginary environment is depicted in Figure 23.13. The sending process produces (writes to) the stream of bytes, and the receiving process consumes (reads from) them.

Figure 23.13 *Stream delivery*

Sending and Receiving Buffers Because the sending and the receiving processes may not write or read data at the same speed, 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 I-byte locations as shown in Figure 23.14. 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 23.14 *Sending and receiving buffers*

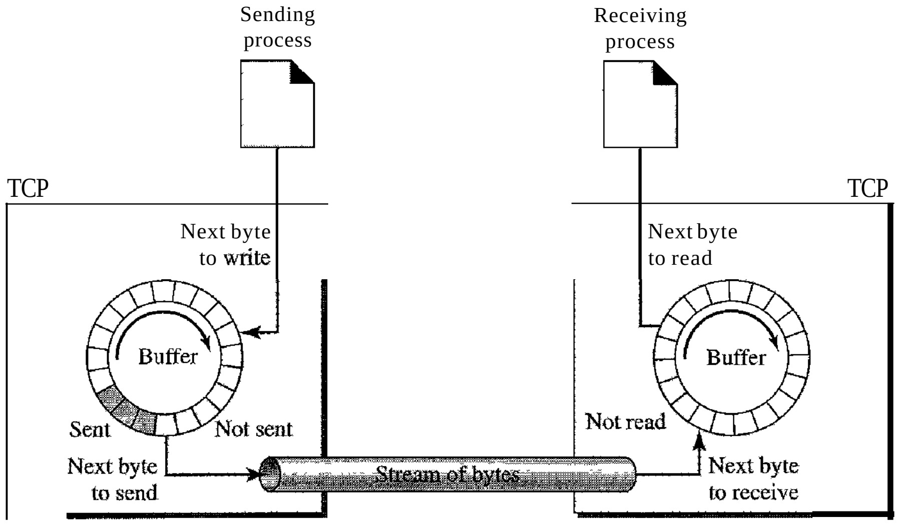
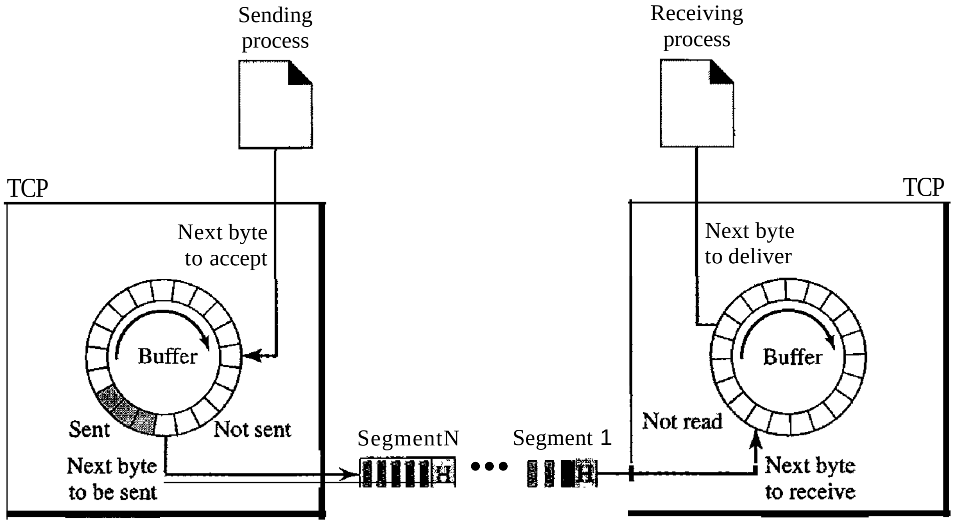


Figure 23.14 shows the movement of the data in one direction. At the sending site, the buffer has three types of chambers. The white section contains empty chambers that can be filled by the sending process (producer). The gray area holds bytes that have been sent but not yet acknowledged. TCP keeps these bytes in the buffer until it receives an acknowledgment. The colored 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 colored section. This could be due to the slowness of the receiving process or perhaps to congestion in the network. Also note that after the bytes in the gray 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 site 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 IP datagrams 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 these are handled by TCP with the receiving process unaware of any activities. Figure 23.15 shows how segments are created from the bytes in the buffers.

Figure 23.15 *TCP segments*



Note that the segments are not necessarily the same size. In Figure 23.15, 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.

Full-Duplex Communication

TCP offers full-duplex service, in which data can flow in both directions at the same time. Each TCP then has a sending and receiving buffer, and segments move in both directions.

Connection-Oriented Service

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

1. The two TCPs establish a 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 use 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. The situation is similar to creating a bridge that spans multiple islands and passing all the bytes from one island to another in one single connection. We will discuss this feature later in the chapter.

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.

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 the byte number and not the segment number.

Byte Number TCP numbers all data bytes that are transmitted in a connection. Numbering is independent in each direction. When TCP receives bytes of data from a process, it stores them in the sending buffer and numbers them. The numbering does not necessarily start from 0. Instead, TCP generates a random number between 0 and $2^{32} - 1$ for the number of the first byte. For example, if the random 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 a randomly 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 carried in that segment.

Example 23.3

Suppose a TCP connection is transferring a file of 5000 bytes. The first byte is numbered 1O,00l. 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,(00)
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 of 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 consumes one sequence number as though it carried 1 byte, but there are no actual data. If the randomly generated sequence number is x, the first data byte is numbered $x + 1$. The byte x is considered a phony byte that is used for a control segment to open a connection, as we will see shortly.

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 I 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 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 receiver of the data controls the amount of data that are to be sent by the sender. This is done to prevent the receiver from being overwhelmed with data. The numbering system allows TCP to use a byte-oriented flow control.

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 detennined by the level of congestion in the network.

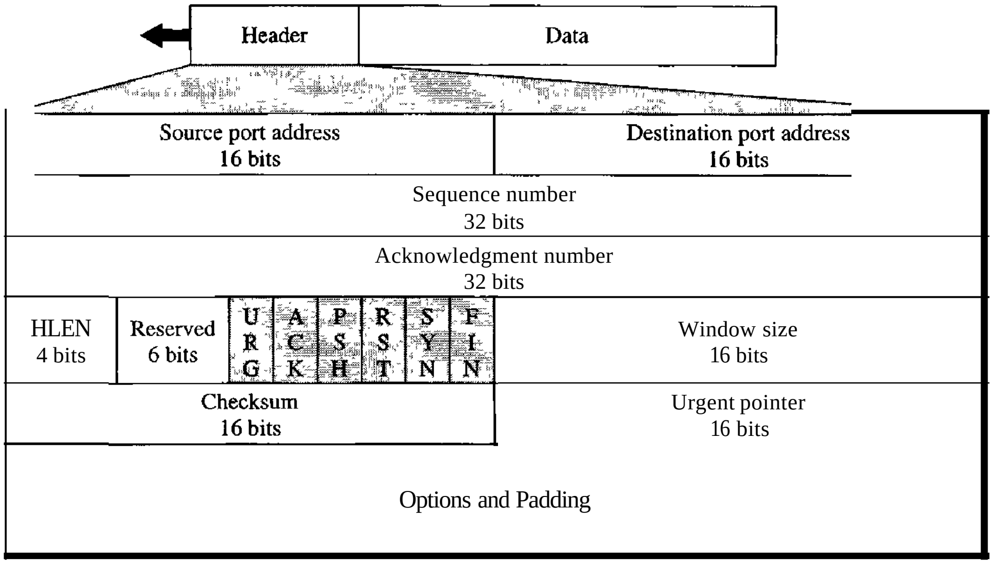
Segment

Before we discuss TCP in greater 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 23.16.

Figure 23.16 *TCP segment format*



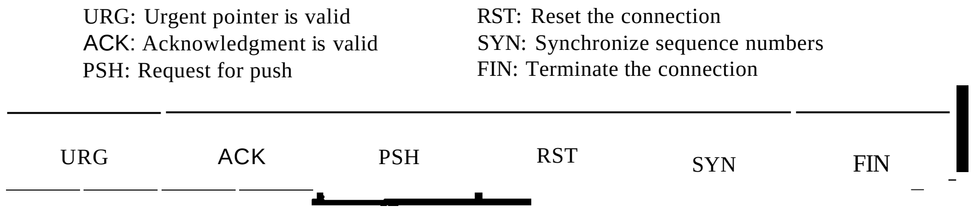
The segment consists of a 20- to 60-byte header, followed by data from the application program. The header is 20 bytes if there are no options and up to 60 bytes if it contains options. We will discuss some of the header fields in this section. The meaning and purpose of these will become clearer as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

- 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.
- 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.
- 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 comprises the first byte in the segment. During connection establishment, 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 defines x + I 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 can be between 5 (5 x 4 =20) and 15 (15 x 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 23.17. One or more of these bits can be set at a time.

Figure 23.17 *Control field*



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 Table 23.3. We will discuss them further when we study the detailed operation of TCP later in the chapter.

Table 23.3 *Description offlags in the controlfield*

Flag	*Description*
URG	The value of the urgent pointer field is valid.
ACK	The value of the acknowledgment field is valid.
PSH	Push the data.
RST	Reset the connection.
SYN	Synchronize sequence numbers during connection.
FIN	Terminate the connection.

- Window size. This field defines the size of the window, in bytes, that the other party must maintain. 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 inclusion of the checksum in the UDP datagram is optional, whereas the inclusion 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.

- 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 the number 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 not discuss these options here; please refer to the reference list for more information.

A TCP Connection

TCP is connection-oriented. A connection-oriented transport protocol establishes a virtual path between the source and destination. All 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

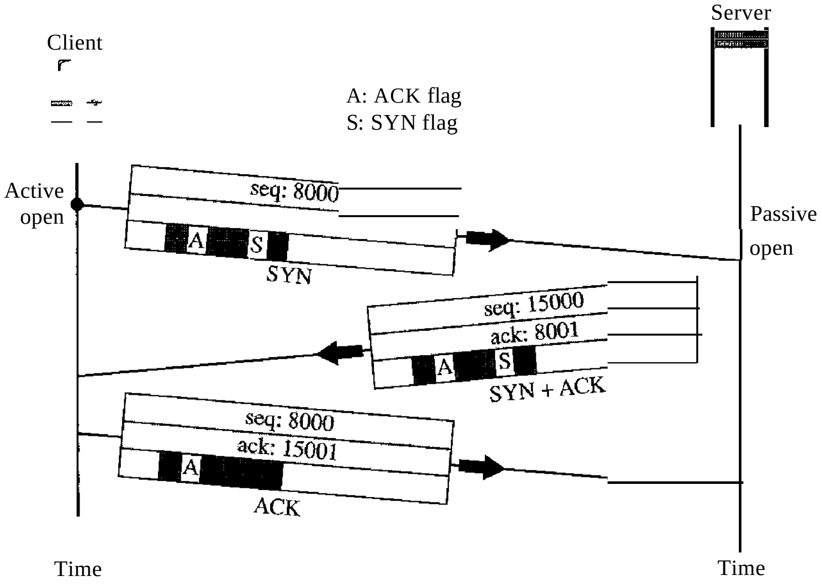
TCP transmits data in full-duplex mode. When two TCPs in two machines are connected, they are able to send segments to each other simultaneously. This implies that each party must initialize communication and get approval from the other party before any data are transferred.

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 is called a request for a *passive open.* Although the server TCP is ready to accept any 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 that it needs to be connected to that particular server. TCP can now start the three-way handshaking process as shown in Figure 23.18. To show the process, we use two time lines: one at each site. 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,

Figure 23.18 *Connection establishment using three-way handshaking*



the acknowledgment number, the control flags (only those that are set), and the window size, if not empty. 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. It consumes one sequence number. When the data transfer starts, the sequence number is incremented by 1. We can say that the SYN segment carries no real data, but we can think of it as containing 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 2 flag bits set: SYN and ACK. This segment has a dual purpose. It is a SYN segment for communication in the other direction and serves as the acknowledgment for the SYN segment. It consumes one sequence number.

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.

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

Simultaneous Open A rare situation, called a simultaneous open, 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.

SYN Flooding Attack The connection establishment procedure in TCP is susceptible to a serious security problem called the SYN flooding attack. This happens when a malicious attacker sends a large number of SYN segments to a server, pretending that each of them is corning 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 communication tables and setting timers. The TCP server then sends the SYN + ACK segments to the fake clients, which are lost. During this time, however, a lot of resources are occupied without being used. If, during this short time, the number of SYN segments is large, the server eventually runs out of resources and may crash. This SYN flooding attack belongs to a type of security attack known as a denial-of-service attack, in which an attacker monopolizes a system with so many service requests that the system collapses and denies service to every request.

Some implementations of TCP have strategies to alleviate the effects of a SYN attack. Some have imposed a limit on connection requests during a specified period of time. Others filter out datagrams coming from unwanted source addresses. One recent strategy is to postpone resource allocation until the entire connection is set up, using what is called a cookie. SCTP, the new transport layer protocol that we discuss in the next section, uses this strategy.

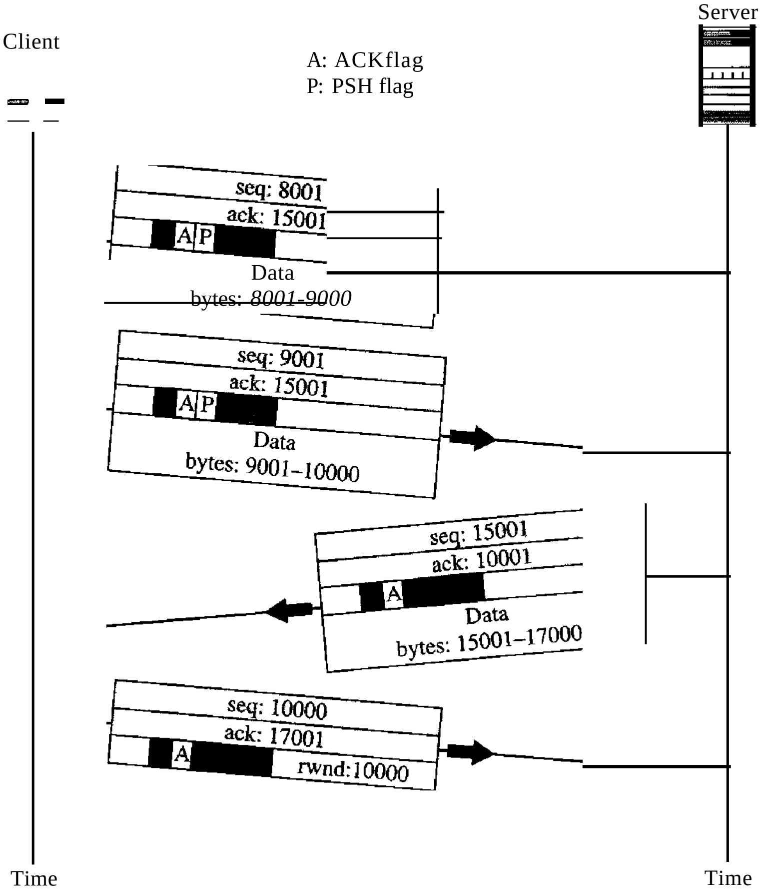
Data Transfer

After connection is established, bidirectional data transfer can take place. The client and server can both send data and acknowledgments. 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 23.19 shows an example. In this example, after connection is established (not shown in the figure), 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 are 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 greater 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, on occasion the application program has no need for this flexibility_ For example, consider an application program that communicates interactively with another

Figure 23.19 *Data transfer*



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 sending site 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 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 are presented from the application program to TCP as a stream of bytes. Each byte of data has a position in the stream. However, on occasion an application program needs to send *urgent* bytes. This means that the sending application program wants a piece of data to be read out of order by the receiving application program. As an example, suppose that the sending application program is sending data to be processed by the receiving application program. When the result of processing comes back, the sending application program finds that everything is wrong. It wants to abort the process, but it has already sent a huge amount of data. If it issues an abort command (control + C), these two characters will be stored at the end of the receiving TCP buffer. It will be delivered to the receiving application program after all the data have been processed.

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 and the start of normal data.

When the receiving TCP receives a segment with the URG bit set, it extracts the urgent data from the segment, using the value of the urgent pointer, and delivers them, out of order, to the receiving application program.

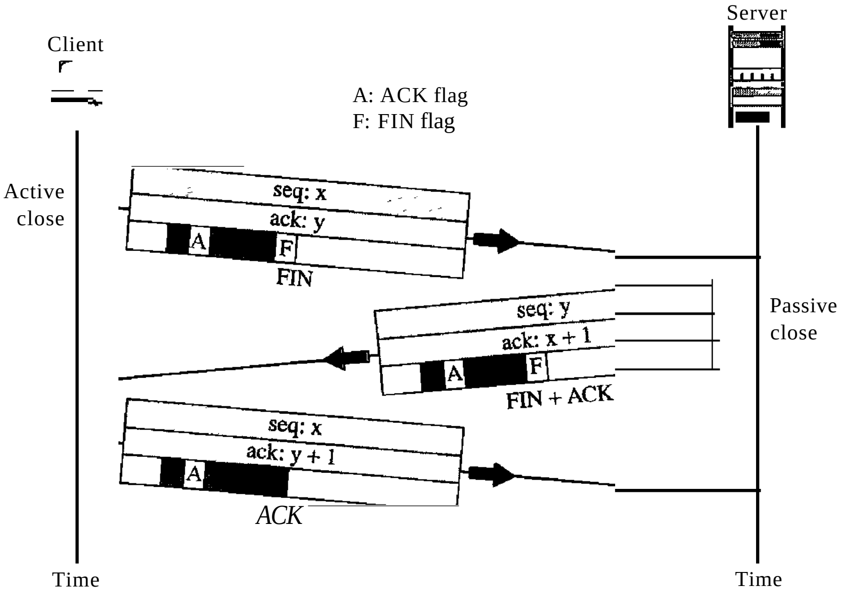
Connection Termination

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 23.20.

1. In a normal 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

Figure 23.20 *Connection termination using three-way handshaking*



can be just a control segment as shown in Figure 23.20. 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.

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.

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

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 1 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. Although either end can issue a half-close, it is normally initiated by the client. 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 the data, can close the connection in the outbound direction. However, the inbound direction must remain open to receive the sorted data. The server, after receiving the data, still needs time for sorting; its outbound direction must remain open.

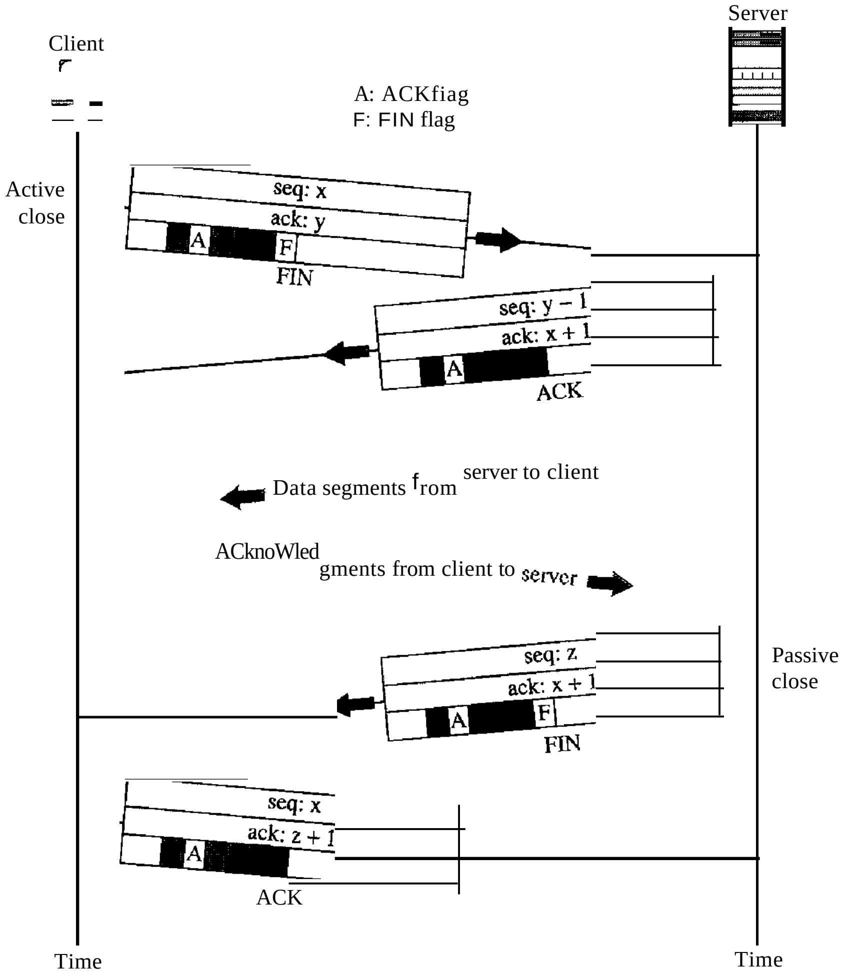
Figure 23.21 shows an example of a half-close. The client half-closes the connection by sending a FIN segment. The server accepts the half-close by sending the ACK segment. The data transfer from the client to the server stops. The server, however, can still send data. When the server has sent all the processed data, it sends a FIN segment, which is acknowledged by an ACK from the client.

After half-closing of 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.

Flow Control

TCP uses a sliding window, as discussed in Chapter 11, to handle flow control. The sliding window protocol used by TCP, however, is something between the *Go-Back-N* and Selective Repeat sliding window. The sliding window protocol in TCP looks like

Figure 23.21 *Half-close*



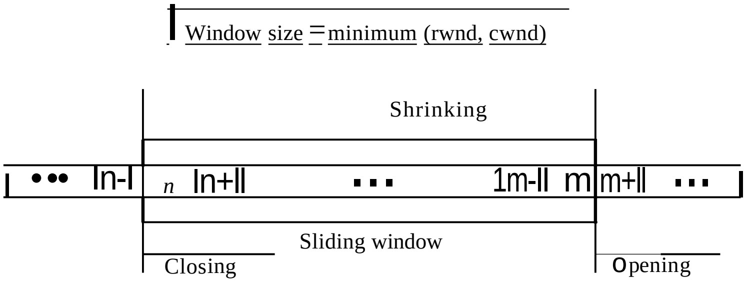
the Go-Back-N protocol because it does not use NAKs; it looks like Selective Repeat because the receiver holds the out-of-order segments until the missing ones arrive. There are two big differences between this sliding window and the one we used at the data link layer. First, the sliding window ofTCP is byte-oriented; the one we discussed in the data link layer is frame-oriented. Second, the TCP's sliding window is of variable size; the one we discussed in the data link layer was of fixed size.

Figure 23.22 shows the sliding window in TCP. The window spans a portion of the buffer containing bytes received from the process. The bytes inside the window are the bytes that can be in transit; they can be sent without worrying about acknowledgment. The imaginary window has two walls: one left and one right.

The window is *opened, closed,* or *shrunk.* These three activities, as we will see, are in the control of the receiver (and depend on congestion in the network), not the sender. The sender must obey the commands of the receiver in this matter.

Opening a window means moving the right wall to the right. This allows more new bytes in the buffer that are eligible for sending. Closing the window means moving the left wall to the right. This means that some bytes have been acknowledged and the sender

Figure 23.22 *Sliding window*



need not worry about them anymore. Shrinking the window means moving the right wall to the left. This is strongly discouraged and not allowed in some implementations because it means revoking the eligibility of some bytes for sending. This is a problem if the sender has already sent these bytes. Note that the left wall cannot move to the left because this would revoke some of the previously sent acknowledgments.

A sliding window is used to make transmission more efficient as weD as to control the flow of data so that the destination does not become overwhelmed with data. TCP sliding windows are byte-oriented.

The size of the window at one end is determined by the lesser of two values: *receiver window (rwnd)* or *congestion window (cwnd)*. The *receiver window* is the value advertised by the opposite end in a segment containing acknowledgment. It is the number of bytes the other end can accept before its buffer overflows and data are discarded. The congestion window is a value determined by the network to avoid congestion. We will discuss congestion later in the chapter.

Example 23.4

What is the value of the receiver window *(rwnd)* for host A if the receiver, host B, has a buffer size of 5000 bytes and 1000 bytes of received and unprocessed data?

Solution

The value of *rwnd* = 5000 - 1000 = 4000. Host B can receive only 4000 bytes of data before overflowing its buffer. Host B advertises this value in its next segment to A.

Example 23.5

What is the size of the window for host A if the value of *rwnd* is 3000 bytes and the value of *cwnd* is 3500 bytes?

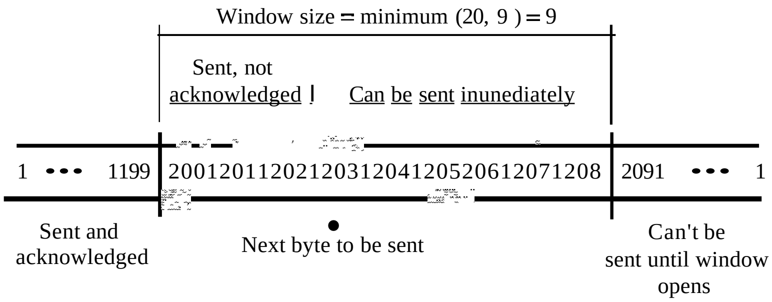
Solution

The size of the window is the smaller of *rwnd* and *cwnd*, which is 3000 bytes.

Example 23.6

Figure 23.23 shows an unrealistic example of a sliding window. The sender has sent bytes up to 202. We assume that cwnd is 20 (in reality this value is thousands of bytes). The receiver has sent

Figure 23.23 *Example 23.6*



an acknowledgment number of 200 with an *rwnd* of 9 bytes (in reality this value is thousands of bytes). The size of the sender window is the minimum of *rwnd* and *cwnd,* or 9 bytes. Bytes 200 to 202 are sent, but not acknowledged. Bytes 203 to 208 can be sent without worrying about acknowledgment. Bytes 209 and above cannot be sent.

Some points about TCP sliding windows:

- The size of the window is the lesser of *rwnd* and *cwnd.*
- The source does not have to send a full window's worth of data.
- The window can be opened or closed by the receiver, but should not be shrunk.
- The destination can send an acknowledgment at any time as long as it does not result in a shrinking window.
- The receiver can temporarily shut down the window; the sender, however, can always send a segment of 1 byte after the window is shut down.

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 corrupted segments, lost segments, out-of-order segments, and duplicated segments. Error control also includes a mechanism for correcting errors after they are detected. Error detection and correction 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 the segment is corrupted, it is discarded by the destination TCP and is considered as lost. TCP uses a 16-bit checksum that is mandatory in every segment. We will see, in Chapter 24, that the 16-bit checksum is considered inadequate for the new transport layer, SCTP. However, it cannot be changed for TCP because this would involve reconfiguration of the entire header format.

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.

Retransmission

The heart of the error control mechanism is the retransmission of segments. When a segment is corrupted, lost, or delayed, it is retransmitted. In modern implementations, a segment is retransmitted on two occasions: when a retransmission timer expires or when the sender receives three duplicate ACKs.

In modern implementations, a retransmission occurs if the retransmission timer expires or three duplicate ACK segments have arrived.

Note that no retransmission occurs for segments that do not consume sequence numbers. In particular, there is no transmission for an ACK segment.

No retransmission timer is set for an ACK segment.

Retransmission After RTO A recent implementation of TCP maintains one retransmission time-out (RTO) timer for all outstanding (sent, but not acknowledged) segments. When the timer matures, the earliest outstanding segment is retransmitted even though lack of a received ACK can be due to a delayed segment, a delayed ACK, or a lost acknowledgment. Note that no time-out timer is set for a segment that carries only an acknowledgment, which means that no such segment is resent. The value of RTO is dynamic in TCP and is updated based on the round-trip time (RTT) of segments. An RTI is the time needed for a segment to reach a destination and for an acknowledgment to be received. It uses a back-off strategy similar to one discussed in Chapter 12.

Retransmission After Three Duplicate ACK Segments The previous rule about retransmission of a segment is sufficient if the value of RTO is not very large. Sometimes, however, one segment is lost and the receiver receives so many out-of-order segments that they cannot be saved (limited buffer size). To alleviate this situation, most implementations today follow the three-duplicate-ACKs rule and retransmit the missing segment immediately. This feature is referred to as fast retransmission, which we will see in an example shortly.

Out-oj-Order Segments

When a segment is delayed, lost, or discarded, the segments following that segment arrive out of order. Originally, TCP was designed to discard all out-of-order segments, resulting in the retransmission of the missing segment and the following segments. Most implementations today do not discard the out-of-order segments. They store them temporarily and flag them as out-of-order segments until the missing segment arrives. Note, however, that the out-of-order segments are not delivered to the process. TCP guarantees that data are delivered to the process in order.

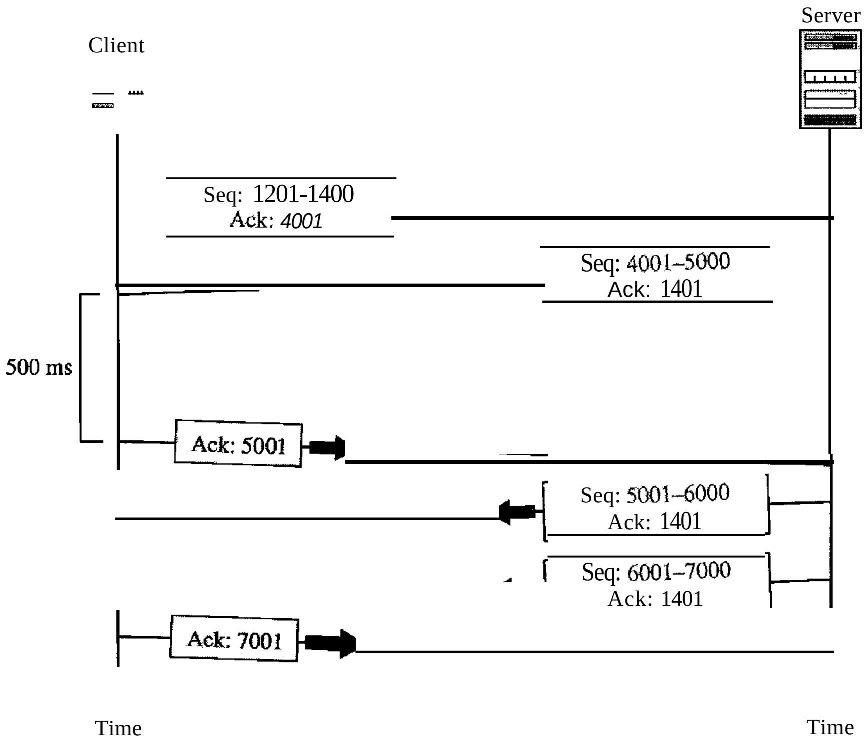
Data may arrive out of order and be temporarily stored by the receiving TCP, but yep guarantees that no out-of-order segment is delivered to the process.

Some Scenarios

In this section we give some examples of scenarios that occur during the operation of TCP. In these scenarios, we show a segment by a rectangle. If the segment carries data, we show the range of byte numbers and the value of the acknowledgment field. If it carries only an acknowledgment, we show only the acknowledgment number in a smaller box.

Normal Operation The first scenario shows bidirectional data transfer between two systems, as in Figure 23.24. The client TCP sends one segment; the server TCP sends three. The figure shows which rule applies to each acknowledgment. There are data to be sent, so the segment displays the next byte expected. When the client receives the first segment from the server, it does not have any more data to send; it sends only an

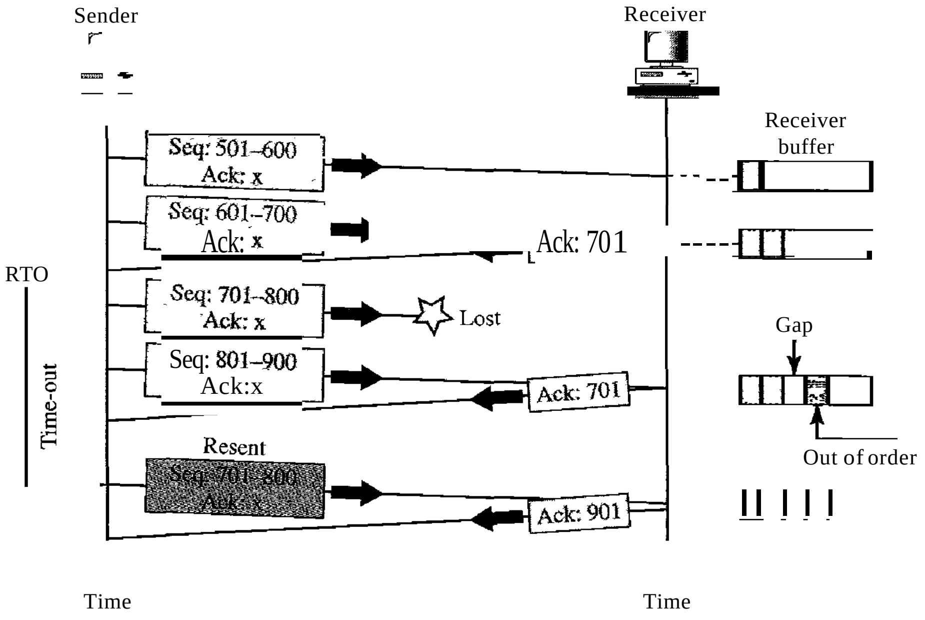
Figure 23.24 *Nonnaloperation*



ACK segment. However, the acknowledgment needs to be delayed for 500 ms to see if any more segments arrive. When the timer matures, it triggers an acknowledgment. This is so because the client has no knowledge if other segments are coming; it cannot delay the acknowledgment forever. When the next segment arrives, another acknowledgment timer is set. However, before it matures, the third segment arrives. The arrival of the third segment triggers another acknowledgment.

Lost Segment In this scenario, we show what happens when a segment is lost or corrupted. A lost segment and a corrupted segment are treated the same way by the receiver. A lost segment is discarded somewhere in the network; a corrupted segment is discarded by the receiver itself. Both are considered lost. Figure 23.25 shows a situation in which a segment is lost and discarded by some router in the network, perhaps due to congestion.

Figure 23.25 *Lost segment*



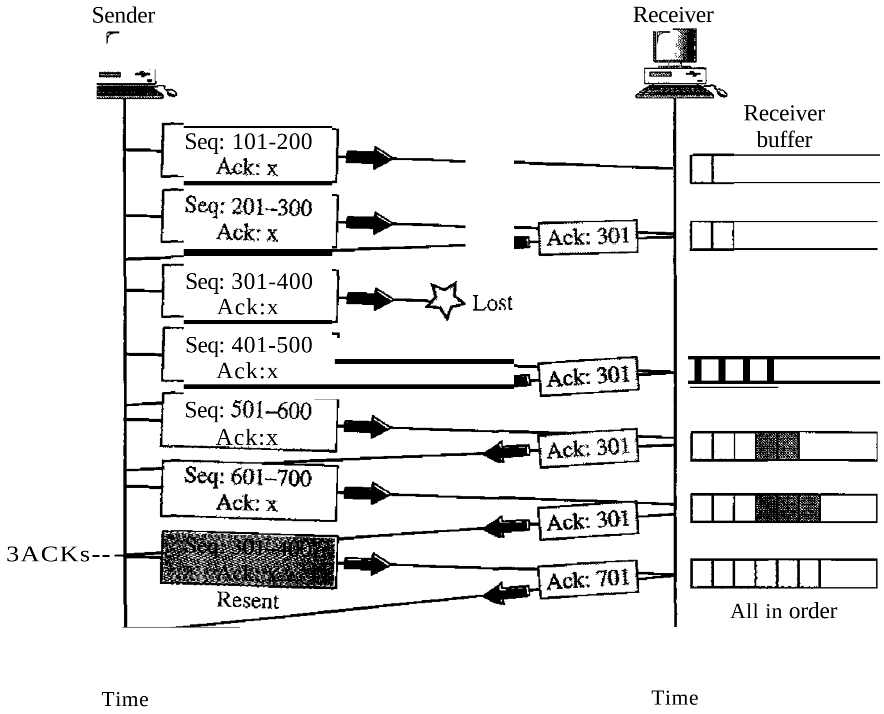
We are assuming that data transfer is unidirectional: one site is sending, the other is receiving. In our scenario, the sender sends segments 1 and 2, which are acknowledged immediately by an ACK. Segment 3, however, is lost. The receiver receives segment 4, which is out of order. The receiver stores the data in the segment in its buffer but leaves a gap to indicate that there is no continuity in the data. The receiver immediately sends an acknowledgment to the sender, displaying the next byte it expects. Note that the receiver stores bytes 801 to 900, but never delivers these bytes to the application until the gap is filled.

The receiver TCP delivers only ordered data to the process.

We have shown the timer for the earliest outstanding segment. The timer for this definitely runs out because the receiver never sends an acknowledgment for lost or out-of-order segments. When the timer matures, the sending TCP resends segment 3, which arrives this time and is acknowledged properly. Note that the value in the second and third acknowledgments differs according to the corresponding rule.

Fast Retransmission In this scenario, we want to show the idea of fast retransmission. Our scenario is the same as the second except that the RTO has a higher value (see Figure 23.26).

Figure 23.26 *Fast retransmission*



When the receiver receives the fourth, fifth, and sixth segments, it triggers an acknowledgment. The sender receives four acknowledgments with the same value (three duplicates). Although the timer for segment 3 has not matured yet, the fast transmission requires that segment 3, the segment that is expected by all these acknowledgments, be resent immediately.

Note that only one segment is retransmitted although four segments are not acknowledged. When the sender receives the retransmitted ACK, it knows that the four segments are safe and sound because acknowledgment is cumulative.

Congestion Control

We discuss congestion control of TCP in Chapter 24.

23.4 SCTP

Stream Control Transmission Protocol (SCTP) is a new reliable, message-oriented transport layer protocol. SCTP, however, is mostly designed for Internet applications that have recently been introduced. These new applications, such as IUA (ISDN over IP), M2UA and M3UA (telephony signaling), H.248 (media gateway control), H.323 (IP telephony), and SIP (IP telephony), need a more sophisticated service than TCP can provide. SCTP provides this enhanced performance and reliability. We briefly compare UDP, TCP, and SCTP:

- UDP is a message-oriented protocol. A process delivers a message to UDP, which is encapsulated in a user datagram and sent over the network. UDP *conserves the message boundaries;* each message is independent of any other message. This is a desirable feature when we are dealing with applications such as IP telephony and transmission of real-time data, as we will see later in the text. However, UDP is unreliable; the sender cannot know the destiny of messages sent. A message can be lost, duplicated, or received out of order. UDP also lacks some other features, such as congestion control and flow control, needed for a friendly transport layer protocol.
- TCP is a byte-oriented protocol. It receives a message or messages from a process, stores them as a stream of bytes, and sends them in segments. There is no preservation of the message boundaries. However, TCP is a reliable protocol. The duplicate segments are detected, the lost segments are resent, and the bytes are delivered to the end process in order. TCP also has congestion control and flow control mechanisms.
- SCTP combines the best features of UDP and TCP. SCTP is a reliable message-oriented protocol. It preserves the message boundaries and at the same time detects lost data, duplicate data, and out-of-order data. It also has congestion control and flow control mechanisms. Later we will see that SCTP has other innovative features unavailable in UDP and TCP.

SCTP is a *message-oriented, reliable* protocol that combines the best features of UDP and TCP.

SCTP Services

Before we discuss the operation of SCTP, let us explain the services offered by SCTP to the application layer processes.

Process-to-Process Communication

SCTP uses all well-known ports in the TCP space. Table 23.4 lists some extra port numbers used by SCTP.

Multiple Streams

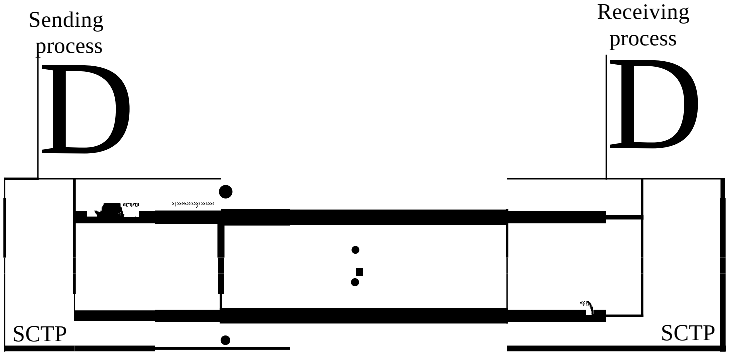
We learned in the previous section that TCP is a stream-oriented protocol. Each connection between a TCP client and a TCP server involves one single stream. The problem

Table 23.4 *Some SCTP applications*

Protocol	*Port Number*	*Description*
IVA	9990	ISDN overIP
M2UA	2904	SS7 telephony signaling
M3UA	2905	SS7 telephony signaling
H.248	2945	Media gateway control
H.323	1718,1719, 1720, 11720	IP telephony
SIP	5060	IP telephony

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. The idea is similar to multiple lanes on a highway. Each lane can be used for a different type of traffic. For example, one lane can be used for regular traffic, another for car pools. If the traffic is blocked for regular vehicles, car pool vehicles can still reach their destinations. Figure 23.27 shows the idea of multiple-stream delivery.

Figure 23.27 *Multiple-stream concept*

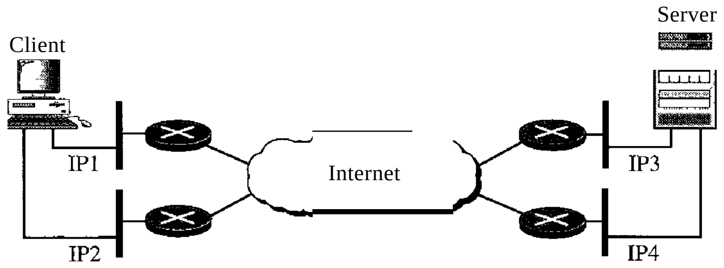


An association in SCTP can involve multiple streams.

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 23.28 shows the idea of multihoming.

Figure 23.28 *Multihoming concept*



In Figure 23.28, 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 IF 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.

SCTP association allows multiple IP addresses for each end.

Full-Duplex Communication

Like TCP, SCTP offers full-duplex service, in which 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. When a process at site A wants to send and receive data from another process at site B, the following occurs:

1. The two SCTPs establish an association between each other.
2. Data are exchanged in both directions.
3. The association is terminated.

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 the section on error control.

SCTP Features

Let us first discuss the general features of SCTP and then compare them with those ofTCP.

Transmission Sequence Number

The unit of data in TCP is a byte. Data transfer in TCP is controlled by numbering bytes by using a sequence number. On the other hand, the unit of data in SCTP is a DATA chunk which mayor may not have a one-to-one relationship with the message coming from the process because of fragmentation (discussed later), 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 to 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.

In **SCTP,** a data chunk is numbered using a TSN.

Stream Identifier

In TCP, there is only one stream in each connection. In SCTP, there may be several streams in each association. Each stream in SCTP needs to be identified by 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 51 is a 16-bit number starting from 0.

To distinguish between different streams, SCTP uses an SI.

Stream Sequence Number

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).

To distinguish between different data chunks belonging
to the same stream, SCTP uses SSNs.

Packets

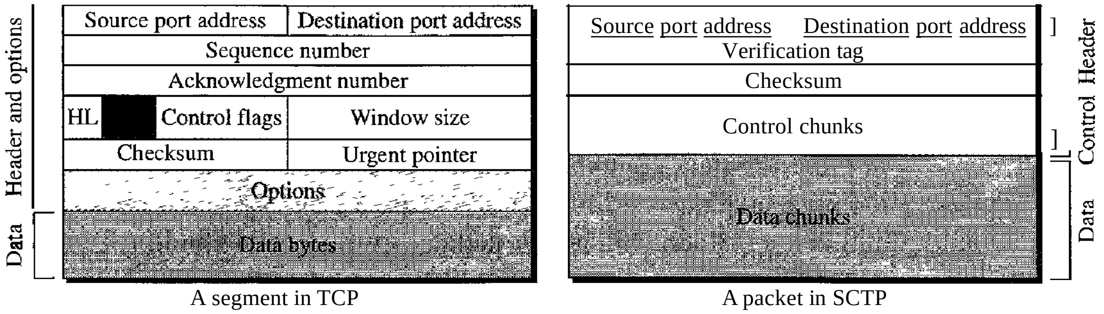
In TCP, a segment carries data and control information. Data are carried as a collection of bytes; control information is defined by six control flags in the header. The design of SCTP is totally different: data are carried as data chunks, control information is carried as control chunks. Several control 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 23.29 compares a segment in TCP and a packet in SCTP. Let us briefly list the differences between an SCTP packet and a TCP segment:

TCP has segments; SCTP has packets.

---

1. The control information in TCP is part of the header; the control information in SCTP is included in the control chunks. There are several types of control chunks; each is used for a different purpose.

Figure 23.29 *Comparison between a TCP segment and an SCTP packet*



2. The data in a TCP segment treated as one entity; an SCTP packet can carry several data chunks; each can belong to a different stream.
3. The options section, which can be part of a TCP segment, does not exist in an SCTP packet. Options in SCTP are handled by defining new chunk types.
4. The mandatory part of the TCP header is 20 bytes, while the general header in SCTP is only 12 bytes. The SCTP header is shorter due to the following:
   a. An SCTP sequence number (TSN) belongs to each data chunk and hence is located in the chunk's header.
   b. The acknowledgment number and window size are part of each control chunk.
   c. There is no need for a header length field (shown as HL in the TCP segment) because there are no options to make the length of the header variable; the SCTP header length is fixed (12 bytes).
   d. There is no need for an urgent pointer in SCTP.
5. The checksum in TCP is 16 bits; in SCTP, it is 32 bits.
6. The verification tag in SCTP is an association identifier, which does not exist in TCP. In TCP, the combination of IP and port addresses defines a connection; in SCTP we may have multihorning using different IP addresses. A unique verification tag is needed to define each association.
7. TCP includes one sequence number in the header, which defines the number of the first byte in the data section. An SCTP packet can include several different data chunks. TSNs, SIs, and SSNs define each data chunk.
8. Some segments in TCP that carry control information (such as SYN and FIN) need to consume one sequence number; control chunks in SCTP never use a TSN, SI, or SSN. These three identifiers belong only to data chunks, not to the whole packet.

In SCTP, control information and data information are carried in separate chunks.

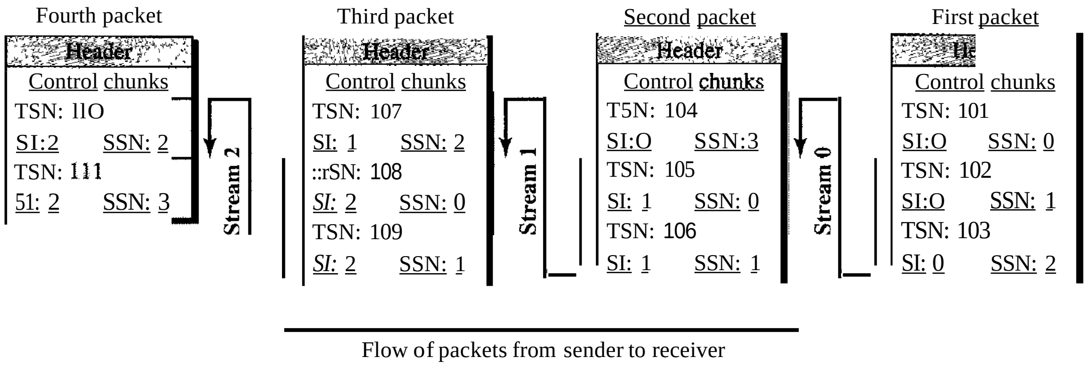
In SCTP, we have data chunks, streams, and packets. An association may send many packets, a packet may contain several chunks, and chunks may belong to different streams. To make the definitions of these terms clear, let us suppose that process A needs to send 11 messages to process B in three streams. 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 23.30. 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 packets; those in stream 2 are carried in the third and fourth packets.

Figure 23.30 *Packet, data chunks, and streams*



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.

Data chunks are identified by three items: TSN, SI, and SSN.
TSN is a cumulative number identifying the association;
SI defines the stream; SSN defines the chunk in a stream.

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.

> In **SCTP,** acknowledgment numbers are used to acknowledge only data chunks; control chunks are acknowledged by other control chunks if necessary.

Flow Control

Like TCP, SCTP implements flow control to avoid overwhelming the receiver. We will discuss SCTP flow control later in the chapter.

Error Control

Like TCP, SCTP implements error control to provide reliability. TSN numbers and acknowledgment numbers are used for error control. We will discuss error control later in the chapter.

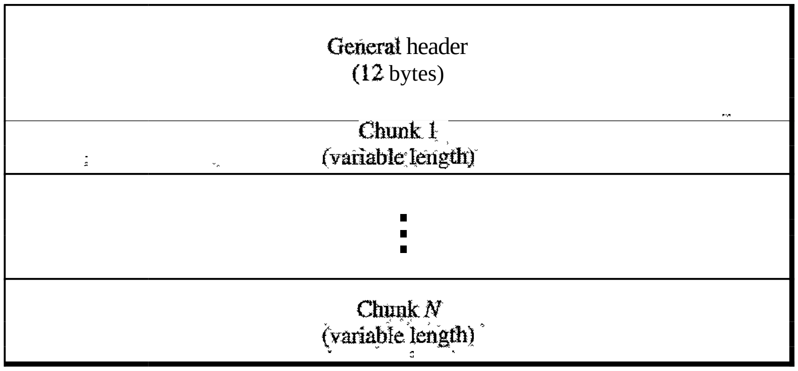
Congestion Control

Like TCP, SCTP implements congestion control to determine how many data chunks can be injected into the network. We will discuss congestion control in Chapter 24.

Packet Format

In this section, we show the format of a packet and different types of chunks. Most of the information presented in this section will become clear later; this section can be skipped in the first reading or used only as a reference. An SCTP packet has a mandatory general header and a set of blocks called chunks. There are two types of chunks: control chunks and data chunks. A control chunk controls and maintains the association; a data chunk carries user data. In a packet, the control chunks corne before the data chunks. Figure 23.31 shows the general format of an SCTP packet.

Figure 23.31 *SCTP packet format*



In an SCTP packet, control chunks come before data chunks.

General Header

The **general header** (packet header) defines the endpoints of each association to which the packet belongs, guarantees that the packet belongs to a particular association, and preserves the integrity of the contents of the packet including the header itself. The fonnat of the general header is shown in Figure 23.32.

Figure 23.32 *General header*

<table>
<tr><td>Source port address
16 bits</td><td>Destination port address
16 bits</td></tr>
<tr><td colspan="2">Verification tag
32 bits</td></tr>
<tr><td colspan="2">Checksum
32 bits</td></tr>
</table>

There are four fields in the general header:

- **Source port address.** This is a 16-bit field that defines the port number of the process sending the packet.
- **Destination port address.** This is a 16-bit field that defines the port number of the process receiving the packet.
- **Verification tag.** This is a number that matches a packet to an association. This prevents a packet from a previous association from being mistaken as a packet in this association. It serves as an identifier for the association; it is repeated in every packet during the association. There is a separate verification used for each direction in the association.
- **Checksum.** This 32-bit field contains a CRC-32 checksum. Note that the size of the checksum is increased from 16 (in UDP, TCP, and IP) to 32 bits to allow the use of the CRC-32 checksum.

Chunks

Control infonnation or user data are carried in chunks. The detailed fonnat of each chunk is beyond the scope of this book. See [For06] for details. The first three fields are common to all chunks; the information field depends on the type of chunk. The important point to remember is that SCTP requires the information section to be a multiple of 4 bytes; if not, padding bytes (eight as) are added at the end of the section. See Table 23.5 for a list of chunks and their descriptions.

An SCTP Association

SCTP, like TCP, is a connection-oriented protocol. However, a connection in SCTP is called an *association* to emphasize multihoming.

Table 23.5 *Chunks*

Type	*Chunk*	*Description*
0	DATA	User data
1	INIT	Sets up an association
2	INITACK	Acknowledges INIT chunk
3	SACK	Selective acknowledgment
4	HEARTBEAT	Probes the peer for liveliness
5	HEARTBEAT ACK	Acknowledges HEARTBEAT chunk
6	ABORT	Aborts an association
7	SHUTDOWN	Terminates an association
8	SHUTDOWN ACK	Acknowledges SHUTDOWN chunk
9	ERROR	Reports errors without shutting down
10	COOKIE ECHO	Third packet in association establishment
11	COOKIEACK	Acknowledges COOKIE ECHO chunk
14	SHUTDOWN COMPLETE	Third packet in association termination
192	FORWARDTSN	For adjusting cumulative TSN

A connection in SCTP is called an association.

Association Establishment

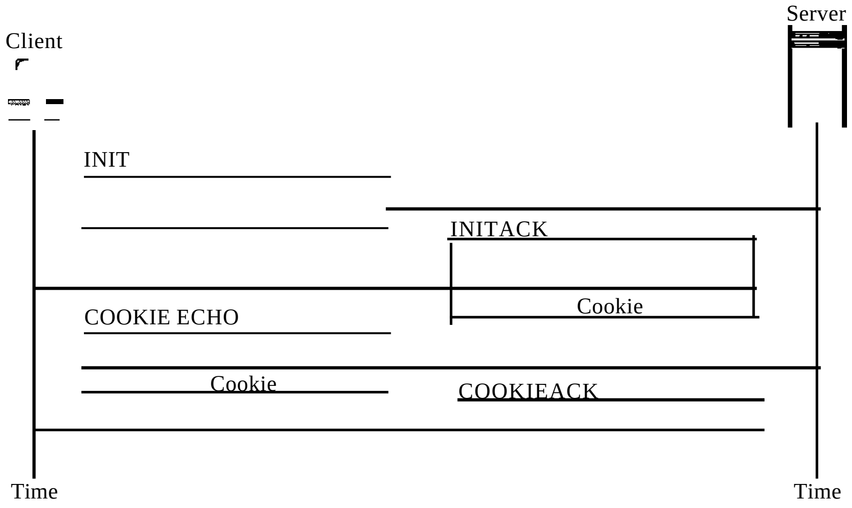
Association establishment in SCTP requires a four-way handshake. In this procedure, a process, normally a client, wants to establish an association with another process, normally a server, using SCTP as the transport layer protocol. Similar to TCP, the SCTP server needs to be prepared to receive any association (passive open). Association establishment, however, is initiated by the client (active open). SCTP association establishment is shown in Figure 23.33. The steps, in a normal situation, are as follows:

1. The client sends the first packet, which contains an INIT chunk.
2. The server sends the second packet, which contains an INIT ACK chunk.
3. The client sends the third packet, which includes a COOKIE ECHO chunk. This is a very simple chunk that echoes, without change, the cookie sent by the server. SCTP allows the inclusion of data chunks in this packet.
4. The server sends the fourth packet, which includes the COOKIE ACK chunk that acknowledges the receipt of the COOKIE ECHO chunk. SCTP allows the inclusion of data chunks with this packet.

No other chunk is allowed in a packet carrying an INIT or INIT ACK chunk.
A COOKIE ECHO or a COOKIE ACK chunk can carry data chunks.

Cookie We discussed a SYN flooding attack in the previous section. With TCP, a malicious attacker can flood a TCP server with a huge number of phony SYN segments using different forged IP addresses. Each time the server receives a SYN segment, it

Figure 23.33 *Four-way handshaking*



sets up a state table and allocates other resources while waiting for the next segment to arrive. After a while, however, the server may collapse due to the exhaustion of resources.

The designers of SCTP have a strategy to prevent this type of attack. The strategy is to postpone the allocation of resources until the reception of the third packet, when the IP address of the sender is verified. The information received in the first packet must somehow be saved until the third packet arrives. But if the server saved the information, that would require the allocation of resources (memory); this is the dilemma. The solution is to pack the information and send it back to the client. This is called generating a cookie. The cookie is sent with the second packet to the address received in the first packet. There are two potential situations.

1. If the sender of the first packet is an attacker, the server never receives the third packet; the cookie is lost and no resources are allocated. The only effort for the server is "baking" the cookie.
2. If the sender of the first packet is an honest client that needs to make a connection, it receives the second packet, with the cookie. It sends a packet (third in the series) with the cookie, with no changes. The server receives the third packet and knows that it has come from an honest client because the cookie that the sender has sent is there. The server can now allocate resources.

The above strategy works if no entity can "eat" a cookie "baked" by the server. To guarantee this, the server creates a digest (see Chapter 30) from the information, using its own secret key. The information and the digest together make the cookie, which is sent to the client in the second packet. When the cookie is returned in the third packet, the server calculates the digest from the information. If the digest matches the one that is sent, the cookie has not been changed by any other entity.

Data Transfer

The whole purpose of an association is to transfer data between two ends. After the association is established, bidirectional data transfer can take place. The client and the server can both send data. Like TCP, SCTP supports piggybacking.

There is a major difference, however, between data transfer in TCP and SCTP. TCP receives messages from a process as a stream of bytes without recognizing any boundary between them. The process may insert some boundaries for its peer use, but TCP treats that mark as part of the text. In other words, TCP takes each message and appends it to its buffer. A segment can carry parts of two different messages. The only ordering system imposed by TCP is the byte numbers.

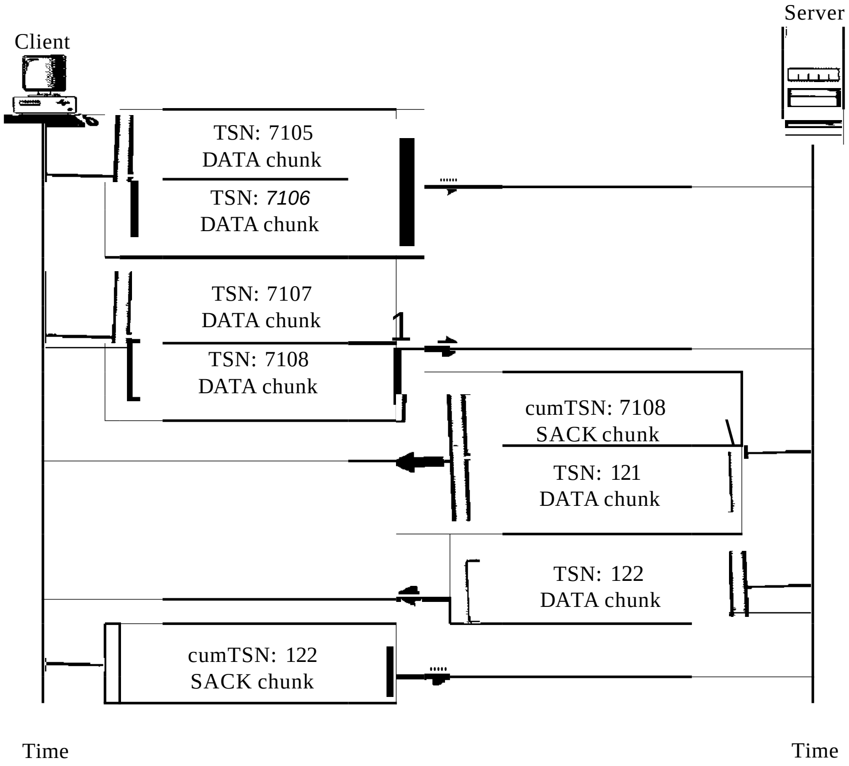
SCTP, on the other hand, recognizes and maintains boundaries. Each message coming from the process is treated as one unit and inserted into a DATA chunk unless it is fragmented (discussed later). In this sense, SCTP is like UDP, with one big advantage: data chunks are related to each other.

A message received from a process becomes a DATA chunk, or chunks if fragmented, by adding a DATA chunk header to the message. Each DATA chunk formed by a message or a fragment of a message has one TSN. We need to remember that only DATA chunks use TSNs and only DATA chunks are acknowledged by SACK chunks.

In SCTP, only DATA chunks consume TSNs;
DATA chunks are the only chunks that are acknowledged.

Let us show a simple scenario in Figure 23.34. In this figure a client sends four DATA chunks and receives two DATA chunks from the server. Later, we will discuss

Figure 23.34 *Simple data transfer*



the use of flow and error control in SCTP. For the moment, we assume that everything goes well in this scenario.

1. The client sends the first packet carrying two DATA chunks with TSNs 7105 and 7106.
2. The client sends the second packet carrying two DATA chunks with TSNs 7107 and 7108.
3. The third packet is from the server. It contains the SACK chunk needed to acknowledge the receipt of DATA chunks from the client. Contrary to TCP, SCTP acknowledges the last in-order TSN received, not the next expected. The third packet also includes the first DATA chunk from the server with TSN 121.
4. After a while, the server sends another packet carrying the last DATA chunk with TSN 122, but it does not include a SACK chunk in the packet because the last DATA chunk received from the client was already acknowledged.
5. Finally, the client sends a packet that contains a SACK chunk acknowledging the receipt of the last two DATA chunks from the server.

The acknowledgment in SCTP defines the cumulative TSN,
the TSN of the last data chunk received in order.

Multihoming Data Transfer We discussed the multihoming capability of SCTP, a feature that distinguishes SCTP from UDP and TCP. Multihoming allows both ends to define multiple IP addresses for communication. However, only one of these addresses can be defined as the primary address; the rest are alternative addresses. The primary address is defined during association establishment. The interesting point is that the primary address of an end is determined by the other end. In other words, a source defines the primary address for a destination.

Multistream Delivery One interesting feature of SCTP is the distinction between data transfer and data delivery. SCTP uses TSN numbers to handle data transfer, movement of data chunks between the source and destination. The delivery of the data chunks is controlled by SIs and SSNs. SCTP can support multiple streams, which means that the sender process can define different streams and a message can belong to one of these streams. Each stream is assigned a stream identifier (SI) which uniquely defines that stream.

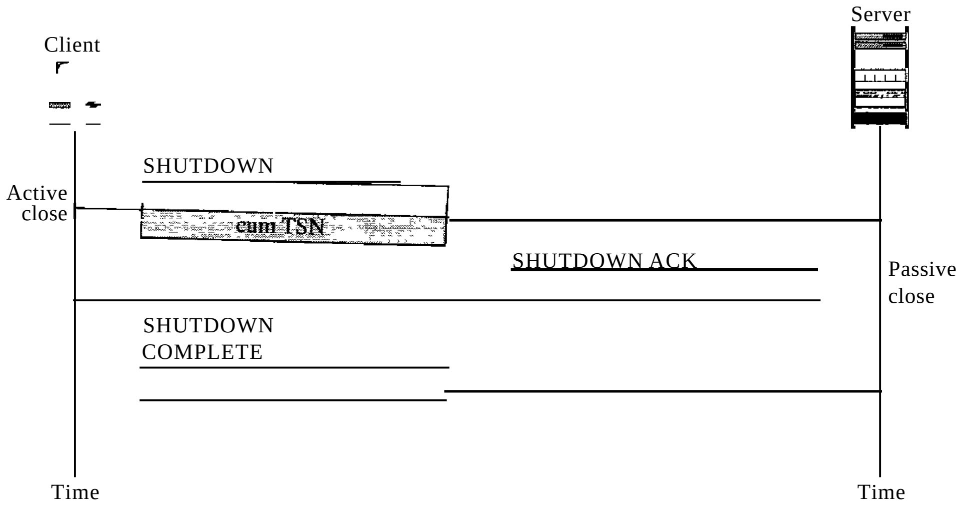
Fragmentation Another issue in data transfer is fragmentation. Although SCTP shares this term with IP, fragmentation in IP and in SCTP belongs to different levels: the former at the network layer, the latter at the transport layer.

SCTP preserves the boundaries of the message from process to process when creating a DATA chunk from a message if the size of the message (when encapsulated in an IP datagram) does not exceed the MTU of the path. The size of an IP datagram carrying a message can be determined by adding the size of the message, in bytes, to the four overheads: data chunk header, necessary SACK chunks, SCTP general header, and IP header. If the total size exceeds the MTU, the message needs to be fragmented.

Association Termination

In SCTP, like TCP, either of the two parties involved in exchanging data (client or server) can close the connection. However, unlike TCP, SCTP does not allow a half-close situation. If one end closes the association, the other end must stop sending new data. If any data are left over in the queue of the recipient of the termination request, they are sent and the association is closed. Association **termination** uses three packets, as shown in Figure 23.35. Note that although the figure shows the case in which termination is initiated by the client, it can also be initiated by the server. Note that there can be several scenarios of association termination. We leave this discussion to the references mentioned at the end of the chapter.

Figure 23.35 *Association termination*



Flow Control

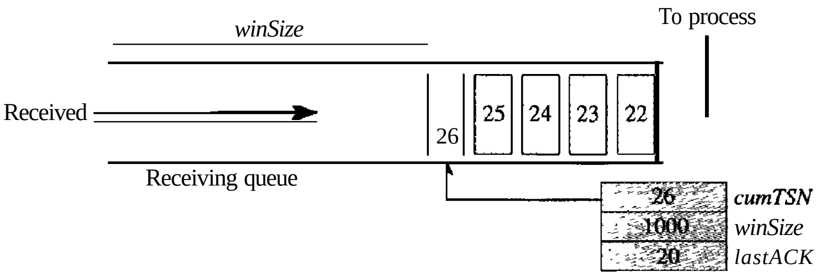
Flow control in SCTP is similar to that in TCP. In TCP, we need to deal with only one unit of data, the byte. In SCTP, we need to handle two units of data, the byte and the chunk. The values of *rwnd* and *cwnd* are expressed in bytes; the values of TSN and acknowledgments are expressed in chunks. To show the concept, we make some unrealistic assumptions. We assume that there is never congestion in the network and that the network is error-free. In other words, we assume that cwnd is infinite and no packet is lost or delayed or arrives out of order. We also assume that data transfer is unidirectional. We correct our unrealistic assumptions in later sections. Current SCTP implementations still use a byte-oriented window for flow control. We, however, show the buffer in terms of chunks to make the concept easier to understand.

Receiver Site

The receiver has one buffer (queue) and three variables. The queue holds the received data chunks that have not yet been read by the process. The first variable holds the last TSN received, *cumTSN.* The second variable holds the available buffer size, *winsize.*

The third variable holds the last accumulative acknowledgment, *lastACK.* Figure 23.36 shows the queue and variables at the receiver site.

Figure 23.36 *Flow control, receiver site*

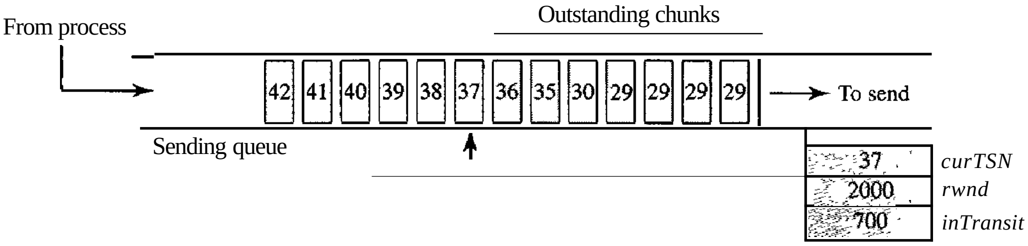


1. When the site receives a data chunk, it stores it at the end of the buffer (queue) and subtracts the size of the chunk from *winSize.* The TSN number of the chunk is stored in the *cumTSN* variable.
2. When the process reads a chunk, it removes it from the queue and adds the size of the removed chunk to *winSize* (recycling).
3. When the receiver decides to send a SACK, it checks the value of *lastAck;* if it is less than *cumTSN,* it sends a SACK with a cumulative TSN number equal to the *cumTSN.* It also includes the value of *winSize* as the advertised window size.

Sender Site

The sender has one buffer (queue) and three variables: *curTSN, rwnd,* and *inTransit,* as shown in Figure 23.37. We assume each chunk is 100 bytes long.

Figure 23.37 *Flow control, sender site*



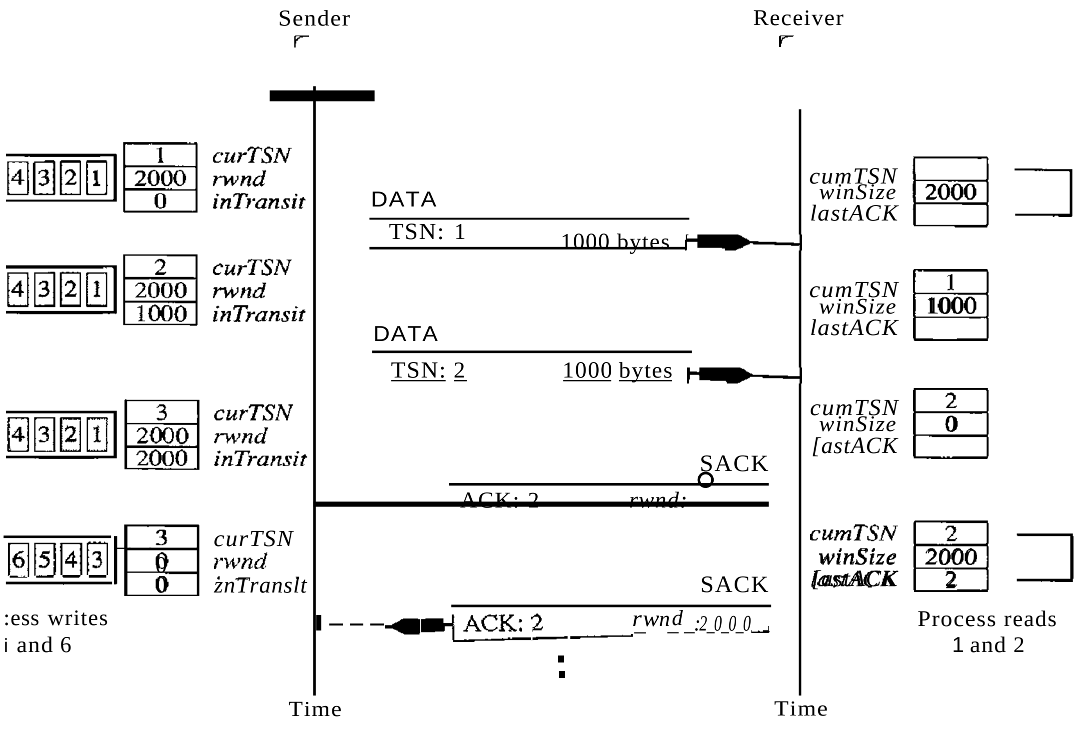
The buffer holds the chunks produced by the process that either have been sent or are ready to be sent. The first variable, *curTSN,* refers to the next chunk to be sent. All chunks in the queue with a TSN less than this value have been sent, but not acknowledged; they are outstanding. The second variable, *rwnd,* holds the last value advertised by the receiver (in bytes). The third variable, *inTransit,* holds the number of bytes in transit, bytes sent but not yet acknowledged. The following is the procedure used by the sender.

1. A chunk pointed to by *curTSN* can be sent if the size of the data is less than or equal to the quantity *rwnd* - *inTransit.* After sending the chunk, the value of *curTSN* is incremented by 1 and now points to the next chunk to be sent. The value of *inTransit* is incremented by the size of the data in the transmitted chunk.
2. When a SACK is received, the chunks with a TSN less than or equal to the cumulative TSN in the SACK are removed from the queue and discarded. The sender does not have to worry about them any more. The value of *inTransit* is reduced by the total size of the discarded chunks. The value of *rwnd* is updated with the value of the advertised window in the SACK.

A Scenario

Let us give a simple scenario as shown in Figure 23.38. At the start the value of *rwnd* at the sender site and the value of *winSize* at the receiver site are 2000 (advertised during association establishment). Originally, there are four messages in the sender queue. The sender sends one data chunk and adds the number of bytes (1000) to the *inTransit* variable. After awhile, the sender checks the difference between the *rwnd* and *inTransit,* which is 1000 bytes, so it can send another data chunk. Now the difference between the two variables is 0 and no more data chunks can be sent. After awhile, a SACK arrives that acknowledges data chunks 1 and 2. The two chunks are removed from the queue. The value of *inTransit* is now 0. The SACK, however, advertised a receiver window of value 0, which makes the sender.update *rwnd* to 0. Now the sender is blocked; it cannot send any data chunks (with one exception explained later).

Figure 23.38 *Flow control scenario*



At the receiver site, the queue is empty at the beginning. After the first data chunk is received, there is one message in the queue and the value of *cumTSN* is 1. The value of *winSize* is reduced to 1000 because the first message occupies 1000 bytes. After the second data chunk is received, the value of window size is a and *cumTSN* is 2. Now, as we will see, the receiver is required to send a SACK with cumulative TSN of 2. After the first SACK was sent, the process reads the two messages, which means that there is now room in the queue; the receiver advertises the situation with a SACK to allow the sender to send more data chunks. The remaining events are not shown in the figure.

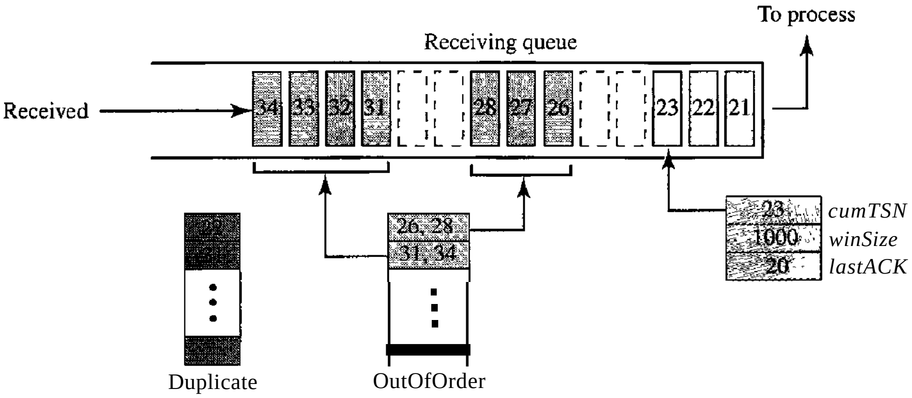
Error Control

SCTP, like TCP, is a reliable transport layer protocol. It uses a SACK chunk to report the state of the receiver buffer to the sender. Each implementation uses a different set of entities and timers for the receiver and sender sites. We use a very simple design to convey the concept to the reader.

Receiver Site

In our design, the receiver stores all chunks that have arrived in its queue including the out-of-order ones. However, it leaves spaces for any missing chunks. It discards duplicate messages, but keeps track of them for reports to the sender. Figure 23.39 shows a typical design for the receiver site and the state of the receiving queue at a particular point in time.

Figure 23.39 *Error control, receiver site*

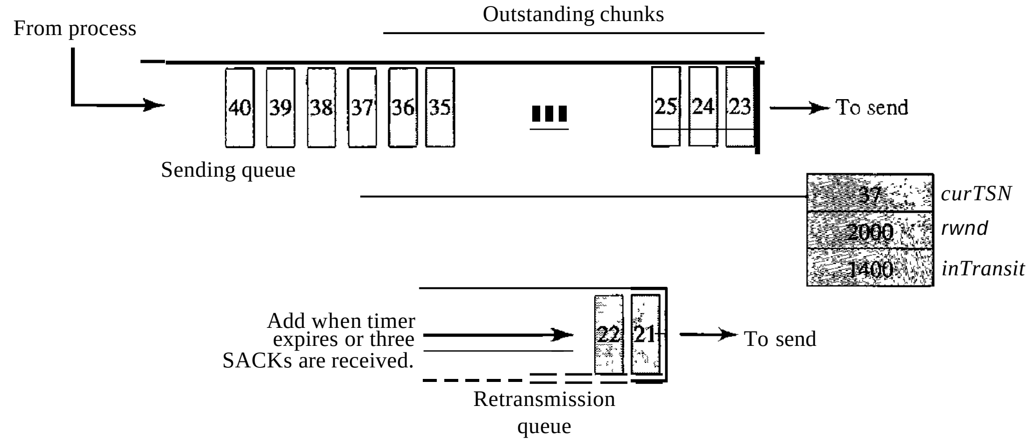


The last acknowledgment sent was for data chunk 20. The available window size is 1000 bytes. Chunks 21 to 23 have been received in order. The first out-of-order block contains chunks 26 to 28. The second out-of-order block contains chunks 31 to 34. A variable holds the value of *cumTSN*. An array of variables keeps track of the beginning and the end of each block that is out of order. An array of variables holds the duplicate chunks received. Note that there is no need for storing duplicate chunks in the queue; they will be discarded. The figure also shows the SACK chunk that will be sent to report the state of the receiver to the sender. The TSN numbers for out-of-order chunks are relative (offsets) to the cumulative TSN.

Sender Site

At the sender site, our design demands two buffers (queues): a sending queue and a retransmission queue. We also use the three variables *rwnd, inTransit,* and *curTSN* as described in the previous section. Figure 23.40 shows a typical design.

Figure 23.40 *Error control, sender site*



The sending queue holds chunks 23 to 40. The chunks 23 to 36 have already been sent, but not acknowledged; they are outstanding chunks. The *curTSN* points to the next chunk to be sent (37). We assume that each chunk is 100 bytes, which means that 1400 bytes of data (chunks 23 to 36) is in transit. The sender at this moment has a retransmission queue. When a packet is sent, a retransmission timer starts for that packet (all data chunks in that packet). Some implementations use one single timer for the entire association, but we continue with our tradition of one timer for each packet for simplification. When the retransmission timer for a packet expires, or four duplicate SACKs arrive that declare a packet as missing (fast retransmission was discussed in Chapter 12), the chunks in that packet are moved to the retransmission queue to be resent. These chunks are considered lost, rather than outstanding. The chunks in the retransmission queue have priority. In other words, the next time the sender sends a chunk, it would be chunk 21 from the retransmission queue.

Sending Data Chunks

An end can send a data packet whenever there are data chunks in the sending queue with a TSN greater than or equal to *curTSN* or if there are data chunks in the retransmission queue. The retransmission queue has priority. However, the total size of the data chunk or chunks included in the packet must not exceed *rwnd - inTransit,* and the total size of the frame must not exceed the MTU size as we discussed in previous sections.

Retransmission To control a lost or discarded chunk, SCTP, like TCP, employs two strategies: using retransmission timers and receiving four SACKs with the same missing chunks.

Generating SACK Chunks

Another issue in error control is the generation of SACK chunks. The rules for generating SCTP SACK chunks are similar to the rules used for acknowledgment with the TCP ACK flag.

Congestion Control

SCTP, like TCP, is a transport layer protocol with packets subject to congestion in the network. The SCTP designers have used the same strategies we will describe for congestion control in Chapter 24 for TCP. SCTP has slow start (exponential increase), congestion avoidance (additive increase), and congestion detection (multiplicative decrease) phases. Like TCP, SCTP also uses fast retransmission and fast recovery.

23.5 RECOMMENDED READING

For more details about subjects discussed in this chapter, we recommend the following books and sites. The items in brackets [...] refer to the reference list at the end of the text.

Books

UDP is discussed in Chapter 11 of [For06], Chapter 11 of [Ste94], and Chapter 12 of [ComOO]. TCP is discussed in Chapter 12 of [For06], Chapters 17 to 24 of [Ste94], and Chapter 13 of [ComOO]. SCTP is discussed in Chapter 13 of [For06] and [SX02]. Both UDP and TCP are discussed in Chapter 6 of [Tan03].

Sites

- www.ietf.org/rfc.html Information about RFCs

RFCs

A discussion of UDP can be found in RFC 768.
A discussion ofTCP can be found in the following RFCs:

675,700,721,761,793,879,896,1078,1106,1110,1144, 1145, 1146, 1263,1323,1337,
1379,1644,1693,1901,1905,2001,2018,2488,2580

A discussion of SCTP can be found in the following RFCs:

2960,3257,3284,3285,3286,3309,3436,3554,3708,3758

23.6 KEY TERMS

acknowledgment number
association
association establishment
association termination
byte-oriented
chunk
client
client/server paradigm
connection abortion
connection-oriented service
connectionless service
connectionless, unreliable transport protocol
cookie
COOKIE ACK chunk
COOKIE ECHO chunk
cumulative TSN
DATA chunk
data transfer
denial-of-service attack
dynamic port
ephemeral port number
error control
fast retransmission
flow control
four-way handshaking
fragmentation
full-duplex service
general header
half-close
inbound stream
INIT ACK chunk
INIT chunk
initial sequence number (ISN)
message-oriented
multihoming service
multistream service
port number
primary address
process-to-process delivery
pseudoheader
queue
registered port
retransmission time-out (RTO)
retransmission timer
round-trip time (RTT)
SACK chunk
segment
sequence number
server
simultaneous close
simultaneous open
socket address
stream identifier (SI)
stream sequence number (SSN)
SYN flooding attack
three-way handshaking
Transmission Control Protocol (TCP)
transmission sequence number (TSN)
transport layer
user datagram
User Datagram Protocol (UDP)
verification tag
well-known port number

23.7 SUMMARY

- In the client/server paradigm, an application program on the local host, called the client, needs services from an application program on the remote host, called a server.

- Each application program has a port number that distinguishes it from other programs running at the same time on the same machine.
- The client program is assigned a random port number called an ephemeral port number; the server program is assigned a universal port number called a well-known port number.
- The ICANN has specified ranges for the different types of port numbers.
- The combination of the IP address and the port number, called the socket address, defines a process and a host.
- UDP is a connectionless, unreliable transport layer protocol with no embedded flow or error control mechanism except the checksum for error detection.
- The UDP packet is called a user datagram. A user datagram is encapsulated in the data field of an IP datagram.
- Transmission Control Protocol (TCP) is one of the transport layer protocols in the TCP/IP protocol suite.
- TCP provides process-to-process, full-duplex, and connection-oriented service.
- The unit of data transfer between two devices using TCP software is called a segment; it has 20 to 60 bytes of header, followed by data from the application program.
- A TCP connection normally consists of three phases: connection establishment, data transfer, and connection termination.
- Connection establishment requires three-way handshaking; connection termination requires three- or four-way handshaking.
- TCP uses flow control, implemented as a sliding window mechanism, to avoid overwhelming a receiver with data.
- The TCP window size is determined by the receiver-advertised window size *(rwnd)* or the congestion window size *(cwnd),* whichever is smaller. The window can be opened or closed by the receiver, but should not be shrunk.
- The bytes of data being transferred in each connection are numbered by TCP. The numbering starts with a randomly generated number.
- TCP uses error control to provide a reliable service. Error control is handled by the checksum, acknowledgment, and time-out. Corrupted and lost segments are retransmitted, and duplicate segments are discarded. Data may arrive out of order and are temporarily stored by the receiving TCP, but TCP guarantees that no out-of-order segment is delivered to the process.
- In modem implementations, a retransmission occurs if the retransmission timer expires or three duplicate ACK segments have arrived.
- SCTP is a message-oriented, reliable protocol that combines the good features of UDP and TCP.
- SCTP provides additional services not provided by UDP or Tep, such as multiple-stream and multihoming services.
- SCTP is a connection-oriented protocol. An SCTP connection is called an association.
- SCTP uses the term *packet* to define a transportation unit.
- In SCTP, control information and data information are carried in separate chunks.

- An SCTP packet can contain control chunks and data chunks with control chunks coming before data chunks.
- In SCTP, each data chunk is numbered using a transmission sequence number (TSN).
- To distinguish between different streams, SCTP uses the sequence identifier (SI).
- To distinguish between different data chunks belonging to the same stream, SCTP uses the stream sequence number (SSN).
- Data chunks are identified by three identifiers: TSN, SI, and SSN. TSN is a cumulative number recognized by the whole association; SSN starts from 0 in each stream.
- SCTP acknowledgment numbers are used only to acknowledge data chunks; control chunks are acknowledged, if needed, by another control chunk.
- An SCTP association is normally established using four packets (four-way handshaking). An association is normally terminated using three packets (three-way handshaking).
- An SCTP association uses a cookie to prevent blind flooding attacks and a verification tag to avoid insertion attacks.
- SCTP provides flow control, error control, and congestion control.
- The SCTP acknowledgment SACK reports the cumulative TSN, the TSN of the last data chunk received in order, and selective TSNs that have been received.

23.8 PRACTICE SET

Review Questions

1. In cases where reliability is not of primary importance, UDP would make a good transport protocol. Give examples of specific cases.
2. Are both UDP and IP unreliable to the same degree? Why or why not?
3. Do port addresses need to be unique? Why or why not? Why are port addresses shorter than IP addresses?
4. What is the dictionary definition of the word *ephemeral?* How does it apply to the concept of the ephemeral port number?
5. What is the minimum size of a UDP datagram?
6. What is the maximum size of a UDP datagram?
7. What is the minimum size of the process data that can be encapsulated in a UDP datagram?
8. What is the maximum size of the process data that can be encapsulated in a UDP datagram?
9. Compare the TCP header and the UDP header. List the fields in the TCP header that are missing from UDP header. Give the reason for their absence.
10. UDP is a message-oriented protocol. TCP is a byte-oriented protocol. If an application needs to protect the boundaries of its message, which protocol should be used, UDP or TCP?

11. What can you say about the TCP segment in which the value of the control field is one of the following?
    a. 000000
    b. 000001
    c. 010001
12. What is the maximum size of the TCP header? What is the minimum size of the TCP header?

Exercises

13. Show the entries for the header of a UDP user datagram that carries a message from a TFTP client to a TFTP server. Fill the checksum field with Os. Choose an appropriate ephemeral port number and the correct well-known port number. The length of data is 40 bytes. Show the UDP packet, using the format in Figure 23.9.
14. An SNMP client residing on a host with IP address 122.45.12.7 sends a message to an SNMP server residing on a host with IP address 200.112.45.90. What is the pair of sockets used in this communication?
15. A TFTP server residing on a host with IP address 130.45.12.7 sends a message to a TFTP client residing on a host with IP address 14.90.90.33. What is the pair of sockets used in this communication?
16. A client has a packet of 68,000 bytes. Show how this packet can be transferred by using only one UDP user datagram.
17. A client uses UDP to send data to a server. The data are 16 bytes. Calculate the efficiency of this transmission at the UDP level (ratio of useful bytes to total bytes).
18. Redo Exercise 17, calculating the efficiency of transmission at the IP leveL Assume no options for the IP header.
19. Redo Exercise 18, calculating the efficiency of transmission at the data link layer. Assume no options for the IP header and use Ethernet at the data link layer.
20. The following is a dump of a UDP header in hexadecimal format.

```
0632000DOO lCE217
```

    a. What is the source port number?
    b. What is the destination port number?
    c. What is the total length of the user datagram?
    d. What is the length of the data?
    e. Is the packet directed from a client to a server or vice versa?
    f. What is the client process?
21. An IP datagram is carrying a TCP segment destined for address 130.14.16.17/16. The destination port address is corrupted, and it arrives at destination 130.14.16.19/16. How does the receiving TCP react to this error?
22. In TCP, if the value of HLEN is 0111, how many bytes of option are included in the segment?

23. Show the entries for the header of a TCP segment that carries a message from an FTP client to an FTP server. Fill the checksum field with 0s. Choose an appropriate ephemeral port number and the correct well-known port number. The length of the data is 40 bytes.
24. The following is a dump of a TCP header in hexadecimal format.

```
05320017 00000001 00000000 5OO207FF 00000000
```

a. What is the source port number?
b. What is the destination port number?
c. What the sequence number?
d. What is the acknowledgment number?
e. What is the length of the header?
f. What is the type of the segment?
g. What is the window size?

25. To make the initial sequence number a random number, most systems start the counter at 1 during bootstrap and increment the counter by 64,000 every 0.5 s. How long does it take for the counter to wrap around?
26. In a connection, the value of *cwnd* is 3000 and the value of *rwnd* is 5000. The host has sent 2000 bytes which has not been acknowledged. How many more bytes can be sent?
27. TCP opens a connection using an initial sequence number (ISN) of 14,534. The other party opens the connection with an ISN of 21,732. Show the three TCP segments during the connection establishment.
28. A client uses TCP to send data to a server. The data are 16 bytes. Calculate the efficiency of this transmission at the TCP level (ratio of useful bytes to total bytes). Calculate the efficiency of transmission at the IP level. Assume no options for the IP header. Calculate the efficiency of transmission at the data link layer. Assume no options for the IF header and use Ethernet at the data link layer.
29. TCP is sending data at 1 Mbyte/s. If the sequence number starts with 7000, how long does it take before the sequence number goes back to zero?
30. A TCP connection is using a window size of 10,000 bytes, and the previous acknowledgment number was 22,001. It receives a segment with acknowledgment number 24,001 and window size advertisement of 12,000. Draw a diagram to show the situation of the window before and after.
31. A window holds bytes 2001 to 5000. The next byte to be sent is 3001. Draw a figure to show the situation of the window after the following two events.
    a. An ACK segment with the acknowledgment number 2500 and window size advertisement 4000 is received.
    b. A segment carrying 1000 bytes is sent.
32. In SCTP, the value of the cumulative TSN in a SACK is 23. The value of the previous cumulative TSN in the SACK was 29. What is the problem?
33. In SCTP, the state of a receiver is as follows:
    a. The receiving queue has chunks 1 to 8, 11 to 14, and 16 to 20.
    b. There are 1800 bytes of space in the queue.

   c. The value of *lastAck* is 4.
   d. No duplicate chunk has been received.
   e. The value of *cumTSN* is 5.

   Show the contents of the receiving queue and the variables.

34. In SCTP, the state of a sender is as follows:
    a. The sending queue has chunks 18 to 23.
    b. The value of *cumTSN* is 20.
    c. The value of the window size is 2000 bytes.
    d. The value of *inTransit* is 200.

    If each data chunk contains 100 bytes of data, how many DATA chunks can be sent now? What is the next DATA chunk to be sent?

Research Activities

35. Find more information about ICANN. What was it called before its name was changed?
36. TCP uses a transition state diagram to handle sending and receiving segments. Find out about this diagram and how it handles flow and control.
37. SCTP uses a transition state diagram to handle sending and receiving segments. Find out about this diagram and how it handles flow and control.
38. What is the half-open case in TCP?
39. What is the half-duplex close case in TCP?
40. The *tcpdump* command in UNIX or LINUX can be used to print the headers of packets of a network interface. Use *tcpdump* to see the segments sent and received.
41. In SCTP, find out what happens if a SACK chunk is delayed or lost.
42. Find the name and functions of timers used in TCP.
43. Find the name and functions of timers used in SCTP.
44. Find out more about ECN in SCTP. Find the format of these two chunks.
45. Some application programs, such as FTP, need more than one connection when using TCP. Find how the multistream service of SCTP can help these applications establish only one association with several streams.