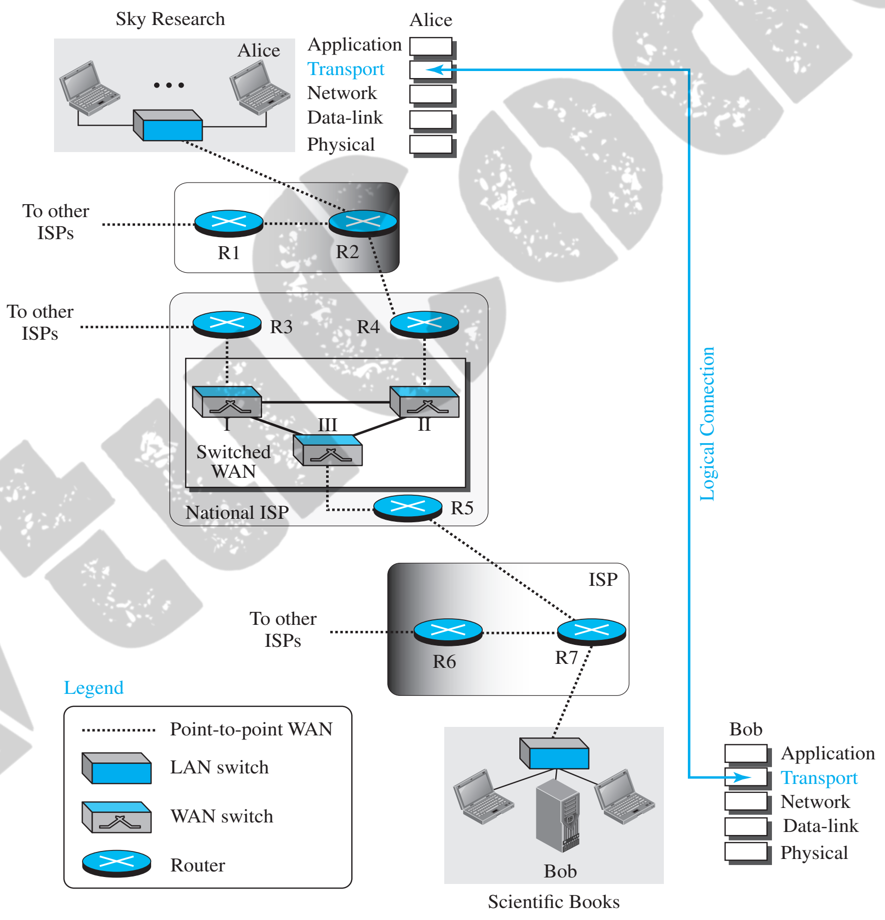
MODULE 4

23.1 INTRODUCTION

The transport layer is located between the application layer and the network layer. It provides a process-to-process communication between two application layers, one at the local host and the other at the remote host. Communication is provided using a logical connection, which means that the two application layers, which can be located in different parts of the globe, assume that there is an imaginary direct connection through which they can send and receive messages. Figure 23.1 shows the idea behind this logical connection.

Figure 23.1 *Logical connection at the transport layer*



The figure shows the same scenario we used in Chapter 3 for the physical layer (Figure 3.1). Alice's host in the Sky Research company creates a logical connection with Bob's host in the Scientific Books company at the transport layer. The two companies communicate at the transport layer as though there is a real connection between them. Figure 23.1 shows that only the two end systems (Alice's and Bob's computers) use the services of the transport layer; all intermediate routers use only the first three layers.

23.1.1 Transport-Layer Services

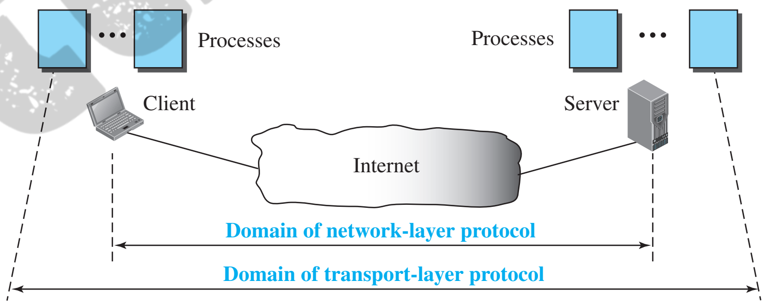
As we discussed in Chapter 2, the transport layer is located between the network layer and the application layer. The transport layer is responsible for providing services to the application layer; it receives services from the network layer. In this section, we discuss the services that can be provided by the transport layer; in the next section, we discuss several transport-layer protocols.

Process-to-Process Communication

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

The network layer (discussed in Chapters 18 to 22) is responsible for communication at the computer level (host-to-host communication). A network-layer protocol can deliver the message only to the destination computer. However, this is an incomplete delivery. The message still needs to be handed to the correct process. This is where a transport-layer protocol takes over. A transport-layer protocol is responsible for delivery of the message to the appropriate process. Figure 23.2 shows the domains of a network layer and a transport layer.

Figure 23.2 *Network layer versus transport layer*


Processes
Processes
Client
Server
Internet
Domain of network-layer protocol
Domain of transport-layer protocol

Addressing: Port Numbers

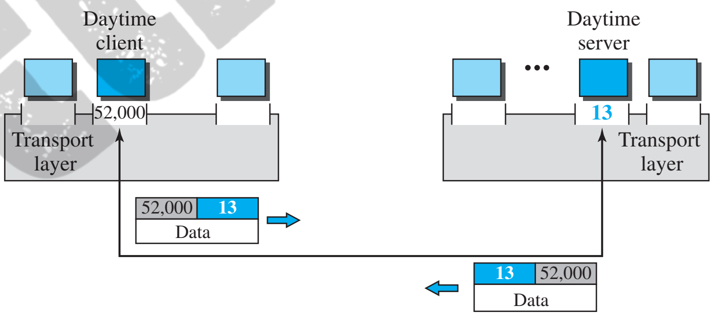
Although there are a few ways to achieve process-to-process communication, the most common is through the **client-server paradigm** (see Chapter 25). A process on the local host, called a *client,* needs services from a process usually on the remote host, called a *server.*

However, operating systems today support both multiuser and multiprogramming environments. A remote computer can run several server programs at the same time, just as several local computers can run one or more client programs at the same time. For communication, we must define the local host, local process, remote host, and remote process. The local host and the remote host are defined using IP addresses (discussed in Chapter 18). To define the processes, we need second identifiers, called ***port numbers.*** In the TCP/IP protocol suite, the port numbers are integers between 0 and 65,535 (16 bits).

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

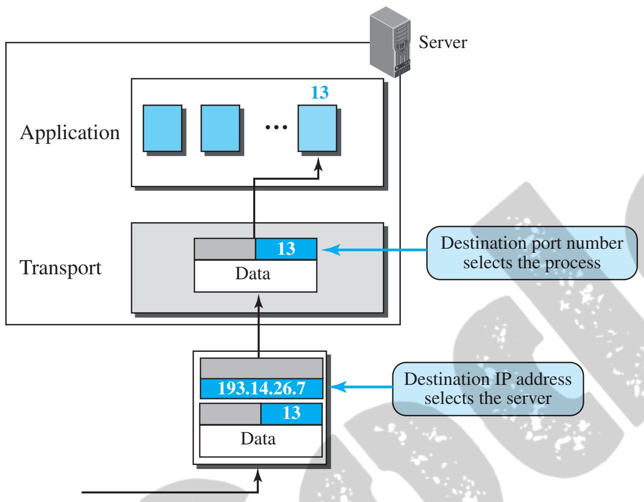
The server process must also define itself with a port number. This port number, however, cannot be chosen randomly. If the computer at the server site runs a server process and assigns a random number as the port number, the process at the client site that wants to access that server and use its services will not know the port number. Of course, one solution would be to send a special packet and request the port number of a specific server, but this creates more overhead. TCP/IP 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, a well-known client program, can use an ephemeral (temporary) port number, 52,000, to identify itself, the daytime server process must use the well-known (permanent) port number 13. Figure 23.3 shows this concept.

Figure 23.3 *Port numbers*



It should be clear by now that the IP addresses and port numbers play different roles in selecting the final destination of data. The destination IP address defines the host among the different hosts in the world. After the host has been selected, the port number defines one of the processes on this particular host (see Figure 23.4).

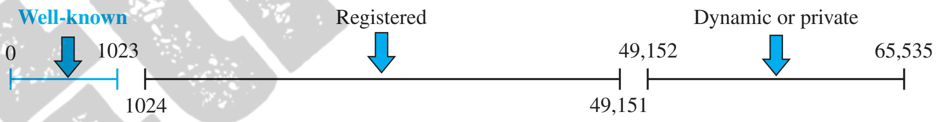
Figure 23.4 *IP addresses versus port numbers*


Server
Application
13
Transport
13
Data
Destination port number selects the process
193.14.26.7
13
Data
Destination IP address selects the server

ICANN Ranges

ICANN (see Chapter 18) has divided the port numbers into three ranges: well-known, registered, and dynamic (or private), as shown in Figure 23.5.

Figure 23.5 *ICANN ranges*



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

Example 23.1

In UNIX, the well-known ports are stored in a file called /etc/services. 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 TFTP. Note that TFTP can use port 69 on either UDP or TCP.

SNMP (see Chapter 27) uses two port numbers (161 and 162), each for a different purpose.

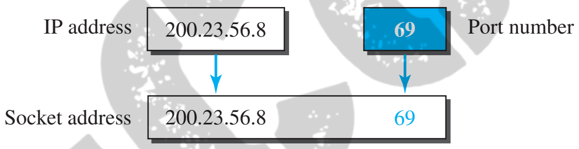
```
$grep   tftp/etc/services
tftp 69/tcp
tftp 69/udp
```

```
$grep   snmp/etc/services
snmp161/tcp#Simple Net Mgmt Proto
snmp161/udp#Simple Net Mgmt Proto
snmptrap162/udp#Traps for SNMP
```

Socket Addresses

A transport-layer protocol in the TCP suite needs both the IP address and the port number, at each end, to make a connection. The combination of an IP address and a port number is called a ***socket address.*** The client socket address defines the client process uniquely just as the server socket address defines the server process uniquely (see Figure 23.6).

Figure 23.6 *Socket address*



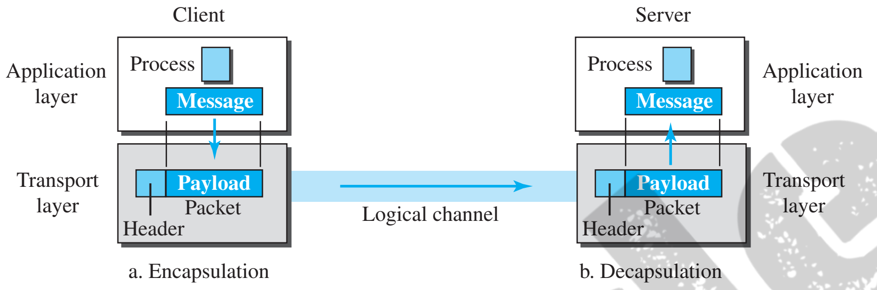
To use the services of the transport layer in the Internet, we need a pair of socket addresses: the client socket address and the server socket address. These four pieces of information are part of the network-layer packet header and the transport-layer packet header. The first header contains the IP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

Encapsulation and Decapsulation

To send a message from one process to another, the transport-layer protocol encapsulates and decapsulates messages (Figure 23.7). Encapsulation happens at the sender site. When a process has a message to send, it passes the message to the transport layer along with a pair of socket addresses and some other pieces of information, which depend on the transport-layer protocol. The transport layer receives the data and adds the transport-layer header. The packets at the transport layer in the Internet are called *user datagrams, segments,* or *packets,* depending on what transport-layer protocol we use. In general discussion, we refer to transport-layer payloads as *packets.*

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

Figure 23.7 *Encapsulation and decapsulation*



Multiplexing and Demultiplexing

Whenever an entity accepts items from more than one source, this is referred to as ***multiplexing*** (many to one); whenever an entity delivers items to more than one source, this is referred to as ***demultiplexing*** (one to many). The transport layer at the source performs multiplexing; the transport layer at the destination performs demultiplexing (Figure 23.8).

Figure 23.8 shows communication between a client and two servers. Three client processes are running at the client site, P1, P2, and P3. The processes P1 and P3 need to send requests to the corresponding server process running in a server. The client process P2 needs to send a request to the corresponding server process running at another server. The transport layer at the client site accepts three messages from the three processes and creates three packets. It acts as a *multiplexer*. The packets 1 and 3 use the same logical channel to reach the transport layer of the first server. When they arrive at the server, the transport layer does the job of a *demultiplexer* and distributes the messages to two different processes. The transport layer at the second server receives packet 2 and delivers it to the corresponding process. Note that we still have demultiplexing although there is only one message.

Flow Control

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

Pushing or Pulling

Delivery of items from a producer to a consumer can occur in one of two ways: *pushing* or *pulling*. If the sender delivers items whenever they are produced—without a prior request from the consumer—the delivery is referred to as *pushing*. If the producer delivers the items after the consumer has requested them, the delivery is referred to as *pulling*. Figure 23.9 shows these two types of delivery.

Figure 23.8 *Multiplexing and demultiplexing*

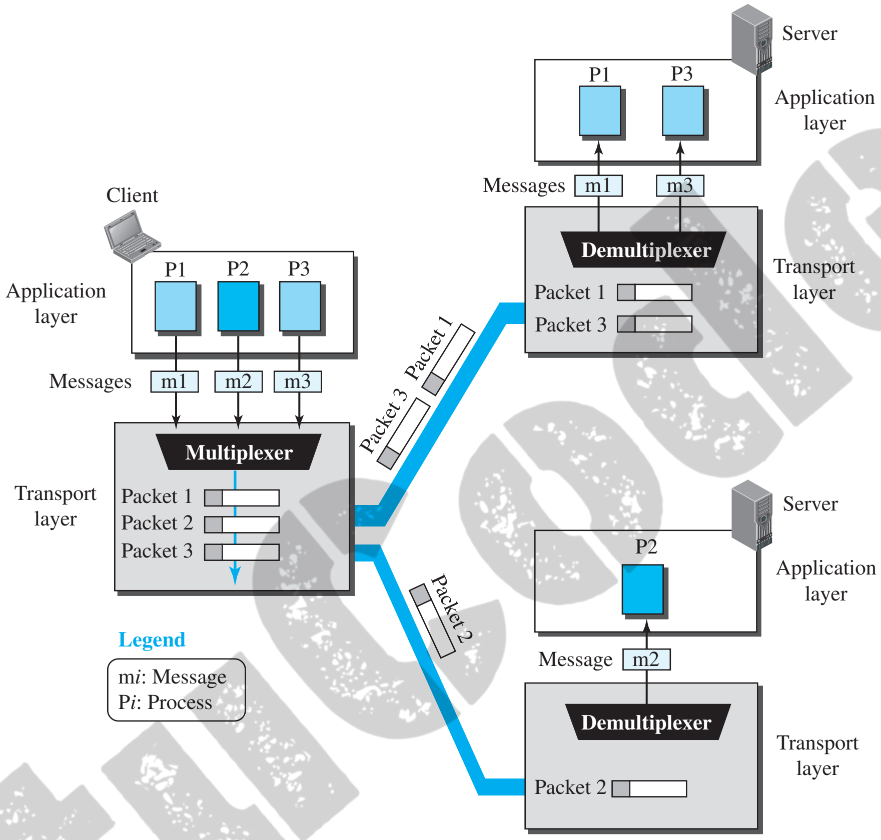
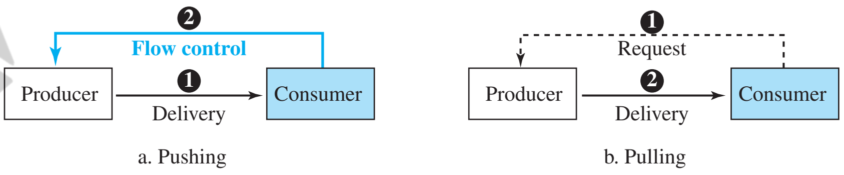


Figure 23.9 *Pushing or pulling*



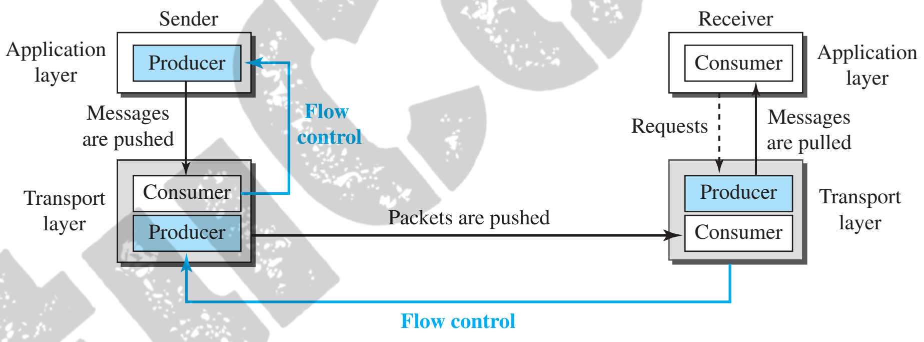
When the producer *pushes* the items, the consumer may be overwhelmed and there is a need for flow control, in the opposite direction, to prevent discarding of the items. In other words, the consumer needs to warn the producer to stop the delivery and to inform the producer when it is again ready to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

Flow Control at Transport Layer

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

Figure 23.10 shows that we need at least two cases of flow control: from the sending transport layer to the sending application layer and from the receiving transport layer to the sending transport layer.

Figure 23.10 *Flow control at the transport layer*



Buffers

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

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

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

Example 23.2

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

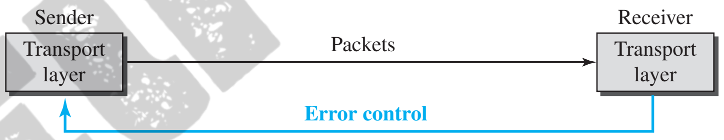
Error Control

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

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

Error control, unlike flow control, involves only the sending and receiving transport layers. We are assuming that the message chunks exchanged between the application and transport layers are error free. Figure 23.11 shows the error control between the sending and receiving transport layers. As with the case of flow control, the receiving transport layer manages error control, most of the time, by informing the sending transport layer about the problems.

Figure 23.11 *Error control at the transport layer*



Sequence Numbers

Error control requires that the sending transport layer knows which packet is to be resent and the receiving transport layer knows which packet is a duplicate, or which packet has arrived out of order. This can be done if the packets are numbered. We can add a field to the transport-layer packet to hold the **sequence number** of the packet. When a packet is corrupted or lost, the receiving transport layer can somehow inform the sending transport layer to resend that packet using the sequence number. The receiving transport layer can also detect duplicate packets if two received packets have the same sequence number. The out-of-order packets can be recognized by observing gaps in the sequence numbers.

Packets are numbered sequentially. However, because we need to include the sequence number of each packet in the header, we need to set a limit. If the header of the packet allows *m* bits for the sequence number, the sequence numbers range from 0 to $2^m - 1$. For example, if m is 4, the only sequence numbers are 0 through 15, inclusive. However, we can wrap around the sequence. So the sequence numbers in this case are

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ...

In other words, the sequence numbers are modulo 2^m.

For error control, the sequence numbers are modulo 2^m, where m is the size of the sequence number field in bits.

Acknowledgment

We can use both positive and negative signals as error control, but we discuss only positive signals, which are more common at the transport layer. The receiver side can send an acknowledgment (ACK) for each of a collection of packets that have arrived safe and sound. The receiver can simply discard the corrupted packets. The sender can detect lost packets if it uses a timer. When a packet is sent, the sender starts a timer. If an ACK does not arrive before the timer expires, the sender resends the packet. Duplicate packets can be silently discarded by the receiver. Out-of-order packets can be either discarded (to be treated as lost packets by the sender), or stored until the missing one arrives.

Combination of Flow and Error Control

We have discussed that flow control requires the use of two buffers, one at the sender site and the other at the receiver site. We have also discussed that error control requires the use of sequence and acknowledgment numbers by both sides. These two requirements can be combined if we use two numbered buffers, one at the sender, one at the receiver.

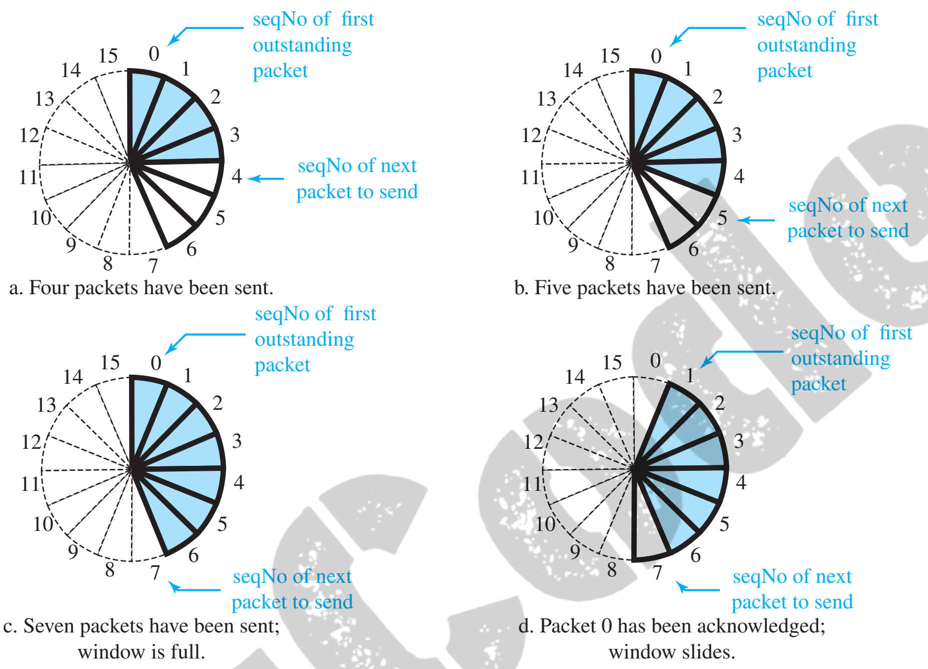
At the sender, when a packet is prepared to be sent, we use the number of the next free location, x, in the buffer as the sequence number of the packet. When the packet is sent, a copy is stored at memory location x, awaiting the acknowledgment from the other end. When an acknowledgment related to a sent packet arrives, the packet is purged and the memory location becomes free.

At the receiver, when a packet with sequence number y arrives, it is stored at the memory location y until the application layer is ready to receive it. An acknowledgment can be sent to announce the arrival of packet y.

Sliding Window

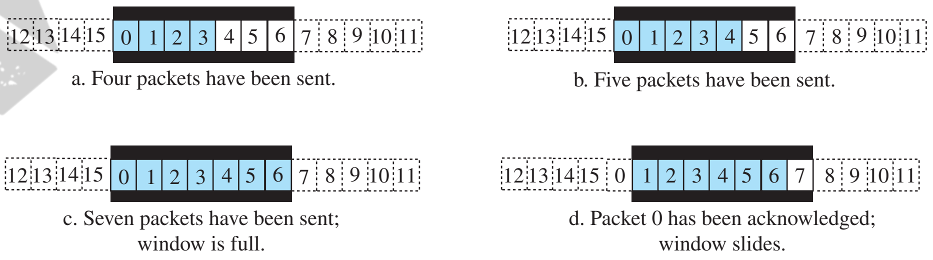
Since the sequence numbers use modulo 2^m, a circle can represent the sequence numbers from 0 to $2^m - 1$ (Figure 23.12). The buffer is represented as a set of slices, called the ***sliding window,*** that occupies part of the circle at any time. At the sender site, when a packet is sent, the corresponding slice is marked. When all the slices are marked, it means that the buffer is full and no further messages can be accepted from the application layer. When an acknowledgment arrives, the corresponding slice is unmarked. If some consecutive slices from the beginning of the window are unmarked, the window slides over the range of the corresponding sequence numbers to allow more free slices at the end of the window. Figure 23.12 shows the sliding window at the sender. The sequence numbers are in modulo 16 ($m = 4$) and the size of the window is 7. Note that the sliding window is just an abstraction: the actual situation uses computer variables to hold the sequence numbers of the next packet to be sent and the last packet sent.

Figure 23.12 *Sliding window in circular format*


seqNo of first outstanding packet
seqNo of next packet to send
a. Four packets have been sent.
b. Five packets have been sent.
c. Seven packets have been sent; window is full.
d. Packet 0 has been acknowledged; window slides.

Most protocols show the sliding window using linear representation. The idea is the same, but it normally takes less space on paper. Figure 23.13 shows this representation. Both representations tell us the same thing. If we take both sides of each diagram in Figure 23.13 and bend them up, we can make the same diagram as in Figure 23.12.

Figure 23.13 *Sliding window in linear format*


12 13 14 15 0 1 2 3 4 5 6 7 8 9 10 11
a. Four packets have been sent.
b. Five packets have been sent.
c. Seven packets have been sent; window is full.
d. Packet 0 has been acknowledged; window slides.

Congestion Control

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

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

Congestion in a network or internetwork occurs because routers and switches have queues—buffers that hold the packets before and after processing. A router, for example, has an input queue and an output queue for each interface. If a router cannot process the packets at the same rate at which they arrive, the queues become overloaded and congestion occurs. Congestion at the transport layer is actually the result of congestion at the network layer, which manifests itself at the transport layer. We discussed congestion at the network layer and its causes in Chapter 18. Later in this chapter, we show how TCP, assuming that there is no congestion control at the network layer, implements its own congestion control mechanism.

23.1.2 Connectionless and Connection-Oriented Protocols

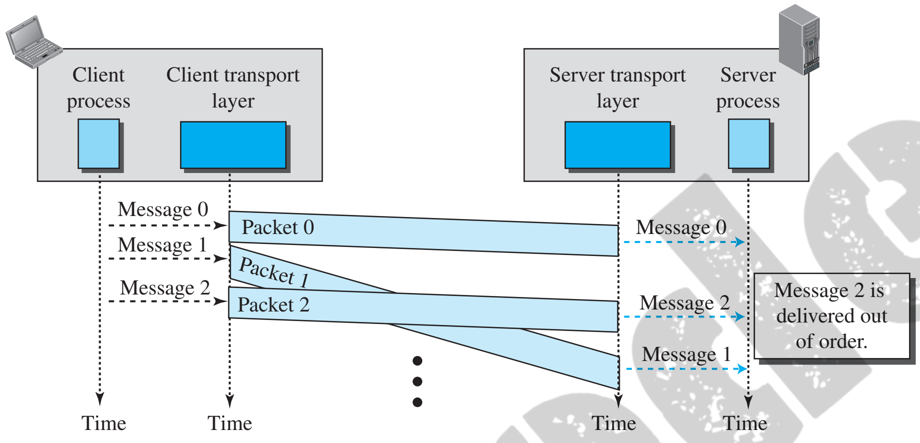
A transport-layer protocol, like a network-layer protocol, can provide two types of services: connectionless and connection-oriented. The nature of these services at the transport layer, however, is different from the ones at the network layer. At the network layer, a connectionless service may mean different paths for different datagrams belonging to the same message. At the transport layer, we are not concerned about the physical paths of packets (we assume a logical connection between two transport layers). Connectionless service at the transport layer means independency between packets; connection-oriented means dependency. Let us elaborate on these two services.

Connectionless Service

In a connectionless service, the source process (application program) needs to divide its message into chunks of data of the size acceptable by the transport layer and deliver them to the transport layer one by one. The transport layer treats each chunk as a single unit without any relation between the chunks. When a chunk arrives from the application layer, the transport layer encapsulates it in a packet and sends it. To show the independency of packets, assume that a client process has three chunks of messages to send to a server process. The chunks are handed over to the connectionless transport protocol in order. However, since there is no dependency between the packets at the transport layer, the packets may arrive out of order at the destination and will be delivered out of order to the server process (Figure 23.14).

In Figure 23.14, we have shown the movement of packets using a time line, but we have assumed that the delivery of the process to the transport layer and vice versa are instantaneous. The figure shows that at the client site, the three chunks of messages are delivered to the client transport layer in order (0, 1, and 2). Because of the extra delay in transportation of the second packet, the delivery of messages at the server is not in

Figure 23.14 *Connectionless service*



order (0, 2, 1). If these three chunks of data belong to the same message, the server process may have received a strange message.

The situation would be worse if one of the packets were lost. Since there is no numbering on the packets, the receiving transport layer has no idea that one of the messages has been lost. It just delivers two chunks of data to the server process.

The above two problems arise from the fact that the two transport layers do not coordinate with each other. The receiving transport layer does not know when the first packet will come nor when all of the packets have arrived.

We can say that no flow control, error control, or congestion control can be effectively implemented in a connectionless service.

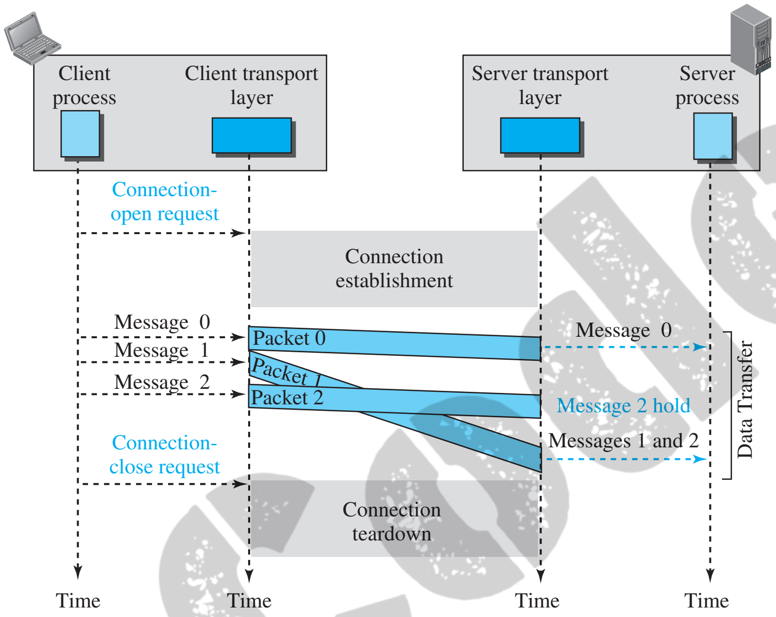
Connection-Oriented Service

In a connection-oriented service, the client and the server first need to establish a logical connection between themselves. The data exchange can only happen after the connection establishment. After data exchange, the connection needs to be torn down (Figure 23.15).

As we mentioned before, the connection-oriented service at the transport layer is different from the same service at the network layer. In the network layer, connection-oriented service means a coordination between the two end hosts and all the routers in between. At the transport layer, connection-oriented service involves only the two hosts; the service is end to end. This means that we should be able to make a connection-oriented protocol at the transport layer over either a connectionless or connection-oriented protocol at the network layer. Figure 23.15 shows the connection-establishment, data-transfer, and tear-down phases in a connection-oriented service at the transport layer.

We can implement flow control, error control, and congestion control in a connection-oriented protocol.

Figure 23.15 *Connection-oriented service*



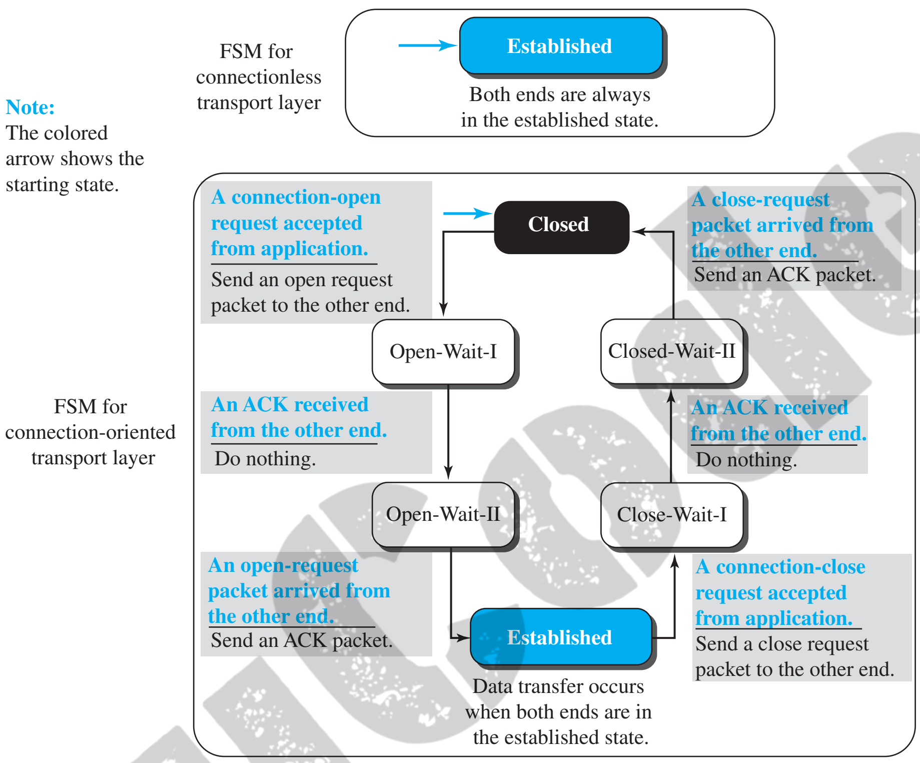
Finite State Machine

The behavior of a transport-layer protocol, both when it provides a connectionless and when it provides a connection-oriented protocol, can be better shown as a **finite state machine (FSM).** Figure 23.16 shows a representation of a transport layer using an FSM. Using this tool, each transport layer (sender or receiver) is taught as a machine with a finite number of states. The machine is always in one of the states until an *event* occurs. Each event is associated with two reactions: defining the list (possibly empty) of actions to be performed and determining the next state (which can be the same as the current state). One of the states must be defined as the initial state, the state in which the machine starts when it turns on. In this figure we have used rounded-corner rectangles to show states, colored text to show events, and regular black text to show actions. A horizontal line is used to separate the event from the actions, although later we replace the horizontal line with a slash. The arrow shows the movement to the next state.

We can think of a connectionless transport layer as an FSM with only one state: the established state. The machine on each end (client and server) is always in the established state, ready to send and receive transport-layer packets.

An FSM in a connection-oriented transport layer, on the other hand, needs to go through three states before reaching the established state. The machine also needs to go through three states before closing the connection. The machine is in the *closed* state when there is no connection. It remains in this state until a request for opening the connection arrives from the local process; the machine sends an open request packet to the

Figure 23.16 *Connectionless and connection-oriented service represented as FSMs*



remote transport layer and moves to the *open-wait-I* state. When an acknowledgment is received from the other end, the local FSM moves to the *open-wait-II* state. When the machine is in this state, a unidirectional connection has been established, but if a bidirectional connection is needed, the machine needs to wait in this state until the other end also requests a connection. When the request is received, the machine sends an acknowledgment and moves to the *established* state.

Data and data acknowledgment can be exchanged between the two ends when they are both in the established state. However, we need to remember that the established state, both in connectionless and connection-oriented transport layers, represents a set of data transfer states, which we discuss in the next section, Transport-Layer Protocols.

To tear down a connection, the application layer sends a close request message to its local transport layer. The transport layer sends a close-request packet to the other end and moves to *close-wait-I* state. When an acknowledgment is received from the other end, the machine moves to the *close-wait-II* state and waits for the close-request packet from the other end. When this packet arrives, the machine sends an acknowledgment and moves to the *closed* state.

There are several variations of the connection-oriented FSM that we will discuss later. We will also see how the FSM can be condensed or expanded and the names of the states can be changed.

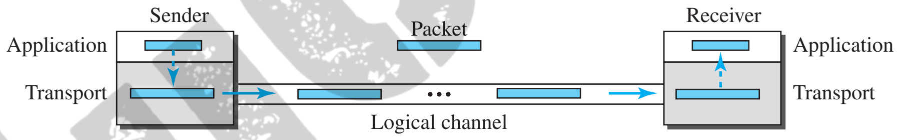
23.2 TRANSPORT-LAYER PROTOCOLS

We can create a transport-layer protocol by combining a set of services described in the previous sections. To better understand the behavior of these protocols, we start with the simplest one and gradually add more complexity. The TCP/IP protocol uses a transport-layer protocol that is either a modification or a combination of some of these protocols. We discuss these general protocols in this section to pave the way for understanding more complex ones in the rest of the chapter. To make our discussion simpler, we first discuss all of these protocols as a unidirectional protocol (i.e., simplex) in which the data packets move in one direction. At the end of the chapter, we briefly discuss how they can be changed to bidirectional protocols where data can be moved in two directions (i.e., full duplex).

23.2.1 Simple Protocol

Our first protocol is a simple connectionless protocol with neither flow nor error control. We assume that the receiver can immediately handle any packet it receives. In other words, the receiver can never be overwhelmed with incoming packets. Figure 23.17 shows the layout for this protocol.

Figure 23.17 *Simple protocol*



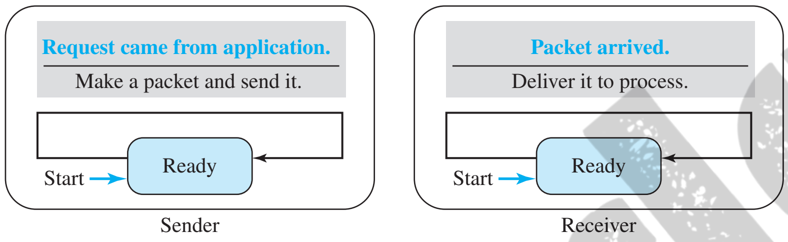
The transport layer at the sender gets a message from its application layer, makes a packet out of it, and sends the packet. The transport layer at the receiver receives a packet from its network layer, extracts the message from the packet, and delivers the message to its application layer. The transport layers of the sender and receiver provide transmission services for their application layers.

FSMs

The sender site should not send a packet until its application layer has a message to send. The receiver site cannot deliver a message to its application layer until a packet arrives. We can show these requirements using two FSMs. Each FSM has only one state, the *ready state*. The sending machine remains in the ready state until a request comes from the process in the application layer. When this event occurs, the sending machine encapsulates the message in a packet and sends it to the receiving machine. The receiving machine remains in the ready state until a packet arrives from the sending machine. When this event occurs, the receiving machine decapsulates the message out of the packet and delivers it to the process at the application layer. Figure 23.18

shows the FSMs for the simple protocol. We see later that the UDP protocol is a slight modification of this protocol.

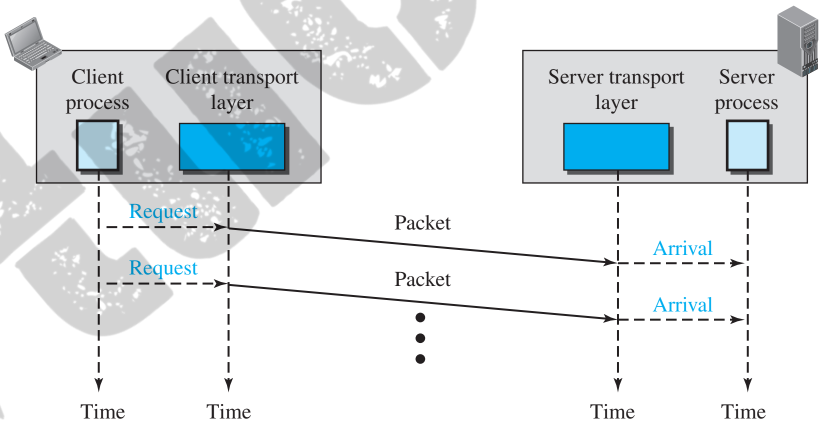
Figure 23.18 *FSMs for the simple protocol*



Example 23.3

Figure 23.19 shows an example of communication using this protocol. It is very simple. The sender sends packets one after another without even thinking about the receiver.

Figure 23.19 *Flow diagram for Example 23.3*

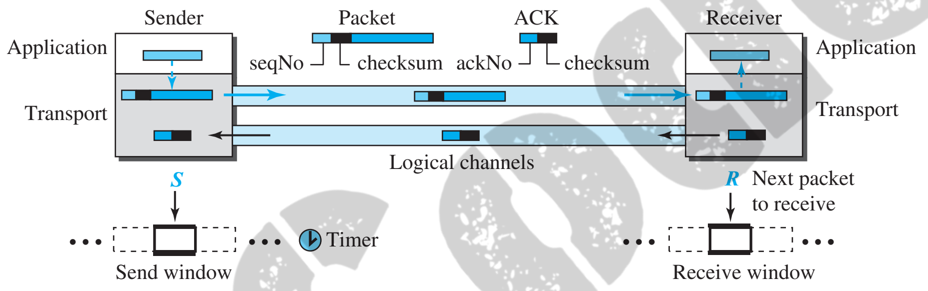


23.2.2 Stop-and-Wait Protocol

Our second protocol is a connection-oriented protocol called the **Stop-and-Wait protocol,** which uses both flow and error control. Both the sender and the receiver use a sliding window of size 1. The sender sends one packet at a time and waits for an acknowledgment before sending the next one. To detect corrupted packets, we need to add a checksum to each data packet. When a packet arrives at the receiver site, it is checked. If its checksum is incorrect, the packet is corrupted and silently discarded.

The silence of the receiver is a signal for the sender that a packet was either corrupted or lost. Every time the sender sends a packet, it starts a timer. If an acknowledgment arrives before the timer expires, the timer is stopped and the sender sends the next packet (if it has one to send). If the timer expires, the sender resends the previous packet, assuming that the packet was either lost or corrupted. This means that the sender needs to keep a copy of the packet until its acknowledgment arrives. Figure 23.20 shows the outline for the Stop-and-Wait protocol. Note that only one packet and one acknowledgment can be in the channels at any time.

Figure 23.20 *Stop-and-Wait protocol*



The Stop-and-Wait protocol is a connection-oriented protocol that provides flow and error control.

Sequence Numbers

To prevent duplicate packets, the protocol uses sequence numbers and acknowledgment numbers. A field is added to the packet header to hold the sequence number of that packet. One important consideration is the range of the sequence numbers. Since we want to minimize the packet size, we look for the smallest range that provides unambiguous communication. Let us discuss the range of sequence numbers we need. Assume we have used x as a sequence number; we only need to use $x + 1$ after that. There is no need for $x + 2$. To show this, assume that the sender has sent the packet with sequence number x. Three things can happen.

1. The packet arrives safe and sound at the receiver site; the receiver sends an acknowledgment. The acknowledgment arrives at the sender site, causing the sender to send the next packet numbered $x + 1$.
2. The packet is corrupted or never arrives at the receiver site; the sender resends the packet (numbered x) after the time-out. The receiver returns an acknowledgment.
3. The packet arrives safe and sound at the receiver site; the receiver sends an acknowledgment, but the acknowledgment is corrupted or lost. The sender resends the packet (numbered x) after the time-out. Note that the packet here is a duplicate. The receiver can recognize this fact because it expects packet $x + 1$ but packet x was received.

We can see that there is a need for sequence numbers x and $x + 1$ because the receiver needs to distinguish between case 1 and case 3. But there is no need for a packet to be numbered $x + 2$. In case 1, the packet can be numbered x again because packets x and $x + 1$ are acknowledged and there is no ambiguity at either site. In cases 2 and 3, the new packet is $x + 1$, not $x + 2$. If only x and $x + 1$ are needed, we can let $x = 0$ and $x + 1 = 1$. This means that the sequence is 0, 1, 0, 1, 0, and so on. This is referred to as modulo 2 arithmetic.

Acknowledgment Numbers

Since the sequence numbers must be suitable for both data packets and acknowledgments, we use this convention: The acknowledgment numbers always announce the sequence number of the *next packet expected* by the receiver. For example, if packet 0 has arrived safe and sound, the receiver sends an ACK with acknowledgment 1 (meaning packet 1 is expected next). If packet 1 has arrived safe and sound, the receiver sends an ACK with acknowledgment 0 (meaning packet 0 is expected).

In the Stop-and-Wait protocol, the acknowledgment number always announces, in modulo-2 arithmetic, the sequence number of the next packet expected.

The sender has a control variable, which we call S (sender), that points to the only slot in the send window. The receiver has a control variable, which we call R (receiver), that points to the only slot in the receive window.

FSMs

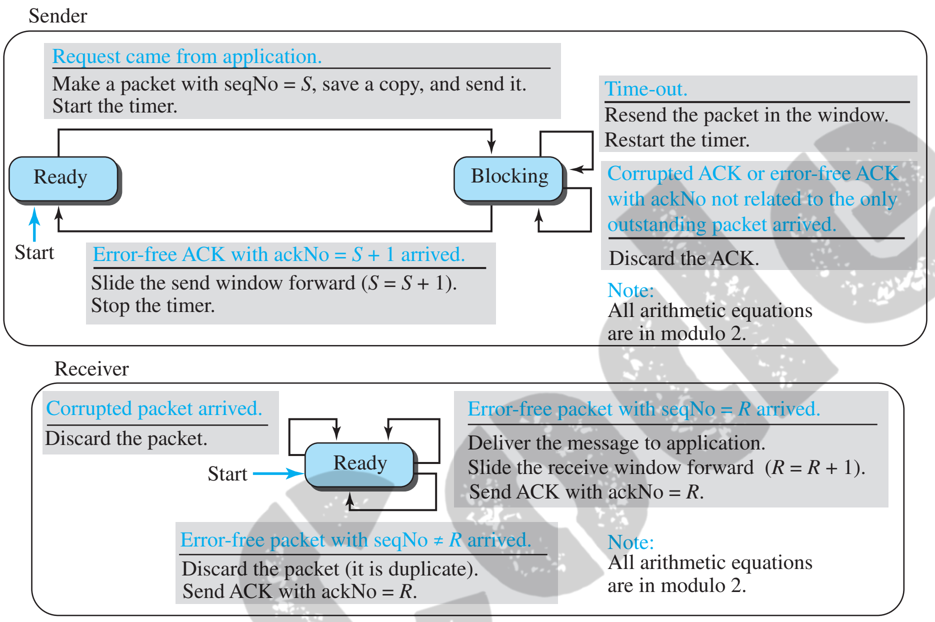
Figure 23.21 shows the FSMs for the Stop-and-Wait protocol. Since the protocol is a connection-oriented protocol, both ends should be in the *established* state before exchanging data packets. The states are actually nested in the *established* state.

Sender

The sender is initially in the ready state, but it can move between the ready and blocking state. The variable S is initialized to 0.

- ❑ ***Ready state.*** When the sender is in this state, it is only waiting for one event to occur. If a request comes from the application layer, the sender creates a packet with the sequence number set to S. A copy of the packet is stored, and the packet is sent. The sender then starts the only timer. The sender then moves to the blocking state.
- ❑ ***Blocking state.*** When the sender is in this state, three events can occur:
  - **a.** If an error-free ACK arrives with the ackNo related to the next packet to be sent, which means ackNo = $(S + 1)$ modulo 2, then the timer is stopped. The window slides, $S = (S + 1)$ modulo 2. Finally, the sender moves to the ready state.
  - **b.** If a corrupted ACK or an error-free ACK with the ackNo $\neq$ $(S + 1)$ modulo 2 arrives, the ACK is discarded.
  - **c.** If a time-out occurs, the sender resends the only outstanding packet and restarts the timer.

Figure 23.21 *FSMs for the Stop-and-Wait protocol*


Sender
Request came from application.
Make a packet with seqNo = S, save a copy, and send it.
Start the timer.
Time-out.
Resend the packet in the window.
Restart the timer.
Ready
Blocking
Corrupted ACK or error-free ACK with ackNo not related to the only outstanding packet arrived.
Discard the ACK.
Start
Error-free ACK with ackNo = S + 1 arrived.
Slide the send window forward (S = S + 1).
Stop the timer.
Note:
All arithmetic equations are in modulo 2.
Receiver
Corrupted packet arrived.
Discard the packet.
Error-free packet with seqNo = R arrived.
Deliver the message to application.
Slide the receive window forward (R = R + 1).
Send ACK with ackNo = R.
Start
Ready
Error-free packet with seqNo ≠ R arrived.
Discard the packet (it is duplicate).
Send ACK with ackNo = R.
Note:
All arithmetic equations are in modulo 2.

Receiver

The receiver is always in the *ready* state. Three events may occur:

a. If an error-free packet with seqNo = R arrives, the message in the packet is delivered to the application layer. The window then slides, $R = (R + 1)$ modulo 2. Finally an ACK with ackNo = R is sent.

b. If an error-free packet with seqNo ≠ R arrives, the packet is discarded, but an ACK with ackNo = R is sent.

c. If a corrupted packet arrives, the packet is discarded.

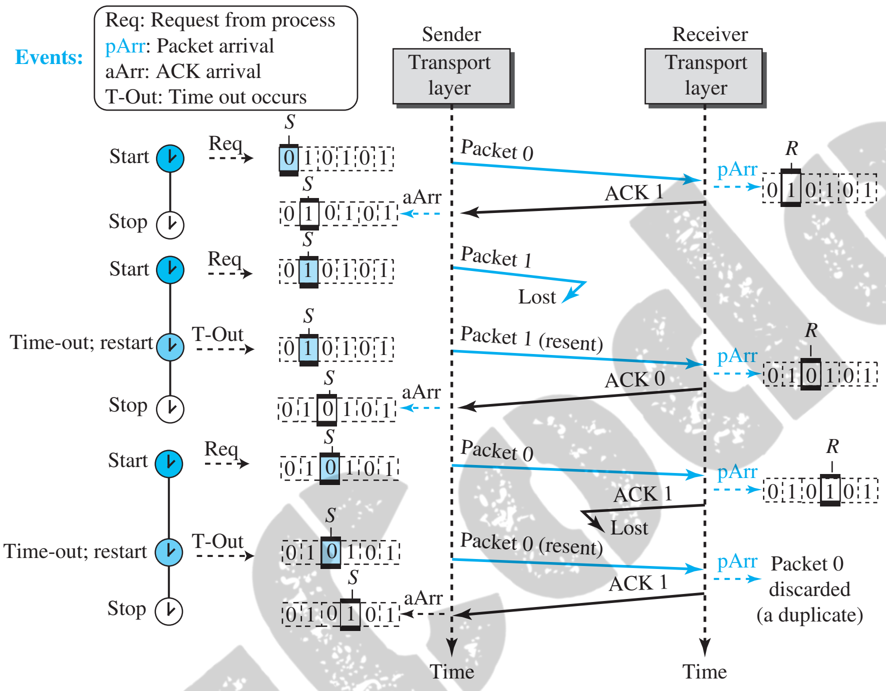
Example 23.4

Figure 23.22 shows an example of the Stop-and-Wait protocol. Packet 0 is sent and acknowledged. Packet 1 is lost and resent after the time-out. The resent packet 1 is acknowledged and the timer stops. Packet 0 is sent and acknowledged, but the acknowledgment is lost. The sender has no idea if the packet or the acknowledgment is lost, so after the time-out, it resends packet 0, which is acknowledged.

Efficiency

The Stop-and-Wait protocol is very inefficient if our channel is *thick* and *long*. By *thick,* we mean that our channel has a large bandwidth (high data rate); by *long,* we mean the round-trip delay is long. The product of these two is called the **bandwidth-delay product.** We can think of the channel as a pipe. The bandwidth-delay product then is the volume of the pipe in bits. The pipe is always there. It is not efficient if it

Figure 23.22 *Flow diagram for Example 23.4*



is not used. The bandwidth-delay product is a measure of the number of bits a sender can transmit through the system while waiting for an acknowledgment from the receiver.

Example 23.5

Assume that, in a Stop-and-Wait system, the bandwidth of the line is 1 Mbps, and 1 bit takes 20 milliseconds to make a round trip. What is the bandwidth-delay product? If the system data packets are 1,000 bits in length, what is the utilization percentage of the link?

Solution

The bandwidth-delay product is $(1 \times 10^6) \times (20 \times 10^{-3}) = 20{,}000$ bits. The system can send 20,000 bits during the time it takes for the data to go from the sender to the receiver and the acknowledgment to come back. However, the system sends only 1,000 bits. We can say that the link utilization is only 1,000/20,000, or 5 percent. For this reason, in a link with a high bandwidth or long delay, the use of Stop-and-Wait wastes the capacity of the link.

Example 23.6

What is the utilization percentage of the link in Example 23.5 if we have a protocol that can send up to 15 packets before stopping and worrying about the acknowledgments?

Solution

The bandwidth-delay product is still 20,000 bits. The system can send up to 15 packets or 15,000 bits during a round trip. This means the utilization is 15,000/20,000, or 75 percent. Of course, if there are damaged packets, the utilization percentage is much less because packets have to be resent.

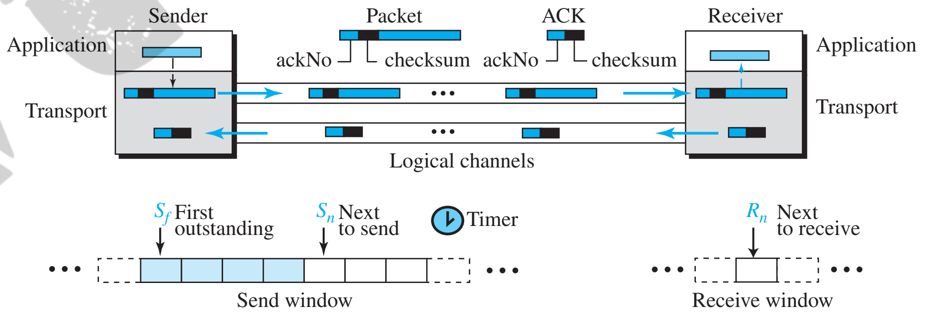
Pipelining

In networking and in other areas, a task is often begun before the previous task has ended. This is known as **pipelining.** There is no pipelining in the Stop-and-Wait protocol because a sender must wait for a packet to reach the destination and be acknowledged before the next packet can be sent. However, pipelining does apply to our next two protocols because several packets can be sent before a sender receives feedback about the previous packets. Pipelining improves the efficiency of the transmission if the number of bits in transition is large with respect to the bandwidth-delay product.

23.2.3 Go-Back-*N* Protocol (GBN)

To improve the efficiency of transmission (to fill the pipe), multiple packets must be in transition while the sender is waiting for acknowledgment. In other words, we need to let more than one packet be outstanding to keep the channel busy while the sender is waiting for acknowledgment. In this section, we discuss one protocol that can achieve this goal; in the next section, we discuss a second. The first is called ***Go-Back-N*** **(GBN)** (the rationale for the name will become clear later). The key to Go-back-*N* is that we can send several packets before receiving acknowledgments, but the receiver can only buffer one packet. We keep a copy of the sent packets until the acknowledgments arrive. Figure 23.23 shows the outline of the protocol. Note that several data packets and acknowledgments can be in the channel at the same time.

Figure 23.23 *Go-Back-*N *protocol*



Sequence Numbers

As we mentioned before, the sequence numbers are modulo 2^m, where m is the size of the sequence number field in bits.

Acknowledgment Numbers

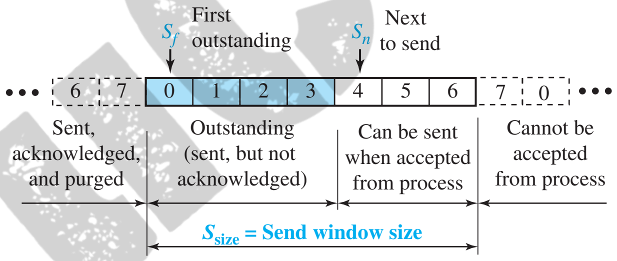
An acknowledgment number in this protocol is cumulative and defines the sequence number of the next packet expected. For example, if the acknowledgment number (ackNo) is 7, it means all packets with sequence number up to 6 have arrived, safe and sound, and the receiver is expecting the packet with sequence number 7.

In the Go-Back-*N* protocol, the acknowledgment number is cumulative and defines the sequence number of the next packet expected to arrive.

Send Window

The send window is an imaginary box covering the sequence numbers of the data packets that can be in transit or can be sent. In each window position, some of these sequence numbers define the packets that have been sent; others define those that can be sent. The maximum size of the window is $2^m - 1$, for reasons that we discuss later. In this chapter, we let the size be fixed and set to the maximum value, but we will see later that some protocols may have a variable window size. Figure 23.24 shows a sliding window of size 7 ($m = 3$) for the Go-Back-*N* protocol.

Figure 23.24 *Send window for Go-Back-*N



S_f First outstanding
S_n Next to send
6
7
0
1
2
3
4
5
6
7
0
Sent, acknowledged, and purged
Outstanding (sent, but not acknowledged)
Can be sent when accepted from process
Cannot be accepted from process
S_{size} = Send window size

The send window at any time divides the possible sequence numbers into four regions. The first region, left of the window, defines the sequence numbers belonging to packets that are already acknowledged. The sender does not worry about these packets and keeps no copies of them. The second region, colored, defines the range of sequence numbers belonging to the packets that have been sent, but have an unknown status. The sender needs to wait to find out if these packets have been received or were lost. We call these *outstanding* packets. The third range, white in the figure, defines the range of sequence numbers for packets that can be sent; however, the corresponding data have not yet been received from the application layer. Finally, the fourth region, right of the window, defines sequence numbers that cannot be used until the window slides.

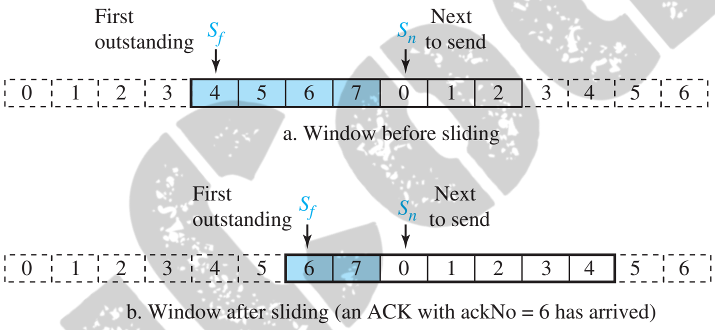
The window itself is an abstraction; three variables define its size and location at any time. We call these variables S_f (send window, the first outstanding packet), S_n (send window, the next packet to be sent), and S_{size} (send window, size). The variable S_f defines the sequence number of the first (oldest) outstanding packet. The variable

S_n holds the sequence number that will be assigned to the next packet to be sent. Finally, the variable S_{size} defines the size of the window, which is fixed in our protocol.

The send window is an abstract concept defining an imaginary box of maximum size = $2^m - 1$ with three variables: S_f, S_n, and S_{size}.

Figure 23.25 shows how a send window can slide one or more slots to the right when an acknowledgment arrives from the other end. In the figure, an acknowledgment with ackNo = 6 has arrived. This means that the receiver is waiting for packets with sequence number 6.

Figure 23.25 *Sliding the send window*

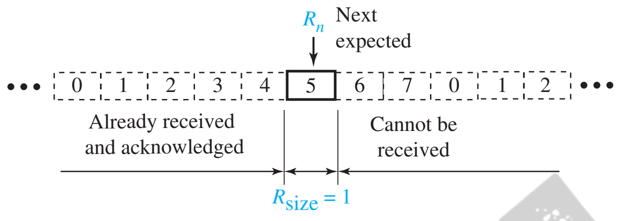


The send window can slide one or more slots when an error-free ACK with ackNo greater than or equal to S_f and less than S_n (in modular arithmetic) arrives.

Receive Window

The receive window makes sure that the correct data packets are received and that the correct acknowledgments are sent. In Go-Back-*N*, the size of the receive window is always 1. The receiver is always looking for the arrival of a specific packet. Any packet arriving out of order is discarded and needs to be resent. Figure 23.26 shows the receive window. Note that we need only one variable, R_n (receive window, next packet expected), to define this abstraction. The sequence numbers to the left of the window belong to the packets already received and acknowledged; the sequence numbers to the right of this window define the packets that cannot be received. Any received packet with a sequence number in these two regions is discarded. Only a packet with a sequence number matching the value of R_n is accepted and acknowledged. The receive window also slides, but only one slot at a time. When a correct packet is received, the window slides, $R_n = (R_n + 1)$ modulo 2^m.

Figure 23.26 *Receive window for Go-Back-*N



The receive window is an abstract concept defining an imaginary box of size 1 with a single variable R_n. The window slides when a correct packet has arrived; sliding occurs one slot at a time.

Timers

Although there can be a timer for each packet that is sent, in our protocol we use only one. The reason is that the timer for the first outstanding packet always expires first. We resend all outstanding packets when this timer expires.

Resending packets

When the timer expires, the sender resends all outstanding packets. For example, suppose the sender has already sent packet 6 ($S_n = 7$), but the only timer expires. If $S_f = 3$, this means that packets 3, 4, 5, and 6 have not been acknowledged; the sender goes back and resends packets 3, 4, 5, and 6. That is why the protocol is called *Go-Back-*N. On a time-out, the machine goes back N locations and resends all packets.

FSMs

Figure 23.27 shows the FSMs for the GBN protocol.

Sender

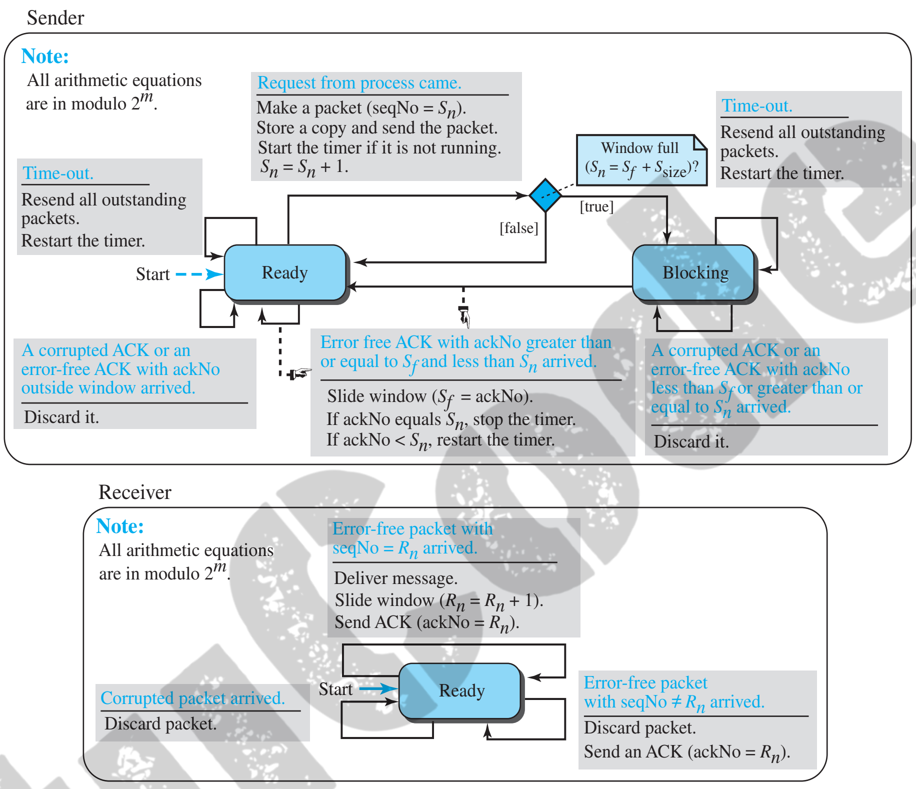
The sender starts in the ready state, but thereafter it can be in one of the two states: *ready* or *blocking*. The two variables are normally initialized to 0 ($S_f = S_n = 0$).

❑ ***Ready state.*** Four events may occur when the sender is in ready state.

  a. If a request comes from the application layer, the sender creates a packet with the sequence number set to S_n. A copy of the packet is stored, and the packet is sent. The sender also starts the only timer if it is not running. The value of S_n is now incremented, ($S_n = S_n + 1$) modulo 2^m. If the window is full, $S_n = (S_f + S_{\text{size}})$ modulo 2^m, the sender goes to the blocking state.

  b. If an error-free ACK arrives with ackNo related to one of the outstanding packets, the sender slides the window (set S_f = ackNo), and if all outstanding packets are acknowledged (ackNo = S_n), then the timer is stopped. If all outstanding packets are not acknowledged, the timer is restarted.

Figure 23.27 *FSMs for the Go-Back-N protocol*



Sender
Note:
All arithmetic equations are in modulo 2^m.
Request from process came.
Make a packet (seqNo = S_n).
Store a copy and send the packet.
Start the timer if it is not running.
$S_n = S_n + 1$.
Time-out.
Resend all outstanding packets.
Restart the timer.
Window full ($S_n = S_f + S_{size}$)?
[false]
[true]
Time-out.
Resend all outstanding packets.
Restart the timer.
Start
Ready
Blocking
A corrupted ACK or an error-free ACK with ackNo outside window arrived.
Discard it.
Error free ACK with ackNo greater than or equal to S_f and less than S_n arrived.
Slide window (S_f = ackNo).
If ackNo equals S_n, stop the timer.
If ackNo < S_n, restart the timer.
A corrupted ACK or an error-free ACK with ackNo less than S_f or greater than or equal to S_n arrived.
Discard it.
Receiver
Note:
All arithmetic equations are in modulo 2^m.
Error-free packet with seqNo = R_n arrived.
Deliver message.
Slide window ($R_n = R_n + 1$).
Send ACK (ackNo = R_n).
Corrupted packet arrived.
Discard packet.
Start
Ready
Error-free packet with seqNo ≠ R_n arrived.
Discard packet.
Send an ACK (ackNo = R_n).

c. If a corrupted ACK or an error-free ACK with ackNo not related to the outstanding packet arrives, it is discarded.

d. If a time-out occurs, the sender resends all outstanding packets and restarts the timer.

- ***Blocking state.*** Three events may occur in this case:

a. If an error-free ACK arrives with ackNo related to one of the outstanding packets, the sender slides the window (set S_f = ackNo) and if all outstanding packets are acknowledged (ackNo = S_n), then the timer is stopped. If all outstanding packets are not acknowledged, the timer is restarted. The sender then moves to the ready state.

b. If a corrupted ACK or an error-free ACK with the ackNo not related to the outstanding packets arrives, the ACK is discarded.

c. If a time-out occurs, the sender sends all outstanding packets and restarts the timer.

Receiver

The receiver is always in the *ready* state. The only variable, R_n, is initialized to 0. Three events may occur:

a. If an error-free packet with seqNo = R_n arrives, the message in the packet is delivered to the application layer. The window then slides, $R_n = (R_n + 1)$ modulo 2^m. Finally an ACK is sent with ackNo = R_n.

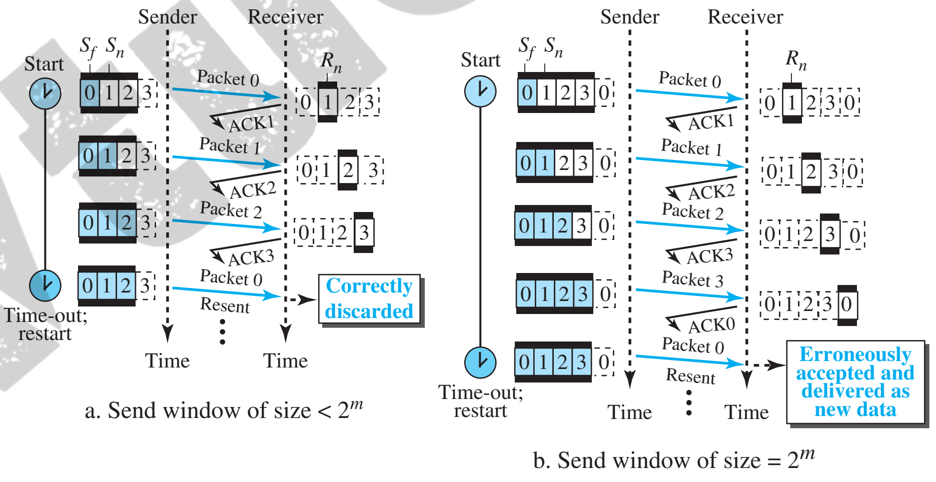
b. If an error-free packet with seqNo outside the window arrives, the packet is discarded, but an ACK with ackNo = R_n is sent.

c. If a corrupted packet arrives, it is discarded.

Send Window Size

We can now show why the size of the send window must be less than 2^m. As an example, we choose $m = 2$, which means the size of the window can be $2^m - 1$, or 3. Figure 23.28 compares a window size of 3 against a window size of 4. If the size of the window is 3 (less than 2^m) and all three acknowledgments are lost, the only timer expires and all three packets are resent. The receiver is now expecting packet 3, not packet 0, so the duplicate packet is correctly discarded. On the other hand, if the size of the window is 4 (equal to 2^2) and all acknowledgments are lost, the sender will send a duplicate of packet 0. However, this time the window of the receiver expects to receive packet 0 (in the next cycle), so it accepts packet 0, not as a duplicate, but as the first packet in the next cycle. This is an error. This shows that the size of the send window must be less than 2^m.

Figure 23.28 *Send window size for Go-Back-*N

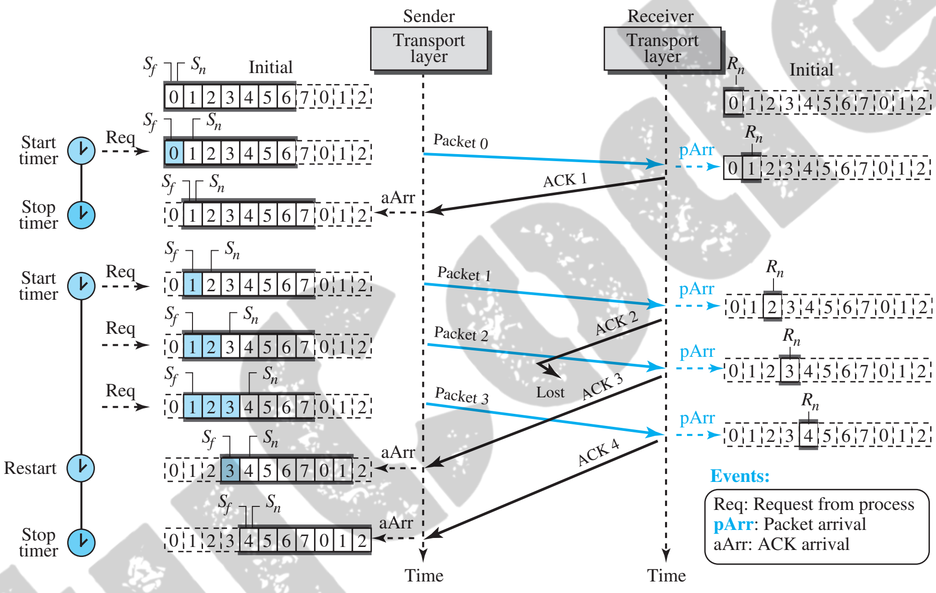


In the Go-Back-*N* protocol, the size of the send window must be less than 2^m; the size of the receive window is always 1.

Example 23.7

Figure 23.29 shows an example of Go-Back-*N*. This is an example of a case where the forward channel is reliable, but the reverse is not. No data packets are lost, but some ACKs are delayed and one is lost. The example also shows how cumulative acknowledgments can help if acknowledgments are delayed or lost.

Figure 23.29 *Flow diagram for Example 23.7*



After initialization, there are some sender events. Request events are triggered by message chunks from the application layer; arrival events are triggered by ACKs received from the network layer. There is no time-out event here because all outstanding packets are acknowledged before the timer expires. Note that although ACK 2 is lost, ACK 3 is cumulative and serves as both ACK 2 and ACK 3. There are four events at the receiver site.

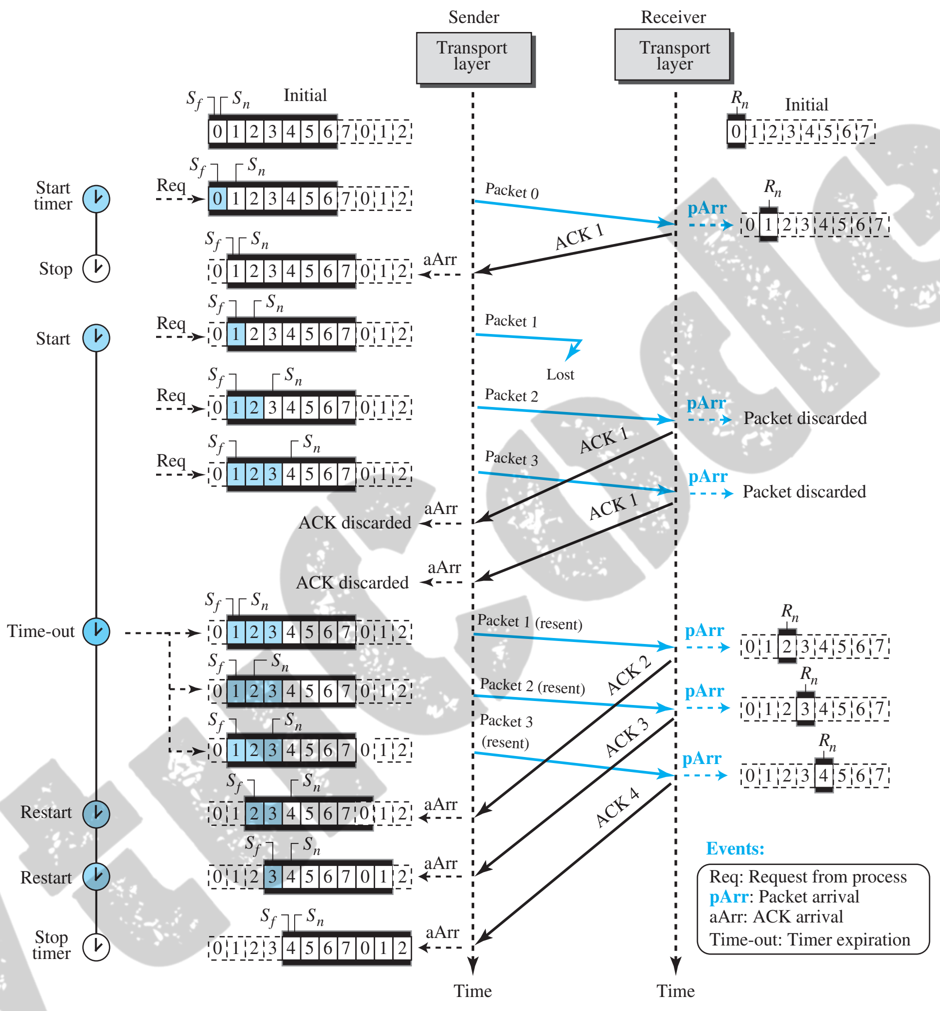
Example 23.8

Figure 23.30 shows what happens when a packet is lost. Packets 0, 1, 2, and 3 are sent. However, packet 1 is lost. The receiver receives packets 2 and 3, but they are discarded because they are received out of order (packet 1 is expected). When the receiver receives packets 2 and 3, it sends ACK1 to show that it expects to receive packet 1. However, these ACKs are not useful for the sender because the ackNo is equal to S_f, not greater than S_f. So the sender discards them. When the time-out occurs, the sender resends packets 1, 2, and 3, which are acknowledged.

Go-Back-N versus Stop-and-Wait

The reader may find that there is a similarity between the Go-Back-*N* protocol and the Stop-and-Wait protocol. The Stop-and-Wait protocol is actually a Go-Back-*N* protocol in

Figure 23.30 *Flow diagram for Example 23.8*



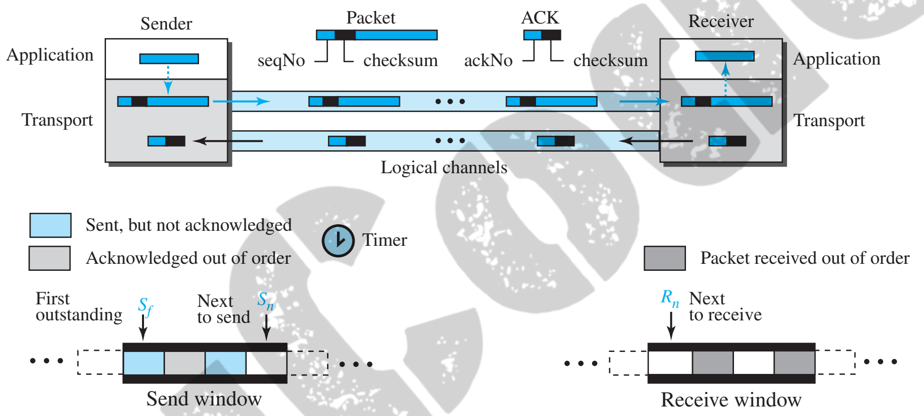
which there are only two sequence numbers and the send window size is 1. In other words, $m = 1$ and $2^m - 1 = 1$. In Go-Back-*N*, we said that the arithmetic is modulo 2^m; in Stop-and-Wait it is modulo 2, which is the same as 2^m when $m = 1$.

23.2.4 Selective-Repeat Protocol

The Go-Back-*N* protocol simplifies the process at the receiver. The receiver keeps track of only one variable, and there is no need to buffer out-of-order packets; they are simply discarded. However, this protocol is inefficient if the underlying network protocol loses a lot of packets. Each time a single packet is lost or corrupted, the sender resends all outstanding packets, even though some of these packets may have been received safe and sound but out of order. If the network layer is losing many packets because of congestion in the network, the resending of all of these outstanding packets makes the congestion worse, and eventually more packets are lost. This has an avalanche effect that may result in the total collapse of the network.

Another protocol, called the **Selective-Repeat (SR) protocol,** has been devised, which, as the name implies, resends only selective packets, those that are actually lost. The outline of this protocol is shown in Figure 23.31.

Figure 23.31 *Outline of Selective-Repeat*



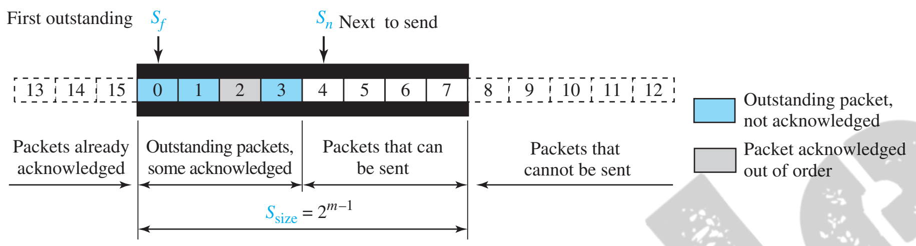
Windows

The Selective-Repeat protocol also uses two windows: a send window and a receive window. However, there are differences between the windows in this protocol and the ones in Go-Back-*N*. First, the maximum size of the send window is much smaller; it is 2^{m-1}. The reason for this will be discussed later. Second, the receive window is the same size as the send window.

The send window maximum size can be 2^{m-1}. For example, if $m = 4$, the sequence numbers go from 0 to 15, but the maximum size of the window is just 8 (it is 15 in the Go-Back-*N* Protocol). We show the Selective-Repeat send window in Figure 23.32 to emphasize the size.

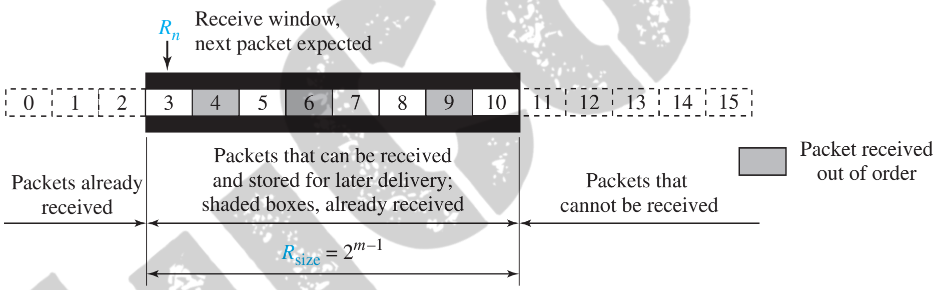
The receive window in Selective-Repeat is totally different from the one in Go-Back-*N*. The size of the receive window is the same as the size of the send window (maximum 2^{m-1}). The Selective-Repeat protocol allows as many packets as the size of the receive window to arrive out of order and be kept until there is a set of consecutive packets to be delivered to the application layer. Because the sizes of the send window and receive window are the same, all the packets in the send packet can arrive out of order and be stored until they can be delivered. We need, however, to emphasize that in a reliable protocol the receiver *never* delivers packets out of order to the application layer. Figure 23.33 shows the receive window in Selective-Repeat. Those slots inside the

Figure 23.32 *Send window for Selective-Repeat protocol*


First outstanding
S_f
S_n Next to send
13 14 15 0 1 2 3 4 5 6 7 8 9 10 11 12
Outstanding packet, not acknowledged
Packet acknowledged out of order
Packets already acknowledged
Outstanding packets, some acknowledged
Packets that can be sent
Packets that cannot be sent
$S_{size} = 2^{m-1}$

window that are shaded define packets that have arrived out of order and are waiting for the earlier transmitted packet to arrive before delivery to the application layer.

Figure 23.33 *Receive window for Selective-Repeat protocol*


R_n
Receive window, next packet expected
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Packet received out of order
Packets already received
Packets that can be received and stored for later delivery; shaded boxes, already received
Packets that cannot be received
$R_{size} = 2^{m-1}$

Timer

Theoretically, Selective-Repeat uses one timer for each outstanding packet. When a timer expires, only the corresponding packet is resent. In other words, GBN treats outstanding packets as a group; SR treats them individually. However, most transport-layer protocols that implement SR use only a single timer. For this reason, we use only one timer.

Acknowledgments

There is yet another difference between the two protocols. In GBN an ackNo is cumulative; it defines the sequence number of the next packet expected, confirming that all previous packets have been received safe and sound. The semantics of acknowledgment is different in SR. In SR, an ackNo defines the sequence number of a single packet that is received safe and sound; there is no feedback for any other.

In the Selective-Repeat protocol, an acknowledgment number defines the sequence number of the error-free packet received.

Example 23.9

Assume a sender sends 6 packets: packets 0, 1, 2, 3, 4, and 5. The sender receives an ACK with ackNo = 3. What is the interpretation if the system is using GBN or SR?

Solution

If the system is using GBN, it means that packets 0, 1, and 2 have been received uncorrupted and the receiver is expecting packet 3. If the system is using SR, it means that packet 3 has been received uncorrupted; the ACK does not say anything about other packets.

FSMs

Figure 23.34 shows the FSMs for the Selective-Repeat protocol. It is similar to the ones for the GBN, but there are some differences.

Sender

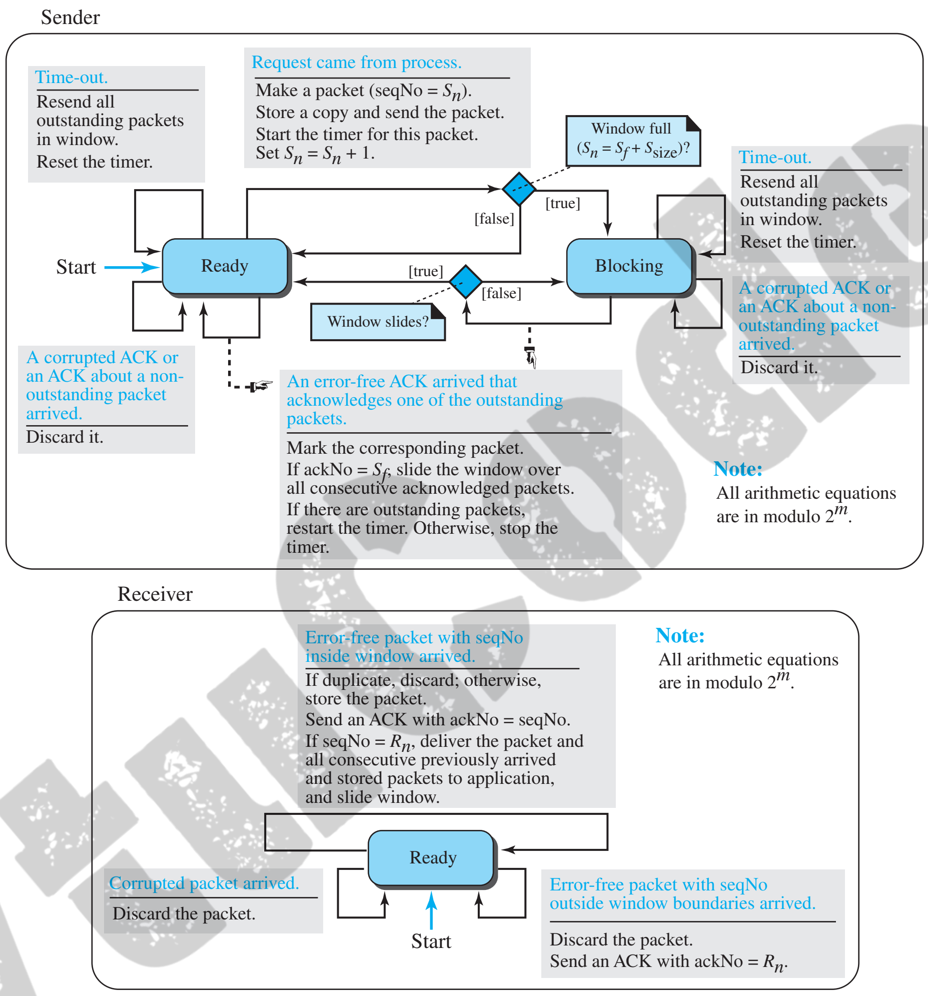
The sender starts in the *ready* state, but later it can be in one of the two states: *ready* or *blocking*. The following shows the events and the corresponding actions in each state.

- ***Ready state.*** Four events may occur in this case:
  - **a.** If a request comes from the application layer, the sender creates a packet with the sequence number set to S_n. A copy of the packet is stored, and the packet is sent. If the timer is not running, the sender starts the timer. The value of S_n is now incremented, $S_n = (S_n + 1)$ modulo 2^m. If the window is full, $S_n = (S_f + S_{size})$ modulo 2^m, the sender goes to the blocking state.
  - **b.** If an error-free ACK arrives with ackNo related to one of the outstanding packets, that packet is marked as acknowledged. If the ackNo = S_f, the window slides to the right until the S_f points to the first unacknowledged packet (all consecutive acknowledged packets are now outside the window). If there are outstanding packets, the timer is restarted; otherwise, the timer is stopped.
  - **c.** If a corrupted ACK or an error-free ACK with ackNo not related to an outstanding packet arrives, it is discarded.
  - **d.** If a time-out occurs, the sender resends all unacknowledged packets in the window and restarts the timer.
- ***Blocking state.*** Three events may occur in this case:
  - **a.** If an error-free ACK arrives with ackNo related to one of the outstanding packets, that packet is marked as acknowledged. In addition, if the ackNo = S_f, the window is slid to the right until the S_f points to the first unacknowledged packet (all consecutive acknowledged packets are now outside the window). If the window has slid, the sender moves to the ready state.
  - **b.** If a corrupted ACK or an error-free ACK with the ackNo not related to outstanding packets arrives, the ACK is discarded.
  - **c.** If a time-out occurs, the sender resends all unacknowledged packets in the window and restarts the timer.

Receiver

The receiver is always in the *ready* state. Three events may occur:

Figure 23.34 *FSMs for SR protocol*


Sender
Time-out.
Resend all outstanding packets in window.
Reset the timer.
Request came from process.
Make a packet (seqNo = S_n).
Store a copy and send the packet.
Start the timer for this packet.
Set $S_n = S_n + 1$.
Window full ($S_n = S_f + S_{size}$)?
[true]
[false]
Time-out.
Resend all outstanding packets in window.
Reset the timer.
Start
Ready
Blocking
[true]
[false]
Window slides?
A corrupted ACK or an ACK about a non-outstanding packet arrived.
Discard it.
A corrupted ACK or an ACK about a non-outstanding packet arrived.
Discard it.
An error-free ACK arrived that acknowledges one of the outstanding packets.
Mark the corresponding packet.
If ackNo = S_f, slide the window over all consecutive acknowledged packets.
If there are outstanding packets, restart the timer. Otherwise, stop the timer.
Note: All arithmetic equations are in modulo 2^m.
Receiver
Error-free packet with seqNo inside window arrived.
If duplicate, discard; otherwise, store the packet.
Send an ACK with ackNo = seqNo.
If seqNo = R_n, deliver the packet and all consecutive previously arrived and stored packets to application, and slide window.
Note: All arithmetic equations are in modulo 2^m.
Ready
Corrupted packet arrived.
Discard the packet.
Start
Error-free packet with seqNo outside window boundaries arrived.
Discard the packet.
Send an ACK with ackNo = R_n.

a. If an error-free packet with seqNo in the window arrives, the packet is stored and an ACK with ackNo = seqNo is sent. In addition, if the seqNo = R_n, then the packet and all previously arrived consecutive packets are delivered to the application layer and the window slides so that the R_n points to the first empty slot.

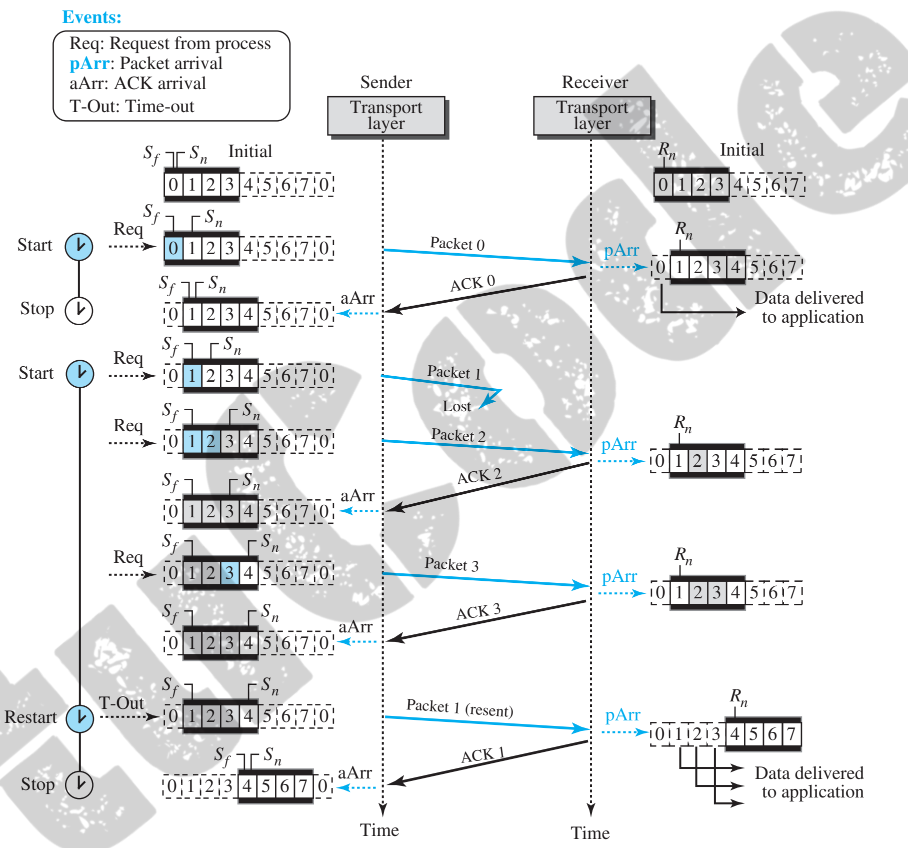
b. If an error-free packet with seqNo outside the window arrives, the packet is discarded, but an ACK with ackNo = R_n is returned to the sender. This is needed to let the sender slide its window if some ACKs related to packets with seqNo < R_n were lost.

c. If a corrupted packet arrives, the packet is discarded.

Example 23.10

This example is similar to Example 23.8 (Figure 23.30) in which packet 1 is lost. We show how Selective-Repeat behaves in this case. Figure 23.35 shows the situation.

Figure 23.35 *Flow diagram for Example 23.10*



At the sender, packet 0 is transmitted and acknowledged. Packet 1 is lost. Packets 2 and 3 arrive out of order and are acknowledged. When the timer times out, packet 1 (the only unacknowledged packet) is resent and is acknowledged. The send window then slides.

At the receiver site we need to distinguish between the acceptance of a packet and its delivery to the application layer. At the second arrival, packet 2 arrives and is stored and marked (shaded slot), but it cannot be delivered because packet 1 is missing. At the next arrival, packet 3 arrives and is marked and stored, but still none of the packets can be delivered. Only at the last arrival, when finally a copy of packet 1 arrives, can packets 1, 2, and 3 be delivered to the application layer. There are two conditions for the delivery of packets to the application layer: First, a set of consecutive packets must have arrived. Second, the set starts from the beginning of the window. After the first arrival, there was only one packet and it started from the beginning of the window. After the last arrival, there are three packets and the first one starts from the beginning of the window. The key is that a reliable transport layer promises to deliver packets in order.

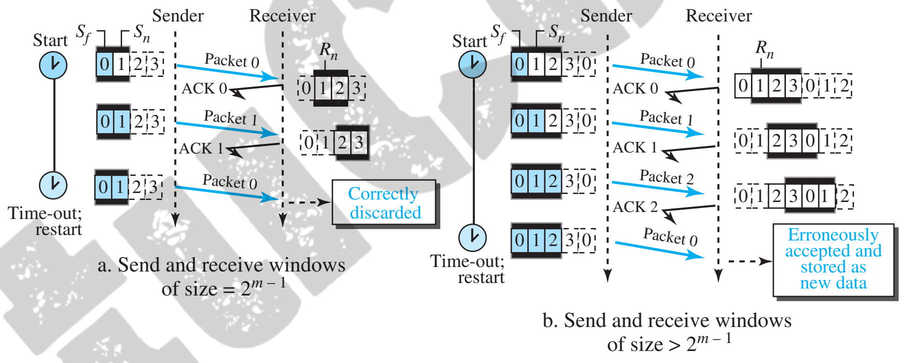
Window Sizes

We can now show why the size of the sender and receiver windows can be at most one-half of 2^m. For an example, we choose $m = 2$, which means the size of the window is $2^m/2$ or $2^{(m-1)} = 2$. Figure 23.36 compares a window size of 2 with a window size of 3.

If the size of the window is 2 and all acknowledgments are lost, the timer for packet 0 expires and packet 0 is resent. However, the window of the receiver is now expecting packet 2, not packet 0, so this duplicate packet is correctly discarded (the sequence number 0 is not in the window). When the size of the window is 3 and all acknowledgments are lost, the sender sends a duplicate of packet 0. However, this time, the window of the receiver expects to receive packet 0 (0 is part of the window), so it accepts packet 0, not as a duplicate, but as a packet in the next cycle. This is clearly an error.

In Selective-Repeat, the size of the sender and receiver window can be at most one-half of 2^m.

Figure 23.36 *Selective-Repeat, window size*

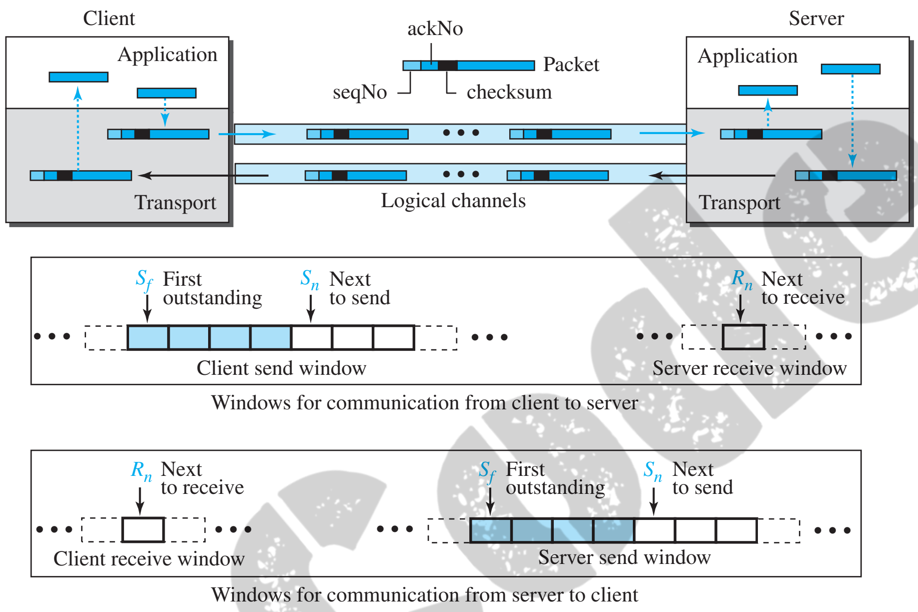

Sender
Receiver
Start
S_f
S_n
R_n
Packet 0
ACK 0
Packet 1
ACK 1
Packet 2
ACK 2
Time-out; restart
Correctly discarded
Erroneously accepted and stored as new data
a. Send and receive windows of size = 2^{m-1}
b. Send and receive windows of size > 2^{m-1}

23.2.5 Bidirectional Protocols: Piggybacking

The four protocols we discussed earlier in this section are all unidirectional: data packets flow in only one direction and acknowledgments travel in the other direction. In real life, data packets are normally flowing in both directions: from client to server and from server to client. This means that acknowledgments also need to flow in both directions. A technique called **piggybacking** is used to improve the efficiency of the bidirectional protocols. When a packet is carrying data from A to B, it can also carry acknowledgment feedback about arrived packets from B; when a packet is carrying data from B to A, it can also carry acknowledgment feedback about the arrived packets from A.

Figure 23.37 shows the layout for the GBN protocol implemented bidirectionally using piggybacking. The client and server each use two independent windows: send and receive.

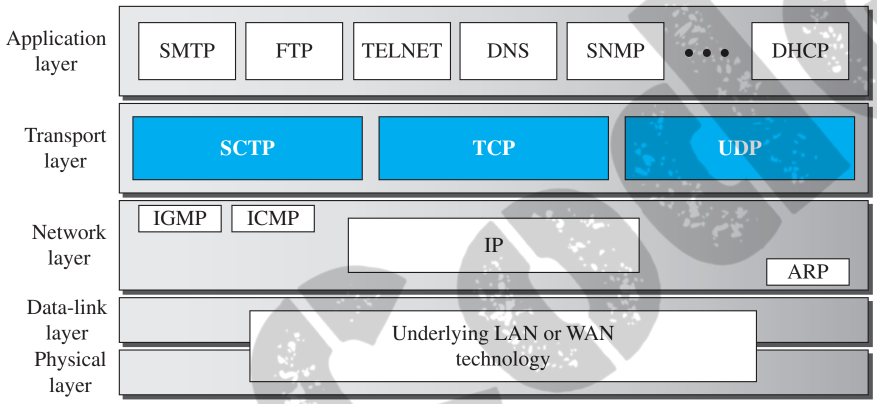
Figure 23.37 *Design of piggybacking in Go-Back-*N



24.1 INTRODUCTION

After discussing the general principle behind the transport layer in the previous chapter, we concentrate on the transport protocols in the Internet in this chapter. Figure 24.1 shows the position of these three protocols in the TCP/IP protocol suite.

Figure 24.1 *Position of transport-layer protocols in the TCP/IP protocol suite*



24.1.1 Services

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

UDP

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

TCP

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

SCTP

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

24.1.2 Port Numbers

As discussed in the previous chapter, a transport-layer protocol usually has several responsibilities. One is to create a process-to-process communication; these protocols use port numbers to accomplish this. Port numbers provide end-to-end addresses at the transport layer and allow multiplexing and demultiplexing at this layer, just as IP addresses do at the network layer. Table 24.1 gives some common port numbers for all three protocols we discuss in this chapter.

Table 24.1 *Some well-known ports used with UDP and TCP*

Port	*Protocol*	*UDP*	*TCP*	*SCTP*	*Description*
7	Echo	√	√	√	Echoes back a received datagram
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	FTP-data		√	√	File Transfer Protocol
21	FTP-21		√	√	File Transfer Protocol
23	TELNET		√	√	Terminal Network
25	SMTP		√	√	Simple Mail Transfer Protocol
53	DNS	√	√	√	Domain Name Service
67	DHCP	√	√	√	Dynamic Host Configuration Protocol
69	TFTP	√	√	√	Trivial File Transfer Protocol
80	HTTP		√	√	HyperText Transfer Protocol
111	RPC	√	√	√	Remote Procedure Call
123	NTP	√	√	√	Network Time Protocol
161	SNMP-server	√			Simple Network Management Protocol
162	SNMP-client	√			Simple Network Management Protocol

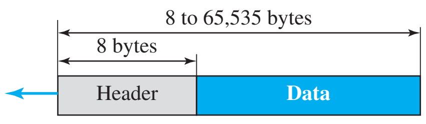
24.2 USER DATAGRAM PROTOCOL

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

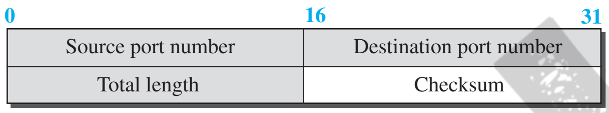
24.2.1 User Datagram

UDP packets, called ***user datagrams,*** have a fixed-size header of 8 bytes made of four fields, each of 2 bytes (16 bits). Figure 24.2 shows the format of a user datagram. The first two fields define the source and destination port numbers. The third field defines the total length of the user datagram, header plus data. The 16 bits can define a total length of 0 to 65,535 bytes. However, the total length needs to be less because a UDP user datagram is stored in an IP datagram with the total length of 65,535 bytes. The last field can carry the optional checksum (explained later).

Figure 24.2 *User datagram packet format*



a. UDP user datagram



b. Header format

Example 24.1

The following is the content of a UDP header in hexadecimal format.

CB84000D001C001C

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?

Solution

a. The source port number is the first four hexadecimal digits $(CB84)_{16}$, which means that the source port number is 52100.
b. The destination port number is the second four hexadecimal digits $(000D)_{16}$, which means that the destination port number is 13.
c. The third four hexadecimal digits $(001C)_{16}$ define the length of the whole UDP packet as 28 bytes.
d. The length of the data is the length of the whole packet minus the length of the header, or $28 - 8 = 20$ bytes.
e. Since the destination port number is 13 (well-known port), the packet is from the client to the server.
f. The client process is the Daytime (see Table 24.1).

24.2.2 UDP Services

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

Process-to-Process Communication

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

Connectionless Services

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

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

Flow Control

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

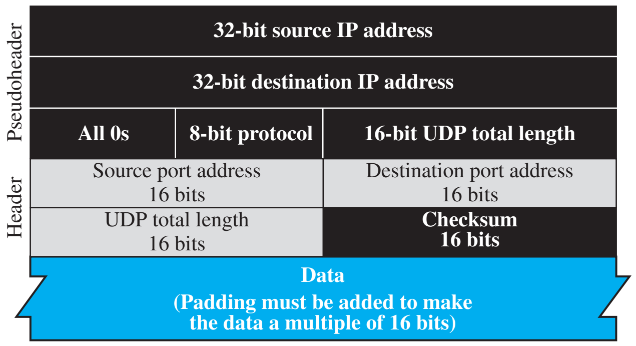
Error Control

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

Checksum

We discussed checksum and its calculation in Chapter 10. UDP checksum calculation includes three sections: a pseudoheader, the UDP header, and the data coming from the application layer. The *pseudoheader* is the part of the header of the IP packet (discussed in Chapter 19) in which the user datagram is to be encapsulated with some fields filled with 0s (see Figure 24.3).

Figure 24.3 *Pseudoheader for checksum calculation*


Pseudoheader
32-bit source IP address
32-bit destination IP address
All 0s
8-bit protocol
16-bit UDP total length
Header
Source port address 16 bits
Destination port address 16 bits
UDP total length 16 bits
Checksum 16 bits
Data
(Padding must be added to make the data a multiple of 16 bits)

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

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

Optional Inclusion of Checksum

The sender of a UDP packet can choose not to calculate the checksum. In this case, the checksum field is filled with all 0s before being sent. In the situation where the sender decides to calculate the checksum, but it happens that the result is all 0s, the checksum is changed to all 1s before the packet is sent. In other words, the sender complements the sum two times. Note that this does not create confusion because the value of the checksum is never all 1s in a normal situation (see the next example).

Example 24.2

What value is sent for the checksum in each one of the following hypothetical situations?

a. The sender decides not to include the checksum.
b. The sender decides to include the checksum, but the value of the sum is all 1s.
c. The sender decides to include the checksum, but the value of the sum is all 0s.

Solution

a. The value sent for the checksum field is all 0s to show that the checksum is not calculated.
b. When the sender complements the sum, the result is all 0s; the sender complements the result again before sending. The value sent for the checksum is all 1s. The second complement operation is needed to avoid confusion with the case in part a.
c. This situation never happens because it implies that the value of every term included in the calculation of the sum is all 0s, which is impossible; some fields in the pseudoheader have nonzero values.

Congestion Control

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

Encapsulation and Decapsulation

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

Queuing

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

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

Multiplexing and Demultiplexing

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

Comparison between UDP and Generic Simple Protocol

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

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

24.2.3 UDP Applications

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

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

UDP Features

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

Connectionless Service

As we mentioned previously, UDP is a connectionless protocol. Each UDP packet is independent from other packets sent by the same application program. This feature can be considered as an advantage or disadvantage depending on the application requirements. It is an advantage if, for example, a client application needs to send a short request to a server and to receive a short response. If the request and response can each fit in a single user datagram, a connectionless service may be preferable. The overhead to establish and close a connection may be significant in this case. In the connection-oriented service, to achieve the above goal, at least 9 packets are exchanged between the client and the server; in connectionless service only 2 packets are exchanged. The connectionless service provides less delay; the connection-oriented service creates more delay. If delay is an important issue for the application, the connectionless service is preferred.

Example 24.3

A client-server application such as DNS (see Chapter 26) uses the services of UDP because a client needs to send a short request to a server and to receive a quick response from it. The request and response can each fit in one user datagram. Since only one message is exchanged in each direction, the connectionless feature is not an issue; the client or server does not worry that messages are delivered out of order.

Example 24.4

A client-server application such as SMTP (see Chapter 27), which is used in electronic mail, cannot use the services of UDP because a user might send a long e-mail message, which could include multimedia (images, audio, or video). If the application uses UDP and the message does not fit in one user datagram, the message must be split by the application into different user datagrams. Here the connectionless service may create problems. The user datagrams may arrive and be delivered to the receiver application out of order. The receiver application may not be able to reorder the pieces. This means the connectionless service has a disadvantage for an application program that sends long messages. In SMTP, when we send a message, we do not expect to receive a response quickly (sometimes no response is required). This means that the extra delay inherent in connection-oriented service is not crucial for SMTP.

Lack of Error Control

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

Example 24.5

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

Example 24.6

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

Lack of Congestion Control

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

Typical Applications

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

- UDP is suitable for a process that requires simple request-response communication with little concern for flow and error control. It is not usually used for a process such as FTP that needs to send bulk data (see Chapter 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 27).
- UDP is used for some route updating protocols such as Routing Information Protocol (RIP) (see Chapter 20).
- UDP is normally used for interactive real-time applications that cannot tolerate uneven delay between sections of a received message (see Chapter 28).

24.3 TRANSMISSION CONTROL PROTOCOL

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

24.3.1 TCP Services

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

Process-to-Process Communication

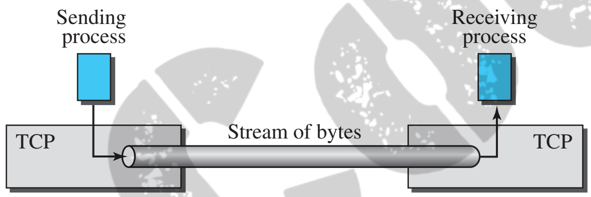
As with UDP, TCP provides process-to-process communication using port numbers. We have already given some of the port numbers used by TCP in Table 24.1 in the previous section.

Stream Delivery Service

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

TCP, on the other hand, allows the sending process to deliver data as a stream of bytes and allows the receiving process to obtain data as a stream of bytes. TCP creates an environment in which the two processes seem to be connected by an imaginary "tube" that carries their bytes across the Internet. This imaginary environment is depicted in Figure 24.4. The sending process produces (writes to) the stream and the receiving process consumes (reads from) it.

Figure 24.4 *Stream delivery*



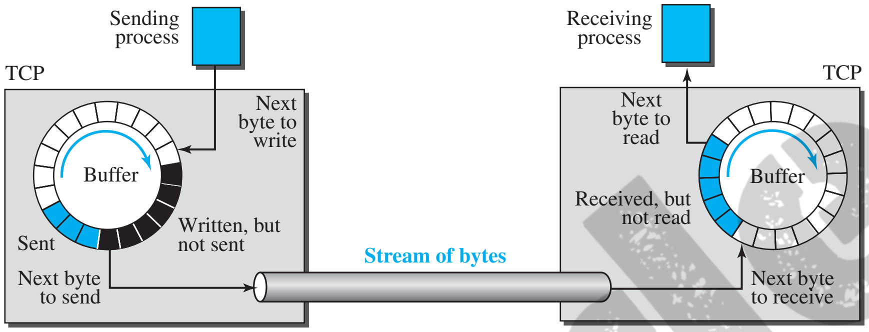
Sending and Receiving Buffers

Because the sending and the receiving processes may not necessarily write or read data at the same rate, TCP needs buffers for storage. There are two buffers, the sending buffer and the receiving buffer, one for each direction. We will see later that these buffers are also necessary for flow- and error-control mechanisms used by TCP. One way to implement a buffer is to use a circular array of 1-byte locations as shown in Figure 24.5. 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.

The figure shows the movement of the data in one direction. At the sender, the buffer has three types of chambers. The white section contains empty chambers that can be filled by the sending process (producer). The colored area holds bytes that have been sent but not yet acknowledged. The TCP sender keeps these bytes in the buffer until it receives an acknowledgment. The shaded area contains bytes to be sent by the sending TCP. However, as we will see later in this chapter, TCP may be able to send only part of this shaded section. This could be due to the slowness of the receiving process or to congestion in the network. Also note that, after the bytes in the colored chambers are acknowledged, the chambers are recycled and available for use by the sending process. This is why we show a circular buffer.

The operation of the buffer at the receiver is simpler. The circular buffer is divided into two areas (shown as white and colored). The white area contains empty chambers to be filled by bytes received from the network. The colored sections contain received

Figure 24.5 *Sending and receiving buffers*

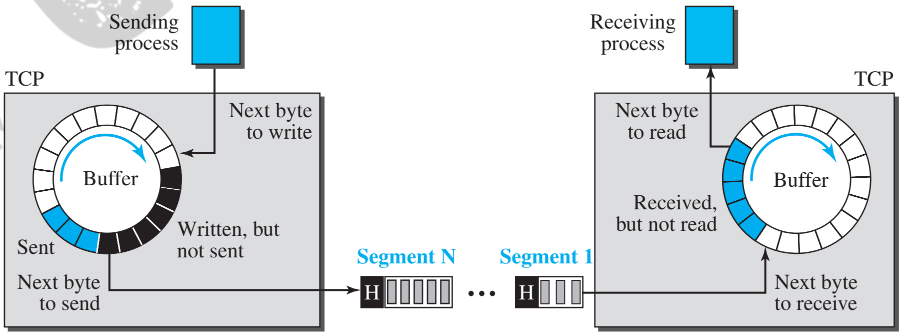


bytes that can be read by the receiving process. When a byte is read by the receiving process, the chamber is recycled and added to the pool of empty chambers.

Segments

Although buffering handles the disparity between the speed of the producing and consuming processes, we need one more step before we can send data. The network layer, as a service provider for TCP, needs to send data in packets, not as a stream of bytes. At the transport layer, TCP groups a number of bytes together into a packet called a *segment*. TCP adds a header to each segment (for control purposes) and delivers the segment to the network layer for transmission. The segments are encapsulated in an IP datagram and transmitted. This entire operation is transparent to the receiving process. Later we will see that segments may be received out of order, lost or corrupted, and resent. All of these are handled by the TCP receiver with the receiving application process unaware of TCP's activities. Figure 24.6 shows how segments are created from the bytes in the buffers.

Figure 24.6 *TCP segments*



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

Full-Duplex Communication

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

Multiplexing and Demultiplexing

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

Connection-Oriented Service

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

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

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

Reliable Service

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

24.3.2 TCP Features

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

Numbering System

Although the TCP software keeps track of the segments being transmitted or received, there is no field for a segment number value in the segment header. Instead, there are two fields, called the *sequence number* and the *acknowledgment number.* These two fields refer to a byte number and not a segment number.

Byte Number

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

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

Sequence Number

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

1. The sequence number of the first segment is the ISN (initial sequence number), which is a random number.
2. The sequence number of any other segment is the sequence number of the previous segment plus the number of bytes (real or imaginary) carried by the previous segment. Later, we show that some control segments are thought of as carrying one imaginary byte.

Example 24.7

Suppose a TCP connection is transferring a file of 5000 bytes. The first byte is numbered 10001. 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:	10001	**Range:**	10001	to	11000
Segment 2	→	Sequence Number:	11001	**Range:**	11001	to	12000
Segment 3	→	Sequence Number:	12001	**Range:**	12001	to	13000
Segment 4	→	Sequence Number:	13001	**Range:**	13001	to	14000
Segment 5	→	Sequence Number:	14001	**Range:**	14001	to	15000

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

When a segment carries a combination of data and control information (piggybacking), it uses a sequence number. If a segment does not carry user data, it does not logically define a sequence number. The field is there, but the value is not valid. However, some segments, when carrying only control information, need a sequence number to allow an acknowledgment from the receiver. These segments are used for connection establishment, termination, or abortion. Each of these segments consume one sequence number as though it carries one byte, but there are no actual data. We will elaborate on this issue when we discuss connections.

Acknowledgment Number

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

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

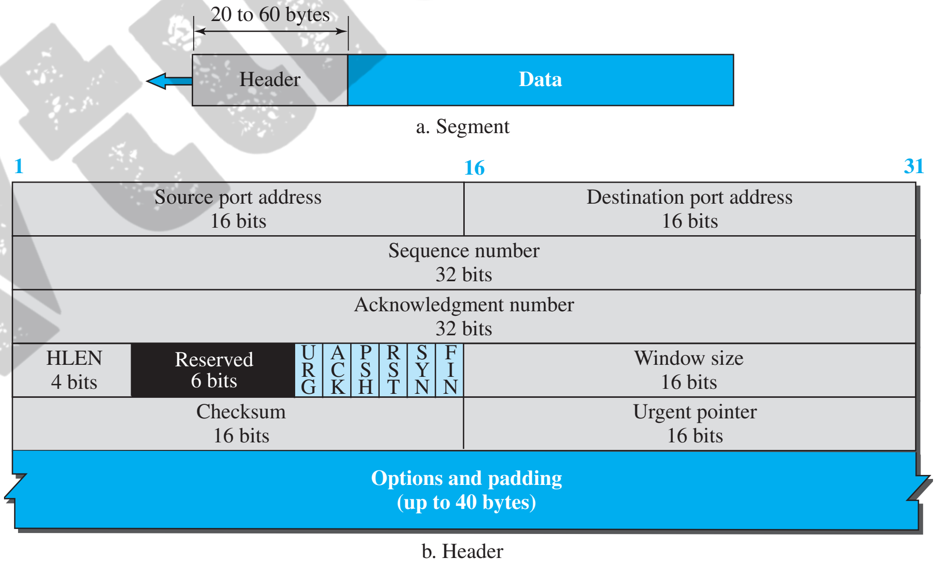
24.3.3 Segment

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

Format

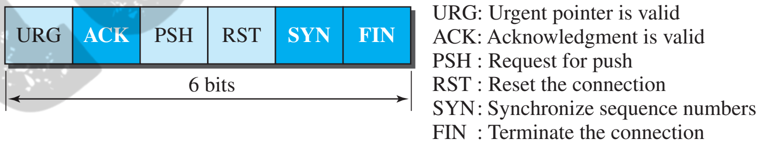
The format of a segment is shown in Figure 24.7. The segment consists of a header of 20 to 60 bytes, followed by data from the application program. The header is 20 bytes if there are no options and up to 60 bytes if it contains options. We will discuss some of the header fields in this section. The meaning and purpose of these will become clearer as we proceed through the section.

Figure 24.7 *TCP segment format*



- ***Source port address.*** This is a 16-bit field that defines the port number of the application program in the host that is sending the segment.
- ***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.
- ***Sequence number.*** This 32-bit field defines the number assigned to the first byte of data contained in this segment. As we said before, TCP is a stream transport protocol. To ensure connectivity, each byte to be transmitted is numbered. The sequence number tells the destination which byte in this sequence is the first byte in the segment. During connection establishment (discussed later) each party uses a random number generator to create an **initial sequence number** (ISN), which is usually different in each direction.
- ***Acknowledgment number.*** This 32-bit field defines the byte number that the receiver of the segment is expecting to receive from the other party. If the receiver of the segment has successfully received byte number x from the other party, it returns $x + 1$ as the acknowledgment number. Acknowledgment and data can be piggybacked together.
- ***Header length.*** This 4-bit field indicates the number of 4-byte words in the TCP header. The length of the header can be between 20 and 60 bytes. Therefore, the value of this field is always between 5 ($5 \times 4 = 20$) and 15 ($15 \times 4 = 60$).
- ***Control.*** This field defines 6 different control bits or flags, as shown in Figure 24.8. One or more of these bits can be set at a time. These bits enable flow control, connection establishment and termination, connection abortion, and the mode of data transfer in TCP. A brief description of each bit is shown in the figure. We will discuss them further when we study the detailed operation of TCP later in the chapter.

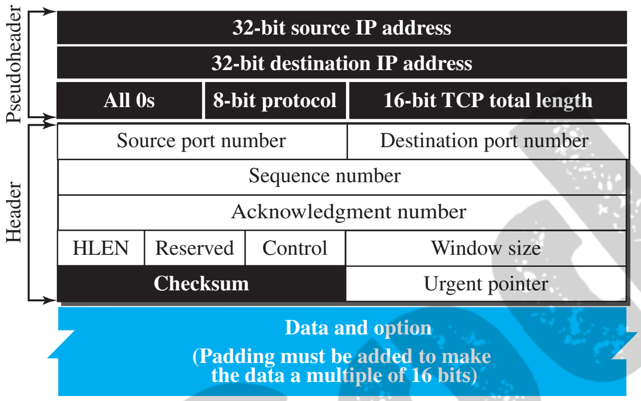
Figure 24.8 *Control field*


URG
ACK
PSH
RST
SYN
FIN
6 bits
URG: Urgent pointer is valid
ACK: Acknowledgment is valid
PSH : Request for push
RST : Reset the connection
SYN: Synchronize sequence numbers
FIN : Terminate the connection

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

purpose, is added to the segment. For the TCP pseudoheader, the value for the protocol field is 6. See Figure 24.9.

Figure 24.9 *Pseudoheader added to the TCP datagram*



The use of the checksum in TCP is mandatory.

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

Encapsulation

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

24.3.4 A TCP Connection

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

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

Connection Establishment

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

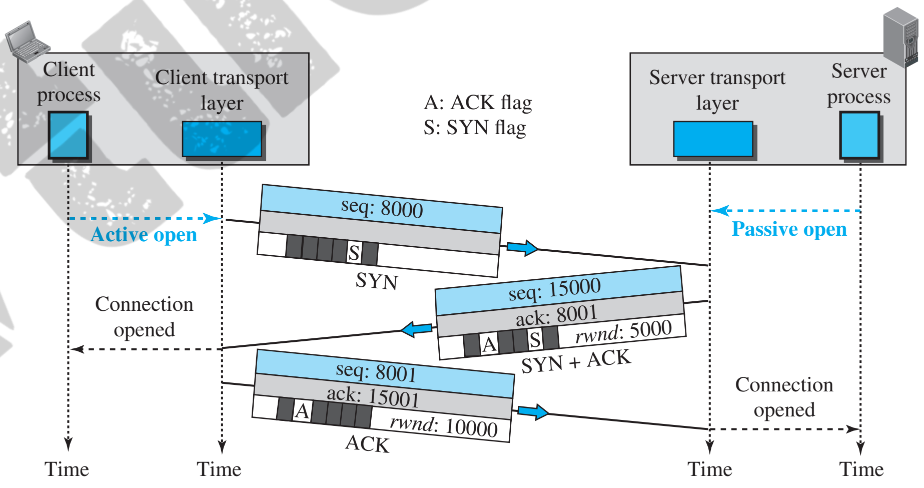
Three-Way Handshaking

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

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

The client program issues a request for an *active open*. A client that wishes to connect to an open server tells its TCP to connect to a particular server. TCP can now start the three-way handshaking process, as shown in Figure 24.10.

Figure 24.10 *Connection establishment using three-way handshaking*



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

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

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

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

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

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

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

SYN Flooding Attack

The connection establishment procedure in TCP is susceptible to a serious security problem called ***SYN flooding attack.*** This happens when one or more malicious attackers send a large number of SYN segments to a server pretending that each of them is coming from a different client by faking the source IP addresses in the datagrams. The server, assuming that the clients are issuing an active open, allocates the necessary resources, such as creating transfer control block (TCB) tables and setting timers. The

TCP server then sends the SYN + ACK segments to the fake clients, which are lost. When the server waits for the third leg of the handshaking process, however, resources are allocated without being used. If, during this short period of time, the number of SYN segments is large, the server eventually runs out of resources and may be unable to accept connection requests from valid clients. This SYN flooding attack belongs to a group of security attacks known as a ***denial of service attack,*** in which an attacker monopolizes a system with so many service requests that the system overloads and denies service to valid requests.

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

Data Transfer

After connection is established, bidirectional data transfer can take place. The client and server can send data and acknowledgments in both directions. We will study the rules of acknowledgment later in the chapter; for the moment, it is enough to know that data traveling in the same direction as an acknowledgment are carried on the same segment. The acknowledgment is piggybacked with the data. Figure 24.11 shows an example.

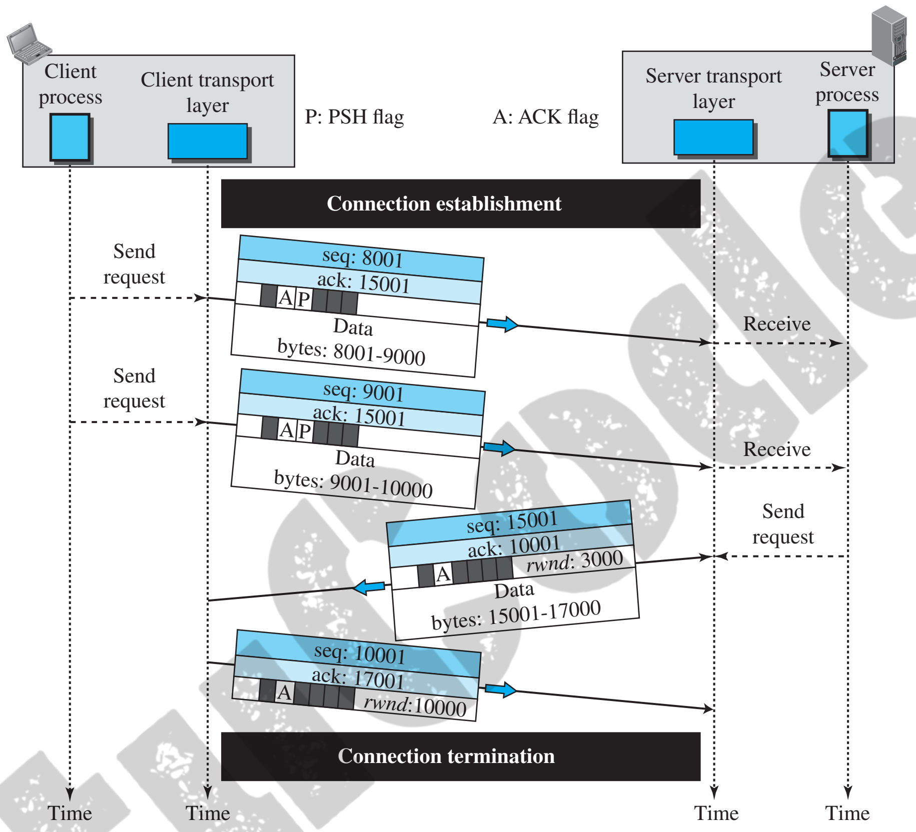
In this example, after a connection is established, the client sends 2,000 bytes of data in two segments. The server then sends 2,000 bytes in one segment. The client sends one more segment. The first three segments carry both data and acknowledgment, but the last segment carries only an acknowledgment because there is no more data to be sent. Note the values of the sequence and acknowledgment numbers. The data segments sent by the client have the PSH (push) flag set so that the server TCP knows to deliver data to the server process as soon as they are received. We discuss the use of this flag in more detail later. The segment from the server, on the other hand, does not set the push flag. Most TCP implementations have the option to set or not to set this flag.

Pushing Data

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

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

Figure 24.11 *Data transfer*



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

Urgent Data

TCP is a stream-oriented protocol. This means that the data is presented from the application program to TCP as a stream of bytes. Each byte of data has a position in the stream. However, there are occasions in which an application program needs to send *urgent* bytes, some bytes that need to be treated in a special way by the application at the other end. The solution is to send a segment with the URG bit set. The sending application program tells the sending TCP that the piece of data is urgent. The sending TCP creates a segment and inserts the urgent data at the beginning of the segment. The rest of the segment can contain normal data from the buffer. The urgent pointer field in the header defines the end of the urgent data (the last byte of urgent data). For example, if the segment sequence number is 15000 and the value of the urgent pointer is 200, the first byte of urgent data is the byte 15000 and the last byte is the byte 15200. The rest of the bytes in the segment (if present) are nonurgent.

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

Connection Termination

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

Three-Way Handshaking

Most implementations today allow *three-way handshaking* for connection termination, as shown in Figure 24.12.

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

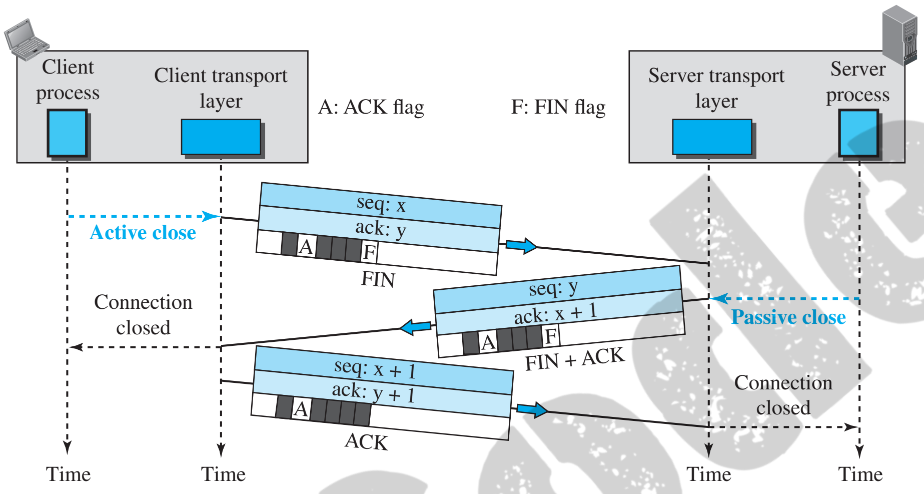
The FIN segment consumes one sequence number if it does not carry data.

2. The server TCP, after receiving the FIN segment, informs its process of the situation and sends the second segment, a FIN + ACK segment, to confirm the receipt of the FIN segment from the client and at the same time to announce the closing of the connection in the other direction. This segment can also contain the last chunk of data from the server. If it does not carry data, it consumes only one sequence number because it needs to be acknowledged.
3. The client TCP sends the last segment, an ACK segment, to confirm the receipt of the FIN segment from the TCP server. This segment contains the acknowledgment number, which is one plus the sequence number received in the FIN segment from the server. This segment cannot carry data and consumes no sequence numbers.

Half-Close

In TCP, one end can stop sending data while still receiving data. This is called a ***half-close.*** Either the server or the client can issue a half-close request. It can occur when the server needs all the data before processing can begin. A good example is sorting. When

Figure 24.12 *Connection termination using three-way handshaking*



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

the client sends data to the server to be sorted, the server needs to receive all the data before sorting can start. This means the client, after sending all data, can close the connection in the client-to-server direction. However, the server-to-client direction must remain open to return the sorted data. The server, after receiving the data, still needs time for sorting; its outbound direction must remain open. Figure 24.13 shows an example of a half-close.

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

After half-closing the connection, data can travel from the server to the client and acknowledgments can travel from the client to the server. The client cannot send any more data to the server.

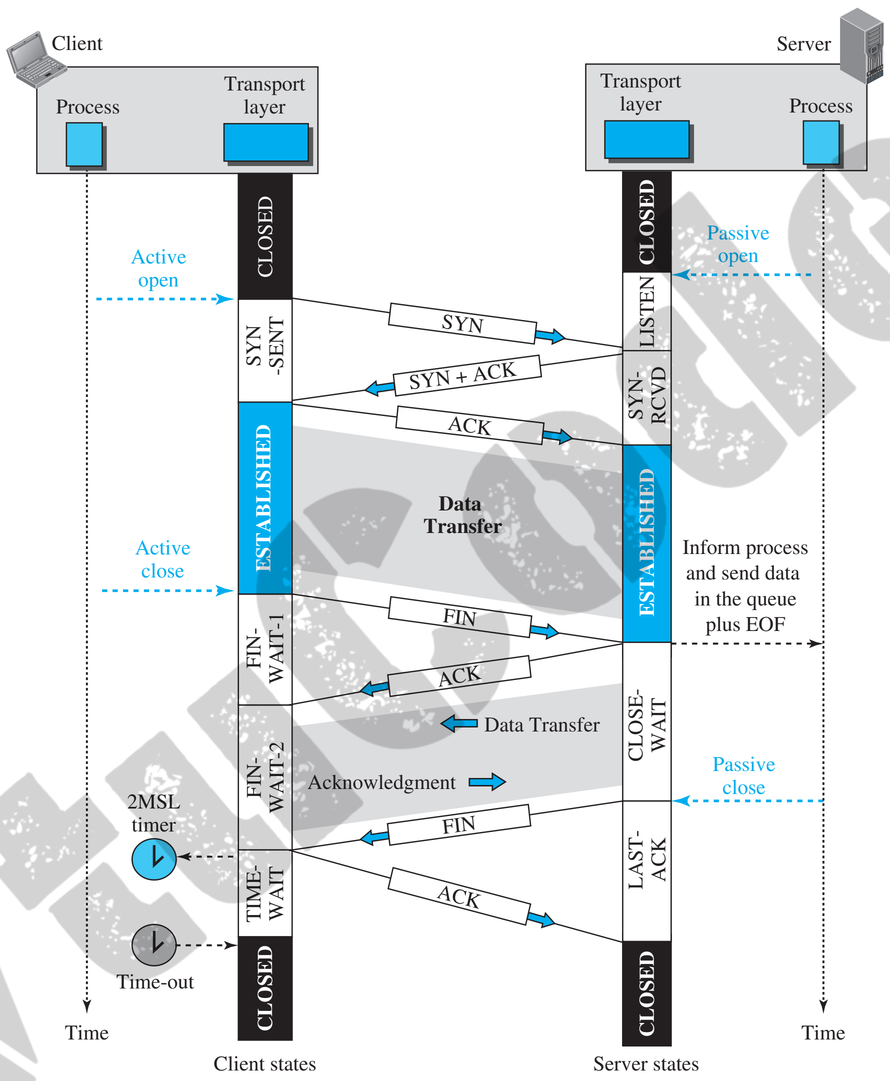
Connection Reset

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

24.3.5 State Transition Diagram

To keep track of all the different events happening during connection establishment, connection termination, and data transfer, TCP is specified as the finite state machine (FSM) as shown in Figure 24.14.

Figure 24.16 *Time-line diagram for a common scenario*



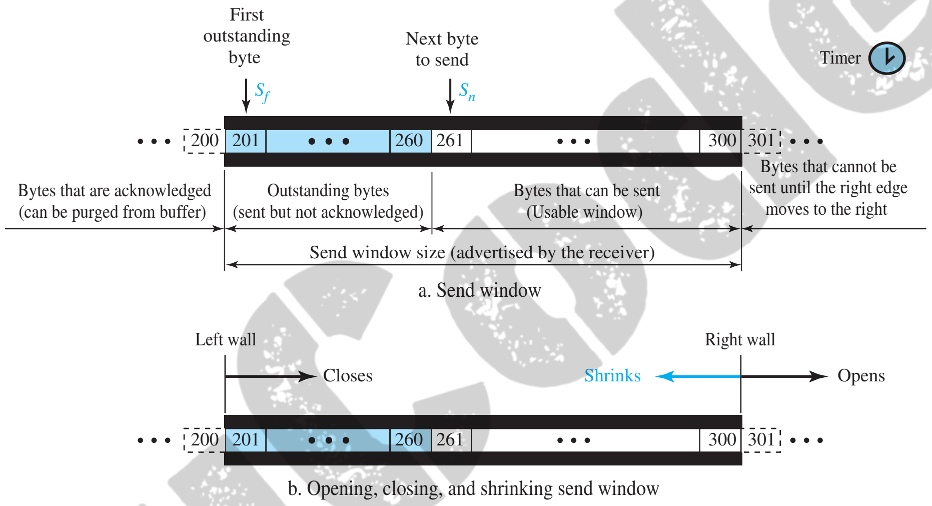
24.3.6 Windows in TCP

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

Send Window

Figure 24.17 shows an example of a send window. The window size is 100 bytes, but later we see that the send window size is dictated by the receiver (flow control) and the congestion in the underlying network (congestion control). The figure shows how a send window *opens*, *closes*, or *shrinks*.

Figure 24.17 *Send window in TCP*


First outstanding byte
S_f
Next byte to send
S_n
Timer
200
201
260
261
300
301
Bytes that are acknowledged (can be purged from buffer)
Outstanding bytes (sent but not acknowledged)
Bytes that can be sent (Usable window)
Bytes that cannot be sent until the right edge moves to the right
Send window size (advertised by the receiver)
a. Send window
Left wall
Closes
Right wall
Shrinks
Opens
b. Opening, closing, and shrinking send window

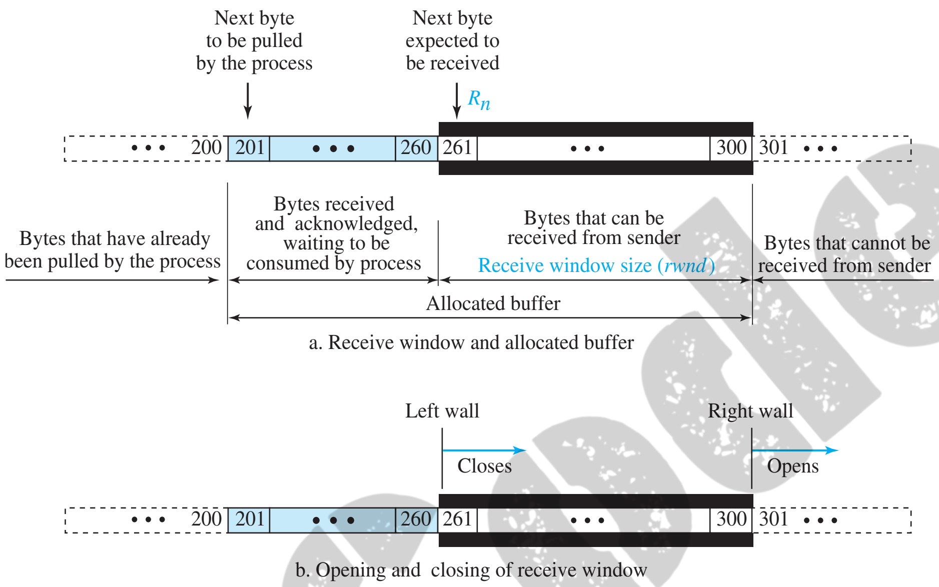
The send window in TCP is similar to the one used with the Selective-Repeat protocol, but with some differences:

1. One difference is the nature of entities related to the window. The window size in SR is the number of packets, but the window size in TCP is the number of bytes. Although actual transmission in TCP occurs segment by segment, the variables that control the window are expressed in bytes.
2. The second difference is that, in some implementations, TCP can store data received from the process and send them later, but we assume that the sending TCP is capable of sending segments of data as soon as it receives them from its process.
3. Another difference is the number of timers. The theoretical Selective-Repeat protocol may use several timers for each packet sent, but as mentioned before, the TCP protocol uses only one timer.

Receive Window

Figure 24.18 shows an example of a receive window. The window size is 100 bytes. The figure also shows how the receive window opens and closes; in practice, the window should never shrink.

Figure 24.18 *Receive window in TCP*


Next byte to be pulled by the process
Next byte expected to be received
R_n
200
201
260
261
300
301
Bytes that have already been pulled by the process
Bytes received and acknowledged, waiting to be consumed by process
Bytes that can be received from sender
Receive window size (*rwnd*)
Bytes that cannot be received from sender
Allocated buffer
a. Receive window and allocated buffer
Left wall
Right wall
Closes
Opens
b. Opening and closing of receive window

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

1. The first difference is that TCP allows the receiving process to pull data at its own pace. This means that part of the allocated buffer at the receiver may be occupied by bytes that have been received and acknowledged, but are waiting to be pulled by the receiving process. The receive window size is then always smaller than or equal to the buffer size, as shown in Figure 24.18. The receive window size determines the number of bytes that the receive window can accept from the sender before being overwhelmed (flow control). In other words, the receive window size, normally called *rwnd,* can be determined as:

   ***rwnd* = buffer size − number of waiting bytes to be pulled**

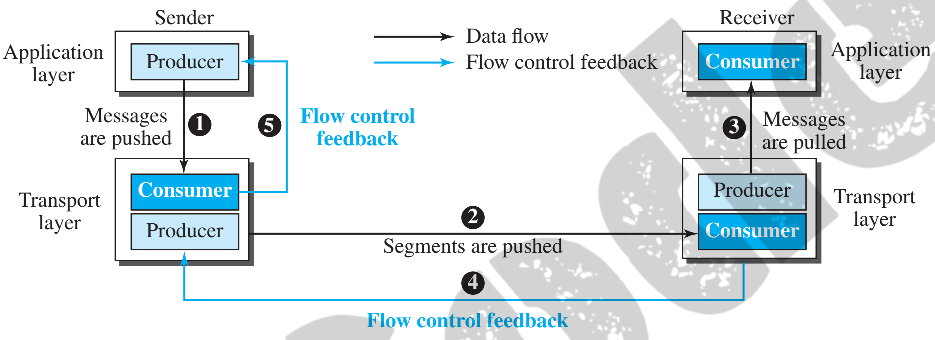
2. The second difference is the way acknowledgments are used in the TCP protocol. Remember that an acknowledgement in SR is selective, defining the uncorrupted packets that have been received. The major acknowledgment mechanism in TCP is a cumulative acknowledgment announcing the next expected byte to receive (in this way TCP looks like GBN, discussed earlier). The new version of TCP, however, uses both cumulative and selective acknowledgments; we will discuss these options on the book website.

24.3.7 Flow Control

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

Figure 24.19 shows unidirectional data transfer between a sender and a receiver; bidirectional data transfer can be deduced from the unidirectional process.

Figure 24.19 *Data flow and flow control feedbacks in TCP*



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

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

Opening and Closing Windows

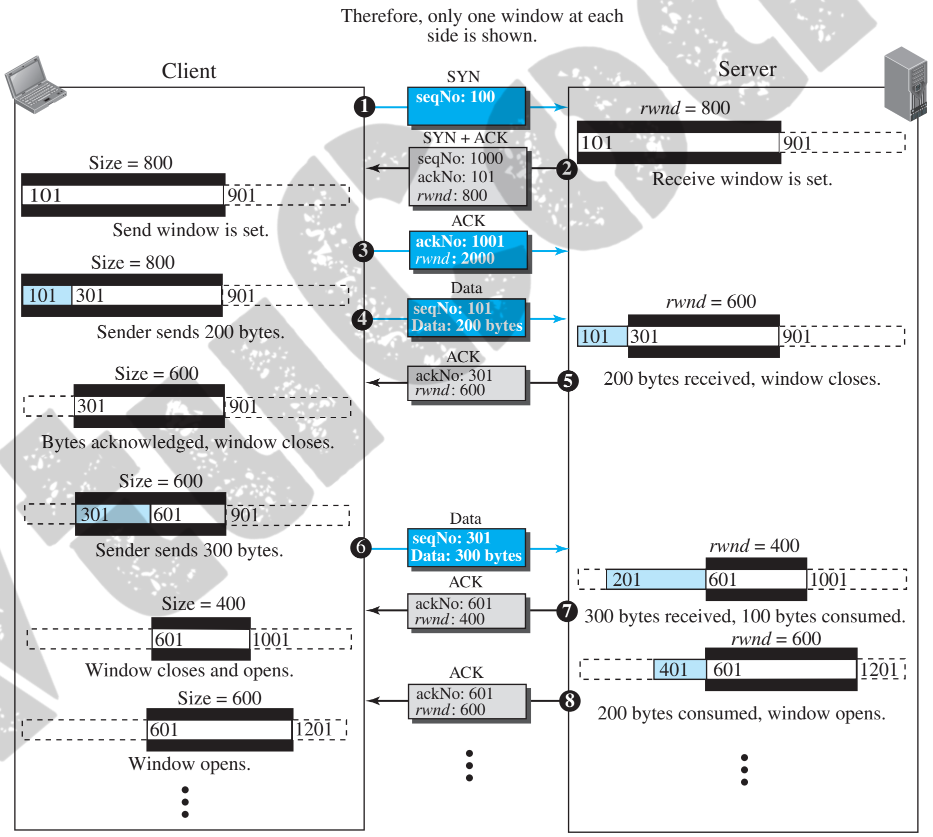
To achieve flow control, TCP forces the sender and the receiver to adjust their window sizes, although the size of the buffer for both parties is fixed when the connection is established. The receive window closes (moves its left wall to the right) when more bytes arrive from the sender; it opens (moves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

The opening, closing, and shrinking of the send window is controlled by the receiver. The send window closes (moves its left wall to the right) when a new acknowledgment allows it to do so. The send window opens (its right wall moves to the right) when the receive window size (*rwnd*) advertised by the receiver allows it to do so (new ackNo + new *rwnd* > last ackNo + last *rwnd*). The send window shrinks in the event this situation does not occur.

A Scenario

We show how the send and receive windows are set during the connection establishment phase, and how their situations will change during data transfer. Figure 24.20 shows a simple example of unidirectional data transfer (from client to server). For the time being, we ignore error control, assuming that no segment is corrupted, lost, duplicated, or has arrived out of order. Note that we have shown only two windows for unidirectional data transfer. Although the client defines server's window size of 2000 in the third segment, we have not shown that window because the communication is only unidirectional.

Figure 24.20 *An example of flow control*



Eight segments are exchanged between the client and server:

1. The first segment is from the client to the server (a SYN segment) to request connection. The client announces its initial seqNo = 100. When this segment arrives at the server, it allocates a buffer size of 800 (an assumption) and sets its window to cover the whole buffer (*rwnd* = 800). Note that the number of the next byte to arrive is 101.

2. The second segment is from the server to the client. This is an ACK + SYN segment. The segment uses ackNo = 101 to show that it expects to receive bytes starting from 101. It also announces that the client can set a buffer size of 800 bytes.
3. The third segment is the ACK segment from the client to the server. Note that the client has defined a *rwnd* of size 2000, but we do not use this value in our figure because the communication is only in one direction.
4. After the client has set its window with the size (800) dictated by the server, the process pushes 200 bytes of data. The TCP client numbers these bytes 101 to 300. It then creates a segment and sends it to the server. The segment shows the starting byte number as 101 and the segment carries 200 bytes. The window of the client is then adjusted to show that 200 bytes of data are sent but waiting for acknowledgment. When this segment is received at the server, the bytes are stored, and the receive window closes to show that the next byte expected is byte 301; the stored bytes occupy 200 bytes of buffer.
5. The fifth segment is the feedback from the server to the client. The server acknowledges bytes up to and including 300 (expecting to receive byte 301). The segment also carries the size of the receive window after decrease (600). The client, after receiving this segment, purges the acknowledged bytes from its window and closes its window to show that the next byte to send is byte 301. The window size, however, decreases to 600 bytes. Although the allocated buffer can store 800 bytes, the window cannot open (moving its right wall to the right) because the receiver does not let it.
6. Segment 6 is sent by the client after its process pushes 300 more bytes. The segment defines seqNo as 301 and contains 300 bytes. When this segment arrives at the server, the server stores them, but it has to reduce its window size. After its process has pulled 100 bytes of data, the window closes from the left for the amount of 300 bytes, but opens from the right for the amount of 100 bytes. The result is that the size is only reduced 200 bytes. The receiver window size is now 400 bytes.
7. In segment 7, the server acknowledges the receipt of data, and announces that its window size is 400. When this segment arrives at the client, the client has no choice but to reduce its window again and set the window size to the value of *rwnd* = 400 advertised by the server. The send window closes from the left by 300 bytes, and opens from the right by 100 bytes.
8. Segment 8 is also from the server after its process has pulled another 200 bytes. Its window size increases. The new *rwnd* value is now 600. The segment informs the client that the server still expects byte 601, but the server window size has expanded to 600. We need to mention that the sending of this segment depends on the policy imposed by the implementation. Some implementations may not allow advertisement of the *rwnd* at this time; the server then needs to receive some data before doing so. After this segment arrives at the client, the client opens its window by 200 bytes without closing it. The result is that its window size increases to 600 bytes.

Shrinking of Windows

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

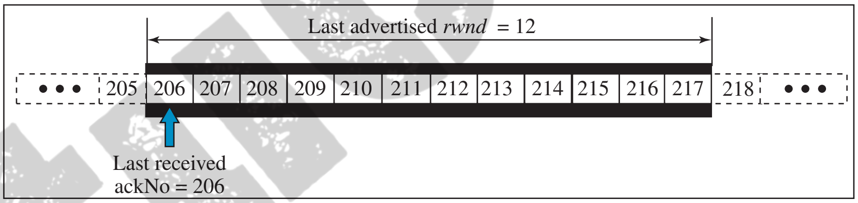
$$\textbf{new ackNo} + \textbf{new } \boldsymbol{rwnd} \geq \textbf{last ackNo} + \textbf{last } \boldsymbol{rwnd}$$

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

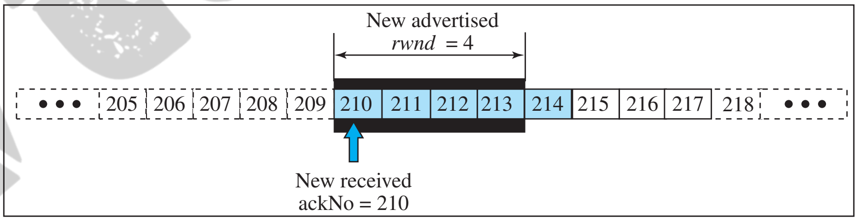
Example 24.8

Figure 24.21 shows the reason for this mandate.

Figure 24.21 *Example 24.8*



a. The window after the last advertisement



b. The window after the new advertisement; window has shrunk

Part a of the figure shows the values of the last acknowledgment and *rwnd*. Part b shows the situation in which the sender has sent bytes 206 to 214. Bytes 206 to 209 are acknowledged and purged. The new advertisement, however, defines the new value of *rwnd* as 4, in which $210 + 4 < 206 + 12$. When the send window shrinks, it creates a problem: byte 214, which has already been sent, is outside the window. The relation discussed before forces the receiver to maintain the right-hand wall of the window to be as shown in part a, because the receiver does not know which of the bytes 210 to 217 has already been sent. One way to prevent this situation is to let the receiver postpone its feedback until enough buffer locations are available in its window. In other words, the receiver should wait until more bytes are consumed by its process to meet the relationship described above.

Window Shutdown

We said that shrinking the send window by moving its right wall to the left is strongly discouraged. However, there is one exception: the receiver can temporarily shut down the window by sending a *rwnd* of 0. This can happen if for some reason the receiver does not want to receive any data from the sender for a while. In this case, the sender does not actually shrink the size of the window, but stops sending data until a new advertisement has arrived. As we will see later, even when the window is shut down by an order from the receiver, the sender can always send a segment with 1 byte of data. This is called *probing* and is used to prevent a deadlock (see the section on TCP timers).

Silly Window Syndrome

A serious problem can arise in the sliding window operation when either the sending application program creates data slowly or the receiving application program consumes data slowly, or both. Any of these situations results in the sending of data in very small segments, which reduces the efficiency of the operation. For example, if TCP sends segments containing only 1 byte of data, it means that a 41-byte datagram (20 bytes of TCP header and 20 bytes of IP header) transfers only 1 byte of user data. Here the overhead is 41/1, which indicates that we are using the capacity of the network very inefficiently. The inefficiency is even worse after accounting for the data-link layer and physical-layer overhead. This problem is called the ***silly window syndrome.*** For each site, we first describe how the problem is created and then give a proposed solution.

Syndrome Created by the Sender

The sending TCP may create a silly window syndrome if it is serving an application program that creates data slowly, for example, 1 byte at a time. The application program writes 1 byte at a time into the buffer of the sending TCP. If the sending TCP does not have any specific instructions, it may create segments containing 1 byte of data. The result is a lot of 41-byte segments that are traveling through an internet.

The solution is to prevent the sending TCP from sending the data byte by byte. The sending TCP must be forced to wait and collect data to send in a larger block. How long should the sending TCP wait? If it waits too long, it may delay the process. If it does not wait long enough, it may end up sending small segments. Nagle found an elegant solution. **Nagle's algorithm** is simple:

1. The sending TCP sends the first piece of data it receives from the sending application program even if it is only 1 byte.
2. After sending the first segment, the sending TCP accumulates data in the output buffer and waits until either the receiving TCP sends an acknowledgment or until enough data have accumulated to fill a maximum-size segment. At this time, the sending TCP can send the segment.
3. Step 2 is repeated for the rest of the transmission. Segment 3 is sent immediately if an acknowledgment is received for segment 2, or if enough data have accumulated to fill a maximum-size segment.

The elegance of Nagle's algorithm is in its simplicity and in the fact that it takes into account the speed of the application program that creates the data and the speed of the network that transports the data. If the application program is faster than the network, the segments are larger (maximum-size segments). If the application program is slower than the network, the segments are smaller (less than the maximum segment size).

Syndrome Created by the Receiver

The receiving TCP may create a silly window syndrome if it is serving an application program that consumes data slowly, for example, 1 byte at a time. Suppose that the sending application program creates data in blocks of 1 kilobyte, but the receiving application program consumes data 1 byte at a time. Also suppose that the input buffer of the receiving TCP is 4 kilobytes. The sender sends the first 4 kilobytes of data. The receiver stores it in its buffer. Now its buffer is full. It advertises a window size of zero, which means the sender should stop sending data. The receiving application reads the first byte of data from the input buffer of the receiving TCP. Now there is 1 byte of space in the incoming buffer. The receiving TCP announces a window size of 1 byte, which means that the sending TCP, which is eagerly waiting to send data, takes this advertisement as good news and sends a segment carrying only 1 byte of data. The procedure will continue. One byte of data is consumed and a segment carrying 1 byte of data is sent. Again we have an efficiency problem and the silly window syndrome.

Two solutions have been proposed to prevent the silly window syndrome created by an application program that consumes data more slowly than they arrive. **Clark's solution** is to send an acknowledgment as soon as the data arrive, but to announce a window size of zero until either there is enough space to accommodate a segment of maximum size or until at least half of the receive buffer is empty. The second solution is to delay sending the acknowledgment. This means that when a segment arrives, it is not acknowledged immediately. The receiver waits until there is a decent amount of space in its incoming buffer before acknowledging the arrived segments. The delayed acknowledgment prevents the sending TCP from sliding its window. After the sending TCP has sent the data in the window, it stops. This kills the syndrome.

Delayed acknowledgment also has another advantage: it reduces traffic. The receiver does not have to acknowledge each segment. However, there also is a disadvantage in that the delayed acknowledgment may result in the sender unnecessarily retransmitting the unacknowledged segments.

The protocol balances the advantages and disadvantages. It now defines that the acknowledgment should not be delayed by more than 500 ms.

24.3.8 Error Control

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

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

Checksum

Each segment includes a checksum field, which is used to check for a corrupted segment. If a segment is corrupted, as detected by an invalid checksum, the segment is discarded by the destination TCP and is considered as lost. TCP uses a 16-bit checksum that is mandatory in every segment. We discuss checksum calculation in Chapter 10.

Acknowledgment

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

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

Acknowledgment Type

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

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

Selective Acknowledgment (SACK) More and more implementations are adding another type of acknowledgment called *selective acknowledgment,* or SACK. A SACK does not replace an ACK, but reports additional information to the sender. A SACK reports a block of bytes that is out of order, and also a block of bytes that is duplicated, i.e., received more than once. However, since there is no provision in the TCP header for adding this type of information, SACK is implemented as an option at the end of the TCP header. We discuss this new feature when we discuss options in TCP on the book website.

Generating Acknowledgments

When does a receiver generate acknowledgments? During the evolution of TCP, several rules have been defined and used by several implementations. We give the most common rules here. The order of a rule does not necessarily define its importance.

1. When end A sends a data segment to end B, it must include (piggyback) an acknowledgment that gives the next sequence number it expects to receive. This rule decreases the number of segments needed and therefore reduces traffic.
2. When the receiver has no data to send and it receives an in-order segment (with expected sequence number) and the previous segment has already been acknowledged, the receiver delays sending an ACK segment until another segment

arrives or until a period of time (normally 500 ms) has passed. In other words, the receiver needs to delay sending an ACK segment if there is only one outstanding in-order segment. This rule reduces ACK segments.

3. When a segment arrives with a sequence number that is expected by the receiver, and the previous in-order segment has not been acknowledged, the receiver immediately sends an ACK segment. In other words, there should not be more than two in-order unacknowledged segments at any time. This prevents the unnecessary retransmission of segments that may create congestion in the network.
4. When a segment arrives with an out-of-order sequence number that is higher than expected, the receiver immediately sends an ACK segment announcing the sequence number of the next expected segment. This leads to the *fast retransmission* of missing segments (discussed later).
5. When a missing segment arrives, the receiver sends an ACK segment to announce the next sequence number expected. This informs the receiver that segments reported missing have been received.
6. If a duplicate segment arrives, the receiver discards the segment, but immediately sends an acknowledgment indicating the next in-order segment expected. This solves some problems when an ACK segment itself is lost.

Retransmission

The heart of the error control mechanism is the retransmission of segments. When a segment is sent, it is stored in a queue until it is acknowledged. When the retransmission timer expires or when the sender receives three duplicate ACKs for the first segment in the queue, that segment is retransmitted.

Retransmission after RTO

The sending TCP maintains one **retransmission time-out (RTO)** for each connection. When the timer matures, i.e. times out, TCP resends the segment in the front of the queue (the segment with the smallest sequence number) and restarts the timer. Note that again we assume $S_f < S_n$. We will see later that the value of RTO is dynamic in TCP and is updated based on the **round-trip time (RTT)** of segments. RTT is the time needed for a segment to reach a destination and for an acknowledgment to be received.

Retransmission after Three Duplicate ACK Segments

The previous rule about retransmission of a segment is sufficient if the value of RTO is not large. To expedite service throughout the Internet by allowing senders to retransmit without waiting for a time out, most implementations today follow the three duplicate ACKs rule and retransmit the missing segment immediately. This feature is called ***fast retransmission.*** In this version, if three duplicate acknowledgments (i.e., an original ACK plus three exactly identical copies) arrive for a segment, the next segment is retransmitted without waiting for the time-out. We come back to this feature later in the chapter.

Out-of-Order Segments

TCP implementations today do not discard out-of-order segments. They store them temporarily and flag them as out-of-order segments until the missing segments arrive.

Note, however, that out-of-order segments are never 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 TCP guarantees that no out-of-order data are delivered to the process.

FSMs for Data Transfer in TCP

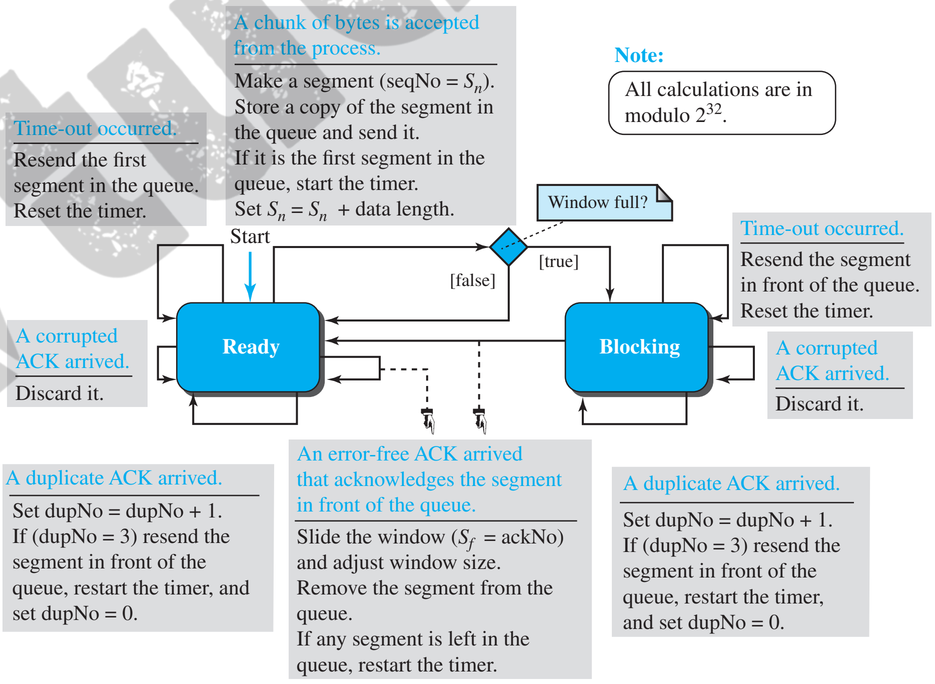
Data transfer in TCP is close to the Selective-Repeat protocol with a slight similarity to GBN. Since TCP accepts out-of-order segments, TCP can be thought of as behaving more like the SR protocol, but since the original acknowledgments are cumulative, it looks like GBN. However, if the TCP implementation uses SACKs, then TCP is closest to SR.

TCP can best be modeled as a Selective-Repeat protocol.

Sender-Side FSM

Let us show a simplified FSM for the sender side of the TCP protocol similar to the one we discussed for the SR protocol, but with some changes specific to TCP. We assume that the communication is unidirectional and the segments are acknowledged using ACK segments. We also ignore selective acknowledgments and congestion control for the moment. Figure 24.22 shows the simplified FSM for the sender site. Note that the

Figure 24.22 *Simplified FSM for the TCP sender side*



A chunk of bytes is accepted from the process.
Make a segment (seqNo = S_n). Store a copy of the segment in the queue and send it. If it is the first segment in the queue, start the timer. Set $S_n = S_n$ + data length.
Note:
All calculations are in modulo 2^{32}.
Time-out occurred.
Resend the first segment in the queue. Reset the timer.
Window full?
Start
[false]
[true]
Ready
Blocking
Time-out occurred.
Resend the segment in front of the queue. Reset the timer.
A corrupted ACK arrived.
Discard it.
A corrupted ACK arrived.
Discard it.
A duplicate ACK arrived.
Set dupNo = dupNo + 1. If (dupNo = 3) resend the segment in front of the queue, restart the timer, and set dupNo = 0.
An error-free ACK arrived that acknowledges the segment in front of the queue.
Slide the window (S_f = ackNo) and adjust window size. Remove the segment from the queue. If any segment is left in the queue, restart the timer.
A duplicate ACK arrived.
Set dupNo = dupNo + 1. If (dupNo = 3) resend the segment in front of the queue, restart the timer, and set dupNo = 0.

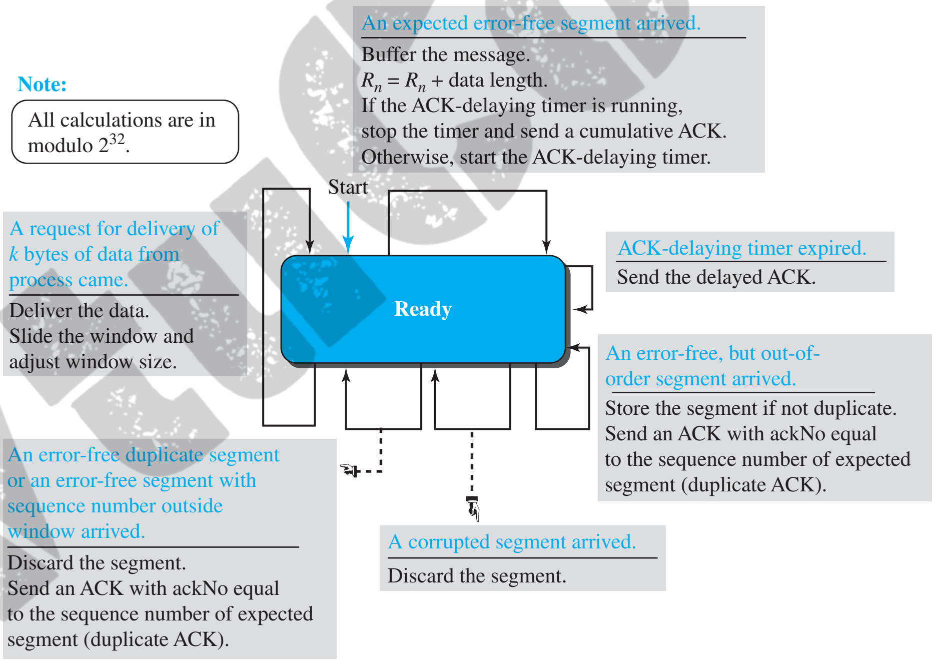
FSM is rudimentary; it does not include issues such as silly window syndrome (Nagle's algorithm) or window shutdown. It defines a unidirectional communication, ignoring all issues that affect bidirectional communication.

There are some differences between the FSM in Figure 24.22 and the one we discussed for an SR protocol. One difference is the fast transmission (three duplicate ACKs). The other is the window size adjustment based on the value of *rwnd* (ignoring congestion control for the moment).

Receiver-Side FSM

Now let us show a simplified FSM for the receiver-side TCP protocol similar to the one we discuss for the SR protocol, but with some changes specific to TCP. We assume that the communication is unidirectional and the segments are acknowledged using ACK segments. We also ignore the selective acknowledgment and congestion control for the moment. Figure 24.23 shows the simplified FSM for the receiver. Note that we ignore some issues such as silly window syndrome (Clark's solution) and window shutdown.

Figure 24.23 *Simplified FSM for the TCP receiver side*


Note:
All calculations are in modulo 2^{32}.
An expected error-free segment arrived.
Buffer the message.
$R_n = R_n$ + data length.
If the ACK-delaying timer is running, stop the timer and send a cumulative ACK.
Otherwise, start the ACK-delaying timer.
Start
Ready
A request for delivery of k bytes of data from process came.
Deliver the data.
Slide the window and adjust window size.
ACK-delaying timer expired.
Send the delayed ACK.
An error-free, but out-of-order segment arrived.
Store the segment if not duplicate.
Send an ACK with ackNo equal to the sequence number of expected segment (duplicate ACK).
An error-free duplicate segment or an error-free segment with sequence number outside window arrived.
Discard the segment.
Send an ACK with ackNo equal to the sequence number of expected segment (duplicate ACK).
A corrupted segment arrived.
Discard the segment.

Again, there are some differences between this FSM and the one we discussed for an SR protocol. One difference is the ACK delaying in unidirectional communication. The other difference is the sending of duplicate ACKs to allow the sender to implement fast retransmission policy.

We also need to emphasize that bidirectional FSM for the receiver is not as simple as the one for SR; we need to consider some policies such as sending an immediate ACK if the receiver has some data to return.

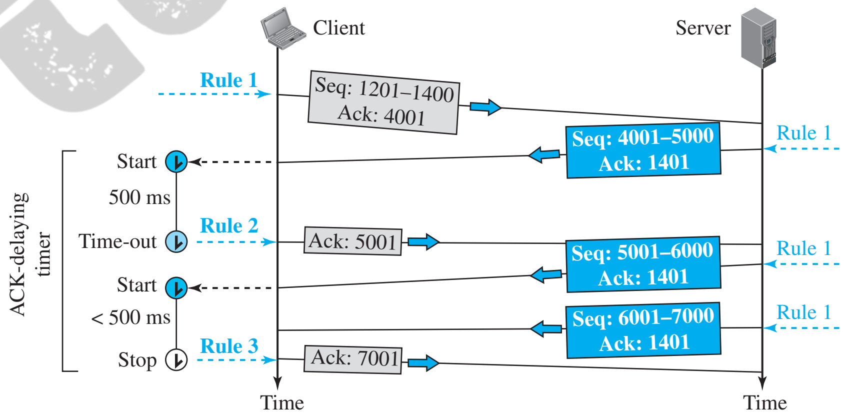
Some Scenarios

In this section we give some examples of scenarios that occur during the operation of TCP, considering only error control issues. 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 shown in Figure 24.24. The client TCP sends one segment; the server TCP sends three. The figure shows which rule applies to each acknowledgment. At the server site, only rule 1 applies. 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 needs to send only an ACK segment. However, according to rule 2, the acknowledgment needs to be delayed for 500 ms to see if any more segments arrive. When the ACK-delaying timer matures, it triggers an acknowledgment. This is because the client has no knowledge of whether other segments are coming; it cannot delay the acknowledgment forever. When the next segment arrives, another ACK-delaying timer is set. However, before it matures, the third segment arrives. The arrival of the third segment triggers another acknowledgment based on rule 3. We have not shown the RTO timer because no segment is lost or delayed. We just assume that the RTO timer performs its duty.

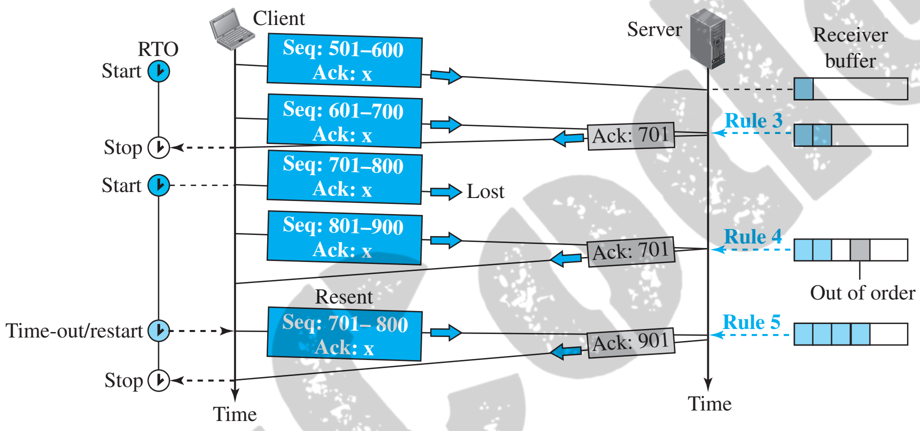
Figure 24.24 *Normal operation*



Lost Segment

In this scenario, we show what happens when a segment is lost or corrupted. A lost or corrupted segment is 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 24.25 shows a situation in which a segment is lost (probably discarded by some router in the network due to congestion).

Figure 24.25 *Lost segment*



We are assuming that data transfer is unidirectional: one site is sending, the other receiving. In our scenario, the sender sends segments 1 and 2, which are acknowledged immediately by an ACK (rule 3). 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 (rule 4). 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.

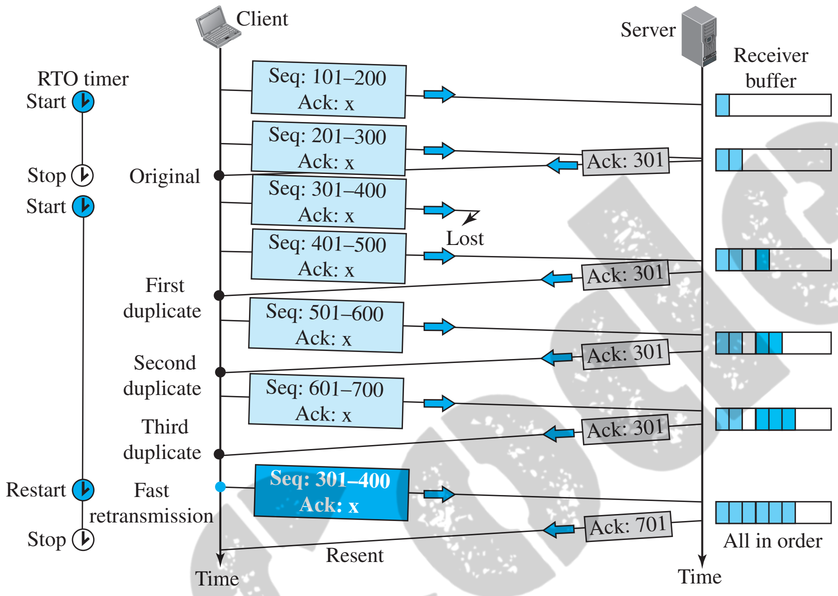
The sender TCP keeps one RTO timer for the whole period of connection. When the third segment times out, the sending TCP resends segment 3, which arrives this time and is acknowledged properly (rule 5).

Fast Retransmission

In this scenario, we want to show *fast retransmission.* Our scenario is the same as the second except that the RTO has a larger value (see Figure 24.26).

Each time the receiver receives a subsequent segment, it triggers an acknowledgment (rule 4). The sender receives four acknowledgments with the same value (three duplicates). Although the timer has not matured, the rule for fast retransmission

Figure 24.26 *Fast retransmission*



requires that segment 3, the segment that is expected by all of these duplicate acknowledgments, be resent immediately. After resending this segment, the timer is restarted.

Delayed Segment

The fourth scenario features a delayed segment. TCP uses the services of IP, which is a connectionless protocol. Each IP datagram encapsulating a TCP segment may reach the final destination through a different route with a different delay. Hence TCP segments may be delayed. Delayed segments sometimes may time out and be resent. If the delayed segment arrives after it has been resent, it is considered a duplicate segment and discarded.

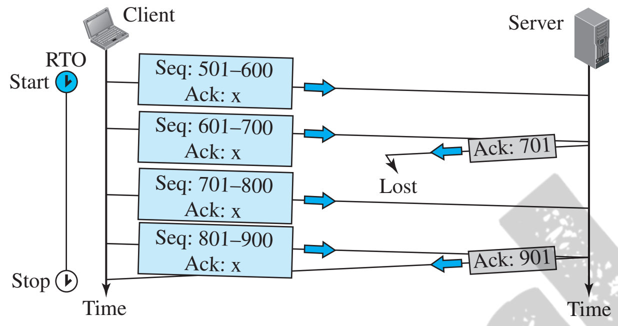
Duplicate Segment

A duplicate segment can be created, for example, by a sending TCP when a segment is delayed and treated as lost by the receiver. Handling the duplicated segment is a simple process for the destination TCP. The destination TCP expects a continuous stream of bytes. When a segment arrives that contains a sequence number equal to an already received and stored segment, it is discarded. An ACK is sent with ackNo defining the expected segment.

Automatically Corrected Lost ACK

This scenario shows a situation in which information in a lost acknowledgment is contained in the next one, a key advantage of using cumulative acknowledgments. Figure 24.27 shows a lost acknowledgment sent by the receiver of data. In the TCP acknowledgment mechanism, a lost acknowledgment may not even be noticed by the source TCP. TCP uses cumulative acknowledgment. We can say that the next acknowledgment automatically corrects the loss of the previous acknowledgment.

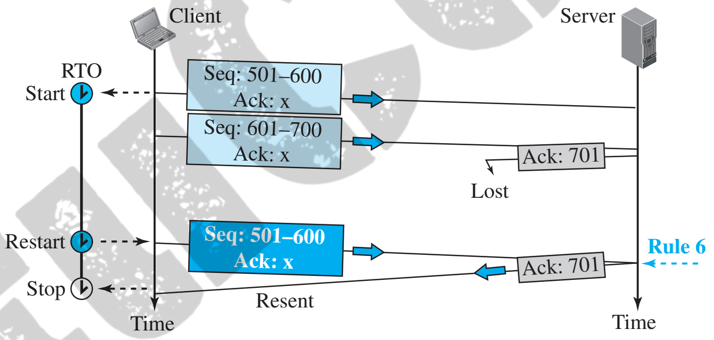
Figure 24.27 *Lost acknowledgment*



Lost Acknowledgment Corrected by Resending a Segment

Figure 24.28 shows a scenario in which an acknowledgment is lost.

Figure 24.28 *Lost acknowledgment corrected by resending a segment*



If the next acknowledgment is delayed for a long time or there is no next acknowledgment (the lost acknowledgment is the last one sent), the correction is triggered by the RTO timer. A duplicate segment is the result. When the receiver receives a duplicate segment, it discards it and resends the last ACK immediately to inform the sender that the segment or segments have been received.

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

Deadlock Created by Lost Acknowledgment

There is one situation in which loss of an acknowledgment may result in system deadlock. This is the case in which a receiver sends an acknowledgment with *rwnd* set to 0 and requests that the sender shut down its window temporarily. After a while, the receiver wants to remove the restriction; however, if it has no data to send, it sends an ACK segment and removes the restriction with a nonzero value for *rwnd*. A problem arises if this acknowledgment is lost. The sender is waiting for an acknowledgment that announces the nonzero *rwnd*. The receiver thinks that the sender has received this and is waiting for data. This situation is called a ***deadlock;*** each end is waiting for a response from the other end and nothing is happening. A retransmission timer is not set. To prevent deadlock, a persistence timer was designed that we will study later in the chapter.

Lost acknowledgments may create deadlock if they are not properly handled.

24.3.9 TCP Congestion Control

TCP uses different policies to handle the congestion in the network. We describe these policies in this section.

Congestion Window

When we discussed flow control in TCP, we mentioned that the size of the send window is controlled by the receiver using the value of *rwnd,* which is advertised in each segment traveling in the opposite direction. The use of this strategy guarantees that the receive window is never overflowed with the received bytes (no end congestion). This, however, does not mean that the intermediate buffers, buffers in the routers, do not become congested. A router may receive data from more than one sender. No matter how large the buffers of a router may be, it may be overwhelmed with data, which results in dropping some segments sent by a specific TCP sender. In other words, there is no congestion at the other end, but there may be congestion in the middle. TCP needs to worry about congestion in the middle because many segments lost may seriously affect the error control. More segment loss means resending the same segments again, resulting in worsening the congestion, and finally the collapse of the communication.

TCP is an end-to-end protocol that uses the service of IP. The congestion in the router is in the IP territory and should be taken care of by IP. However, as we discussed in Chapters 18 and 19, IP is a simple protocol with no congestion control. TCP, itself, needs to be responsible for this problem.

TCP cannot ignore the congestion in the network; it cannot aggressively send segments to the network. The result of such aggressiveness would hurt the TCP itself, as we mentioned before. TCP cannot be very conservative, either, sending a small number of segments in each time interval, because this means not utilizing the available bandwidth of the network. TCP needs to define policies that accelerate the data transmission when there is no congestion and decelerate the transmission when congestion is detected.

To control the number of segments to transmit, TCP uses another variable called a *congestion window, cwnd,* whose size is controlled by the congestion situation in the network (as we will explain shortly). The *cwnd* variable and the *rwnd* variable together define the size of the send window in TCP. The first is related to the congestion in the middle (network); the second is related to the congestion at the end. The actual size of the window is the minimum of these two.

Actual window size = minimum (*rwnd*, *cwnd*)

Congestion Detection

Before discussing how the value of *cwnd* should be set and changed, we need to describe how a TCP sender can detect the possible existence of congestion in the network. The TCP sender uses the occurrence of two events as signs of congestion in the network: time-out and receiving three duplicate ACKs.

The first is the *time-out*. If a TCP sender does not receive an ACK for a segment or a group of segments before the time-out occurs, it assumes that the corresponding segment or segments are lost and the loss is due to congestion.

Another event is the receiving of three duplicate ACKs (four ACKs with the same acknowledgment number). Recall that when a TCP receiver sends a duplicate ACK, it is the sign that a segment has been delayed, but sending three duplicate ACKs is the sign of a missing segment, which can be due to congestion in the network. However, the congestion in the case of three duplicate ACKs can be less severe than in the case of time-out. When a receiver sends three duplicate ACKs, it means that one segment is missing, but three segments have been received. The network is either slightly congested or has recovered from the congestion.

We will show later that an earlier version of TCP, called *Taho TCP,* treated both events (time-out and three duplicate ACKs) similarly, but the later version of TCP, called *Reno TCP,* treats these two signs differently.

A very interesting point in TCP congestion is that the TCP sender uses only one feedback from the other end to detect congestion: ACKs. The lack of regular, timely receipt of ACKs, which results in a time-out, is the sign of a strong congestion; the receiving of three duplicate ACKs is the sign of a weak congestion in the network.

Congestion Policies

TCP's general policy for handling congestion is based on three algorithms: slow start, congestion avoidance, and fast recovery. We first discuss each algorithm before showing how TCP switches from one to the other in a connection.

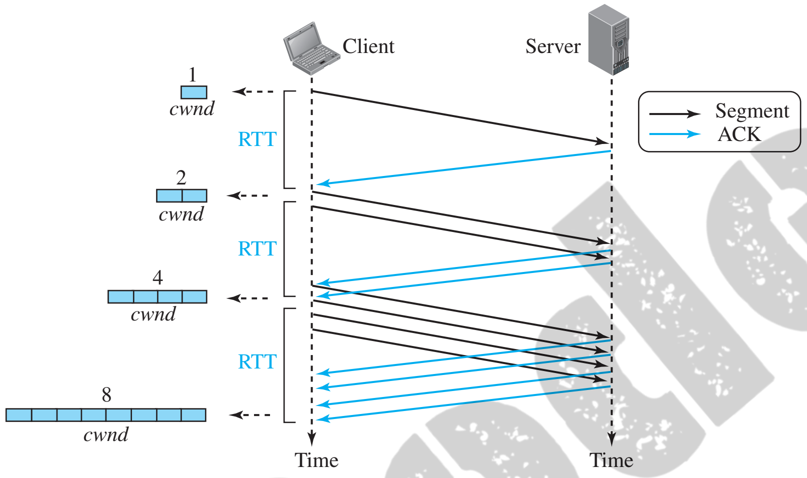
Slow Start: Exponential Increase

The **slow-start algorithm** is based on the idea that the size of the congestion window (*cwnd*) starts with one maximum segment size (MSS), but it increases one MSS each time an acknowledgment arrives. As we discussed before, the MSS is a value negotiated during the connection establishment, using an option of the same name.

The name of this algorithm is misleading; the algorithm starts slowly, but grows exponentially. To show the idea, let us look at Figure 24.29. We assume that *rwnd* is much larger than *cwnd,* so that the sender window size always equals *cwnd*. We also assume that each segment is of the same size and carries MSS bytes. For simplicity, we also ignore the delayed-ACK policy and assume that each segment is acknowledged individually.

The sender starts with *cwnd* = 1. This means that the sender can send only one segment. After the first ACK arrives, the acknowledged segment is purged from the window, which means there is now one empty segment slot in the window. The size of the congestion window is also increased by 1 because the arrival of the acknowledgment is a good sign that there is no congestion in the network. The size of the window is now 2. After sending two segments and receiving two individual acknowledgments for them, the size of the congestion window now becomes 4, and so on. In other words,

Figure 24.29 *Slow start, exponential increase*



the size of the congestion window in this algorithm is a function of the number of ACKs arrived and can be determined as follows.

If an ACK arrives, *cwnd* = *cwnd* + **1.**

If we look at the size of the *cwnd* in terms of round-trip times (RTTs), we find that the growth rate is exponential in terms of each round trip time, which is a very aggressive approach:

Start	→	$cwnd = \mathbf{1} \rightarrow \mathbf{2^0}$
After 1 RTT	→	$cwnd = cwnd + \mathbf{1} = \mathbf{1} + \mathbf{1} = \mathbf{2} \rightarrow \mathbf{2^1}$
After 2 RTT	→	$cwnd = cwnd + \mathbf{2} = \mathbf{2} + \mathbf{2} = \mathbf{4} \rightarrow \mathbf{2^2}$
After 3 RTT	→	$cwnd = cwnd + \mathbf{4} = \mathbf{4} + \mathbf{4} = \mathbf{8} \rightarrow \mathbf{2^3}$

A slow start cannot continue indefinitely. There must be a threshold to stop this phase. The sender keeps track of a variable named *ssthresh* (slow-start threshold). When the size of the window in bytes reaches this threshold, slow start stops and the next phase starts.

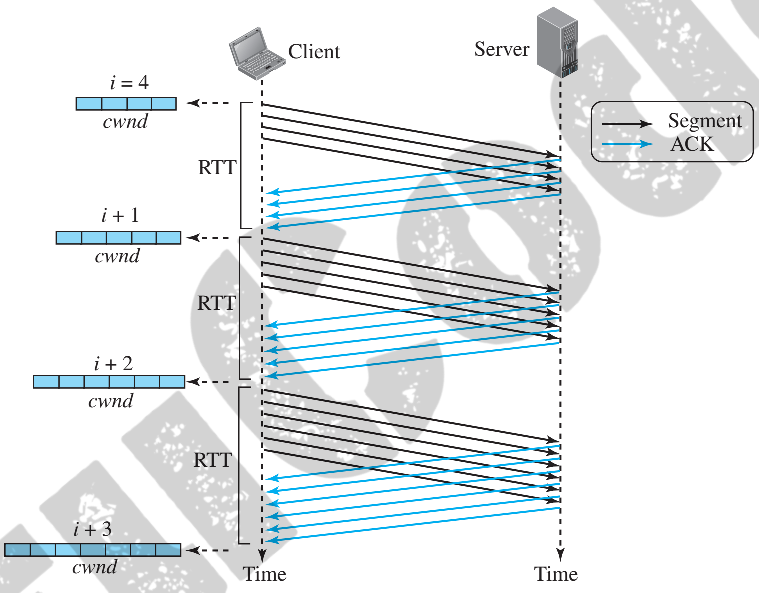
In the slow-start algorithm, the size of the congestion window increases exponentially until it reaches a threshold.

We need, however, to mention that the slow-start strategy is slower in the case of delayed acknowledgments. Remember, for each ACK, the *cwnd* is increased by only 1. Hence, if two segments are acknowledged cumulatively, the size of the *cwnd* increases by only 1, not 2. The growth is still exponential, but it is not a power of 2. With one ACK for every two segments, it is a power of 1.5.

Congestion Avoidance: Additive Increase

If we continue with the slow-start algorithm, the size of the congestion window increases exponentially. To avoid congestion before it happens, we must slow down this exponential growth. TCP defines another algorithm called ***congestion avoidance,*** which increases the *cwnd* additively instead of exponentially. When the size of the congestion window reaches the slow-start threshold in the case where *cwnd* = *i*, the slow-start phase stops and the additive phase begins. In this algorithm, each time the whole "window" of segments is acknowledged, the size of the congestion window is increased by one. A window is the number of segments transmitted during RTT. Figure 24.30 shows the idea.

Figure 24.30 *Congestion avoidance, additive increase*



The sender starts with *cwnd* = 4. This means that the sender can send only four segments. After four ACKs arrive, the acknowledged segments are purged from the window, which means there is now one extra empty segment slot in the window. The size of the congestion window is also increased by 1. The size of window is now 5. After sending five segments and receiving five acknowledgments for them, the size of the congestion window now becomes 6, and so on. In other words, the size of the congestion window in this algorithm is also a function of the number of ACKs that have arrived and can be determined as follows:

If an ACK arrives, *cwnd* = *cwnd* + (1/*cwnd*).

The size of the window increases only 1/*cwnd* portion of MSS (in bytes). In other words, all segments in the previous window should be acknowledged to increase the window 1 MSS bytes.

If we look at the size of the *cwnd* in terms of round-trip times (RTTs), we find that the growth rate is linear in terms of each round-trip time, which is much more conservative than the slow-start approach.

Start	→	$cwnd = i$
After 1 RTT	→	$cwnd = i + 1$
After 2 RTT	→	$cwnd = i + 2$
After 3 RTT	→	$cwnd = i + 3$

In the congestion-avoidance algorithm, the size of the congestion window increases additively until congestion is detected.

Fast Recovery The **fast-recovery** algorithm is optional in TCP. The old version of TCP did not use it, but the new versions try to use it. It starts when three duplicate ACKs arrive, which is interpreted as light congestion in the network. Like congestion avoidance, this algorithm is also an additive increase, but it increases the size of the congestion window when a duplicate ACK arrives (after the three duplicate ACKs that trigger the use of this algorithm). We can say

If a duplicate ACK arrives, $cwnd = cwnd + (1 / cwnd)$.

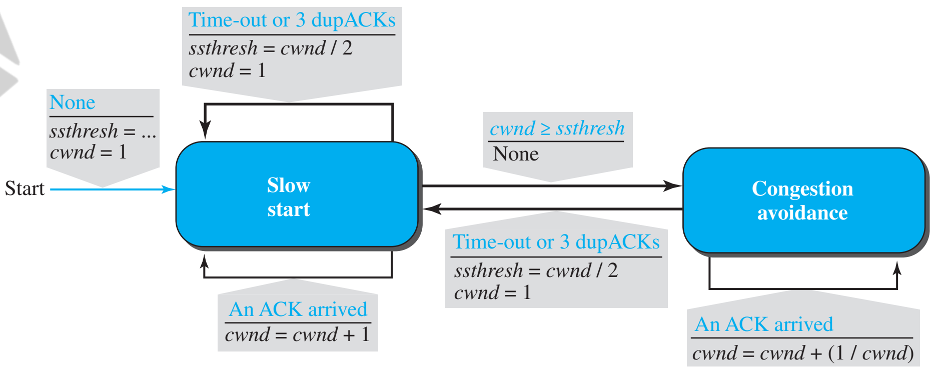
Policy Transition

We discussed three congestion policies in TCP. Now the question is when each of these policies is used and when TCP moves from one policy to another. To answer these questions, we need to refer to three versions of TCP: Taho TCP, Reno TCP, and New Reno TCP.

Taho TCP

The early TCP, known as *Taho TCP,* used only two different algorithms in their congestion policy: *slow start* and *congestion avoidance*. We use Figure 24.31 to show the FSM for this version of TCP. However, we need to mention that we have deleted some small trivial actions, such as incrementing and resetting the number of duplicate ACKs, to make the FSM less crowded and simpler.

Figure 24.31 *FSM for Taho TCP*



Taho TCP treats the two signs used for congestion detection, time-out and three duplicate ACKs, in the same way. In this version, when the connection is established, TCP starts the slow-start algorithm and sets the *ssthresh* variable to a pre-agreed value (normally a multiple of MSS) and the *cwnd* to 1 MSS. In this state, as we said before, each time an ACK arrives, the size of the congestion window is incremented by 1. We know that this policy is very aggressive and exponentially increases the size of the window, which may result in congestion.

If congestion is detected (occurrence of time-out or arrival of three duplicate ACKs), TCP immediately interrupts this aggressive growth and restarts a new slow start algorithm by limiting the threshold to half of the current *cwnd* and resetting the congestion window to 1. In other words, not only does TCP restart from scratch, but it also learns how to adjust the threshold. If no congestion is detected while reaching the threshold, TCP learns that the ceiling of its ambition is reached; it should not continue at this speed. It moves to the congestion avoidance state and continues in that state.

In the congestion-avoidance state, the size of the congestion window is increased by 1 each time a number of ACKs equal to the current size of the window has been received. For example, if the window size is now 5 MSS, five more ACKs should be received before the size of the window becomes 6 MSS. Note that there is no ceiling for the size of the congestion window in this state; the conservative additive growth of the congestion window continues to the end of the data transfer phase unless congestion is detected. If congestion is detected in this state, TCP again resets the value of the *ssthresh* to half of the current *cwnd* and moves to the slow-start state again.

Although in this version of TCP the size of *ssthresh* is continuously adjusted in each congestion detection, this does not mean that it necessarily becomes lower than the previous value. For example, if the original *ssthresh* value is 8 MSS and congestion is detected when TCP is in the congestion avoidance state and the value of the *cwnd* is 20, the new value of the *ssthresh* is now 10, which means it has been increased.

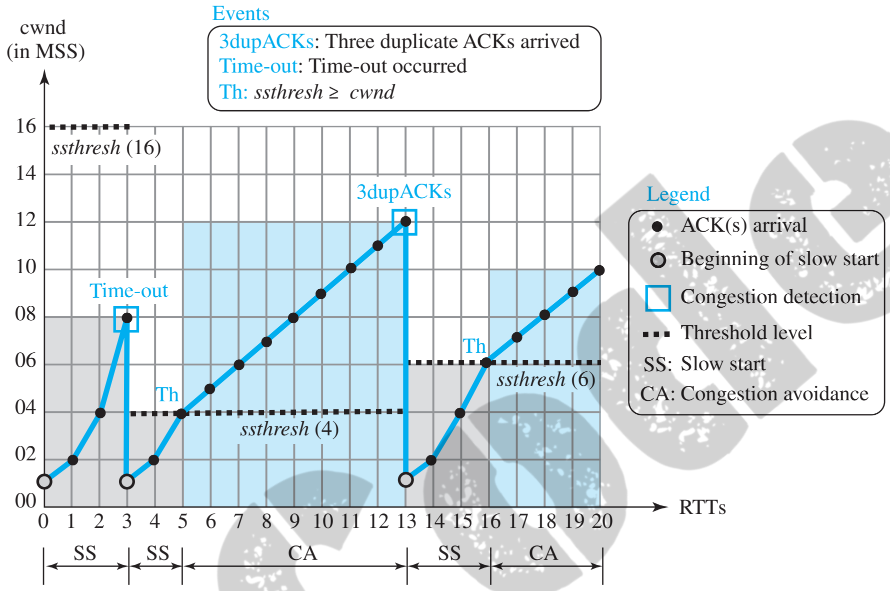
Example 24.9

Figure 24.32 shows an example of congestion control in a Taho TCP. TCP starts data transfer and sets the *ssthresh* variable to an ambitious value of 16 MSS. TCP begins at the slow-start (SS) state with the *cwnd* = 1. The congestion window grows exponentially, but a time-out occurs after the third RTT (before reaching the threshold). TCP assumes that there is congestion in the network. It immediately sets the new *ssthresh* = 4 MSS (half of the current *cwnd,* which is 8) and begins a new slow-start (SA) state with *cwnd* = 1 MSS. The congestion window grows exponentially until it reaches the newly set threshold. TCP now moves to the congestion-avoidance (CA) state and the congestion window grows additively until it reaches *cwnd* = 12 MSS. At this moment, three duplicate ACKs arrive, another indication of congestion in the network. TCP again halves the value of *ssthresh* to 6 MSS and begins a new slow-start (SS) state. The exponential growth of the *cwnd* continues. After RTT 15, the size of *cwnd* is 4 MSS. After sending four segments and receiving only two ACKs, the size of the window reaches the *ssthresh* (6) and TCP moves to the congestion-avoidance state. The data transfer now continues in the congestion-avoidance (CA) state until the connection is terminated after RTT 20.

Reno TCP

A newer version of TCP, called *Reno TCP,* added a new state to the congestion-control FSM, called the fast-recovery state. This version treated the two signals of congestion,

Figure 24.32 *Example 24.9*

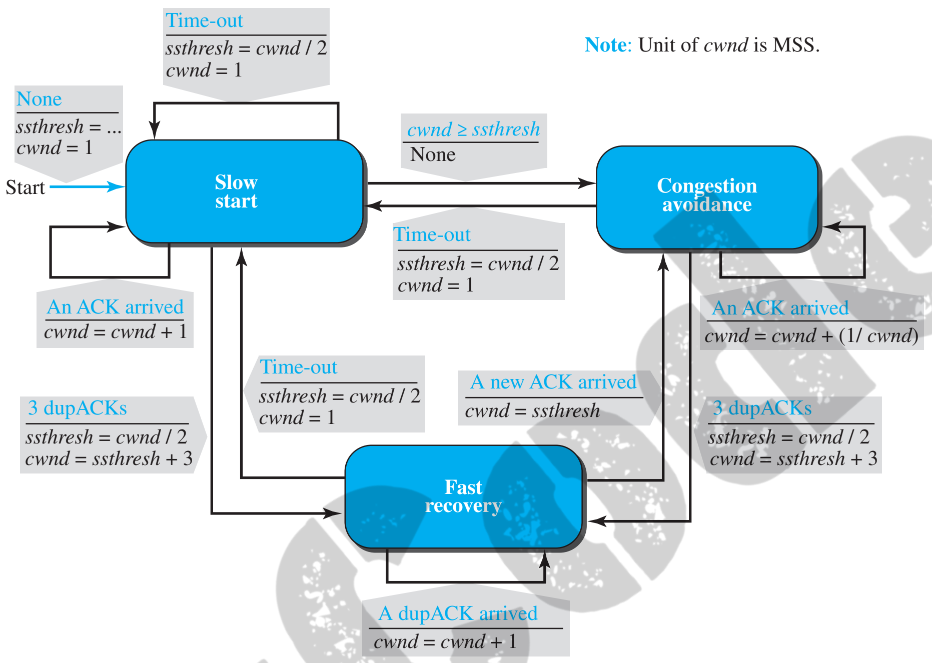


time-out and the arrival of three duplicate ACKs, differently. In this version, if a time-out occurs, TCP moves to the slow-start state (or starts a new round if it is already in this state); on the other hand, if three duplicate ACKs arrive, TCP moves to the fast-recovery state and remains there as long as more duplicate ACKs arrive. The fast-recovery state is a state somewhere between the slow-start and the congestion-avoidance states. It behaves like the slow start, in which the *cwnd* grows exponentially, but the *cwnd* starts with the value of *ssthresh* plus 3 MSS (instead of 1). When TCP enters the fast-recovery state, three major events may occur. If duplicate ACKs continue to arrive, TCP stays in this state, but the *cwnd* grows exponentially. If a time-out occurs, TCP assumes that there is real congestion in the network and moves to the slow-start state. If a new (non-duplicate) ACK arrives, TCP moves to the congestion-avoidance state, but deflates the size of the *cwnd* to the *ssthresh* value, as though the three duplicate ACKs have not occurred, and transition is from the slow-start state to the congestion-avoidance state. Figure 24.33 shows the simplified FSM for Reno TCP. Again, we have removed some trivial events to simplify the figure and discussion.

Example 24.10

Figure 24.34 shows the same situation as Figure 24.32, but in Reno TCP. The changes in the congestion window are the same until RTT 13 when three duplicate ACKs arrive. At this moment, Reno TCP drops the *ssthresh* to 6 MSS (same as Taho TCP), but it sets the *cwnd* to a much higher value (*ssthresh* + 3 = 9 MSS) instead of 1 MSS. Reno TCP now moves to the fast recovery state. We assume that two more duplicate ACKs arrive until RTT 15, where *cwnd* grows exponentially. In this moment, a new ACK (not duplicate) arrives that announces the receipt of the lost segment. Reno TCP now moves to the congestion-avoidance state, but first deflates the

Figure 24.33 *FSM for Reno TCP*


Note: Unit of cwnd is MSS.
Time-out / ssthresh = cwnd / 2, cwnd = 1
None / ssthresh = ..., cwnd = 1
Start
Slow start
cwnd ≥ ssthresh / None
Congestion avoidance
Time-out / ssthresh = cwnd / 2, cwnd = 1
An ACK arrived / cwnd = cwnd + 1
An ACK arrived / cwnd = cwnd + (1/ cwnd)
Time-out / ssthresh = cwnd / 2, cwnd = 1
A new ACK arrived / cwnd = ssthresh
3 dupACKs / ssthresh = cwnd / 2, cwnd = ssthresh + 3
3 dupACKs / ssthresh = cwnd / 2, cwnd = ssthresh + 3
Fast recovery
A dupACK arrived / cwnd = cwnd + 1

congestion window to 6 MSS (the *ssthresh* value) as though ignoring the whole fast-recovery state and moving back to the previous track.

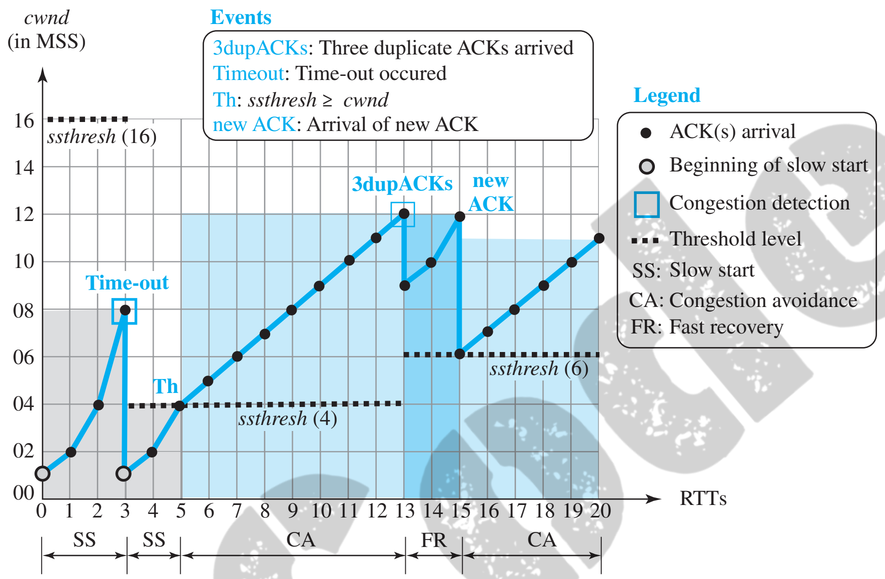
NewReno TCP

A later version of TCP, called *NewReno TCP,* made an extra optimization on the Reno TCP. In this version, TCP checks to see if more than one segment is lost in the current window when three duplicate ACKs arrive. When TCP receives three duplicate ACKs, it retransmits the lost segment until a new ACK (not duplicate) arrives. If the new ACK defines the end of the window when the congestion was detected, TCP is certain that only one segment was lost. However, if the ACK number defines a position between the retransmitted segment and the end of the window, it is possible that the segment defined by the ACK is also lost. NewReno TCP retransmits this segment to avoid receiving more and more duplicate ACKs for it.

Additive Increase, Multiplicative Decrease

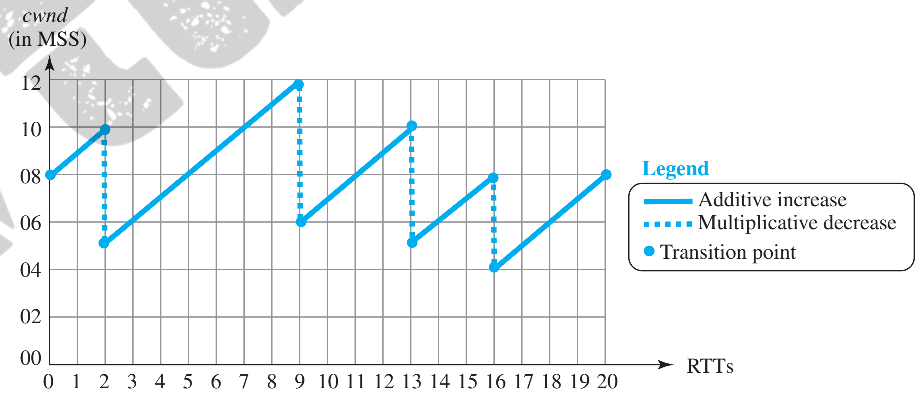
Out of the three versions of TCP, the Reno version is most common today. It has been observed that, in this version, most of the time the congestion is detected and taken care of by observing the three duplicate ACKs. Even if there are some time-out events, TCP recovers from them by aggressive exponential growth. In other words, in a long TCP connection, if we ignore the slow-start states and short exponential growth during fast recovery, the TCP congestion window is *cwnd* = *cwnd* + (1 / cwnd) when an ACK arrives (congestion avoidance), and *cwnd* = *cwnd* / 2 when congestion is detected, as though SS does not exist and the length of FR is reduced to zero. The first is called *additive*

Figure 24.34 *Example 24.10*



increase; the second is called *multiplicative decrease*. This means that the congestion window size, after it passes the initial slow-start state, follows a saw tooth pattern called ***additive increase, multiplicative decrease (AIMD),*** as shown in Figure 24.35.

Figure 24.35 *Additive increase, multiplicative decrease (AIMD)*



TCP Throughput

The throughput for TCP, which is based on the congestion window behavior, can be easily found if the *cwnd* is a constant (flat line) function of RTT. The throughput with this unrealistic assumption is throughput = *cwnd* / RTT. In this assumption, TCP sends a *cwnd* bytes of data and receives acknowledgement for them in RTT time. The behavior of TCP, as shown in Figure 24.35, is not a flat line; it is like saw teeth, with many minimum and maximum values. If each tooth were exactly the same, we could say that the throughput = [(maximum + minimum) / 2] / RTT. However, we know that the value of the maximum is twice the value of the minimum because in each congestion detection the value of *cwnd* is set to half of its previous value. So the throughput can be better calculated as

$$\textbf{throughput} = \textbf{(0.75)}\ \mathbf{W_{max}} / \textbf{RTT}$$

in which W_{max} is the average of window sizes when the congestion occurs.

Example 24.11

If MSS = 10 KB (kilobytes) and RTT = 100 ms in Figure 24.35, we can calculate the throughput as shown below.

$W_{max} = (10 + 12 + 10 + 8 + 8) / 5 = 9.6$ MSS
Throughput = $(0.75\ W_{max}$ / RTT) = 0.75×960 kbps / 100 ms = 7.2 Mbps