Module – 2

TRANSPORT LAYER

Introduction and Transport-Layer Services

A transport-layer protocol provides for **logical communication** between application processes running on different hosts. By *logical communication*, we mean that from an application's perspective, it is as if the hosts running the processes were directly connected transport-layer protocols are implemented in the end systems but not in network routers. On the sending side, the transport layer converts the application-layer messages it receives from a sending application process into transport-layer packets, known as transport-layer **segments** in Internet terminology.

Relationship Between Transport and Network Layers

Transport-layer protocol provides logical communication between *processes* running on different hosts, a network-layer protocol provides logical communication between *hosts*.

Overview of the Transport Layer in the Internet

Recall that the Internet, and more generally a TCP/IP network, makes two distinct transport-layer protocols available to the application layer. One of these protocols is **UDP** (User Datagram Protocol), which provides an unreliable, connectionless service to the invoking application. The second of these protocols is **TCP** (Transmission Control Protocol), which provides a reliable, connection-oriented service to the invoking Application.

The IP service model is a **best-effort delivery service**. This means that IP makes its "best effort" to deliver segments between communicating hosts, *but it makes no guarantees.* It does not guarantee segment delivery, it does not guarantee orderly delivery of segments, and it does not guarantee the integrity of the data in the segments. For these reasons, IP is said to be an **unreliable service.**
The most fundamental responsibility of UDP and TCP is to extend IP's delivery service between two end systems to a delivery service between two processes running on the end systems. Extending host-to-host delivery to process-to-process delivery is called **transport-layer multiplexing** and **demultiplexing**.
TCP, on the other hand, offers several additional services to applications. First and foremost, it provides **reliable data transfer**. Using flow control, sequence numbers, acknowledgments, and timers (techniques we'll explore in detail in this chapter), TCP ensures that data is delivered from sending process to receiving process, correctly and in order. TCP thus converts IP's unreliable service between end systems into a reliable data transport service between processes. TCP also provides **congestion control**. TCP congestion control prevents any one TCP connection from swamping the links and routers between communicating hosts with an excessive amount of traffic. A protocol that provides reliable data transfer and congestion control is necessarily Complex

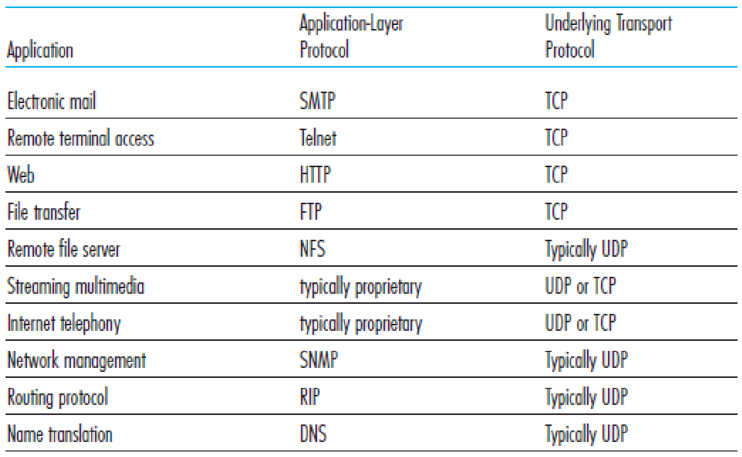
Connectionless Transport: UDP

If the application developer chooses UDP instead of TCP, then the application is almost directly talking with IP. UDP takes messages from the application process, attaches source and destination port number fields for the multiplexing/demultiplexing service, adds two other small fields, and passes the resulting segment to the network layer. The network layer encapsulates the transport-layer segment into an IP datagram and then makes a best-effort attempt to deliver the segment to the receiving host. If the segment arrives at the receiving host, UDP uses the destination port number to deliver the segment's data to the correct application process. Note that with UDP there is no handshaking between sending and receiving transport-layer entities before sending a segment. For this reason, UDP is said to be *connectionless.*
DNS is an example of an application-layer protocol that typically uses UDP.

Why an application developer would ever choose to build an application over UDP rather than over TCP.

Finer application-level control over what data is sent, and when. Under UDP, as soon as an application process passes data to UDP, UDP will package the data inside a UDP segment and immediately pass the segment to the network layer. TCP, on the other hand, has a congestion-control mechanism that throttles the transport-layer TCP sender when one or more links between the source and destination hosts become excessively congested. TCP will also continue to resend a segment until the receipt of the segment has been acknowledged by the destination, regardless of how long reliable delivery takes.
No connection establishment. As we'll discuss later, TCP uses a three-way handshake before it starts to transfer data. UDP just blasts away without any formal preliminaries. Thus UDP does not introduce any delay to establish a connection. This is probably the principal reason why DNS runs over UDP rather than TCP—DNS would be much slower if it ran over TCP. HTTP uses TCP rather than UDP, since reliability is critical for Web pages with text.

No connection state. TCP maintains connection state in the end systems. This connection state includes receive and send buffers, congestion-control parameters, and sequence and acknowledgment number parameters. UDP, on the other hand, does not maintain connection state and does not track any of these parameters.
Small packet header overhead. The TCP segment has 20 bytes of header overhead in every segment, whereas UDP has only 8 bytes of overhead.

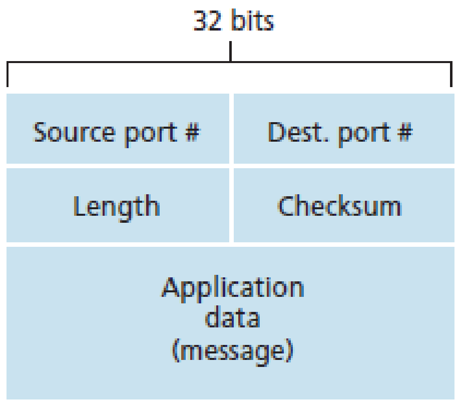


Application	Application-Layer Protocol	Underlying Transport Protocol
Electronic mail	SMTP	TCP
Remote terminal access	Telnet	TCP
Web	HTTP	TCP
File transfer	FTP	TCP
Remote file server	NFS	Typically UDP
Streaming multimedia	typically proprietary	UDP or TCP
Internet telephony	typically proprietary	UDP or TCP
Network management	SNMP	Typically UDP
Routing protocol	RIP	Typically UDP
Name translation	DNS	Typically UDP

Figure 3.6 ♦ Popular Internet applications and their underlying transport protocols

Although commonly done today, running multimedia applications over UDP is controversial. UDP has no congestion control

UDP Segment Structure



The UDP header has only four fields, each consisting of two bytes. The port numbers allow the destination host to pass the application data to the correct process running on the destination end system (that is, to perform the demultiplexing function). The length field specifies the number of bytes in the UDP segment (header plus data). An explicit length value is needed since the size of the data field may differ from one UDP segment to the next. The checksum is used by the receiving host to check whether errors have been introduced into the segment. In truth, the checksum is also calculated over a few of the fields in the IP header in addition to the UDP segment

UDP Checksum

0110011001100000
0101010101010101
1000111100001100

The sum of first two of these 16-bit words is

0110011001100000
0101010101010101
1011101110110101

Adding the third word to the above sum gives

1011101110110101
1000111100001100
0100101011000010

That is, the checksum is used to determine whether bits within the UDP segment have been altered (for example, by noise in the links or while stored in a router) as it moved from source to destination. UDP at the sender side performs the 1s complement of the sum of all the 16-bit words in the segment, with any overflow encountered during the sum being wrapped around. This result is put in the checksum field of the UDP segment. Here we give a simple example of the checksum calculation

Note that this last addition had overflow, which was wrapped around. The 1s complement is obtained by converting all the 0s to 1s and converting all the 1s to 0s.
Thus the 1s complement of the sum 0100101011000010 is 1011010100111101, which becomes the checksum. At the receiver, all four 16-bit words are added, including the checksum. If no errors are introduced into the packet, then clearly the sum at the receiver will be 1111111111111111. If one of the bits is a 0, then we know that errors have been introduced into the packet.

Principles of Reliable Data Transfer

Figure 3.8 illustrates the framework for our study of reliable data transfer. The service abstraction provided to the upper-layer entities is that of a reliable channel through which data can be transferred. With a reliable channel, no transferred data bits are corrupted (flipped from 0 to 1, or vice versa) or lost, and all are delivered in the order in which they were sent. This is precisely the service model offered by TCP to the Internet applications that invoke it.

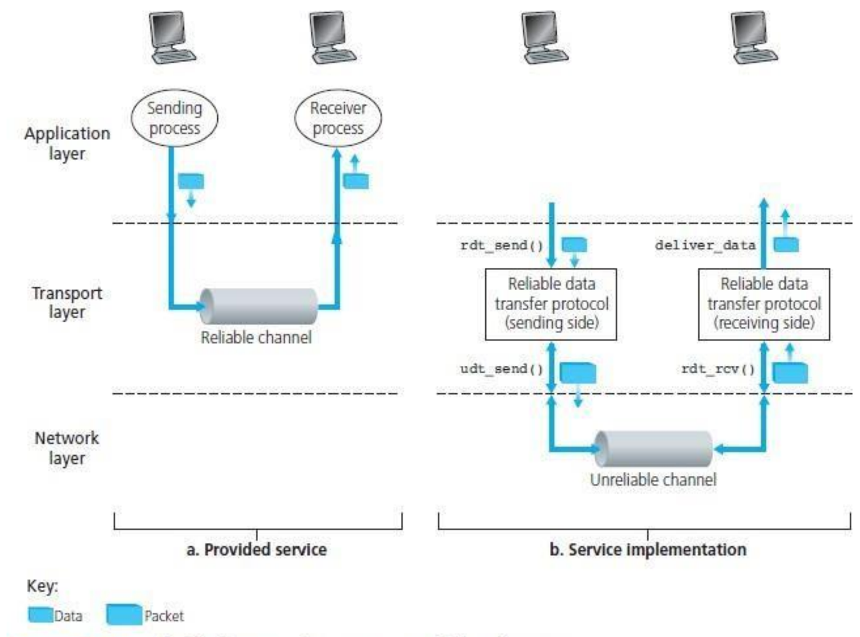
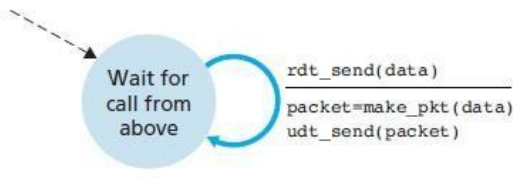


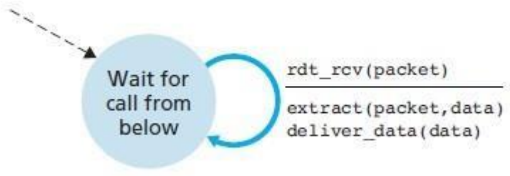
Figure 3.8 ♦ Reliable data transfer: Service model and service implementation

Reliable Data Transfer over a Perfectly Reliable Channel: rdt1.0

The protocol itself, which we will call rdt1.0, is trivial. The finite-state machine (FSM) definitions for the rdt1.0 sender and receiver are shown in Figure 3.9



a. rdt1.0: sending side



b. rdt1.0: receiving side

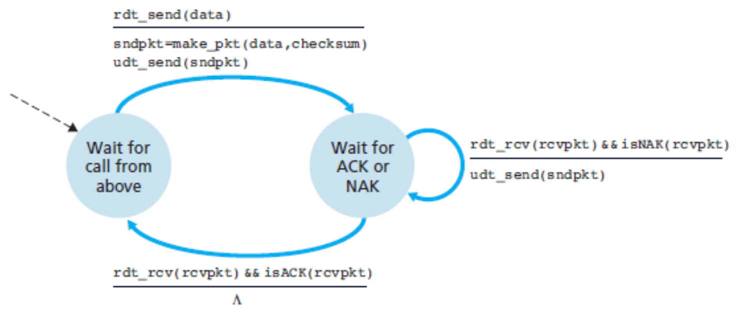
Figure 3.9 ♦ rdt1.0 – A protocol for a completely reliable channel

The sending side of rdt simply accepts data from the upper layer via the rdt_send(data) event, creates a packet containing the data (via the action make_pkt(data)) and sends the packet into the channel. On the receiving side, rdt receives a packet from the underlying channel via the rdt_rcv(packet) event, removes the data from the packet (via the action extract (packet, data)) and passes the data up to the upper layer (via the action deliver_data(data)). In practice, the rdt_rcv(packet) event would result from a procedure call (for example, to rdt_rcv()) from the lowerlayer protocol.

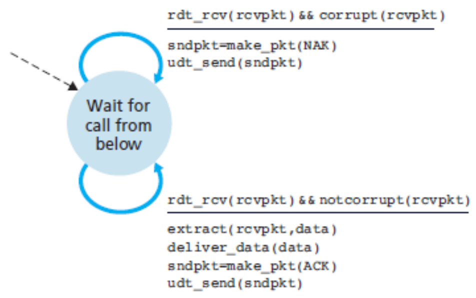
Reliable Data Transfer over a Channel with Bit Errors: rdt2.0

If the message taker hears a garbled sentence, you are asked to repeat the garbled sentence. This message- dictation protocol uses both **positive acknowledgments** (“OK”) and **negative acknowledgments** (“Please repeat that.”). These control messages allow the receiver to let the sender know what has been received correctly, and what has been received in error and thus requires repeating. In a computer network setting, reliable data transfer protocols based on such retransmission are known as **ARQ** (**Automatic Repeat reQuest) protocols**.

Three additional protocol capabilities are required in ARQ protocols to handle the presence of bit errors:
• ***Error detection***. First, a mechanism is needed to allow the receiver to detect when bit errors have occurred.
• ***Receiver feedback***. The positive (ACK) and negative (NAK) acknowledgment replies in the message-dictation scenario are examples of such feedback. Our rdt2.0 protocol will similarly send ACK and NAK packets back from the receiver to the sender.
• ***Retransmission***. A packet that is received in error at the receiver will be retransmitted by the sender.


rdt_send(data)
sndpkt=make_pkt(data,checksum)
udt_send(sndpkt)
Wait for call from above
Wait for ACK or NAK
rdt_rcv(rcvpkt) && isNAK(rcvpkt)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && isACK(rcvpkt)
Λ

a. rdt2.0: sending side



b. rdt2.0: receiving side

Figure 3.10 ♦ rdt2.0–A protocol for a channel with bit errors

The rdt2.1 sender and receiver FSMs each now have twice as many states as before. This is because the protocol state must now reflect whether the packet currently being sent (by the sender) or expected (at the receiver) should have a sequence number of 0 or 1.
Protocol rdt2.1 uses both positive and negative acknowledgments from the receiver to the sender. When an out-of-order packet is received, the receiver sends a positive acknowledgment for the packet it has received. When a corrupted packet is received, the receiver sends a negative acknowledgment. We can accomplish the same effect as a NAK if, instead of sending a NAK, we send an ACK for the last correctly received packet. A sender that receives two ACKs for the same packet (that is, receives **duplicate ACKs**) knows that the receiver did not correctly receive the packet following the packet that is being ACKed twice.

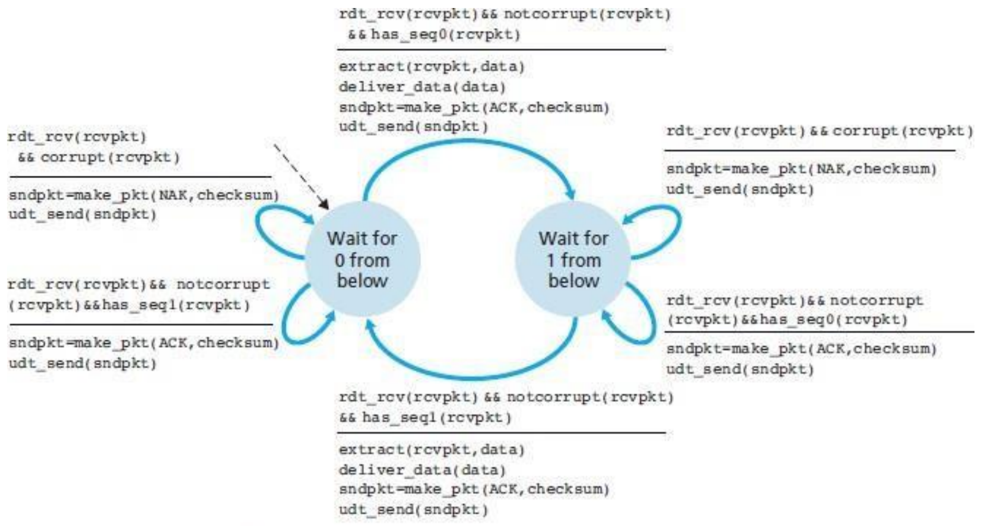

rdt_rcv(rcvpkt)&& notcorrupt(rcvpkt) && has_seq0(rcvpkt)
extract(rcvpkt,data) deliver_data(data) sndpkt=make_pkt(ACK,checksum) udt_send(sndpkt)
rdt_rcv(rcvpkt) && corrupt(rcvpkt)
sndpkt=make_pkt(NAK,checksum) udt_send(sndpkt)
rdt_rcv(rcvpkt) && corrupt(rcvpkt)
sndpkt=make_pkt(NAK,checksum) udt_send(sndpkt)
Wait for 0 from below
Wait for 1 from below
rdt_rcv(rcvpkt)&& notcorrupt (rcvpkt)&&has_seq1(rcvpkt)
sndpkt=make_pkt(ACK,checksum) udt_send(sndpkt)
rdt_rcv(rcvpkt)&& notcorrupt (rcvpkt)&&has_seq0(rcvpkt)
sndpkt=make_pkt(ACK,checksum) udt_send(sndpkt)
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq1(rcvpkt)
extract(rcvpkt,data) deliver_data(data) sndpkt=make_pkt(ACK,checksum) udt_send(sndpkt)

Figure 3.12 ♦ rdt2.1 receiver

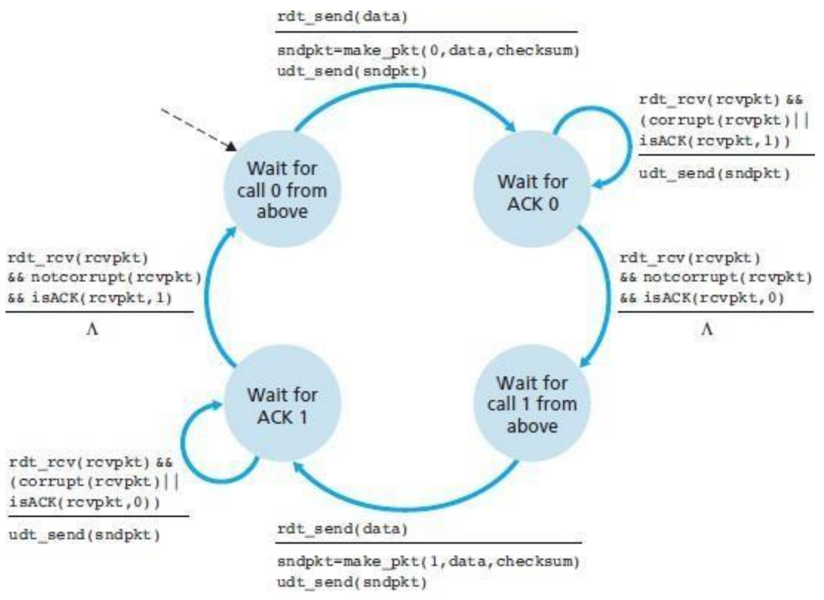

rdt_send(data)
sndpkt=make_pkt(0,data,checksum) udt_send(sndpkt)
Wait for call 0 from above
Wait for ACK 0
rdt_rcv(rcvpkt) && (corrupt(rcvpkt)|| isACK(rcvpkt,1))
udt_send(sndpkt)
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,1)
Λ
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,0)
Λ
Wait for ACK 1
Wait for call 1 from above
rdt_rcv(rcvpkt) && (corrupt(rcvpkt)|| isACK(rcvpkt,0))
udt_send(sndpkt)
rdt_send(data)
sndpkt=make_pkt(1,data,checksum) udt_send(sndpkt)

Figure 3.13 ♦ rdt2.2 sender

Reliable Data Transfer over a Lossy Channel with Bit Errors: rdt3.0

Implementing a time-based retransmission mechanism requires a **countdown timer** that can interrupt the sender after a given amount of time has expired. The sender will thus need to be able to (1) start the timer each time a packet (either a first-time packet or a retransmission) is sent, (2) respond to a timer interrupt (taking appropriate actions), and (3) stop the timer.

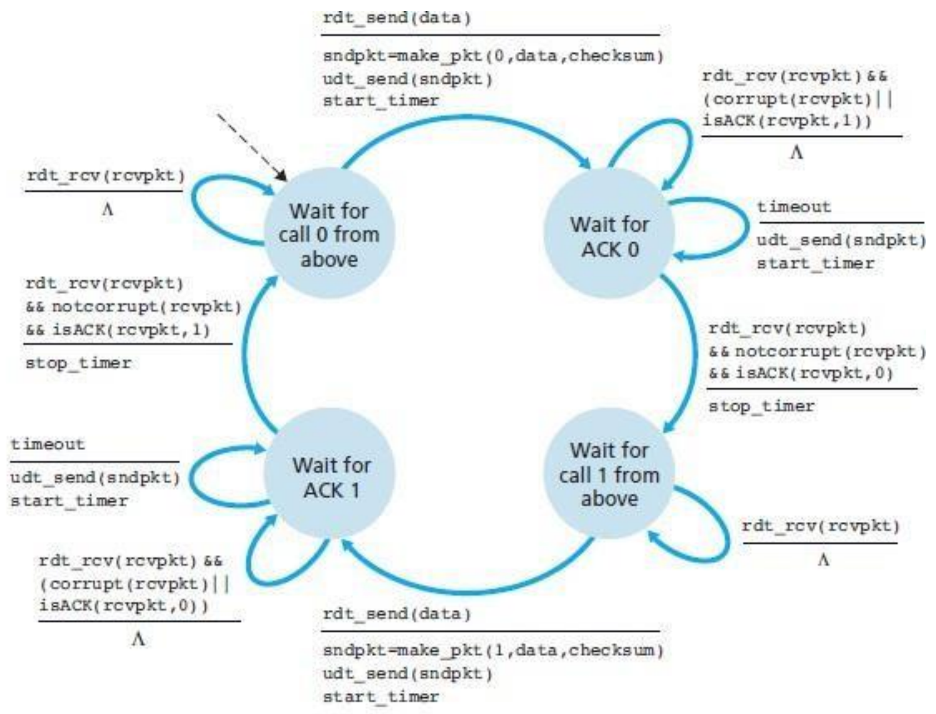

rdt_send(data)
sndpkt=make_pkt(0,data,checksum)
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt) && (corrupt(rcvpkt)|| isACK(rcvpkt,1))
Λ
rdt_rcv(rcvpkt)
Λ
Wait for call 0 from above
Wait for ACK 0
timeout
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,1)
stop_timer
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,0)
stop_timer
timeout
udt_send(sndpkt)
start_timer
Wait for ACK 1
Wait for call 1 from above
rdt_rcv(rcvpkt)
Λ
rdt_rcv(rcvpkt) && (corrupt(rcvpkt)|| isACK(rcvpkt,0))
Λ
rdt_send(data)
sndpkt=make_pkt(1,data,checksum)
udt_send(sndpkt)
start_timer

Figure 3.15 ♦ rdt3.0 sender

Pipelined Reliable Data Transfer Protocols

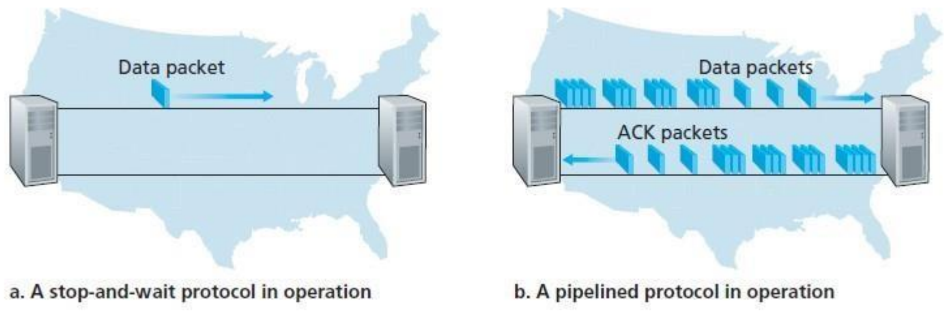

Data packet
Data packets
ACK packets
a. A stop-and-wait protocol in operation
b. A pipelined protocol in operation

Figure 3.17 ♦ Stop-and-wait versus pipelined protocol

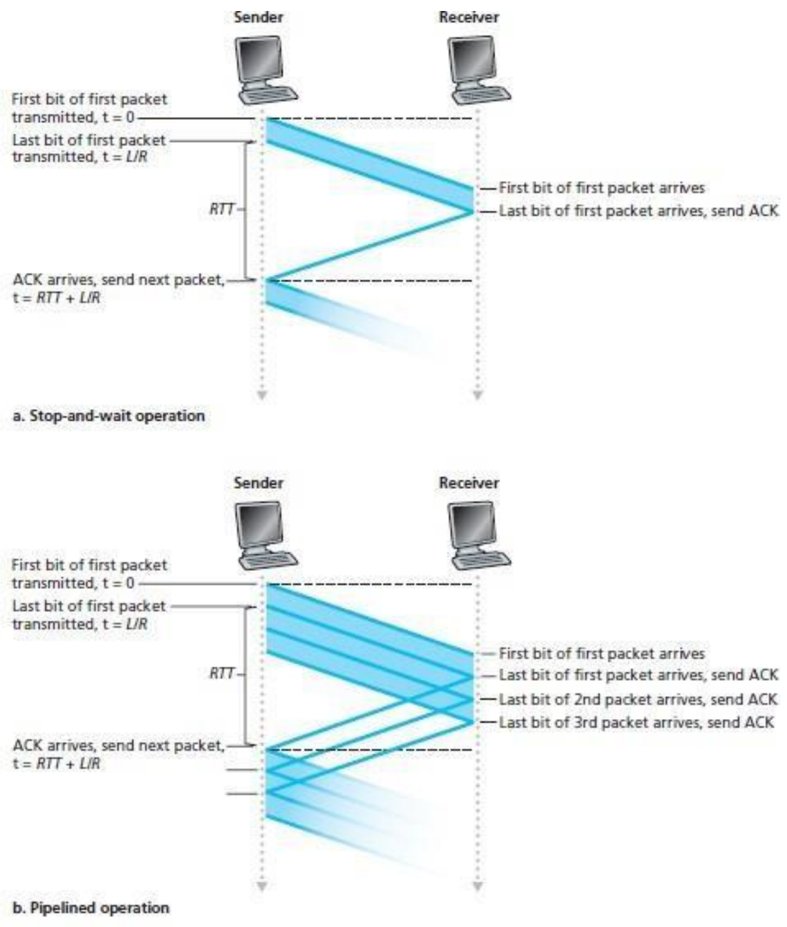

Sender
Receiver
First bit of first packet transmitted, t = 0
Last bit of first packet transmitted, t = L/R
RTT
First bit of first packet arrives
Last bit of first packet arrives, send ACK
ACK arrives, send next packet, t = RTT + L/R
a. Stop-and-wait operation
Sender
Receiver
First bit of first packet transmitted, t = 0
Last bit of first packet transmitted, t = L/R
RTT
First bit of first packet arrives
Last bit of first packet arrives, send ACK
Last bit of 2nd packet arrives, send ACK
Last bit of 3rd packet arrives, send ACK
ACK arrives, send next packet, t = RTT + L/R
b. Pipelined operation

Figure 3.18 ♦ Stop-and-wait and pipelined sending

Go-Back-N (GBN)

In a **Go-Back-N (GBN) protocol**, the sender is allowed to transmit multiple packets (when available) without waiting for an acknowledgment, but is constrained to have no more than some maximum allowable number, *N*, of unacknowledged packets in the pipeline.
If we define base to be the sequence number of the oldest unacknowledged packet and nextseqnum to be the smallest unused sequence number (that is, the sequence number of the next packet to be sent), then four intervals in the range of sequence numbers can be identified. Sequence numbers in the interval [0,base-1] correspond to packets that have already been transmitted and acknowledged. The interval [base,nextseqnum-1] corresponds to packets that have been sent but not yet acknowledged.
Sequence numbers in the interval [nextseqnum,base+N-1] can be used for packets that can be sent immediately, should data arrive from the upper layer. Finally, sequence numbers greater than or equal to base+N cannot be used until an unacknowledged packet currently in the pipeline (specifically, the packet with sequence number base) has been acknowledged. As the protocol operates, this window slides forward over the sequence number space. For this reason, *N* is often referred to as the **window size** and the GBN protocol itself as a **sliding-window protocol**. If *k* is the number of bits in the packet sequence number field, the range of sequence numbers is thus [0,2*k* – 1].

With a finite range of sequence numbers, all arithmetic involving sequence numbers must then be done using modulo 2*k* arithmetic

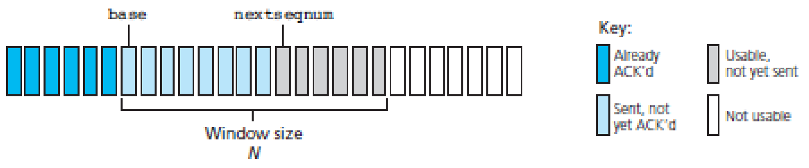
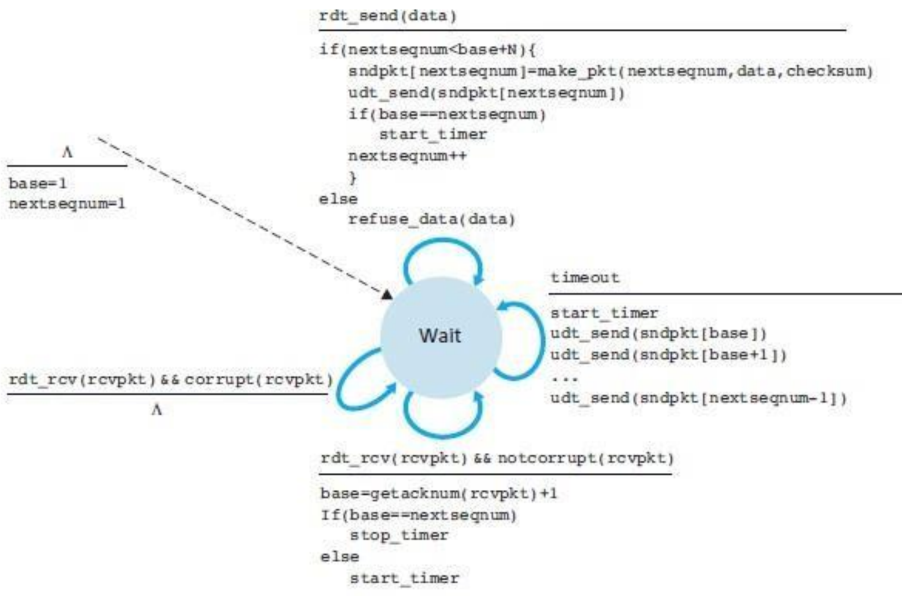


Figure 3.19 ♦ Sender's view of sequence numbers in Go-Back-N

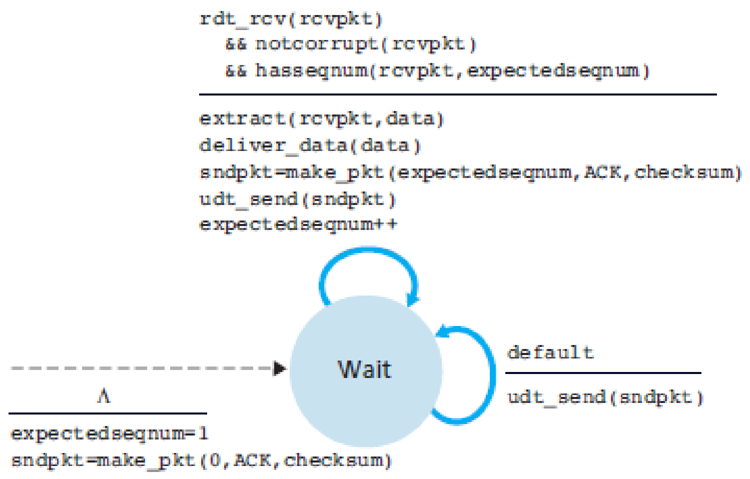


```
rdt_send(data)
_______________
if(nextseqnum<base+N){
   sndpkt[nextseqnum]=make_pkt(nextseqnum,data,checksum)
   udt_send(sndpkt[nextseqnum])
   if(base==nextseqnum)
      start_timer
   nextseqnum++
   }
else
   refuse_data(data)
```

```
timeout
_______________
start_timer
udt_send(sndpkt[base])
udt_send(sndpkt[base+1])
...
udt_send(sndpkt[nextseqnum-1])
```

```
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
_______________
base=getacknum(rcvpkt)+1
If(base==nextseqnum)
   stop_timer
else
   start_timer
```

Figure 3.20 ♦ Extended FSM description of GBN sender



```
rdt_rcv(rcvpkt)
  && notcorrupt(rcvpkt)
  && hasseqnum(rcvpkt,expectedseqnum)
_______________
extract(rcvpkt,data)
deliver_data(data)
sndpkt=make_pkt(expectedseqnum,ACK,checksum)
udt_send(sndpkt)
expectedseqnum++
```

Figure 3.21 ♦ Extended FSM description of GBN receiver

The GBN sender must respond to three types of events:

• ***Invocation from above***. When rdt_send() is called from above, the sender first checks to see if the window is full, that is, whether there are *N* outstanding, unacknowledged packets. If the window is not full, a packet is created and sent, and variables are appropriately updated. If the window is full, the sender simply returns the data back to the upper layer, an implicit indication that the window is full. The upper layer would presumably then have to try again later. In a real implementation, the sender would more likely have either buffered (but not immediately sent) this data, or would have a synchronization mechanism (for example, a semaphore or a flag) that would allow the upper layer to call rdt_send() only when the window is not full.

• ***Receipt of an ACK***. In our GBN protocol, an acknowledgment for a packet with sequence number *n* will be taken to be a **cumulative acknowledgment**, indicating that all packets with a sequence number up to and including *n* have been correctly received at the receiver.

• ***A timeout event***. The protocol's name, "Go-Back-N," is derived from the sender's behavior in the presence of lost or overly delayed packets. As in the stop-and-wait protocol, a timer will again be used to recover from lost data or acknowledgment packets. If a timeout occurs, the sender resends *all* packets that have been previously sent but that have not yet been acknowledged. If an ACK is received but there are still additional transmitted but not yet acknowledged packets, the timer is restarted. If there are no outstanding, unacknowledged packets, the timer is stopped.

The receiver's actions in GBN are also simple. If a packet with sequence number *n* is received correctly and is in order (that is, the data last delivered to the upper layer came from a packet with sequence number *n* – 1), the receiver sends an ACK for packet *n* and delivers the data portion of the packet to the upper layer. In all other cases, the receiver discards the packet and resends an ACK for the most recently received in-order packet. Note that since packets are delivered one at a time to the upper layer, if packet *k* has been received and delivered, then all packets with a sequence number lower than *k* have also been delivered. Thus, the use of cumulative acknowledgments is a natural choice for GBN. In our GBN protocol, the receiver discards out-of-order packets. Although it may seem silly and wasteful to discard a correctly received (but out-of-order) packet, there is some justification for doing so.

Suppose now that packet *n* is expected, but packet *n* + 1 arrives. Because data must be delivered in order, the receiver *could* buffer (save) packet *n* + 1 and then deliver this packet to the upper layer after it had later received and delivered packet *n*. However, if packet *n* is lost, both it and packet *n* + 1 will eventually be retransmitted as a result of the GBN retransmission rule at the sender. Thus, the receiver can simply discard packet *n* + 1. The advantage of this approach is the simplicity of receiver buffering—the receiver need not buffer *any* out-of-order packets.

Figure 3.22 shows the operation of the GBN protocol for the case of a window size of four packets. Because of this window size limitation, the sender sends packets 0 through 3 but then must wait for one or more of these packets to be acknowledged before proceeding. As each successive ACK (for example, ACK0 and ACK1) is received, the window slides forward and the sender can transmit one new packet (pkt4 and pkt5, respectively). On the receiver side, packet 2 is lost and thus packets 3, 4, and 5 are found to be out of order and are discarded.

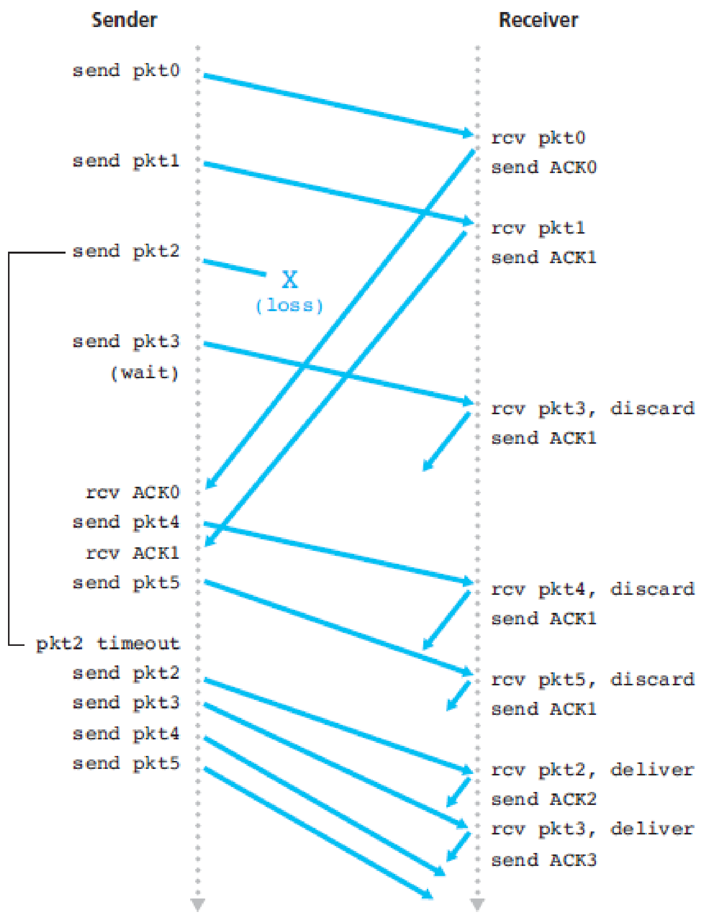


Figure 3.22 ♦ Go-Back-N in operation

Selective Repeat (SR)

GBN itself suffers from performance problems. When the window size and bandwidth-delay product are both large, many packets can be in the pipeline. A single packet error can thus cause GBN to retransmit a large number of packets, many unnecessarily. As the probability of channel errors increases, the pipeline can become filled with these unnecessary retransmissions. Imagine, in our message-dictation scenario, that if every time a word was garbled, the surrounding 1,000 words (for example, a window size of 1,000 words) had to be repeated.

Selective-Repeat protocols avoid unnecessary retransmissions by having the sender retransmit only those packets that it suspects were received in error (that is, were lost or corrupted) at the receiver window size of *N* will again be used to limit the number of outstanding, unacknowledged packets in the pipeline.

However, unlike GBN, the sender will have already received ACKs for some of the packets in the window

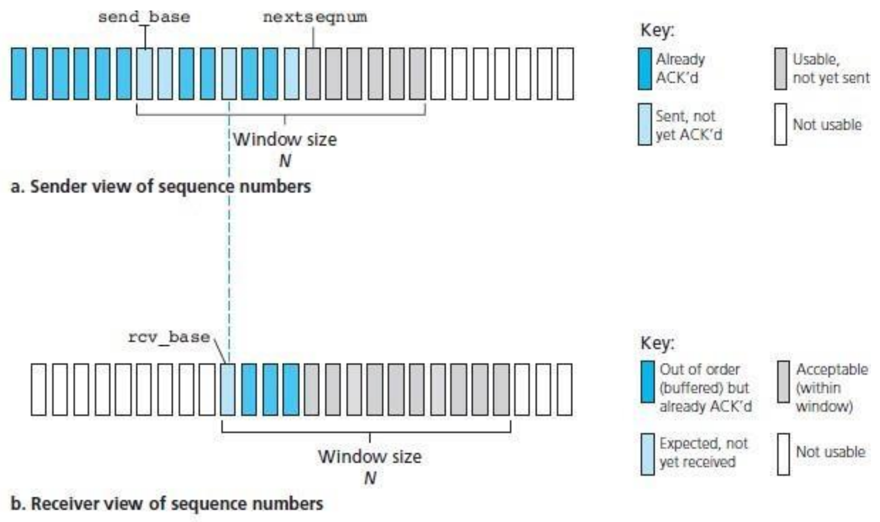

send_base
nextseqnum
Window size
N
a. Sender view of sequence numbers
Key:
Already ACK'd
Sent, not yet ACK'd
Usable, not yet sent
Not usable
rcv_base
Window size
N
b. Receiver view of sequence numbers
Key:
Out of order (buffered) but already ACK'd
Expected, not yet received
Acceptable (within window)
Not usable

Figure 3.23 ♦ Selective-repeat (SR) sender and receiver views of sequence-number space

The SR receiver will acknowledge a correctly received packet whether or not it is in order. Out-of-order packets are buffered until any missing packets (that is, packets with lower sequence numbers) are received, at which point a batch of packets can be delivered in order to the upper layer SR sender events and actions

1. ***Data received from above***. When data is received from above, the SR sender checks the next available sequence number for the packet. If the sequence number is within the sender's window, the data is packetized and sent; otherwise it is either buffered or returned to the upper layer for later transmission, as in GBN.
2. ***Timeout.*** Timers are again used to protect against lost packets. However, each packet must now have its own logical timer, since only a single packet will be transmitted on timeout. A single hardware timer can be used to mimic the operation of multiple logical timers.
3. ***ACK received***. If an ACK is received, the SR sender marks that packet as having been received, provided it is in the window. If the packet's sequence number is equal to send_base, the window base is moved forward to the unacknowledged packet with the smallest sequence number. If the window moves and there are untransmitted packets with sequence numbers that now fall within the window, these packets are transmitted.

SR receiver events and actions

1. ***Packet with sequence number in* [rcv_base, rcv_base+N-1] *is correctly received***. In this case, the received packet falls within the receiver's window and a selective ACK packet is returned to the sender. If the packet was not previously received, it is buffered. If this packet has a sequence number equal to the base of the receive window (rcv_base in Figure 3.22), then this packet, and any previously buffered and consecutively numbered (beginning with rcv_base) packets are delivered to the upper layer. The receive window is then moved forward by the number of packets delivered to the upper layer.
2. ***Packet with sequence number in* [rcv_base-N, rcv_base-1] *is correctly received.*** In this case, an ACK must be generated, even though this is a packet that the receiver has previously acknowledged.
3. ***Otherwise***. Ignore the packet.

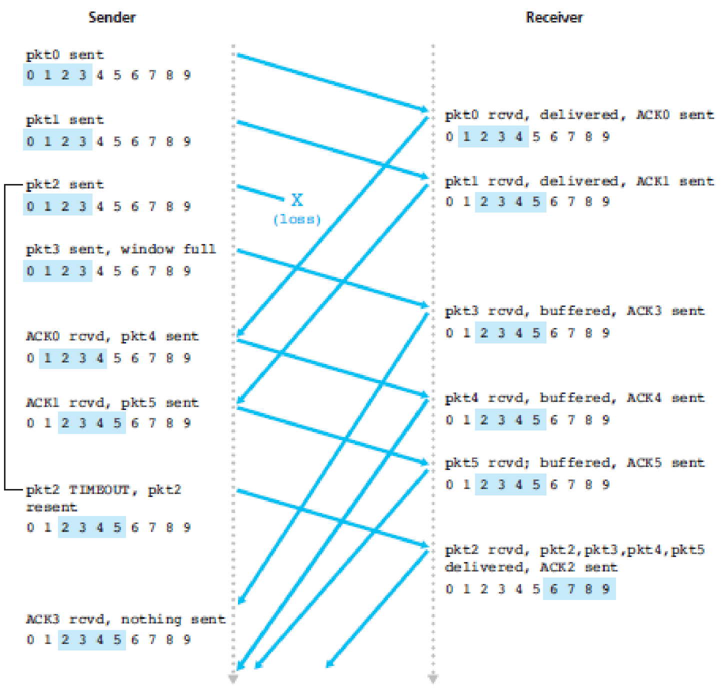
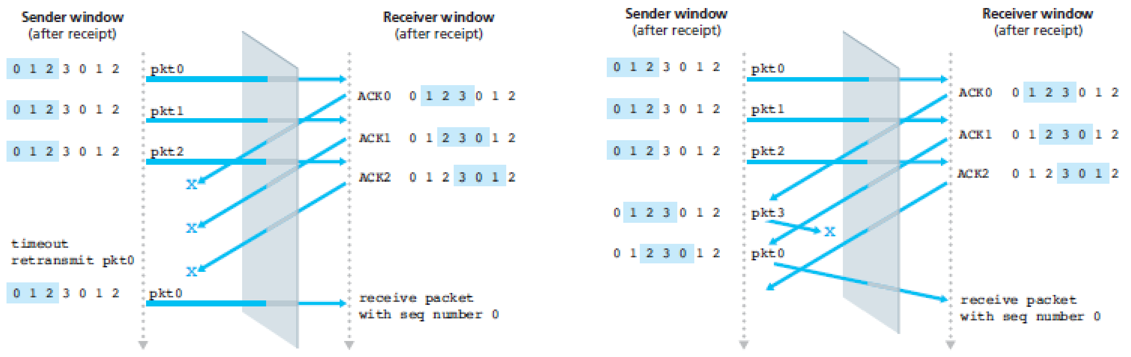
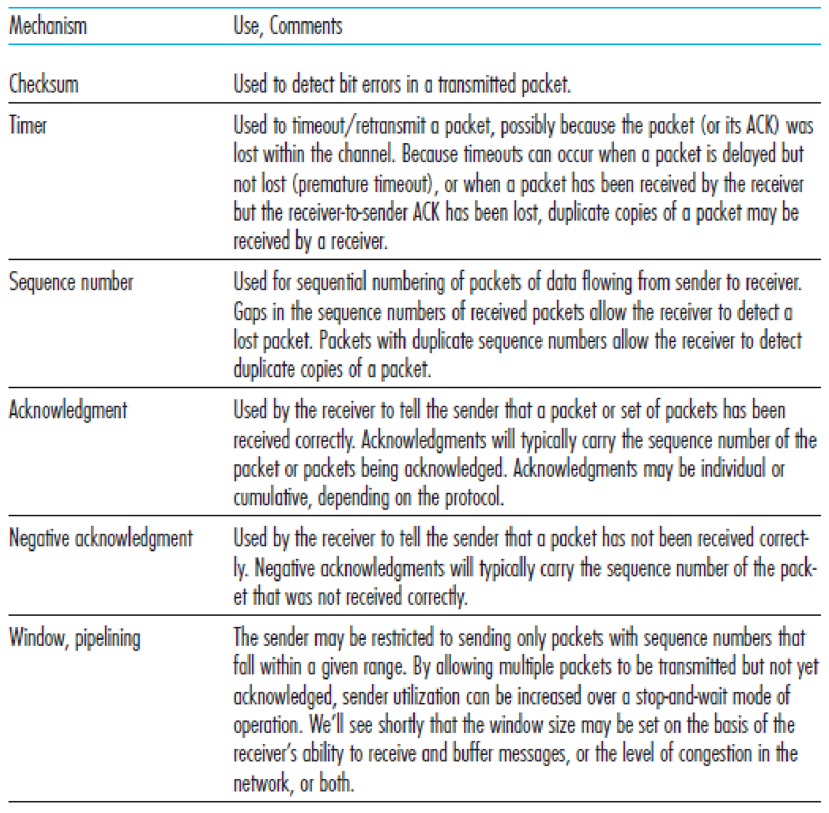

Sender
Receiver
pkt0 sent
pkt1 sent
pkt2 sent
pkt3 sent, window full
ACK0 rcvd, pkt4 sent
ACK1 rcvd, pkt5 sent
pkt2 TIMEOUT, pkt2 resent
ACK3 rcvd, nothing sent
X (loss)
pkt0 rcvd, delivered, ACK0 sent
pkt1 rcvd, delivered, ACK1 sent
pkt3 rcvd, buffered, ACK3 sent
pkt4 rcvd, buffered, ACK4 sent
pkt5 rcvd; buffered, ACK5 sent
pkt2 rcvd, pkt2,pkt3,pkt4,pkt5 delivered, ACK2 sent
0 1 2 3 4 5 6 7 8 9

Figure 3.26 ♦ SR operation

The lack of synchronization between sender and receiver windows has important consequences when we are faced with the reality of a finite range of sequence numbers. Consider what could happen, for example, with a finite range of four packet sequence numbers, 0, 1, 2, 3, and a window size of three. Suppose packets 0 through 2 are transmitted and correctly received and acknowledged at the receiver. At this point, the receiver's window is over the fourth, fifth, and sixth packets, which have sequence numbers 3, 0, and 1, respectively. Now consider two scenarios. In the first scenario, shown in Figure 3.27(a), the ACKs for the first three packets are lost and The lack of synchronization between sender and receiver windows has important consequences when we are faced with the reality of a finite range of sequence numbers. Consider what could happen, for example, with a finite range of four packet sequence numbers, 0, 1, 2, 3, and a window size of three. Suppose packets 0 through 2 are transmitted and correctly received and acknowledged at the receiver. At this point, the receiver's window is over the fourth, fifth, and sixth packets, which have sequence numbers 3, 0, and 1, respectively. Now consider two scenarios.


Sender window (after receipt)
Receiver window (after receipt)
0 1 2 3 0 1 2
pkt0
pkt1
pkt2
pkt3
ACK0
ACK1
ACK2
x
timeout retransmit pkt0
receive packet with seq number 0



Mechanism	Use, Comments
Checksum	Used to detect bit errors in a transmitted packet.
Timer	Used to timeout/retransmit a packet, possibly because the packet (or its ACK) was lost within the channel. Because timeouts can occur when a packet is delayed but not lost (premature timeout), or when a packet has been received by the receiver but the receiver-to-sender ACK has been lost, duplicate copies of a packet may be received by a receiver.
Sequence number	Used for sequential numbering of packets of data flowing from sender to receiver. Gaps in the sequence numbers of received packets allow the receiver to detect a lost packet. Packets with duplicate sequence numbers allow the receiver to detect duplicate copies of a packet.
Acknowledgment	Used by the receiver to tell the sender that a packet or set of packets has been received correctly. Acknowledgments will typically carry the sequence number of the packet or packets being acknowledged. Acknowledgments may be individual or cumulative, depending on the protocol.
Negative acknowledgment	Used by the receiver to tell the sender that a packet has not been received correctly. Negative acknowledgments will typically carry the sequence number of the packet that was not received correctly.
Window, pipelining	The sender may be restricted to sending only packets with sequence numbers that fall within a given range. By allowing multiple packets to be transmitted but not yet acknowledged, sender utilization can be increased over a stop-and-wait mode of operation. We'll see shortly that the window size may be set on the basis of the receiver's ability to receive and buffer messages, or the level of congestion in the network, or both.

Table 3.1 ♦ Summary of reliable data transfer mechanisms and their use

Connection-Oriented Transport: TCP

TCP relies on many principles, error detection, retransmissions, cumulative acknowledgments, timers, and header fields for sequence and acknowledgment numbers.

The TCP Connection

TCP is said to be **connection-oriented** because before one application process can begin to send data to another, the two processes must first "handshake" with each other—that is, they must send some preliminary segments to each other to establish the parameters of the ensuing data transfer.
A TCP connection provides a **full-duplex service**: If there is a TCP connection between Process A on one host and Process B on another host, then application layer data can flow from Process A to Process B at the same time as application layer data flows from Process B to Process A. A TCP connection is also always **point-to-point**, that is, between a single sender and a single receiver Recall that the process that is initiating the connection is called the *client process*, while the other process is called the *server process*. The client application process first informs the client transport layer that it wants to establish a connection to a process in the server
clientSocket.connect((serverName,serverPort))
TCP in the client then proceeds to establish a TCP connection with TCP in the server. At the end of this section we discuss in some detail the connection- establishment procedure.

Once a TCP connection is established, the two application processes can send data to each other.TCP directs this data to the connection's **send buffer**, which is one of the buffers that is set aside during the initial three-way handshake. From time to time, TCP will grab chunks of data from the send buffer and pass the data to the network layer.

The maximum amount of data that can be grabbed and placed in a segment is limited by the **maximum segment size (MSS)**. The MSS is typically set by first determining the length of the largest link-layer frame that can be sent by the local sending host (the so-called **maximum transmission unit**, **MTU**), and then setting the MSS to ensure that a TCP segment (when encapsulated in an IP datagram) plus the TCP/IP header length (typically 40 bytes) will fit into a single link-layer frame.

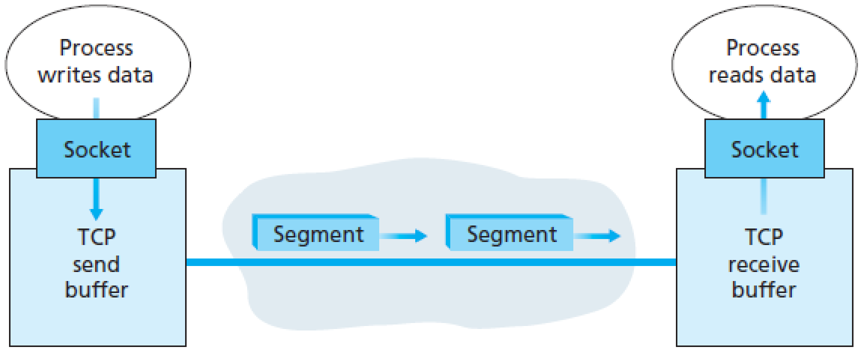


Figure 3.28 ♦ TCP send and receive buffers

TCP pairs each chunk of client data with a TCP header, thereby forming **TCP segments**. The segments are passed down to the network layer, where they are separately encapsulated within network-layer IP datagrams. The IP datagrams are then sent into the network. When TCP receives a segment at the other end, the segment's data is placed in the TCP connection's receive buffer.

TCP Segment Structure

The TCP segment consists of header fields and a data field. The data field contains a chunk of application data. The MSS limits the maximum size of a segment's data field. When TCP sends a large file, such as an image as part of a Web page, it typically breaks the file into chunks of size MSS (except for the last chunk, which will often be less than the MSS).

A TCP segment header also contains the following fields:
The 32-bit **sequence number field** and the 32-bit **acknowledgment number field** are used by the TCP sender and receiver in implementing a reliable data transfer service, as discussed below.
• The 16-bit **receive window** field is used for flow control. Used to indicate the number of bytes that a receiver is willing to accept.
• The 4-bit **header length field** specifies the length of the TCP header in 32-bit words. The TCP header can be of variable length due to the TCP options field.
• The optional and variable-length **options field** is used when a sender and receiver negotiate the maximum segment size (MSS) or as a window scaling factor for use in high-speed networks. A time-stamping option is also defined. ls.
The **flag field** contains 6 bits. The **ACK bit** is used to indicate that the value carried in the acknowledgment field is valid; that is, the segment contains an acknowledgment for a segment that has been successfully received. The **RST**, **SYN**, and **FIN** bits are used for connection setup and teardown, as we will discuss at the end of this section. Setting the **PSH** bit indicates that the receiver should pass the

data to the upper layer immediately. Finally, the **URG** bit is used to indicate that there is data in this segment that the sending- side upper-layer entity has marked as "urgent." The location of the last byte of this urgent data is indicated by the 16-bit **urgent data pointer field**. TCP must inform the receiving- side upper-layer entity when urgent data exists and pass it a pointer to the end of the urgent data.

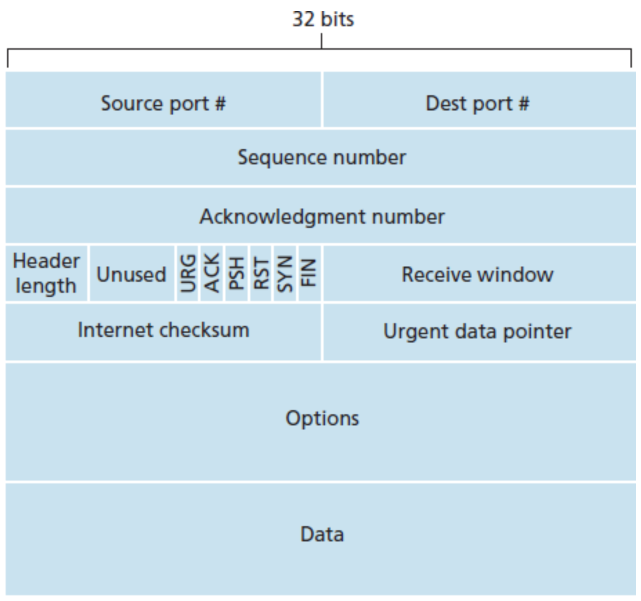


Figure 3.29 ♦ TCP segment structure

Sequence Numbers and Acknowledgment Numbers
TCP's use of sequence numbers reflects this view in that sequence numbers are over the stream of transmitted bytes and *not* over the series of transmitted segments. The **sequence number for a segment** is therefore the byte-stream number of the first byte in the segment.

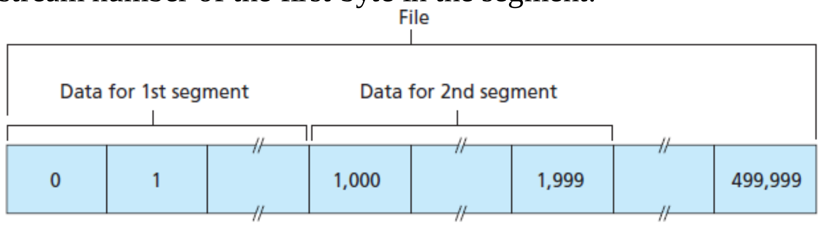


Figure 3.30 ♦ Dividing file data into TCP segments

TCP is full-duplex, so that Host A may be receiving data from Host B while it sends data to Host B (as part of the same TCP connection). Each of the segments that arrive from Host B has a sequence number for the data flowing from B to A. ***The acknowledgment number that Host A puts in its segment is the sequence number of the next byte Host A is expecting from Host B.***
Suppose that Host A has received all bytes numbered 0 through 535 from B and suppose that it is about to send a segment to Host B. Host A is waiting for byte 536 and all the subsequent bytes in Host B's data stream. So Host A puts 536 in the acknowledgment number field of the segment it sends to B. As another example, suppose that Host A has received one segment from Host B containing bytes 0 through 535 and another segment containing bytes 900 through 1,000. For some reason Host A has not yet received bytes 536 through 899. In this example, Host A is still waiting for byte 536 (and beyond) in order to re-create B's data stream. Thus, A's next segment to B will contain 536 in the acknowledgment number field. Because TCP only acknowledges bytes up to the first missing byte in the stream, TCP is said to provide

cumulative acknowledgments.

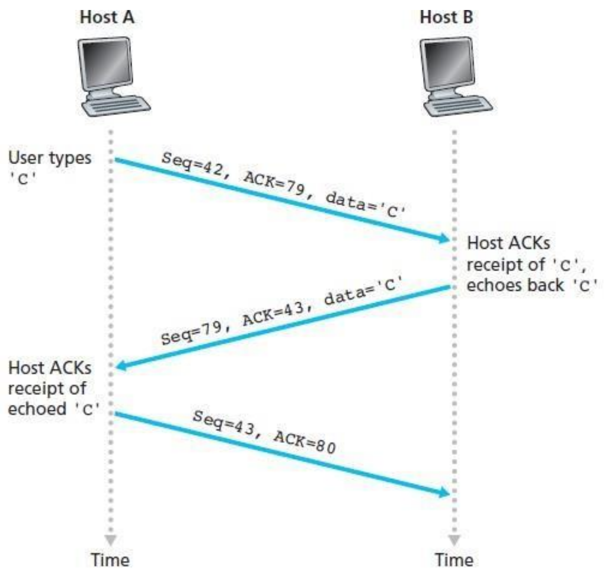


Figure 3.31 ♦ Sequence and acknowledgment numbers for a simple Telnet application over TCP

Round-Trip Time Estimation and Timeout

Estimating the Round-Trip Time

The sample RTT, denoted SampleRTT, for a segment is the amount of time between when the segment is sent and when an acknowledgment for the segment is received. Instead of measuring a SampleRTT for every transmitted segment, most TCP implementations take only one SampleRTT measurement at a time. That is, at any point in time, the SampleRTT is being estimated for only one of the transmitted but currently unacknowledged segments, leading to a new value of SampleRTT approximately once every RTT.

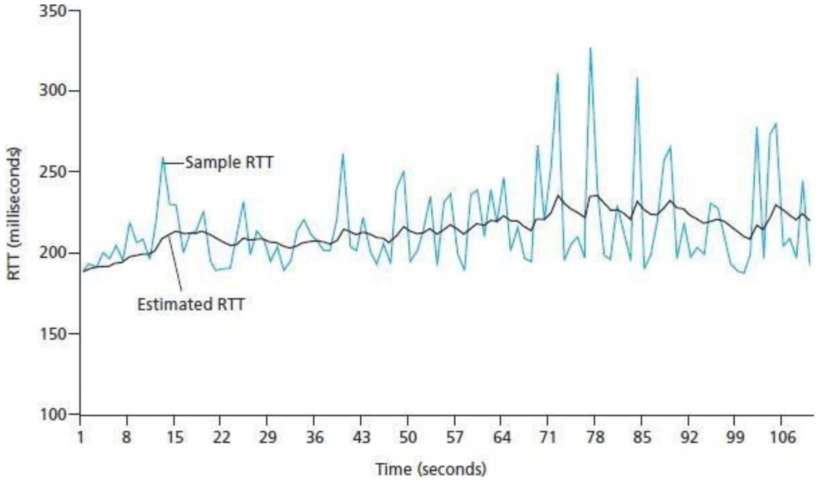
Obviously, the SampleRTT values will fluctuate from segment to segment due to congestion in the routers and to the varying load on the end systems. Because of this fluctuation, any given SampleRTT value may be atypical.
TCP maintains an average, called EstimatedRTT, of the SampleRTT values. Upon obtaining a new SampleRTT, TCP updates EstimatedRTT according to the following formula:
EstimatedRTT = (1 – α) • EstimatedRTT + α• SampleRTT
New value of EstimatedRTT is a weighted combination of the previous value of EstimatedRTT and the new value for SampleRTT. The recommended value of α is α = 0.125
EstimatedRTT = 0.875 • EstimatedRTT + 0.125 • SampleRTT

Note that EstimatedRTT is a weighted average of the SampleRTT values.

In statistics, such an average is called an **exponential weighted moving average (EWMA)**. The word "exponential" appears in EWMA because the weight of a given SampleRTT decays exponentially fast as the updates proceed. In addition to having an estimate of the RTT, it is also valuable to have a measure of the variability of the RTT defines the RTT variation, DevRTT, as an estimate of how much SampleRTT typically deviates from EstimatedRTT:

DevRTT = (1 – β) • DevRTT + β•| SampleRTT – EstimatedRTT |

Note that DevRTT is an EWMA of the difference between SampleRTT and EstimatedRTT. If the SampleRTT values have little fluctuation, then DevRTT will be small; on the other hand, if there is a lot of fluctuation, DevRTT will be large. The recommended value of β is 0.25.



Setting and Managing the Retransmission Timeout Interval

The interval should be greater than or equal to EstimatedRTT, or unnecessary retransmissions would be sent. But the timeout interval should not be too much larger than EstimatedRTT; otherwise, when a segment is lost, TCP would not quickly retransmit the segment, leading to large data transfer delays.
It is therefore desirable to set the timeout equal to the EstimatedRTT plus some margin. The value of DevRTT should thus come into play here. All these considerations are taken into account in TCP's method for determining the retransmission timeout interval: **TimeoutInterval = EstimatedRTT + 4 • DevRTT**

Reliable Data Transfer

TCP creates a **reliable data transfer service** on top of IP's unreliable best effort service. TCP's reliable data transfer service ensures that the data stream that a process reads out of its TCP receive buffer is uncorrupted, without gaps, without duplication, and in sequence; that is, the byte stream is exactly the same byte stream that was sent by the end system on the other side of the connection.

We see that there are three major events related to data transmission and retransmission in the TCP sender: **data received from application above; timer timeout; and ACK receipt.**
Upon the occurrence of the first major event, TCP receives data from the application, encapsulates the data in a segment, and passes the segment to IP. Note that each segment includes a sequence number that is the byte-stream number of the first data byte in the segment. TCP starts the timer when the segment is passed to IP. The expiration interval for this timer is the TimeoutInterval, which is calculated from EstimatedRTT and DevRTT,
The second major event is the timeout. TCP responds to the timeout event by retransmitting the segment that caused the timeout. TCP then restarts the timer.
The third major event that must be handled by the TCP sender is the arrival of an acknowledgment segment (ACK) from the receiver (more specifically, a segment containing a valid ACK field value). On the occurrence of this event, TCP compares the ACK value y with its variable SendBase. The TCP state variable SendBase is the sequence number of the oldest unacknowledged byte.
If **y > SendBase** then the ACK is acknowledging one or more previously unacknowledged segments.

Thus the sender updates its SendBase variable; it also restarts the timer if there currently are any not-yet-acknowledged segments.

A Few Interesting Scenarios

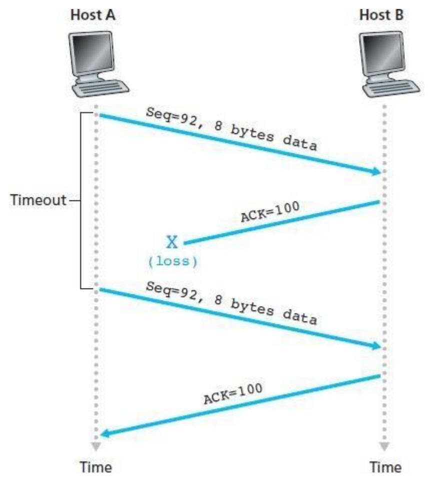


Figure 3.34 ♦ Retransmission due to a lost acknowledgment

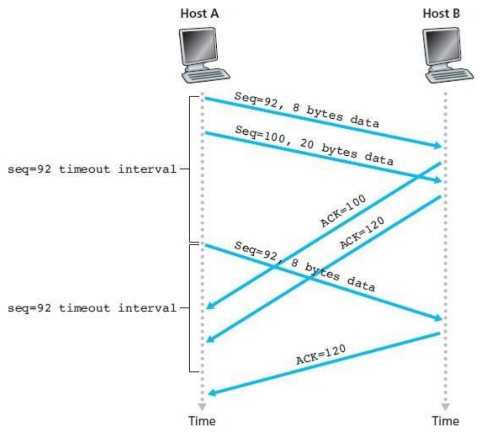


Figure 3.35 ♦ Segment 100 not retransmitted

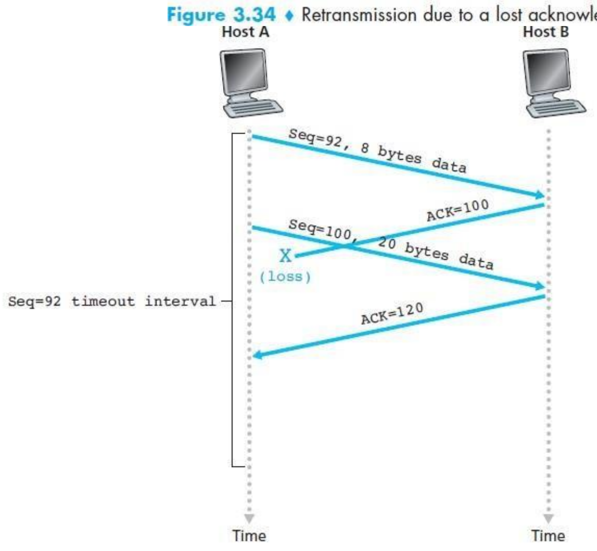


Figure 3.36 ♦ A cumulative acknowledgment avoids retransmission of the first segment

Doubling the Timeout Interval

The first concerns the length of the timeout interval after timer expiration. Whenever the timeout event occurs, TCP retransmits the not-yet acknowledged segment with the smallest sequence number. But each time TCP retransmits, it sets the next timeout interval to twice the previous value, rather than derivng it from the last EstimatedRTT and DevRTT. This modification provides a limited form of congestion control.

Fast Retransmit

When a segment is lost, this long timeout period forces the sender to delay resending the lost packet, thereby increasing the end-to-end delay. Fortunately, the sender can often detect packet loss well before the timeout event occurs by noting so-called duplicate ACKs. A **duplicate ACK** is an ACK that reacknowledges a segment for which the sender has already received an earlier acknowledgment. To understand the sender's response to a duplicate ACK, we must look at why the receiver sends a duplicate ACK in the first place.
When a TCP receiver receives a segment with a sequence number that is larger than the next, expected, in-order sequence number, it detects a gap in the data stream—that is, a missing segment. This gap could be the result of lost or reordered segments within the network. Since TCP does not use negative acknowledgments, the receiver cannot send an explicit negative acknowledgment back to the sender. Instead, it simply reacknowledges (that is, generates a duplicate ACK for) the last in-order byte of data it has received. Because a sender often sends a large number of segments back to back, if one segment is lost, there will likely be many back-to-back duplicate ACKs. If the TCP sender receives three duplicate ACKs for the same data, it takes this as an indication that the segment following the segment that has been ACKed three times has been lost.

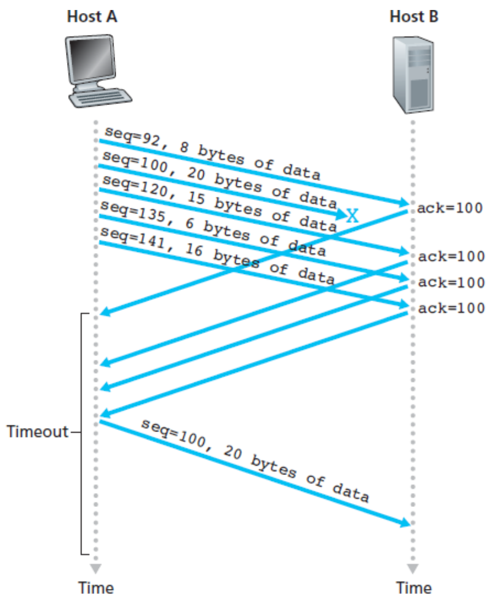


Figure 3.37 ♦ Fast retransmit: retransmitting the missing segment before the segment's timer expires

Flow Control

TCP provides a **flow-control service** to its applications to eliminate the possibility of the sender

overflowing the receiver's buffer. Flow control is thus a speed-matching service—matching the rate at which the sender is sending against the rate at which the receiving application is reading.
TCP provides flow control by having the *sender* maintain a variable called the **receive window**. Informally, the receive window is used to give the sender an idea of how much free buffer space is available at the receiver. Because TCP is full-duplex, the sender at each side of the connection maintains a distinct receive window. Let's investigate the receive window in the context of a file transfer. Suppose that Host Ais sending a large file to Host B over a TCP connection. Host B allocates a receive buffer to this connection; denote its size by RcvBuffer. From time to time, the application process in Host B reads from the buffer. Define the following variables:
LastByteRead: the number of the last byte in the data stream read from the buffer by the application process in B
LastByteRcvd: the number of the last byte in the data stream that has arrived from the network and has been placed in the receive buffer at B.
Because TCP is not permitted to overflow the allocated buffer, we must have
LastByteRcvd – LastByteRead <= RcvBuffer
The receive window, denoted rwnd is set to the amount of spare room in the buffer:
rwnd = RcvBuffer – [LastByteRcvd – LastByteRead]

How does the connection use the variable rwnd to provide the flow-control service? Host B tells Host A how much spare room it has in the connection buffer by placing its current value of rwnd in the receive window field of every segment it sends to A. Initially, Host B sets rwnd = RcvBuffer. Note that to pull this off, Host B must keep track of several connection-specific variables. Host A in turn keeps track of two variables, LastByteSent and Last- ByteAcked, which have obvious meanings. Note that the difference between these two variables, **LastByteSent – LastByteAcked**, is the amount of unacknowledged data that A has sent into the connection. By keeping the amount of unacknowledged data less than the value of rwnd, Host A is assured that it is not overflowing the receive buffer at Host B. Thus, Host A makes sure throughout the connection's life that
LastByteSent – LastByteAcked <= rwnd

There is one minor technical problem with this scheme. To see this, suppose Host B's receive buffer becomes full so that rwnd = 0. After advertising rwnd = 0 to Host A, also suppose that B has *nothing* to send to A. Now consider what happens. As the application process at B empties the buffer, TCP does not send new segments with new rwnd values to Host A; indeed, TCP sends a segment to Host A only if it has data to send or if it has an acknowledgment to send. Therefore, Host A is never informed that some space has opened up in Host B's receive buffer—Host A is blocked and can transmit no more data! To solve this problem, the TCP specification requires Host A to continue to send segments with one data byte when B's receive window is zero. These segments will be acknowledged by the receiver. Eventually the buffer will begin to empty and the acknowledgments will contain a nonzero rwnd value.

TCP Connection Management

The client application process first informs the client TCP that it wants to establish a connection to a process in the server. The TCP in the client then proceeds to establish a TCP connection with the TCP in the server in the following manner:
Step 1. The client-side TCP first sends a special TCP segment to the server-side TCP. This special segment contains no application-layer data. But one of the flag bits in the segment's header, the SYN bit, is set to 1. For this reason, this special segment is referred to as a SYN segment. In addition, the client randomly chooses an initial sequence number (client_isn) and puts this number in the sequence number

field of the initial TCP SYN segment. This segment is encapsulated within an IP datagram and sent to the server.

Step 2. Once the IP datagram containing the TCP SYN segment arrives at the server host (assuming it does arrive!), the server extracts the TCP SYN segment from the datagram, allocates the TCP buffers and variables to the connection, and sends a connection-granted segment to the client TCP. First, the SYN bit is set to 1. Second, the acknowledgment field of the TCP segment header is set to client_isn+1. Finally, the server chooses its own initial sequence number (server_isn) and puts this value in the sequence number field of the TCP segment header. This connection-granted segment is saying, in effect, "I received your SYN packet to start a connection with your initial sequence number, client_isn. I agree to establish this connection. My own initial sequence number is server_isn." The connectiongranted segment is referred to as a **SYNACK segment**.

Step 3. Upon receiving the SYNACK segment, the client also allocates buffers and variables to the connection. The client host then sends the server yet another segment; this last segment acknowledges the server's connection-granted segment (the client does so by putting the value server_isn+1 in the acknowledgment field of the TCP segment header). The SYN bit is set to zero, since the connection is established. this connection establishment procedure is often referred to as a **three-way handshake.**

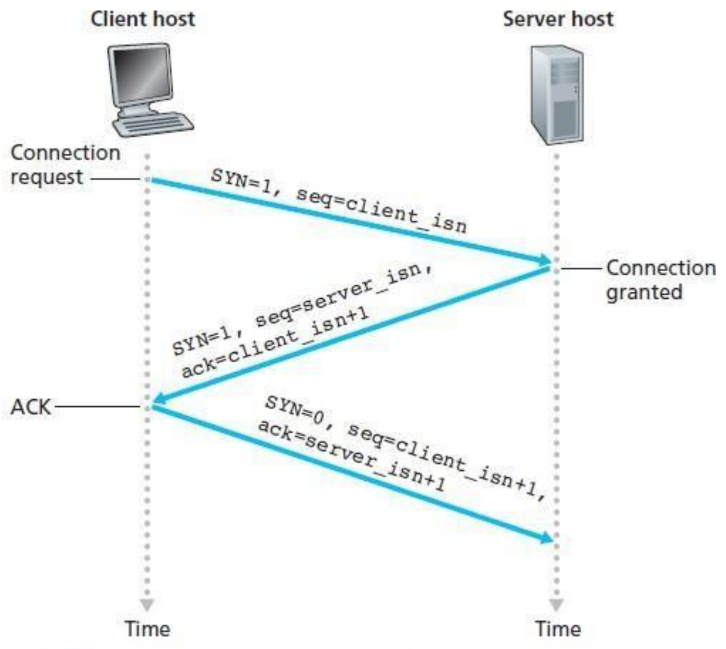


Figure 3.39 ♦ TCP three-way handshake: segment exchange

When a connection ends, the "resources" (that is, the buffers and variables) in the hosts are deallocated During the life of a TCP connection, the TCP protocol running in each host makes transitions through various **TCP states.**

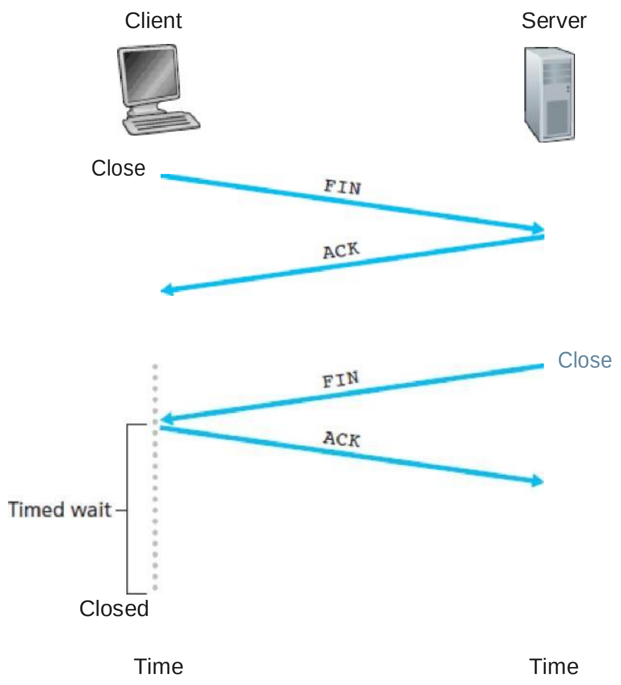


Figure 3 40 t Closing a TCP connection

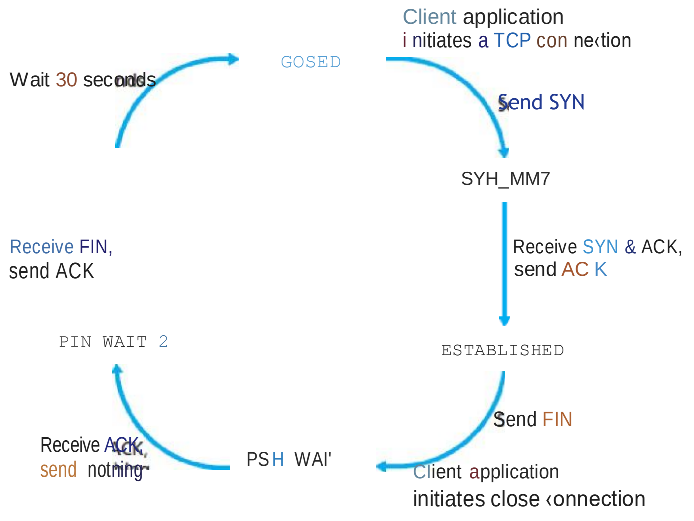


Figure 3. 41 t A typical sequence o* TCP states visited by a client TCP

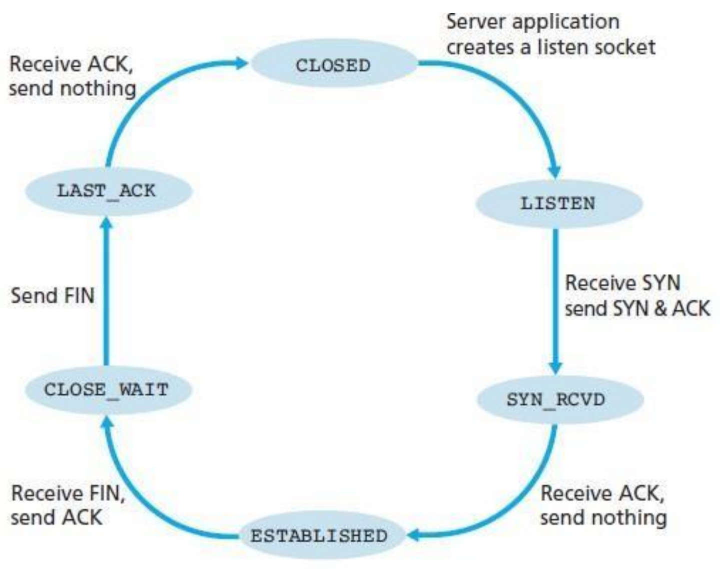

Server application creates a listen socket
CLOSED
LISTEN
Receive SYN send SYN & ACK
SYN_RCVD
Receive ACK, send nothing
ESTABLISHED
Receive FIN, send ACK
CLOSE_WAIT
Send FIN
LAST_ACK
Receive ACK, send nothing

Figure 3.42 ♦ A typical sequence of TCP states visited by a server-side TCP

Principles of Congestion Control

The Causes and the Costs of Congestion

Scenario 1: Two Senders, a Router with Infinite Buffers

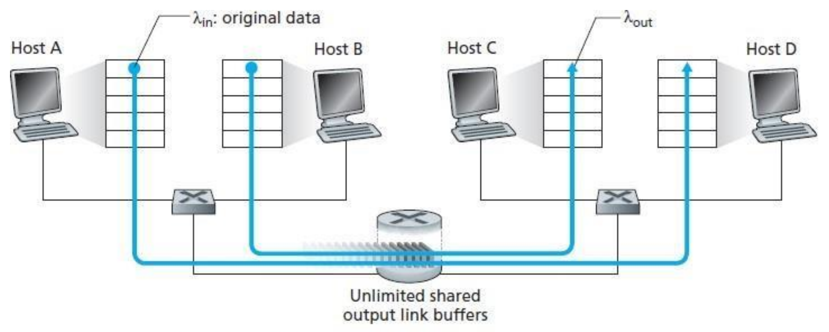


Figure 3.43 ♦ Congestion scenario 1: Two connections sharing a single hop with infinite buffers

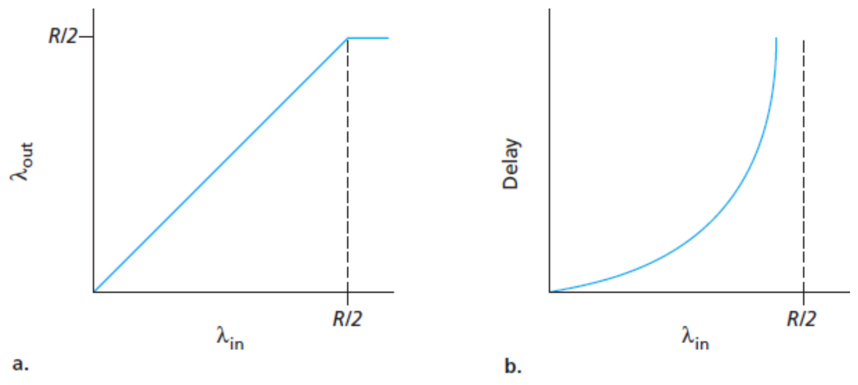


Figure 3.44 ♦ Congestion scenario 1: Throughput and delay as a function of host sending rate

Scenario 2: Two Senders and a Router with Finite Buffers

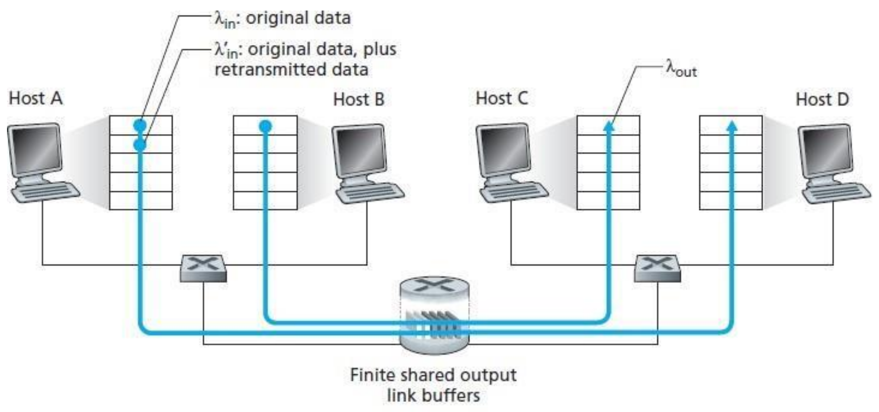

λin: original data
λ'in: original data, plus retransmitted data
λout
Host A
Host B
Host C
Host D
Finite shared output link buffers

Figure 3.45 ♦ Scenario 2: Two hosts (with retransmissions) and a router with finite buffers

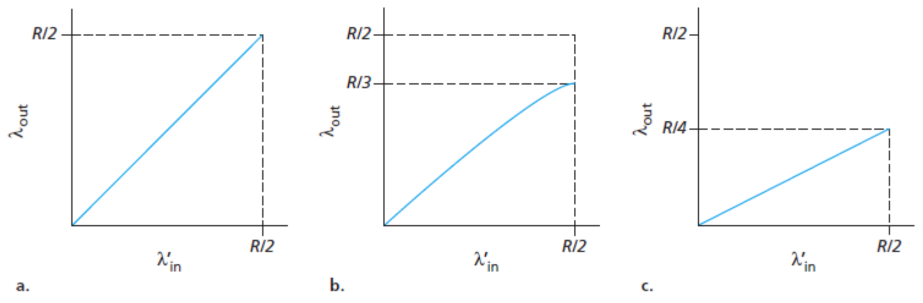


Figure 3.46 ♦ Scenario 2 performance with finite buffers

Scenario 3: Four Senders, Routers with Finite Buffers, and Multihop Paths

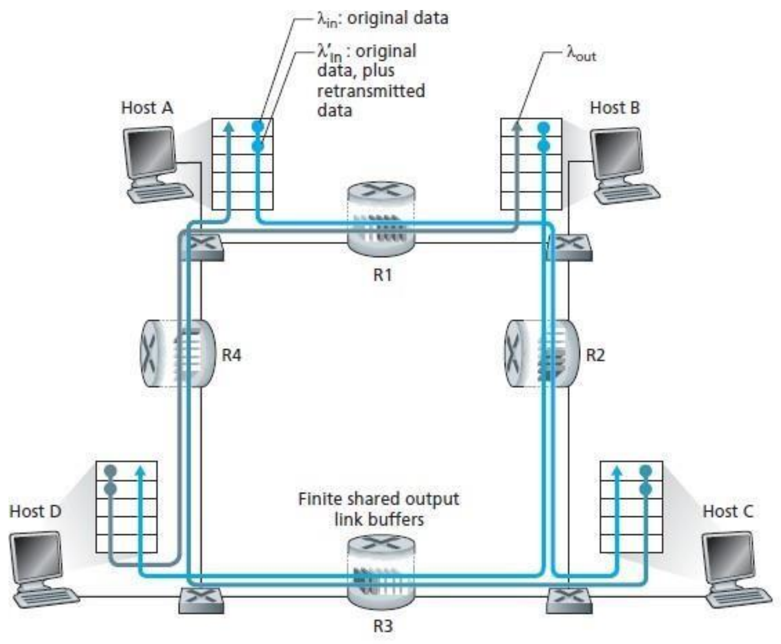

λ_{in}: original data
λ'_{in} : original data, plus retransmitted data
λ_{out}
Host A
Host B
R1
R4
R2
Host D
Finite shared output link buffers
Host C
R3

Figure 3.47 ♦ Four senders, routers with finite buffers, and multihop paths

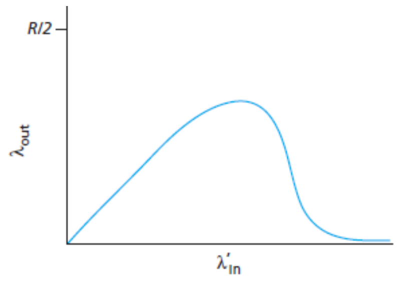


Figure 3.48 ♦ Scenario 3 performance with finite buffers and multihop paths

Approaches to Congestion Control

we can distinguish among congestion-control approaches by whether the network layer provides any explicit assistance to the transport layer for congestion-control purposes:

• *End-to-end congestion control.* In an end-to-end approach to congestion control, the network layer provides *no explicit support* to the transport layer for congestioncontrol purposes. Even the presence of congestion in the network must be inferred by the end systems based only on observed network behavior (for example, packet loss and delay). We will see in Section 3.7 that TCP must necessarily take this endto- end approach toward congestion control, since the IP layer provides no feedback to the end systems regarding network congestion. TCP segment loss (as indicated by a timeout or a triple duplicate acknowledgment) is taken as an indication of network congestion and TCP decreases its window size accordingly. We will also see a more recent proposal for TCP congestion control that uses increasing round-trip delay values as indicators of increased network congestion.

• *Network-assisted congestion control.* With network-assisted congestion control, network-layer components (that is, routers) provide explicit feedback to the sender regarding the congestion state in the network. This feedback may be as simple as a single bit indicating congestion at a link. This approach was taken in the early IBM SNA [Schwartz 1982] and DEC DECnet [Jain 1989; Ramakrishnan 1990] architectures, was recently proposed for TCP/IP networks [Floyd TCP 1994; RFC 3168], and is used in ATM available bit-rate (ABR) congestion control as well, as discussed below. More sophisticated network feedback is also possible. For example, one form of ATM ABR congestion control that we will study shortly allows a router to inform the sender explicitly of the transmission rate it (the router) can support on an outgoing link. The XCP protocol [Katabi 2002] provides router-computed feedback to each source, carried in the packet header, regarding how that source should increase or decrease its transmission rate.

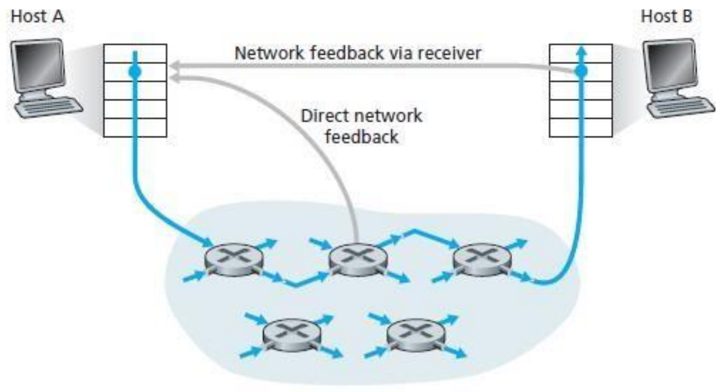


Figure 3.49 ♦ Two feedback pathways for network-indicated congestion information

Network-Assisted Congestion-Control Example:
ATM ABR Congestion Control
Fundamentally ATM takes a virtual-circuit (VC) oriented approach toward packet switching
ABR has been designed as an elastic data transfer service in a manner reminiscent of TCP. When the network is underloaded, ABR service should be able to take advantage of the spare available bandwidth; when the network is congested, ABR service should throttle its transmission rate to some predetermined minimum transmission rate. With ATM ABR service, data cells are transmitted from a source to a destination through a series of intermediate switches. Interspersed with the data cells are **resource-management cells (RM cells)**; these RM cells can be used to convey congestion-related information among the hosts and switches. When an RM cell arrives at a destination, it will be turned around and sent back to the sender (possibly after the destination has modified the contents of the RM cell). It is also possible for a switch to generate an RM cell itself and send this RM cell directly to a source. RM cells can thus be used to provide both direct network feedback and network feedback via the receiver,

ATM ABR congestion control is a rate-based approach. That is, the sender explicitly computes a maximum rate at which it can send and regulates itself accordingly. ABR provides three mechanisms for signaling congestion-related information from the switches to the receiver:
• *EFCI bit.* Each *data cell* contains an **explicit forward congestion indication (EFCI) bit**. A congested network switch can set the EFCI bit in a data cell to 1 to signal congestion to the destination host. The destination must check the EFCI bit in all received data cells. When an RM cell arrives at the destination, if the most recently received data cell had the EFCI bit set to 1, then the destination sets the congestion indication bit (the CI bit) of the RM cell to 1 and sends the RM cell back to the sender. Using the EFCI in data cells and the CI bit in RM cells, a sender can thus be notified about congestion at a network switch.
CI and NI bits. As noted above, sender-to-receiver RM cells are interspersed with data cells. The rate of RM cell interspersion is a tunable parameter, with the default value being one RM cell every 32 data

cells. These RM cells have a **congestion indication (CI) bit** and a **no increase (NI) bit** that can be set by a congested network switch. Specifically, a switch can set the NI bit in a passing RM cell to 1 under mild congestion and can set the CI bit to 1 under severe congestion conditions. When a destination host receives an RM cell, it will send the RM cell back to the sender with its CI and NI bits intact (except that CI may be set to 1 by the destination as a result of the EFCI mechanism described above).
• *ER setting*. Each RM cell also contains a 2-byte **explicit rate (ER) field**. A congested switch may lower the value contained in the ER field in a passing RM cell. In this manner, the ER field will be set to the minimum supportable rate of all switches on the source-to-destination path.

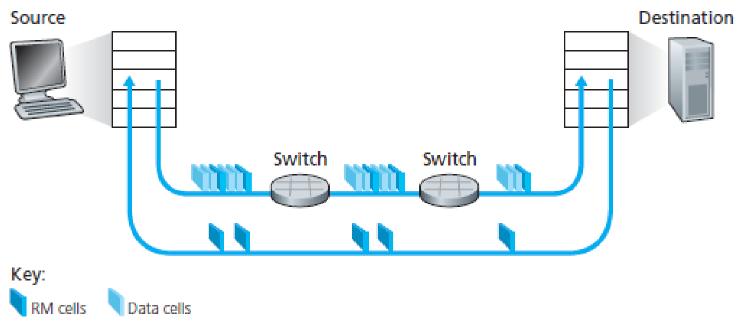


Figure 3.50 ♦ Congestion-control framework for ATM ABR service

TCP Congestion Control
TCP must use end-to-end congestion control rather than network- assisted congestion control
The approach taken by TCP is to have each sender limit the rate at which it sends traffic into its connection as a function of perceived network congestion. If a TCP sender perceives that there is little congestion on the path between itself and the destination, then the TCP sender increases its send rate; if the sender perceives that there is congestion along the path, then the sender reduces its send rate. But this approach raises three questions. First, how does a TCP sender limit the rate at which it sends traffic into its connection? Second, how does a TCP sender perceive that there is congestion on the path between itself and the destination? And third, what algorithm should the sender use to change its send rate as a function of perceived end-to-end congestion?

how a TCP sender limits the rate at which it sends traffic into its connection. we saw that each side of a TCP connection consists of a receive buffer, a send buffer, and several variables (LastByteRead, rwnd, and so on). The TCP congestion-control mechanism operating at the sender keeps track of an additional variable, the **congestion window**. The congestion window, denoted cwnd, imposes a constraint on the rate at which a TCP sender can send traffic into the network. Specifically, the amount of unacknowledged data at a sender may not exceed the minimum of cwnd and rwnd, that is:
LastByteSent – LastByteAcked <= min{cwnd, rwnd}
The constraint above limits the amount of unacknowledged data at the sender and therefore indirectly limits the sender's send rate

Thus the sender's send rate is roughly cwnd/RTT bytes/sec. By adjusting the value of cwnd, the sender can therefore adjust the rate at which it sends data into its connection.

TCP answers these questions using the following guiding principles:
A lost segment implies congestion, and hence, the TCP sender's rate should be decreased when a segment is lost.

An acknowledged segment indicates that the network is delivering the sender's segments to the receiver, and hence, the sender's rate can be increased when an ACK arrives for a previously unacknowledged segment.
Bandwidth probing. Given ACKs indicating a congestion-free source-to-destination path and loss events indicating a congested path, TCP's strategy for adjusting its transmission rate is to increase its rate in response to arriving ACKs until a loss event occurs, at which point, the transmission rate is decreased. The TCP sender thus increases its transmission rate to probe for the rate that at which congestion onset begins, backs off from that rate, and then to begins probing again to see if the congestion onset rate has changed.

Slow Start
Thus, in the **slow-start** state, the value of cwnd begins at 1 MSS and increases by 1 MSS every time a transmitted
segment is first acknowledged. TCP sends the first segment into the network and waits for an acknowledgment. When this acknowledgment arrives, the TCP sender increases the congestion window by one MSS and sends out two maximum-sized segments. These segments are then acknowledged, with the sender increasing the congestion window by 1 MSS for each of the acknowledged segments, giving a congestion window of 4 MSS, and so on. This process results in a doubling of the sending rate every RTT. Thus, the TCP send rate starts slow but grows exponentially during the slow start phase But when should this exponential growth end? Slow start provides several
answers to this question. First, if there is a loss event (i.e., congestion) indicated by a timeout, the TCP sender sets the value of cwnd to 1 and begins the slow start process anew. It also sets the value of a second state variable, ssthresh (shorthand for "slow start threshold") to cwnd/2—half of the value of the congestion window value when congestion was detected. The second way in which slow start may end is directly tied to the value of ssthresh. TCP increases cwnd more cautiously when in congestion-avoidance mode. The final way in which slow start can end is if three duplicate ACKs are detected, in which case TCP performs a fast retransmit (see Section 3.5.4) and enters the fast recovery State.

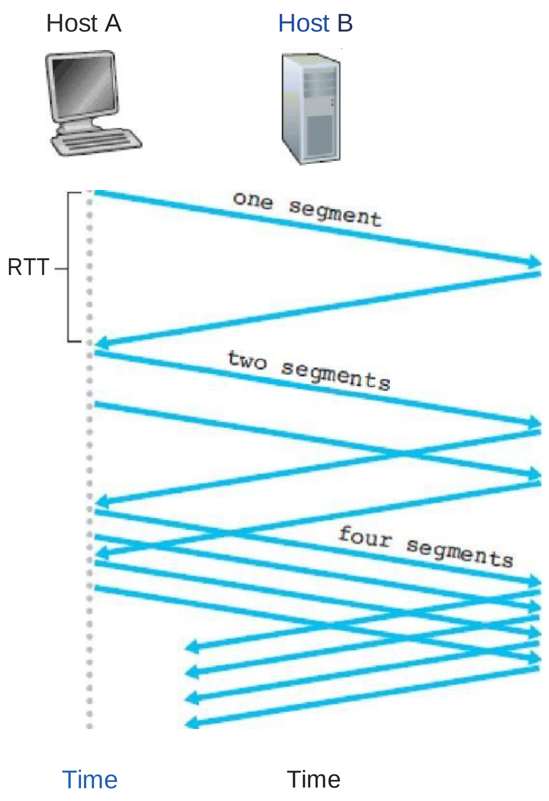


Figure 0.51 TCP slow start

Congestion Avoidance

On entry to the congestion-avoidance state, the value of cwnd is approximately half its value when congestion was last encountered—congestion could be just around the corner! Thus, rather than doubling the value of cwnd every RTT, TCP adopts a more conservative approach and increases the value of cwnd by just a single MSS every RTT. A common approach is for the TCP sender to increase cwnd by MSS bytes (MSS/cwnd) whenever a new acknowledgment arrives. But when should congestion avoidance's linear increase (of 1 MSS per RTT) end? TCP's congestion-avoidance algorithm behaves the same when a timeout occurs. As in the case of slow start: The value of cwnd is set to 1 MSS, and the value of ssthresh is updated to half the value of cwnd when the loss event occurred. Recall, however, that a loss event also can be triggered by a triple duplicate ACK event. TCP halves the value of cwnd and records the value of ssthresh to be half the value of cwnd when the triple duplicate ACKs were received. The fast-recovery state is then entered.

Fast Recovery

In fast recovery, the value of cwnd is increased by 1 MSS for every duplicate ACK received for the missing segment that caused TCP to enter the fast-recovery state. Eventually, when an ACK arrives for the missing segment, TCP enters the congestion-avoidance state after deflating cwnd. If a timeout event occurs, fast recovery transitions to the slow-start state after performing the same actions as in slow start and congestion avoidance: The value of cwnd is set to 1 MSS, and the value of ssthresh is set to half the value of cwnd when the loss event occurred.

Fast recovery is a recommended, but not required, component of TCP. It is interesting that an early version of TCP, known as **TCP Tahoe**, unconditionally cut its congestion window to 1 MSS and entered the slow-start phase after either a timeout-indicated or triple-duplicate-ACK-indicated loss event. The newer version of TCP, **TCP Reno**, incorporated fast recovery.

For the first eight transmission rounds, Tahoe and Reno take identical actions. The congestion window climbs exponentially fast during slow start and hits the threshold at the fourth round of transmission. The congestion window then climbs linearly until a triple duplicate- ACK event occurs, just after transmission round 8. Note that the congestion window is 12 • *MSS* when this loss event occurs. The value of ssthresh is then set to 0.5 • cwnd = 6 • MSS. Under TCP Reno, the congestion window is set to cwnd = 6 • MSS and then grows linearly. Under TCP Tahoe, the congestion window is set to 1 MSS and grows exponentially until it reaches the value of ssthresh, at which point it grows linearly.

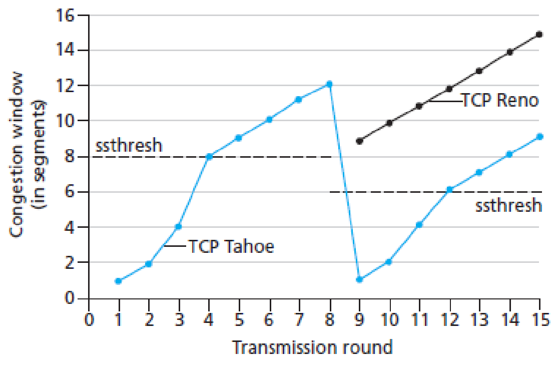


Figure 3.53 ♦ Evolution of TCP's congestion window (Tahoe and Reno)

TCP Congestion Control: Retrospective

Ignoring the initial slow-start period when a connection begins and assuming that losses are indicated by triple duplicate ACKs rather than timeouts, TCP's congestion control consists of linear (additive) increase in cwnd of 1 MSS per RTT and then a halving (multiplicative decrease) of cwnd on a triple duplicate-ACK event. For this reason,
TCP congestion control is often referred to as an **additive-increase, multiplicativedecrease (AIMD)** form of congestion control. AIMD congestion control gives rise to the "saw tooth" behavior.

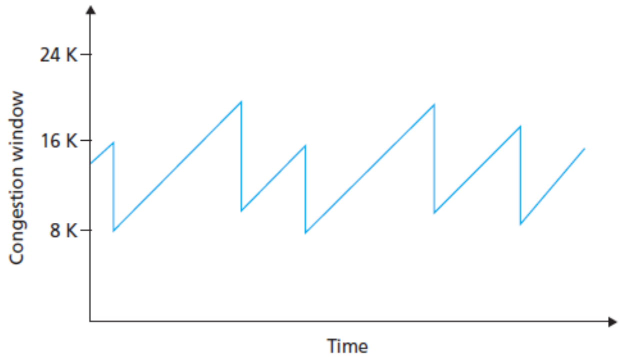


Figure 3.54 ♦ Additive-increase, multiplicative-decrease congestion control

Macroscopic Description of TCP Throughput

average throughput of a connection =(0.75 X *W) / RTT*

TCP Over High-Bandwidth Paths

average throughput of a connection = (1.22 X *MSS) / RTT* √*L*

Fairness

Consider *K* TCP connections, each with a different end-to-end path, but all passing through a bottleneck link with transmission rate *R* bps.

A congestion-control mechanism is said to be *fair* if the average transmission rate of each connection is approximately *R/K;* that is, each connection gets an equal share of the link bandwidth.

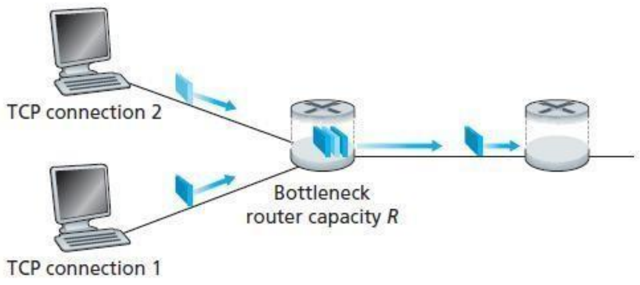


Figure 3.55 ♦ Two TCP connections sharing a single bottleneck link

If TCP is to share the link bandwidth equally between the two connections, then the realized throughput should fall along the 45-degree arrow (equal bandwidth share) emanating from the origin. Ideally, the sum of the two throughputs should equal *R*. (Certainly, each connection receiving an equal, but zero, share of the link capacity is not a desirable situation!) So the goal should be to have the achieved throughputs fall somewhere near the intersection of the equal bandwidth share line and the full bandwidth utilization line

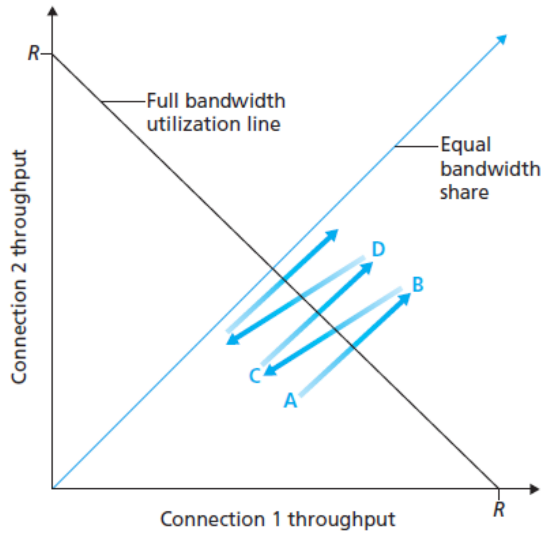


Figure 3.56 ♦ Throughput realized by TCP connections 1 and 2

Fairness and UDP

When running over UDP, applications can pump their audio and video into the network at a constant rate and occasionally lose packets, rather than reduce their rates to “fair” levels at times of congestion and not lose any packets. From the perspective of TCP, the multimedia applications running over UDP are not being fair—they do not cooperate with the other connections nor adjust their transmission rates appropriately. Because TCP congestion control will decrease its transmission rate in the face of increasing congestion (loss), while UDP sources need not, it is possible for UDP sources to crowd out TCP traffic.

Fairness and Parallel TCP Connections

For example, Web browsers often use multiple parallel TCP connections to transfer the multiple objects within a Web page. (The exact number of multiple connections is configurable in most browsers.) When an application uses multiple parallel connections, it gets a larger fraction of the bandwidth in a congested link. As an example, consider a link of rate *R* supporting nine ongoing client-server applications, with each of the applications using one TCP connection. If a new application comes along and also uses one TCP connection, then each application gets approximately the same transmission rate of *R*/10. But if this new application instead uses 11 parallel TCP connections, then the new application gets an unfair allocation of more than *R*/2. Because Web traffic is so pervasive in the Internet, multiple parallel connections are not uncommon.