Chapter Contents

24

UDP: Datagram Transport Service

24.1 Introduction

Previous chapters describe the connectionless packet delivery service provided by IP and the companion protocol used to report errors. This chapter considers UDP, one of the two major transport-layer protocols used in the Internet and the only connectionless transport service. The chapter discusses the UDP packet format and the ways UDP can be used. We will see that although UDP is efficient and flexible, it has the surprising property of using best-effort delivery semantics. In addition to discussing UDP, the chapter covers the important concept of protocol port numbers.

The next chapter continues the discussion by focusing on the other major transport-layer protocol, TCP. Later chapters discuss Internet routing and network management, which each use transport protocols.

24.2 Transport Protocols And End-To-End Communication

As previous chapters show, both IPv4 and IPv6 provide a packet delivery service that spans the Internet (i.e., a datagram can pass from the sending host, across one or more physical networks, to the receiving host). Despite its ability to pass traffic across the Internet, IP lacks an essential feature: IP cannot distinguish among multiple application programs running on a given host. If a user runs an email application and a web browser at the same time or runs multiple copies of a given application, they must be able to communicate independently.

IP is incapable of supporting multiple applications because fields in an datagram header only identify computers. That is, from IP's point of view, the source and destination fields in a datagram identify a host computer; an IP address does not contain additional bits to identify an application program on the host. We say that IP treats a computer as an *endpoint* of communication. In contrast, transport-layer protocols are known as *end-to-end protocols*, because a transport protocol allows an individual application program to be an endpoint of communication. Instead of adding additional features to IP to identify applications, the designers of the TCP/IP protocols placed end-to-end protocols in a separate layer, layer 4.

24.3 The User Datagram Protocol

As we will see, the TCP/IP suite contains two major transport protocols, the *User Datagram Protocol* (*UDP*) and the *Transmission Control Protocol* (*TCP*), that differ dramatically in the service they offer to applications. UDP is less complex and easiest to understand. The simplicity and ease of understanding come with a cost — UDP does not provide the type of service a typical application expects.

UDP can be characterized as:

- *End-to-end.* UDP is a transport protocol that can distinguish among multiple application programs running on a given computer.
- *Connectionless.* The interface that UDP supplies to applications follows a connectionless paradigm.
- *Message-oriented.* An application that uses UDP sends and receives individual messages.
- *Best-effort.* UDP offers applications the same best-effort delivery semantics as IP.
- *Arbitrary Interaction.* UDP allows an application to send to many other applications, receive from many other applications, or communicate with exactly one other application.
- *Operating System Independent.* UDP provides a means of identifying application programs that does not depend on identifiers used by the local operating system.

The most important characteristic of UDP, its best-effort semantics, arises because UDP uses IP for transmission with no further steps taken to correct problems. In fact, UDP is sometimes characterized as a *thin* protocol layer that provides applications with the ability to send and receive IP datagrams. We can summarize:

UDP provides an end-to-end service that allows an application program to send and receive individual messages, each of which travels in a separate datagram. An application can choose to restrict communication to one other application program or communicate with multiple applications.

24.4 The Connectionless Paradigm

UDP uses a *connectionless* communication paradigm, which means that an application using UDP does not need to preestablish communication before sending data, nor does the application need to inform the network when finished. Instead, an application can generate and send data at any time. Moreover, UDP allows an application to delay an arbitrarily long time between the transmission of two messages. UDP does not maintain state, and does not send extra control messages; communication consists only of the data messages themselves. In particular, if a pair of applications stop sending data, no other packets are exchanged. As a result, UDP has extremely low overhead. To summarize:

UDP is connectionless, which means that an application can send data at any time and UDP does not transmit any packets other than the packets that carry user data.

24.5 Message-Oriented Interface

UDP offers application programs a *message-oriented* interface. Each time an application requests that UDP send a block of data, UDP places the data in a single message for transmission. UDP does not divide a message into multiple packets, and does not combine messages for delivery — each message that an application sends is transported across the Internet and delivered to the receiver.

The message-oriented interface has several important consequences for programmers. On the positive side, applications that use UDP can depend on the protocol to preserve data boundaries — each message UDP delivers to a receiving application will be exactly the same as was transmitted by the sender. On the negative side, each UDP message must fit into a single IP datagram. Thus, the IP datagram size forms an absolute limit on the size of a UDP message. More important, UDP message size can lead to inefficient use of the underlying network. If an application sends extremely small messages, the resulting datagrams will have a large ratio of header octets to data octets. If an application sends extremely large messages, the resulting datagrams may be larger than the network MTU, and will be fragmented by IP.

Allowing UDP messages to be large produces an interesting anomaly. Normally, an application programmer can achieve higher efficiency by using large transfers. For example, programmers are encouraged to declare large I/O buffers, and to specify transfers that match the buffer size. With UDP, however, sending large messages leads to less efficiency because large messages cause fragmentation. Even more surprising, the fragmentation can occur on the sending computer — an application sends a large message, UDP places the entire message in a user datagram and encapsulates the user datagram in an Internet datagram, and IP must perform fragmentation before the datagram can be sent. The point is:

> *Although a programmer's intuition suggests that using larger messages will increase efficiency, if a UDP message is larger than the network MTU, IP will fragment the resulting datagram, which reduces efficiency.*

As a consequence, many programmers who use UDP choose a message size that produces datagrams that fit in a standard MTU. In particular, because most parts of the Internet now support an MTU of 1500 octets, programmers often choose a message size of 1400 or 1450 to leave space for IP and UDP headers.

24.6 UDP Communication Semantics

UDP uses IP for all delivery. Furthermore, UDP provides applications with exactly the same best-effort delivery semantics as IP, which means messages can be:

- Lost
- Duplicated
- Delayed
- Delivered out-of-order
- Corrupted

Of course, UDP does not purposefully introduce delivery problems. Instead, UDP merely uses IP to send messages, and does not detect or correct delivery problems. UDP's best-effort delivery semantics have important consequences for applications. An application must either be immune to the problems, or the programmer must take additional steps to detect and correct problems. As an example of an application that can tolerate packet errors, consider an audio transmission. If the sender places a small amount of audio in each message, the loss of a single packet produces a small gap in the playback, which will be heard as a pop or click. Although it is not desirable, the noise is merely annoying. At the opposite extreme, consider an online shopping application. Such applications are not written to use UDP because packet errors can have serious consequences (e.g., duplication of a message that carries a catalog order can result in two orders, with double charges being made to the buyer's credit card).

We can summarize:

Because UDP offers the same best-effort delivery semantics as IP, a UDP message can be lost, duplicated, delayed, delivered out-of-order or bits can be corrupted in transit. UDP only suffices for applications such as voice or video that can tolerate delivery errors.

24.7 Modes Of Interaction And Multicast Delivery

UDP allows four styles of interaction:

- 1-to-1
- 1-to-many
- Many-to-1
- Many-to-many

That is, an application using UDP has a choice. An application can choose a 1-to-1 interaction in which the application exchanges messages with exactly one other application, a 1-to-many interaction in which the application sends a message to multiple recipients, or a many-to-1 interaction in which the application receives messages from multiple senders. Finally, a set of applications can establish a many-to-many interaction in which they exchange messages with one another.

Although a 1-to-many interaction can be achieved by arranging to send an individual copy of a message to each intended recipient, UDP allows the exchange to be efficient. Instead of requiring an application to repeatedly send a message to multiple recipients, UDP allows an application to transmit the message via IP multicast (or IPv4 broadcast). To do so, the sender uses an IP multicast or broadcast address as the destination IP address. For example, delivery to all nodes on the local network can be specified by using IPv4's limited broadcast address, 255.255.255.255, or IPv6's link-local all-nodes multicast address. Delivery via broadcast or multicast is especially useful for Ethernet networks, because the underlying hardware supports both types efficiently.

24.8 Endpoint Identification With Protocol Port Numbers

Exactly how should UDP identify an application program as an endpoint? It might seem that UDP could use the same mechanism that the operating system uses. Unfortunately, because UDP must span heterogeneous computers, no common mechanism exists. For example, some operating systems use process identifiers, others use job names, and others use task identifiers. Thus, an identifier that is meaningful on one system may not be meaningful on another.

To avoid ambiguity, UDP defines an abstract set of identifiers called *protocol port numbers* that are independent of the underlying operating system. Each computer that implements UDP must provide a mapping between protocol port numbers and the program identifiers that the operating system uses. For example, the UDP standard defines protocol port number seven as the port for an *echo* service and port number thirty-seven as the port for a *timeserver* service. All computers running UDP recognize the standard protocol port numbers, independent of the underlying operating system. Thus, when a UDP message arrives for port seven, UDP protocol software must know which application on the local computer implements the echo service and must pass the incoming message to the program.

The communication mode is determined by the way an application fills in addresses and protocol port numbers for a socket. To engage in 1-to-1 communication, an application specifies the local port number, remote IP address, and remote protocol port number; UDP only passes the application messages that arrive from the specified sender. To engage in many-to-1 communication, the application specifies the local port number, but informs UDP that the remote endpoint can be any system. UDP then passes the application all messages that arrive for the specified port†.

24.9 UDP Datagram Format

Each UDP message is called a *user datagram* and consists of two parts: a short header that specifies the sending and receiving application programs and a payload that carries the data being sent. Figure 24.1 illustrates the user datagram format.

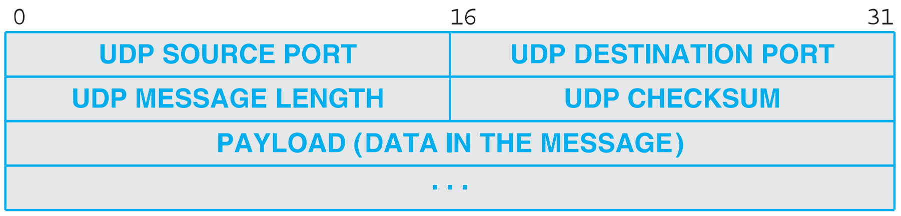

Figure 24.1 The format of a UDP user datagram with an 8-octet header.

The first two fields of the UDP header contain 16-bit protocol port numbers. Field *UDP SOURCE PORT* contains the port number of the sending application, and field *UDP DESTINATION PORT* contains the port number of the application to which the message is being sent. Field *UDP MESSAGE LENGTH* specifies the total size of the UDP message, measured in 8-bit bytes.

†Only one application can request all messages for a given port.

24.10 The UDP Checksum And The Pseudo Header

Although the UDP header contains a sixteen-bit field named *UDP CHECKSUM*, the checksum is optional. A sender can either choose to compute a checksum or set all bits of the checksum field to zero. When a message arrives at the destination, UDP software examines the checksum field, and only verifies the checksum if the value is nonzero†.

Note that the UDP header does not contain any identification of the sender or receiver other than the protocol port numbers. In particular, UDP assumes that the IP source and destination addresses are contained in the IP datagram that carries UDP. Thus, IP addresses are not carried in the UDP header.

Omitting the source and destination IP addresses makes UDP smaller and more efficient, but introduces the possibility of error. In particular, if IP malfunctions and delivers a UDP message to an incorrect destination, UDP cannot use header fields to determine that an error occurred.

To allow UDP to verify that messages reach the correct destination without incurring the overhead of additional header fields, UDP extends the checksum. When computing the checksum, UDP software includes a *pseudo header* that contains the IP source, IP destination, and type (i.e., PROTO or NEXT-HEADER) fields from the IP datagram and a UDP datagram length. That is, the sender computes a checksum as if the UDP header contained extra fields. Similarly, to verify a checksum, a receiver must obtain the UDP length, and the source, destination, and type fields from the IP datagram; the receiver appends them to the UDP message before verifying the checksum. Figure 24.2 illustrates fields in the pseudo header.

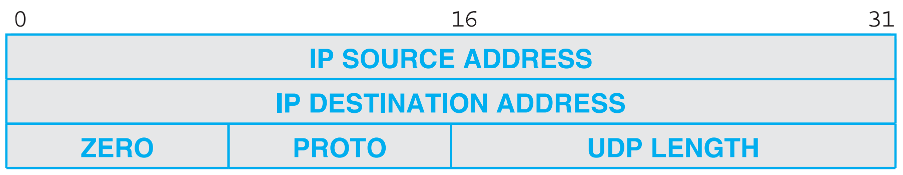

Figure 24.2 Illustration of the pseudo header used to calculate the UDP checksum.

24.11 UDP Encapsulation

Like ICMP, each UDP datagram is encapsulated in an IP datagram for transmission across the Internet. Figure 24.3 illustrates the encapsulation.

†Like IP, UDP uses a ones-complement checksum; if the computed checksum has a value of zero, a sender uses the all-ones form of zero.

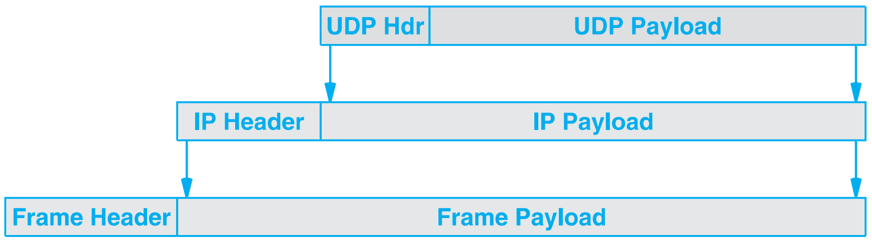

Figure 24.3 The encapsulation of a UDP message in an IP datagram.

24.12 Summary

The User Datagram Protocol (UDP) provides connectionless end-to-end message transport from an application running on one computer to an application running on another computer. UDP offers the same best-effort delivery semantics as IP, which means that messages can be lost, duplicated, or delivered out-of-order. One advantage of a connectionless approach arises from the ability to have 1-to-1, 1-to-many, and many-to-1 interactions among applications.

To remain independent of the underlying operating systems, UDP uses small integer protocol port numbers to distinguish among application programs. Protocol software on a given computer must map each protocol port number to the appropriate mechanism (e.g., process ID) used on the computer.

The UDP checksum is optional — if a sender fills the checksum field with zero, the receiver does not verify the checksum. To verify that the UDP datagram arrived at the correct location, a UDP checksum is computed over the datagram plus a pseudo header

UDP requires two levels of encapsulation. Each UDP message is encapsulated in an IP datagram for transmission across the Internet. The datagram is encapsulated in a frame for transmission across an individual network.

EXERCISES

24.1 List the features of UDP.

24.2 What is the conceptual difference between IP and end-to-end protocols?

24.3 Calculate the size of the largest possible UDP message when using IPv4 and IPv6. (Hint: the entire UDP message must fit in an IP datagram.)

24.4 Do applications need to exchange UDP control messages before exchanging data? Explain.

24.5 If an application uses UDP to send an 8K byte message across an Ethernet, how many frames will traverse the network?

24.6 What happens if a UDP message containing a payload of 1500 bytes is sent across an Ethernet?

24.7 What are the semantics of UDP?

24.8 When a UDP message arrives at a computer, can IP software completely discard the frame header and IP header before passing the UDP message to UDP software for processing? Explain.

24.9 What is a pseudo header, and when is one used?

24.10 What endpoint values must be specified by an application that engages in 1-to-1 communication? In 1-to-many? In many-to-1?

24.11 Given an Ethernet frame, what fields must be examined to determine whether the frame contains an IPv6 datagram that carries a UDP message?

24.12 Answer the previous question for IPv4.

Chapter Contents

25

TCP: Reliable Transport Service

25.1 Introduction

Previous chapters describe the connectionless packet delivery services provided by IP and the User Datagram Protocol that runs over IP. This chapter considers transport protocols in general, and examines TCP, the major transport protocol used in the Internet. The chapter explains how the TCP protocol provides reliable delivery.

TCP achieves a seemingly impossible task: it uses the unreliable datagram service offered by IP when sending across the Internet, but provides a reliable data delivery service to application programs. TCP must compensate for loss, delay, duplication, and out-of-order delivery, and it must do so without overloading the underlying networks and routers. After reviewing the service that TCP provides to applications, the chapter examines the techniques TCP uses to achieve reliability.

25.2 The Transmission Control Protocol

Programmers are trained to think that reliability is fundamental in a computer system. For example, when writing an application that sends data to an I/O device such as a printer, a programmer assumes the data will arrive correctly or the operating system will inform the application that an error has occurred. That is, a programmer assumes the underlying system guarantees that data will be delivered reliably.

To allow programmers to follow conventional techniques when creating applications that communicate across the Internet, protocol software must provide the same semantics as a conventional computer system: the software must guarantee prompt, reliable communication. Data must be delivered in exactly the same order that it was sent, and there must be no loss or duplication.

In the TCP/IP suite, the *Transmission Control Protocol* (*TCP*) provides reliable transport service. TCP is remarkable because it solves a difficult problem well — although other protocols have been created, no general-purpose transport protocol has proved to work better. Consequently, most Internet applications are built to use TCP.

To summarize:

> *In the Internet, the Transmission Control Protocol (TCP) is a transport-layer protocol that provides reliability.*

25.3 The Service TCP Provides To Applications

The service offered by TCP has seven major features:

- *Connection Orientation.* TCP provides connection-oriented service in which an application must first request a connection to a destination, and then use the connection to transfer data.
- *Point-To-Point Communication.* Each TCP connection has exactly two endpoints.
- *Complete Reliability.* TCP guarantees that the data sent across a connection will be delivered exactly as sent, complete and in order.
- *Full Duplex Communication.* A TCP connection allows data to flow in either direction, and allows either application program to send data at any time.
- *Stream Interface.* TCP provides a stream interface, in which an application sends a continuous sequence of octets across a connection. TCP does not group data into records or messages, and does not guarantee to deliver data in the same size pieces that were transferred by the sending application.
- *Reliable Connection Startup.* TCP allows two applications to reliably start communication.
- *Graceful Connection Shutdown.* Before closing a connection, TCP ensures that all data has been delivered and that both sides have agreed to shut down the connection.

To summarize:

TCP provides a reliable, connection-oriented, full-duplex stream transport service that allows two application programs to form a connection, send data in either direction, and then terminate the connection. Each TCP connection is started reliably and terminated gracefully.

25.4 End-To-End Service And Virtual Connections

Like UDP, TCP is classified as an *end-to-end* protocol because it provides communication between an application on one computer to an application on another computer. It is *connection oriented* because applications must request that TCP form a connection before they can transfer data, and must close the connection when transfer is complete.

The connections provided by TCP are called *virtual connections*, because they are achieved by software. Indeed, the underlying Internet does not provide hardware or software support for connections. Instead, the TCP software modules on two machines exchange messages to achieve the illusion of a connection.

Each TCP message is encapsulated in an IP datagram and sent across the Internet. When the datagram arrives on the destination host, IP passes the contents to TCP. Note that although TCP uses IP to carry messages, IP does not read or interpret the messages. In fact, IP treats each TCP message as data to be transferred. Conversely, TCP treats IP as a packet communication system that provides communication between the TCP modules at each end of a connection. Figure 25.1 illustrates how TCP views the underlying Internet.

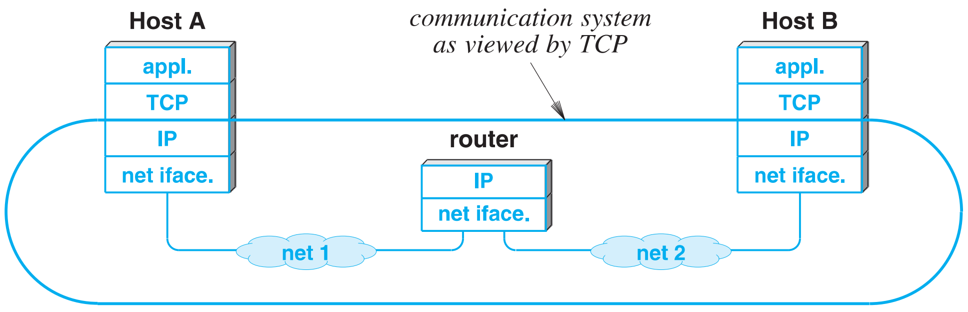

Figure 25.1 Illustration of how TCP views the underlying Internet.

As the figure shows, TCP software is needed at each end of a virtual connection, but not on intermediate routers. From TCP's point of view, the entire Internet is a communication system that can accept and deliver messages without changing or interpreting their contents.

25.5 Techniques That Transport Protocols Use

An end-to-end transport protocol must be carefully designed to achieve efficient, reliable transfer. The major problems are:

- *Unreliable Communication*. Messages sent across the Internet can be lost, duplicated, corrupted, delayed, or delivered out of order.
- *End System Reboot*. At any time during communication, either of the two end systems might crash and reboot. There must be no confusion between sessions, even though some embedded systems can reboot in less time than it takes a packet to cross the Internet.
- *Heterogeneous End Systems*. An application running on a powerful processor can generate data so fast that it overruns an application running on a slow processor.
- *Congestion In The Internet*. If senders aggressively transmit data, intermediate switches and routers can become overrun with packets, analogous to a congested highway.

We have already seen examples of basic techniques data communications systems use to overcome some of the problems. For example, to compensate for bits that are changed during transmission, a protocol might include *parity bits*, a *checksum*, or a *cyclic redundancy check* (*CRC*). The most sophisticated transport protocols do more than detect errors — they employ techniques that can repair or circumvent problems. In particular, transport protocols use a variety of tools to handle some of the most complicated communication problems. The next sections discuss basic mechanisms.

25.5.1 Sequencing To Handle Duplicates And Out-Of-Order Delivery

To handle duplicate packets and out-of-order deliveries, transport protocols use *sequencing*. The sending side attaches a sequence number to each packet. The receiving side stores both the sequence number of the last packet received in order as well as a list of additional packets that arrived out of order. When a packet arrives, the receiver examines the sequence number to determine how the packet should be handled. If the packet is the next one expected (i.e., has arrived in order), the protocol software delivers the packet to the next highest layer, and checks its list to see whether additional packets can also be delivered. If the packet has arrived out of order, the protocol software adds the packet to the list. Sequencing also solves the problem of duplication — a receiver checks for duplicates when it examines the sequence number of an arriving packet. If the packet has already been delivered or the sequence number matches one of the packets waiting on the list, the software discards the new copy.

25.5.2 Retransmission To Handle Lost Packets

To handle packet loss, transport protocols use *positive acknowledgement with retransmission*. Whenever a frame arrives intact, the receiving protocol software sends a small *acknowledgement* (*ACK*) message that reports successful reception. The sender takes responsibility for ensuring that each packet is transferred successfully. Whenever it sends a packet, the sending-side protocol software starts a timer. If an acknowledgement arrives before the timer expires, the software cancels the timer; if the timer expires before an acknowledgement arrives, the software sends another copy of the packet and starts the timer again. The action of sending a second copy is known as *retransmitting*, and the copy is commonly called a *retransmission*.

Of course, retransmission cannot succeed if a hardware failure has permanently disconnected the network or if the receiving computer has crashed. Therefore, protocols that retransmit messages usually bound the maximum number of retransmissions. When the bound has been reached, the protocol stops retransmitting and declares that communication is impossible.

Note that if packets are delayed, retransmission can introduce duplicate packets. Thus, transport protocols that incorporate retransmission are usually designed to handle the problem of duplicate packets.

25.5.3 Techniques To Avoid Replay

Extraordinarily long delays can lead to *replay errors*, in which a delayed packet affects later communication. For example, consider the following sequence of events.

- Two computers agree to communicate at *1* PM.
- One computer sends a sequence of ten packets to the other.
- A hardware problem causes packet *3* to be delayed.
- Routes change to avoid the hardware problem.
- Protocol software on the sending computer retransmits packet *3*, and sends the remaining packets without error.
- At 1:05 PM the two computers agree to communicate again.
- After the second packet arrives, the delayed copy of packet *3* arrives from the earlier conversation.
- Packet *3* arrives from the second conversation.

Unless a transport protocol is designed carefully to avoid such problems, a packet from an earlier conversation might be accepted in a later conversation and the correct packet discarded as a duplicate.

Replay can also occur with control packets (i.e., packets that establish or terminate communication). To understand the scope of the problem, consider a situation in which two application programs form a TCP connection, communicate, close the connection, and then form a new connection. The message that specifies closing the connection might be duplicated and one copy might be delayed long enough for the second connection to be established. A protocol should be designed so the duplicate message will not cause the second connection to be closed.

To prevent replay, protocols mark each session with a unique ID (e.g., the time the session was established), and require the unique ID to be present in each packet. The protocol software discards any arriving packet that contains an incorrect ID. To avoid replay, an ID must not be reused until a reasonable time has passed (e.g., hours).

25.5.4 Flow Control To Prevent Data Overrun

Several techniques are available to prevent a fast computer from sending so much data that it overruns a slow receiver. We use the term *flow control* to refer to techniques that handle the problem. The simplest form of flow control is a *stop-and-go* system in which a sender waits after transmitting each packet. When the receiver is ready for another packet, the receiver sends a control message, usually a form of acknowledgement.

Although stop-and-go protocols prevent overrun, they result in extremely low throughput. To understand why, consider what happens on a network that has a packet size of 1000 octets, a throughput capacity of 2 Mbps, and a delay of 50 milliseconds. The network hardware can transport 2 Mbps from one computer to another. However, after transmitting a packet, the sender must wait 100 msec before sending another packet (i.e., 50 msec for the packet to reach the receiver and 50 msec for an acknowledgement to travel back). Thus, the maximum rate at which data can be sent using stop-and-go is one packet every 100 milliseconds. When expressed as a bit rate, the maximum rate that stop-and-go can achieve is 80,000 bps, which is only 4% of the hardware capacity.

To obtain high throughput rates, transport protocols use a flow control technique known as *sliding window*. The sender and receiver are programmed to use a fixed *window size*, which is the maximum amount of data that can be sent before an acknowledgement arrives. For example, the sender and receiver might agree on a window size of four packets. The sender begins with the data to be sent, extracts data to fill four packets (i.e., the first window), and transmits a copy of each packet. In most transport protocols, the sender retains a copy in case retransmission is needed. The receiver must have preallocated buffer space for the entire window. If a packet arrives in sequence, the receiver passes the packet to the receiving application and transmits an acknowledgement to the sender. When an acknowledgement arrives, the sender discards its copy of the acknowledged packet and transmits the next packet. Figure 25.2 illustrates why the mechanism is known as a *sliding window*.

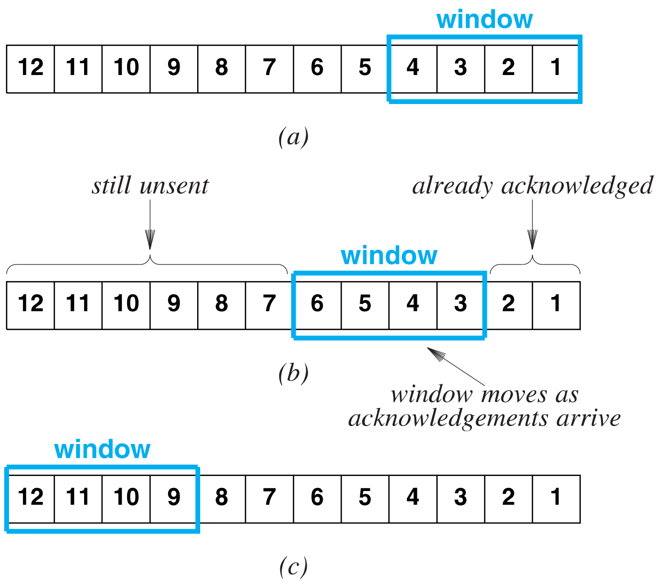

Figure 25.2 An illustration of a sliding window in (a) initial, (b) intermediate, and (c) final positions.

Sliding window can increase throughput dramatically. To understand why, compare the sequence of transmissions with a stop-and-go scheme and a sliding window scheme. Figure 25.3 contains a comparison for a 4-packet transmission.

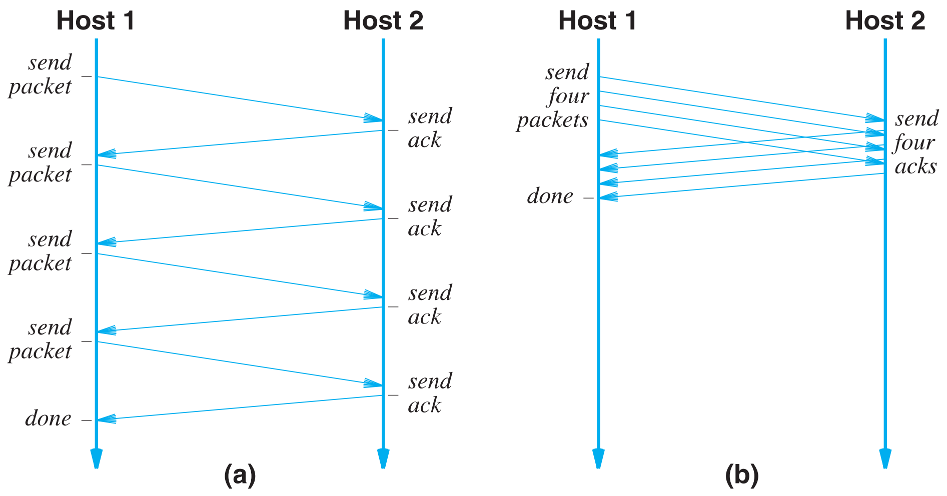

Figure 25.3 Comparison of transmission using (a) stop-and-go, and (b) sliding window.

In Figure 25.3(a), a sender transmits four packets, but waits for an acknowledgement before sending each successive packet. If the delay required to send a single packet on one trip through the network is N, the total time required to send four packets is $8N$. In Figure 25.3(b), a sender transmits all packets in the window before it waits. The figure shows a small delay between successive packet transmissions because transmission is never instantaneous — a short time (usually a few microseconds) is required for the hardware to complete transmission of a packet and begin to transmit the next packet. Thus, the total time required to send four packets is $2N + \varepsilon$, where ε denotes the small delay.

To understand the significance of sliding window, imagine an extended communication that involves many packets. In such cases, the total time required for transmission is so large that ε can be ignored. For such networks, a sliding window protocol can increase performance substantially. The potential improvement is:

$$T_w = T_g \times W \tag{26.1}$$

where T_w is the throughput that can be achieved with a sliding window protocol, T_g is the throughput that can be achieved with a stop-and-go protocol, and W is the window size. The equation explains why the sliding window protocol illustrated in Figure 25.3(b) has approximately four times the throughput of the stop-and-go protocol in Figure 25.3(a). Of course, throughput cannot be increased arbitrarily merely by increasing the window size. The capacity of the underlying network imposes an upper bound — bits cannot be sent faster than the hardware can carry them. Thus, the equation can be rewritten:

$$T_w = min\,(C,\ T_g \times W) \tag{26.2}$$

where C is the underlying hardware capacity†.

25.6 Techniques To Avoid Congestion

To understand how easily congestion can occur, consider four hosts connected by two switches as Figure 25.4 illustrates.

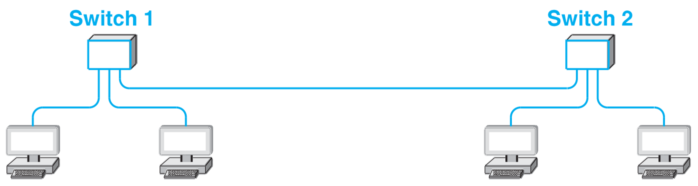

Figure 25.4 Four hosts connected by two switches.

†Networking professionals often use the term *bandwidth* instead of capacity, but strictly speaking the term is not correct because protocols can make the effective network data rate much less than the channel bandwidth.

Assume each connection in the figure operates at 1 Gbps, and consider what happens if both computers attached to Switch 1 attempt to send data to a computer attached to Switch 2. Switch 1 receives data at an aggregate rate of 2 Gbps, but can only forward 1 Gbps to Switch 2. The situation is known as *congestion*. Even if a switch temporarily stores packets in memory, congestion results in increased delay. If congestion persists, the switch will run out of memory and begin discarding packets. Although retransmission can be used to recover lost packets, retransmission sends more packets into the network. Thus, if the situation persists, an entire network can become unusable; the condition is known as *congestion collapse*. In the Internet, congestion usually occurs in routers. Transport protocols attempt to avoid congestion collapse by monitoring the network and reacting quickly once congestion starts. There are two basic approaches:

- Arrange for intermediate systems (i.e., routers) to inform a sender when congestion occurs
- Use increased delay or packet loss as an estimate of congestion

The former scheme is implemented either by having routers send a special message to the source of packets when congestion occurs, or by having routers set a bit in the header of each packet that experiences delay caused by congestion. When the second approach is used, the computer that receives the packet includes information in the acknowledgement to inform the original sender†.

Using delay and loss to estimate congestion is reasonable in the Internet because:

Modern network hardware works well; most delay and loss results from congestion, not hardware failure.

The appropriate response to congestion consists of reducing the rate at which packets are being transmitted. Sliding window protocols can achieve the effect of reducing the rate by temporarily reducing the window size.

25.7 The Art Of Protocol Design

Although the techniques needed to solve specific problems are well-known, protocol design is nontrivial for two reasons. First, to make communication efficient, details must be chosen carefully — small design errors can result in incorrect operation, unnecessary packets, or delays. For example, if sequence numbers are used, each packet must contain a sequence number in the packet header. The field must be large enough so sequence numbers are not reused frequently, but small enough to avoid wasting unnecessary capacity. Second, protocol mechanisms can interact in unexpected ways. For example, consider the interaction between flow control and congestion control mechanisms. A sliding window scheme aggressively uses more of the underlying network capacity to improve throughput. A congestion control mechanism does the opposite by

†A long delay can occur between the time congestion occurs and the original sender is informed.

reducing the number of packets being inserted to prevent the network from collapsing; the balance between sliding window and congestion control can be tricky, and a design that does both well is difficult. That is, aggressive flow control can cause congestion and conservative congestion control can lower the throughput more than necessary. Designs that attempt to switch from aggressive to conservative behavior when congestion occurs tend to oscillate — they slowly increase their use of capacity until the network begins to experience congestion, decrease use until the network becomes stable, and then begin to increase again.

Computer system reboot poses another serious challenge to transport protocol design. Imagine a situation where two application programs establish a connection, begin sending data, and then the computer receiving data reboots. Although protocol software on the rebooted computer has no knowledge of a connection, protocol software on the sending computer considers the connection valid. If a protocol is not designed carefully, a duplicate packet can cause a computer to incorrectly create a connection and begin receiving data in midstream.

25.8 Techniques Used In TCP To Handle Packet Loss

Which of the aforementioned techniques does TCP use to achieve reliable transfer? The answer is complex because TCP uses a variety of schemes that are combined in novel ways. As expected TCP uses *retransmission* to compensate for packet loss. Because TCP provides data flow in both directions, both sides of a communication participate in retransmission. When TCP receives data, it sends an *acknowledgement* back to the sender. Whenever it sends data, TCP starts a timer, and retransmits the data if the timer expires. Thus, basic TCP retransmission operates as Figure 25.5 illustrates.

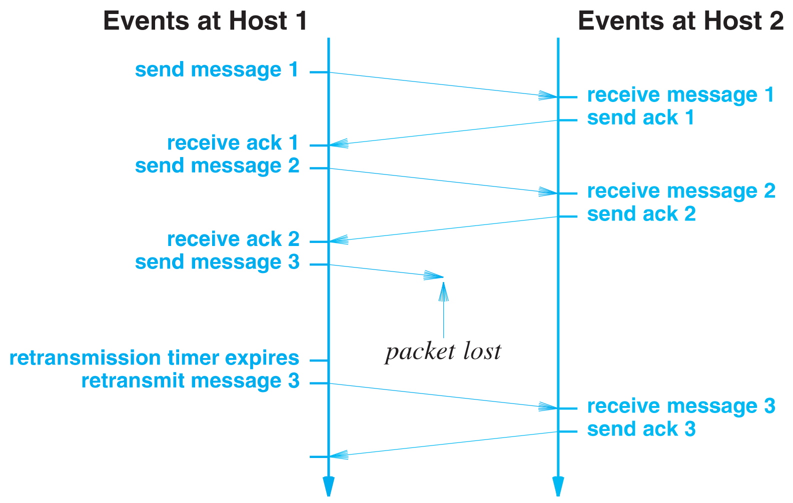

Figure 25.5 Illustration of TCP retransmission after a packet loss.

TCP's retransmission scheme is the key to its success because it handles communication across an arbitrary path through the Internet. For example, one application might send data across a satellite channel to a computer in another country, while another application sends data across a local area network to a computer in the next room. TCP must be ready to retransmit any message that is lost on either connection. The question is: how long should TCP wait before retransmitting? Acknowledgements from a computer on a local area network are expected to arrive within a few milliseconds, but a satellite connection requires hundreds of milliseconds. On the one hand, waiting too long for such an acknowledgement leaves the network idle and does not maximize throughput. Thus, on a local area network, TCP should not delay a long time before retransmitting. On the other hand, retransmitting quickly does not work well on a satellite connection because the unnecessary traffic consumes network bandwidth and lowers throughput.

TCP faces a more difficult challenge than distinguishing between local and remote destinations: bursts of datagrams can cause congestion, which causes transmission delays along a given path to change rapidly. In fact, the total time required to send a message and receive an acknowledgement can increase or decrease by an order of magnitude in a few milliseconds. To summarize:

> *The delay required for data to reach a destination and an acknowledgement to return depends on traffic in the Internet as well as the distance to the destination. Because TCP allows multiple application programs to communicate with multiple destinations concurrently and traffic conditions affect delay, TCP must handle a variety of delays that can change rapidly.*

25.9 Adaptive Retransmission

Before TCP was invented, transport protocols used a fixed value for retransmission delay — the protocol designer or network manager chose a value that was large enough for the expected delay. Designers working on TCP realized that a fixed timeout would not operate well for the Internet. Thus, they chose to make TCP's retransmission *adaptive*. That is, TCP monitors current delay on each connection, and adapts (i.e., changes) the retransmission timer to accommodate changing conditions.

How can TCP monitor Internet delays? In fact, TCP cannot know the exact delays for all parts of the Internet at all times. Instead, TCP estimates *round-trip delay* for each active connection by measuring the time needed to receive a response. Whenever it sends a message to which it expects a response, TCP records the time at which the message was sent. When a response arrives, TCP subtracts the time the message was sent from the current time to produce a new estimate of the round-trip delay for that connection. As it sends data packets and receives acknowledgements, TCP generates a sequence of round-trip estimates and uses a statistical function to produce a weighted average. In addition to a weighted average, TCP keeps an estimate of the variance, and uses a linear combination of the estimated mean and variance when computing the time at which retransmission is needed.

Experience has shown that TCP adaptive retransmission works well. Using the variance helps TCP react quickly when delay increases following a burst of packets. Using a weighted average helps TCP reset the retransmission timer if the delay returns to a lower value after a temporary burst. When the delay remains constant, TCP adjusts the retransmission timeout to a value that is slightly longer than the mean round-trip delay. When delays start to vary, TCP adjusts the retransmission timeout to a value greater than the mean to accommodate peaks.

25.10 Comparison Of Retransmission Times

To understand how adaptive retransmission helps TCP maximize throughput on each connection, consider a case of packet loss on two connections that have different round-trip delays. For example, Figure 25.6 illustrates traffic on two such connections.

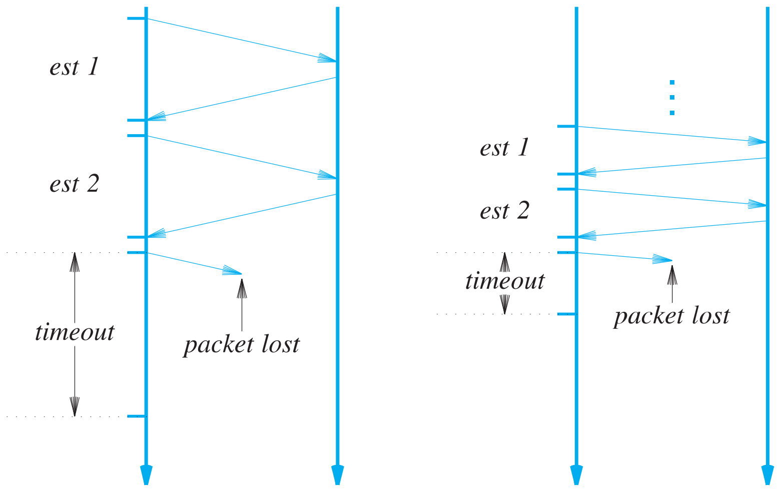

Figure 25.6 Timeout and retransmission on two TCP connections that have different round-trip delays.

As the figure shows, TCP sets the retransmission timeout to be slightly longer than the mean round-trip delay. If the delay is large, TCP uses a large retransmission timeout; if the delay is small, TCP uses a small timeout. The goal is to wait long enough to determine that a packet was lost, without waiting longer than necessary.

25.11 Buffers, Flow Control, And Windows

TCP uses a *window* mechanism to control the flow of data. Unlike the simplistic packet-based window scheme described above, a TCP window is measured in bytes. When a connection is established, each end of the connection allocates a buffer to hold incoming data and sends the size of the buffer to the other end. As data arrives, the receiving TCP sends acknowledgements, which specify the remaining buffer size. TCP uses the term *window* to refer to the amount of buffer space available at any time; a notification that specifies the size of the window is known as a *window advertisement*. A receiver sends a window advertisement with each acknowledgement.

If the receiving application can read data as quickly as it arrives, a receiver will send a positive window advertisement along with each acknowledgement. However, if the sending side operates faster than the receiving side (e.g., because the CPU is faster), incoming data will eventually fill the receiver's buffer, causing the receiver to advertise a *zero window*. A sender that receives a zero window advertisement must stop sending until the receiver again advertises a positive window. Figure 25.7 illustrates window advertisements.

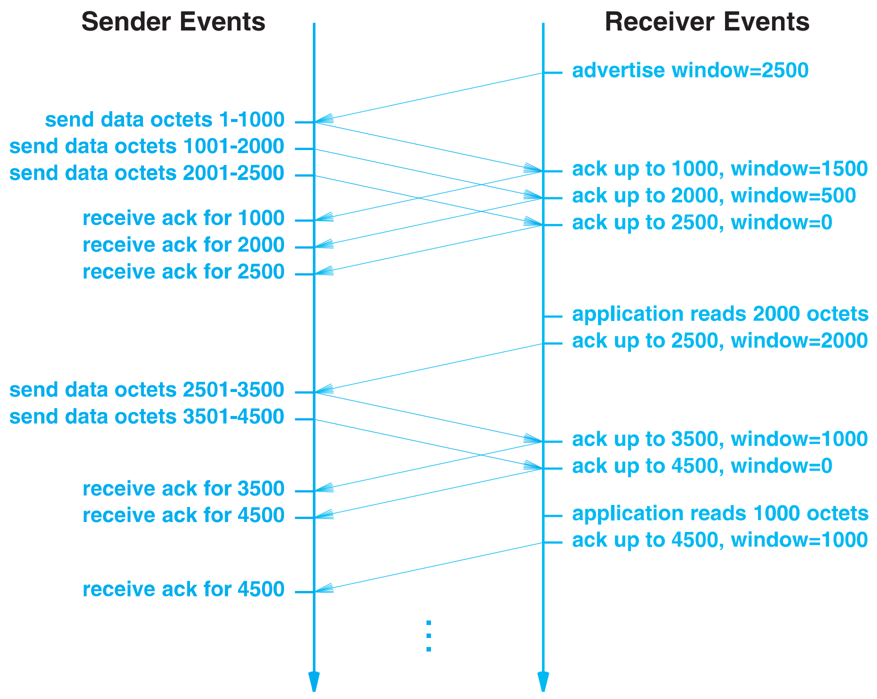
Sender Events
Receiver Events
advertise window=2500
send data octets 1-1000
send data octets 1001-2000
send data octets 2001-2500
ack up to 1000, window=1500
ack up to 2000, window=500
ack up to 2500, window=0
receive ack for 1000
receive ack for 2000
receive ack for 2500
application reads 2000 octets
ack up to 2500, window=2000
send data octets 2501-3500
send data octets 3501-4500
ack up to 3500, window=1000
ack up to 4500, window=0
receive ack for 3500
receive ack for 4500
application reads 1000 octets
ack up to 4500, window=1000
receive ack for 4500
⋮

Figure 25.7 A sequence of messages that illustrates TCP window advertisements for a maximum segment size of *1000* bytes.

In the figure, the sender uses a maximum segment size of *1000* bytes. Transfer begins when the receiver advertises an initial window size of *2500* bytes. The sender immediately transmits three segments, two that contain *1000* bytes of data and one that contains *500* bytes. As the segments arrive, the receiver generates an acknowledgement with the window size reduced by the amount of data that has arrived.

In the example, the first three segments fill the receiver's buffer faster than the receiving application can consume data. Thus, the advertised window size reaches zero, and the sender cannot transmit additional data. After the receiving application consumes *2000* bytes of data, the receiving TCP sends an additional acknowledgement that advertises a window size of *2000* bytes. The window size is always measured beyond the data being acknowledged, so the receiver is advertising that it can accept *2000* bytes beyond the *2500* it has already received. The sender responds by transmitting two additional segments. As each segment arrives, the receiver sends an acknowledgement with the window size reduced by *1000* bytes (i.e., the amount of data that has arrived).

Once again, the window size reaches zero, causing the sender to stop transmission. Eventually, the receiving application consumes some of the data, and the receiving TCP transmits an acknowledgement with a positive window size. If the sender has more data waiting to be sent, the sender can proceed to transmit another segment.

25.12 TCP's Three-Way Handshake

To guarantee that connections are established or terminated reliably, TCP uses a *3-way handshake* in which three messages are exchanged. During the 3-way handshake that is used to start a connection, each side sends a control message that specifies an initial buffer size (for flow control) and a sequence number. Scientists have proved that TCP's 3-way exchange is necessary and sufficient to ensure unambiguous agreement despite packet loss, duplication, delay, and replay events†. Furthermore, the handshake ensures that TCP will not open or close a connection until both ends have agreed.

To understand the 3-way handshake, imagine two applications that want to communicate in an environment where packets can be lost, duplicated, and delayed. Both sides need to agree to start a conversation, and both sides need to know that the other side has agreed. If side *A* sends a request and side *B* responds (a 2-way handshake), then *A* will know that *B* responded, but *B* will not know that *A* received the response. We can think of it this way: every transmission needs an acknowledgement. *B*'s response acknowledges *A*'s request. However, *A* must also acknowledge *B*'s response (i.e., three messages are exchanged).

TCP uses the term *synchronization segment* (*SYN segment*) to describe the control messages used in a 3-way handshake to create a connection, and the term *FIN segment* (*finish segment*) to describe control messages used in a 3-way handshake to close a connection. Figure 25.8 illustrates the 3-way handshake used to create a connection.

†Like other TCP packets, messages used for a 3-way handshake can be retransmitted.

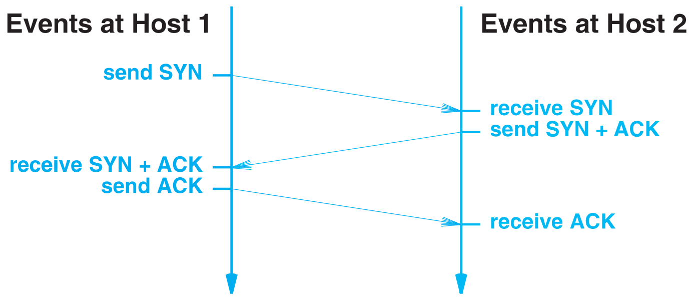

Figure 25.8 The 3-way handshake used to create a TCP connection.

At any time during the exchange, either side can crash and reboot, or a delayed packet from an old exchange can arrive. Computer scientists worked out the details of all possible problems, and added rules to ensure that TCP will establish connections correctly in all cases. For example, a key aspect of the 3-way handshake used to create a connection involves the selection of sequence numbers. TCP requires each end to generate a random 32-bit sequence number that becomes the initial sequence for data sent. If an application attempts to establish a new TCP connection after a computer reboots, TCP chooses a new random number. Because the probability of selecting a random value that matches the sequence used on a previous connection is low, TCP avoids replay problems. That is, if a pair of application programs uses TCP to communicate, closes the connection, and then establishes a new connection, the sequence numbers on the new connection will differ from the sequence numbers used on the old connection, allowing TCP to reject any delayed packets that arrive.

The 3-way handshake used to close a connection uses *FIN* segments. An acknowledgement is sent in each direction along with a FIN segment to guarantee that all data has arrived before the connection is terminated. Figure 25.9 illustrates the exchange.

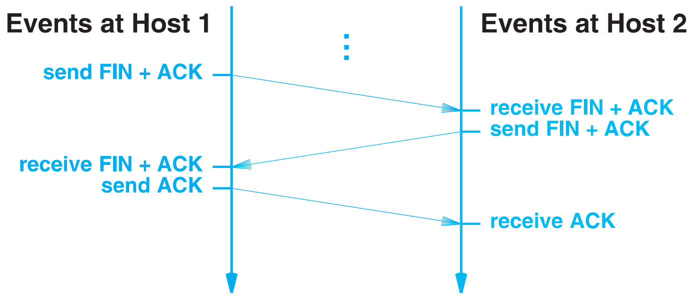

Figure 25.9 The 3-way handshake used to close a connection.

25.13 TCP Congestion Control

One of the most interesting aspects of TCP is a mechanism for *congestion control*. Recall that in the Internet, delay or packet loss is more likely to be caused by congestion than a hardware failure, and that retransmission can exacerbate the problem of congestion by injecting additional copies of a packet. To avoid congestion collapse, TCP uses changes in delay as a measure of congestion, and responds to congestion by reducing the rate at which it retransmits data.

Although we think of reducing the rate of transmission, TCP does not compute a data rate. Instead, TCP bases transmission on buffer size. That is, the receiver advertises a window size, and the sender can transmit data to fill the receiver's window before an ACK is received. To control the data rate, TCP imposes a restriction on the window size — by temporarily reducing the window size, the sending TCP effectively reduces the data rate. The important concept is:

Conceptually, a transport protocol should reduce the rate of transmission when congestion occurs. Because it uses a variable-size window, TCP can achieve a reduction in data rate by temporarily reducing the window size. In the extreme case where loss occurs, TCP temporarily reduces the window to one-half of its current value.

TCP uses a special congestion control mechanism when starting a new connection or when a message is lost. Instead of transmitting enough data to fill the receiver's buffer (i.e., the receiver's window size), TCP begins by sending a single message containing data. If an acknowledgement arrives without additional loss, TCP doubles the amount of data being sent and sends two additional messages. If both acknowledgements arrive, TCP sends four messages, and so on. The exponential increase continues until TCP is sending half of the receiver's advertised window. When one-half of the original window size is reached, TCP slows the rate of increase, and increases the window size linearly as long as congestion does not occur. The approach is known as *slow start*.

Despite the name, TCP's startup is not really slow. The exponential increase means that within a few packet exchanges, TCP transmission ramps up quickly. In the current Internet, round trip times are low (often less than 100 milliseconds), which means that within a second of starting a TCP connection, the throughput approaches the maximum that the network and the communicating hosts can handle. However, the slow start mechanism reacts well in cases where the Internet is badly congested by avoiding sending packets that will make a congested situation worse.

Once a TCP connection is running, the congestion control mechanisms respond well to impending congestion. By backing off quickly, TCP is actually able to alleviate congestion. In essence, TCP avoids adding retransmissions when the Internet becomes congested. More important, if all TCP implementations follow the standard, the congestion control scheme means that all senders back off when congestion occurs.

The important idea is that TCP handles much more than a single connection — the protocol is designed so that if all TCP implementations follow the rules, they will act in concert to avoid global congestion collapse.

25.14 Versions Of TCP Congestion Control

Minor changes to TCP's congestion control algorithm were made over many years, especially in the 1990s. By tradition, each major version is named for a city in Nevada. One of the first major versions, known as *Tahoe*, worked as outlined above. In 1990, a version known as *Reno* introduced *fast recovery* (also called *fast retransmit*) to improve throughput when loss is occasional. The next research version was known as *Vegas*. A version known as *NewReno* refined the heuristics and made further improvements. The operating system vendors who ship TCP/IP protocols with their products tend to wait for new algorithms to be validated before adopting a change. However, most operating systems now run NewReno, which handles transmission over typical networks and congestion avoidance effectively.

25.15 Other Variations: SACK And ECN

As we have seen, TCP measures round-trip delay, and uses variance as an indication of congestion. That is, TCP treats the underlying network as a black box and uses external measures to deduce that congestion has occurred. Similarly, when loss occurs, a sending TCP assumes that one packet has been lost.

Researchers have asked the question: could we make TCP perform better if the network provided more accurate information? In answering the question, the researchers invented two techniques: *Selective Acknowledgement* (*SACK*) and *Explicit Congestion Notification* (*ECN*).

The SACK mechanism changes the acknowledgment scheme, and allows a receiver to specify exactly which pieces of data are missing. A sender can retransmit just the missing pieces and avoid retransmitting data that has arrived. SACK has not paid off as much as researchers expected, because most instances of loss do not involve a random set of packets. The original TCP scheme, known as *cumulative acknowledgement* works fine when a large, contiguous block of packets are lost.

The ECN scheme was proposed as a more accurate way to handle congestion. Under ECN, routers along the path from the source to the destination monitor congestion and mark each TCP segment that passes over a congested network. When a packet reaches its destination, the receiver will know if the path is congested. When the receiver returns an ACK, the receiver tells the sender whether the acknowledged packet experienced congestion. One of the drawbacks of the ECN approach arises from the delay — a sender must wait for an ACK to come back before the sender knows about congestion (and the congestion may subside during the delay). ECN did not prove to be as useful as hoped, and is not widely adopted in the Internet.

25.16 TCP Segment Format

TCP uses a single format for all messages, including messages that carry data, those that carry acknowledgements, and messages that are part of the 3-way handshake used to create or terminate a connection (SYN and FIN). TCP uses the term *segment* to refer to a message. Figure 25.10 illustrates the TCP segment format.

To understand the segment format, it is necessary to remember that a TCP connection contains two streams of data, one flowing in each direction. If the applications at each end are sending data simultaneously, TCP can send a single segment that carries outgoing data, the acknowledgement for incoming data, and a window advertisement that specifies the amount of additional buffer space available for incoming data. Thus, some of the fields in the segment refer to the data stream traveling in the forward direction, while other fields refer to the data stream traveling in the reverse direction.

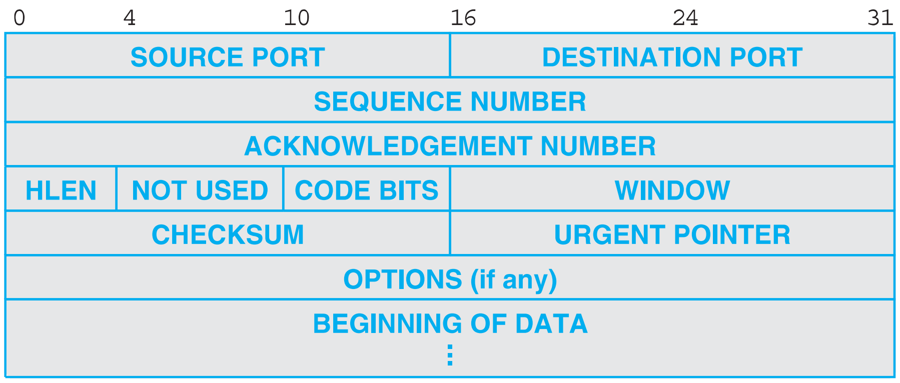

Figure 25.10 The TCP segment format used for both data and control messages.

When a computer sends a segment, the *ACKNOWLEDGEMENT NUMBER* and *WINDOW* fields refer to incoming data: the *ACKNOWLEDGEMENT NUMBER* specifies the sequence number of the data that is expected next, and the *WINDOW* specifies how much additional buffer space is available beyond the acknowledged data. The acknowledgement always refers to the first position for which data is missing; if segments arrive out of order, a receiving TCP generates the same acknowledgement multiple times until the missing data arrives. The *SEQUENCE NUMBER* field refers to outgoing data. It gives the sequence number of the first byte of data being carried in the segment. A receiver uses the sequence number to reorder segments that arrive out of order and to compute an acknowledgement number. Field *DESTINATION PORT* identifies which application program on the receiving computer should receive the data, while field *SOURCE PORT* identifies the application program that sent the data. Finally, the

CHECKSUM field contains a checksum that covers the TCP segment header and the data.

The key ideas regarding sequence and acknowledgement numbering are:

The SEQUENCE NUMBER field in a TCP segment gives the sequence number for the first byte of data carried in the segment in the forward direction; an ACKNOWLEDGEMENT NUMBER gives the first sequence number for which data is missing in the reverse direction.

25.17 Summary

The Transmission Control Protocol (TCP) is the major transport protocol in the TCP/IP protocol suite. TCP provides application programs with a reliable, flow-controlled, full-duplex, stream transport service. After requesting TCP to establish a connection, an application program can use the connection to send or receive data; TCP guarantees to deliver the data in order without duplication. Finally, when the two applications finish using a connection, they request that the connection be terminated.

TCP on one computer communicates with TCP on another computer by exchanging messages. All TCP messages sent from one computer to another use the TCP segment format, including messages that carry data, acknowledgements, and window advertisements, as well as messages used to establish and terminate a connection. Each TCP segment travels in an IP datagram.

In general, transport protocols use a variety of mechanisms to ensure reliable service. TCP has a particularly complex combination of techniques that have proven to be extremely successful. In addition to a checksum in each segment, TCP retransmits any message that is lost. To be useful in the Internet where delays vary over time, TCP's retransmission timeout is adaptive — TCP measures the current round-trip delay separately for each connection, and uses a weighted average of the round-trip times to choose a timeout for retransmission.

EXERCISES

25.1 List the features of TCP.

25.2 Assume that messages sent between two programs can be lost, duplicated, delayed, or delivered out of order. Design a protocol that reliably allows the two programs to agree to communicate. Give your design to someone, and see if they can find a sequence of loss, duplication, and delay that makes the protocol fail.

25.3 What are the main problems a transport protocol must solve to achieve reliable transfer?

25.4 What layers of a protocol stack are used on a router? A host?

25.5 When using a sliding window of size *N*, how many packets can be sent without requiring a single ACK to be received?

25.6 What are the techniques a transport protocol uses?

25.7 Extend the diagrams in Figure 25.3 to show the interaction that occurs when sixteen successive packets are sent.

25.8 Why does a stop-and-go protocol have especially low throughput over a GEO satellite channel that operates at two megabits per second?

25.9 How does TCP handle packet loss?

25.10 What is the chief cause of packet delay and loss in the Internet?

25.11 How does TCP compute a timeout for retransmission?

25.12 What happens to throughput if a protocol waits too long to retransmit? If a protocol does not wait long enough to retransmit?

25.13 What is a *SYN*? A *FIN*?

25.14 What does the TCP window size control?

25.15 What problem in a network causes TCP to reduce its window size temporarily?

25.16 Suppose two programs use TCP to establish a connection, communicate, terminate the connection, and then open a new connection. Further suppose a *FIN* message sent to shut down the first connection is duplicated and delayed until the second connection has been established. If a copy of the old *FIN* is delivered, will TCP terminate the new connection? Why or why not?

25.17 Is the TCP checksum necessary, or can TCP depend on the IP checksum to ensure integrity? Explain.

25.18 Write a computer program to extract and print fields in a TCP segment header.