Module – 2

TRANSPORT LAYER

2.1 Introduction and Transport-Layer Services

- A transport-layer protocol provides for logical communication between application processes running on different hosts.
- Application processes use the logical communication provided by the transport layer to send messages to each other, free from the worry of the details of the physical infrastructure used to carry these messages.
- 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.
- This is done by (possibly) breaking the application messages into smaller chunks and adding a transport-layer header to each chunk to create the transport-layer segment.
- The transport layer then passes the segment to the network layer at the sending end system, where the segment is encapsulated within a network-layer packet (a datagram) and sent to the destination.
- On the receiving side, the network layer extracts the transport-layer segment from the datagram and passes the segment up to the transport layer.
- The transport layer then processes the received segment, making the data in the segment available to the receiving application.
- Internet has two protocols—TCP and UDP. Each of these protocols provides a different set of transport-layer services to the invoking application.

2.1.1 Relationship between Transport and Network Layers

- Transport Layer provides Process to process delivery service whereas network layer provides end to end delivery of data.
- Transport-layer protocol provides logical communication between *processes* running on different hosts, a network-layer protocol provides logical communication between hosts.

- Within an end system, a transport protocol moves messages from application processes to the network edge (that is, the network layer) and vice versa, but it doesn't have any say about how the messages are moved within the network core.
- The services that a transport protocol can provide are often constrained by the service model of the underlying network-layer protocol. If the network-layer protocol cannot provide delay or bandwidth guarantees for transport layer segments sent between hosts, then the transport-layer protocol cannot provide delay or bandwidth guarantees for application messages sent between processes.

2.1.2 Overview of the Transport Layer in the Internet

The Internet supports two transport layer protocols:

1) UDP (User Datagram Protocol), which provides an unreliable, connectionless service to the invoking application.

2) TCP (Transmission Control Protocol), which provides a reliable, connection-oriented service to the invoking application.

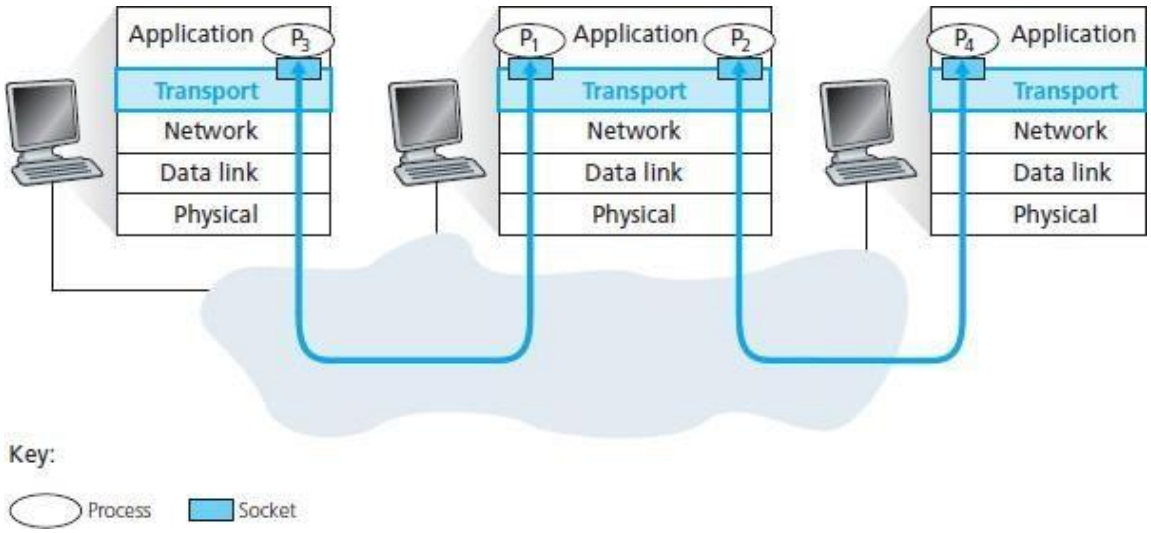
- The Internet's network-layer protocol has Internet Protocol. IP provides logical communication between hosts.
- The IP service model is a best-effort delivery service. This means that IP makes its "best effort" to deliver segments between communicating hosts, but it makes no guarantees. In particular, 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.
- 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.
- UDP and TCP also provide integrity checking by including error detection fields in their segments' headers.
- UDP is an unreliable service it does not guarantee that data sent by one process will arrive intact to the destination process.
- 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, TCP ensures that data is delivered from sending process to receiving process, correctly and in order.

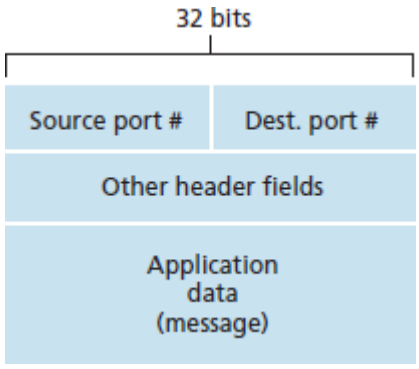
- TCP thus converts IP's unreliable service between end systems into a reliable data transport service between processes.
- TCP also provides congestion control. TCP congestion control prevents any one TCP connection from swamping the links and routers between communicating hosts with an excessive amount of traffic.
- UDP traffic, on the other hand, is unregulated. An application using UDP transport can send at any rate it pleases, for as long as it pleases.

2.2 Multiplexing and Demultiplexing

- At the destination host, the transport layer receives segments from the network layer just below.
- The transport layer has the responsibility of delivering the data in these segments to the appropriate application process running in the host.
- A process can have one or more sockets, doors through which data passes from the network to the process and through which data passes from the process to the network.
- The transport layer in the receiving host does not actually deliver data directly to a process, but instead to an intermediary socket.
- Because at any given time there can be more than one socket in the receiving host, each socket has a unique identifier.
- Each transport-layer segment has a set of fields in the segment to help receiver to deliver data to appropriate process socket.
- At the receiving end, the transport layer examines these fields to identify the receiving socket and then directs the segment to that socket. This job of delivering the data in a transport-layer segment to the correct socket is called **demultiplexing.**
- The job of gathering data chunks at the source host from different sockets, encapsulating each data chunk with header information to create segments, and passing the segments to the network layer is called **multiplexing.**



- Transport-layer multiplexing requires (1) that sockets have unique identifiers, and (2) that each segment have special fields that indicate the socket to which the segment is to be delivered. These special fields are the source port number field and the destination port number field.
- Each port number is a 16-bit number, ranging from 0 to 65535. The port numbers ranging from 0 to 1023 are called well-known port numbers and are restricted, which means that they are reserved for use by well-known application protocols such as HTTP (which uses port number 80) and FTP (which uses port number 21).



- UDP performs connectionless multiplexing and demultiplexing. TCP performs connection oriented multiplexing and demultiplexing.

Connectionless Multiplexing and Demultiplexing

clientSocket = socket(socket.AF_INET, socket.SOCK_DGRAM)

When a UDP socket is created, the transport layer automatically assigns a port number to the socket.

The transport layer assigns a port number in the range 1024 to 65535 that is currently not

being used by any other UDP port in the host.

Associate a specific port number (say, 19157) to this UDP socket via the socket bind() method:

clientSocket.bind((‘’, 19157))

UDP multiplexing/demultiplexing :

Suppose a process in Host A, with UDP port 19157, wants to send a chunk of application data to a process with UDP port 46428 in Host B.

The transport layer in Host A creates a transport-layer segment that includes the application data, the source port number (19157), the destination port number (46428).

The transport layer then passes the resulting segment to the network layer.

The network layer encapsulates the segment in an IP datagram and makes a best-effort attempt to deliver the segment to the receiving host.

If the segment arrives at the receiving Host B, the transport layer at the receiving host examines the destination port number in the segment (46428) and delivers the segment to its socket identified by port 46428.

As UDP segments arrive from the network, Host B directs (demultiplexes) each segment

to the appropriate socket by examining the segment’s destination port number.

UDP socket is fully identified by a two-tuple consisting of a destination IP address and a destination port number.

If two UDP segments have different source IP addresses and/or source port numbers, but have the same *destination* IP address and *destination* port number, then the two segments will be directed to the same destination process via the same destination socket.

Host A-to-B : In the segment , the source port number serves as part of a “return address”—when B wants to send a segment back to A, the destination port in the B-to-A segment will take its value from the source port value of the A-to-B segment.

UDPServer.py, the server uses the recvfrom() method to extract the clientside (source) port number from the segment it receives from the client; it then sends a new segment to the client, with the extracted source port number serving as the destination port number in this new segment.

Connection-Oriented Multiplexing and Demultiplexing

Difference between a TCP socket and a UDP socket is that a TCP socket is identified by a four-tuple: (source IP address, source port number, destination IP address, destination port number).

When a TCP segment arrives from the network to a host, the host uses all four values to direct (demultiplex) the segment to the appropriate socket.

Two arriving TCP segments with different source IP addresses or source port numbers will

be directed to two different sockets.

• The TCP server application has a "welcoming socket," that waits for connection establishment requests from TCP clients on port number 12000.

• The TCP client creates a socket and sends a connection establishment request segment with the lines:

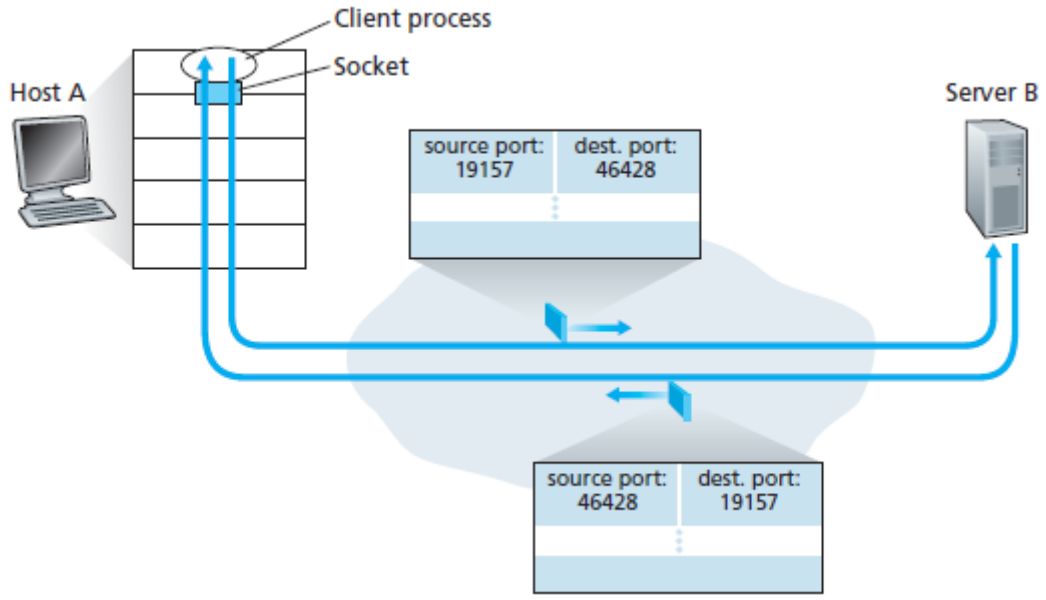
clientSocket = socket(AF_INET, SOCK_STREAM)

clientSocket.connect((serverName,12000))

• Aconnection-establishment request is more a TCP segment with destination port number 12000 and a special connection-establishment bit set in the TCP header. The segment also includes a source port number that was chosen by the client.

• When the host operating system of the computer running the server process receives the incoming connection-request segment with destination port 12000, it locates the server process that is waiting to accept a connection on port number 12000. The server process then creates a new socket:

connectionSocket, addr = serverSocket.accept( )



The transport layer at the server notes the following four values in the connection- request segment:

(1) the source port number in the segment,

(2) the IP address of the source host,

(3) the destination port number in the segment, and

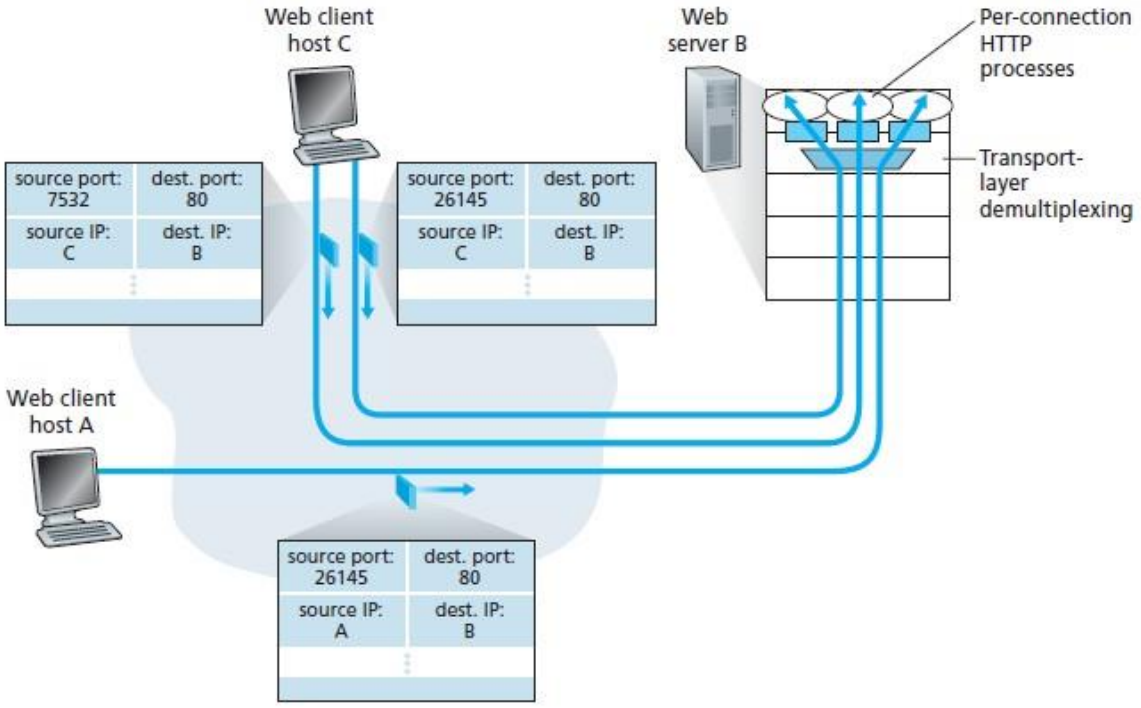
(4) its own IP address.

The newly created connection socket is identified by these four values; all subsequently arriving segments whose source port, source IP address, destination port, and destination IP address match these four values will be demultiplexed to this socket.

With the TCP connection, the client and server can now send data to each other.

The server host may support many simultaneous TCP connection sockets, with each socket attached to a process, and with each socket identified by its own four tuple.

When a TCP segment arrives at the host, all four fields (source IP address, source port, destination IP address, destination port) are used to direct (demultiplex) the segment to the appropriate socket.



Eg. Consider above Figure, in which Host C initiates two HTTP sessions to server B, and Host Ainitiates one HTTP session to B. Hosts Aand C and server B each have their own unique IP address—A, C, and B, respectively. Host C assigns two different source port numbers (26145 and 7532) to its two HTTP connections. Because Host A is choosing source port numbers independently of C, it might also assign a source port of 26145 to its HTTP connection.

Server B will still be able to correctly demultiplex the two connections having the same source port number, since the two connections have different source IP addresses.

Web Servers and TCP

Consider a host running a Web server, such as an Apache Web server, on port 80.

When clients (for example, browsers) send segments to the server, *all* segments will have destination port 80. In particular, both the initial connection-establishment segments and the segments carrying HTTP request messages will have destination port 80.

The server distinguishes the segments from the different clients using source IP addresses and source port numbers.

As shown in Figure, each of these processes has its own connection socket through which HTTP requests arrive and HTTP responses are sent.

There is not always a one-to-one correspondence between connection sockets and processes. Web servers often use only one process and create a new thread with a new connection socket for each new client connection.

2.3 Connectionless Transport: UDP

- UDP is a connectionless protocol which performs only multiplexing/demultiplexing function and some light error checking.
- 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.

Many applications are better suited for UDP for the following reasons:

1) 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.
- Since real-time applications often require a minimum sending rate, do not want to overly delay segment transmission, and can tolerate some data loss, TCP's service model is not particularly well matched to these applications' needs.

2) No connection establishment:

TCP uses a three-way handshake to establish the connection before it starts to transfer data. UDP just sends the data without any formal preliminaries. Thus UDP does not introduce any delay to establish a connection.

3) 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. For this reason, a server devoted to a particular application can typically support many more active clients when the application runs over UDP rather than TCP.

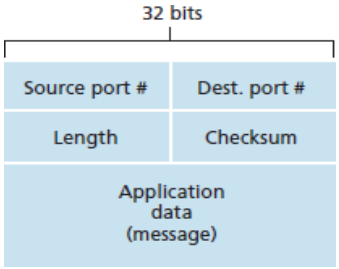
4) Small packet header overhead:

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

Popular Internet applications and their underlying transport protocols

2.3.1 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
- The length field specifies the number of bytes in the UDP segment (header plus data).
- The checksum is used by the receiving host to check whether errors have been introduced into the segment.
- The application data occupies the data field of the UDP segment.

2.3.2 UDP Checksum

The checksum is used to determine whether bits within the UDP segment have been altered as it moved from source to destination.

Step1: Add all the data elements using binary addition (Modulo-2 addition). If you get extra bit wrap it.

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

Step 2: Take 1s complement of the result.

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.

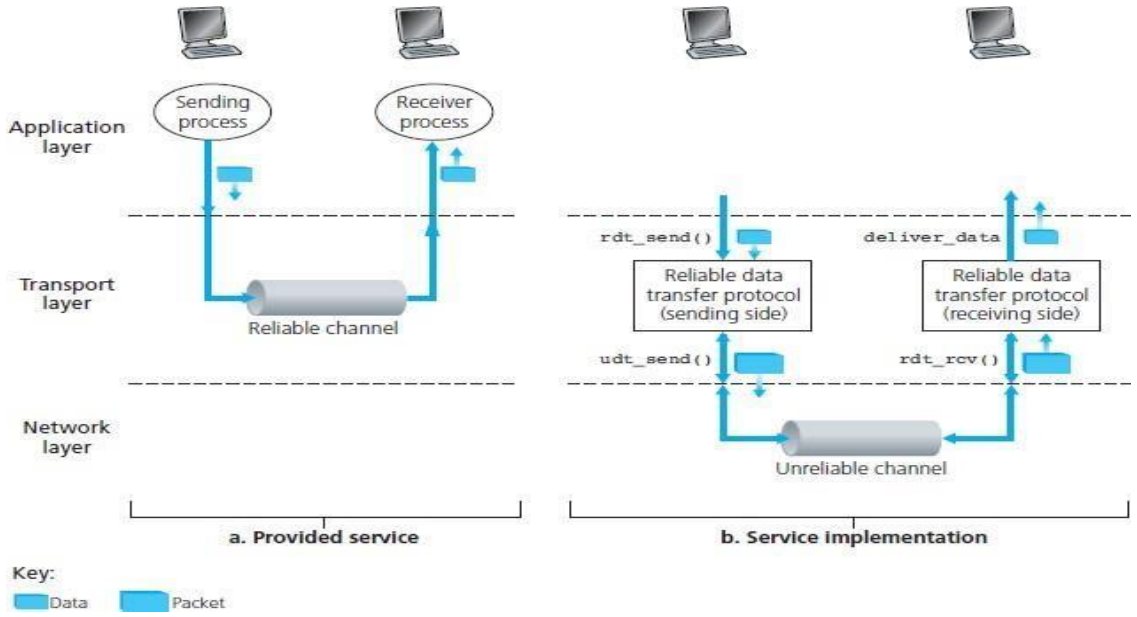
Step 3: Data along with checksum is transmitted to receiver.

Step 4: at the receiver side add all the data and checksum using binary addition. Wrap the extra bit and take 1s complement of the result. This will be the checksum. If checksum is all 0's receiver has received error free data otherwise it has received corrupted data.

2.4 Principles 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.

It is the responsibility of a reliable data transfer protocol to implement this service abstraction. This task is made difficult by the fact that the layer below the reliable data transfer protocol may be unreliable.

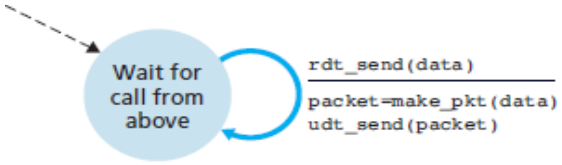


- The sending side of the data transfer protocol will be invoked from above by a call to rdt_send(). It will pass the data to be delivered to the upper layer at the receiving side.
- On the receiving side, rdt_rcv() will be called when a packet arrives from the receiving side of the channel.
- When the rdt protocol wants to deliver data to the upper layer, it will do so by calling deliver_data().
- Both the send and receive sides of rdt send packets to the other side by a call to udt_send()

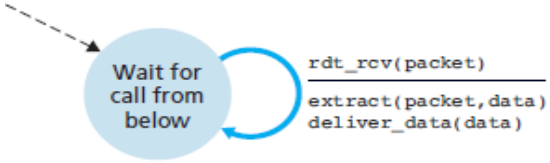
2.4.1 Building a Reliable Data Transfer Protocol

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

- 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)).
- Here all packet flow is from the sender to receiver; with a perfectly reliable channel there is no need for the receiver side to provide any feedback to the sender since nothing can go wrong.
- Also we have assumed that the receiver is able to receive data as fast as the sender happens to send data. Thus, there is no need for the receiver to ask the sender to slow down.



a. rdt1.0: sending side



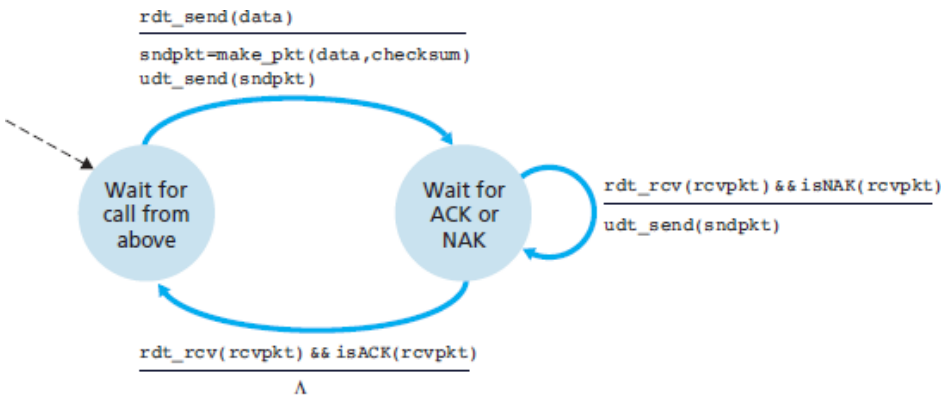
b. rdt1.0: receiving side

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

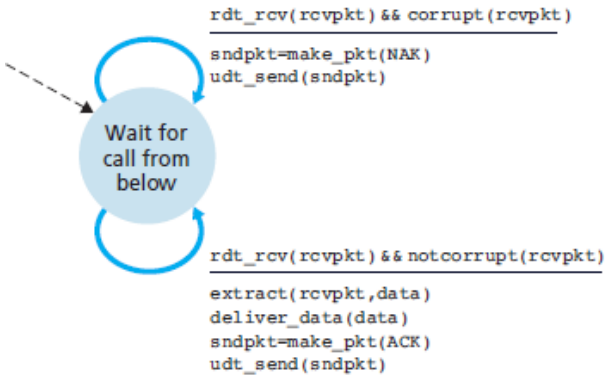
A more realistic model of the underlying channel is one in which bits in a packet may be corrupted.

Three additional protocol capabilities are required in (Automatic Repeat Request) 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. Since the sender and receiver are typically executing on different end systems, possibly separated by thousands of miles, the only way for the sender to learn of the receiver's view of the world is for the receiver to provide explicit feedback to the sender. The positive (ACK) and negative (NAK) acknowledgment replies in the message-dictation scenario are examples of such feedback.
- Retransmission. A packet that is received in error at the receiver will be retransmitted by the sender.



a. rdt2.0: sending side



b. rdt2.0: receiving side

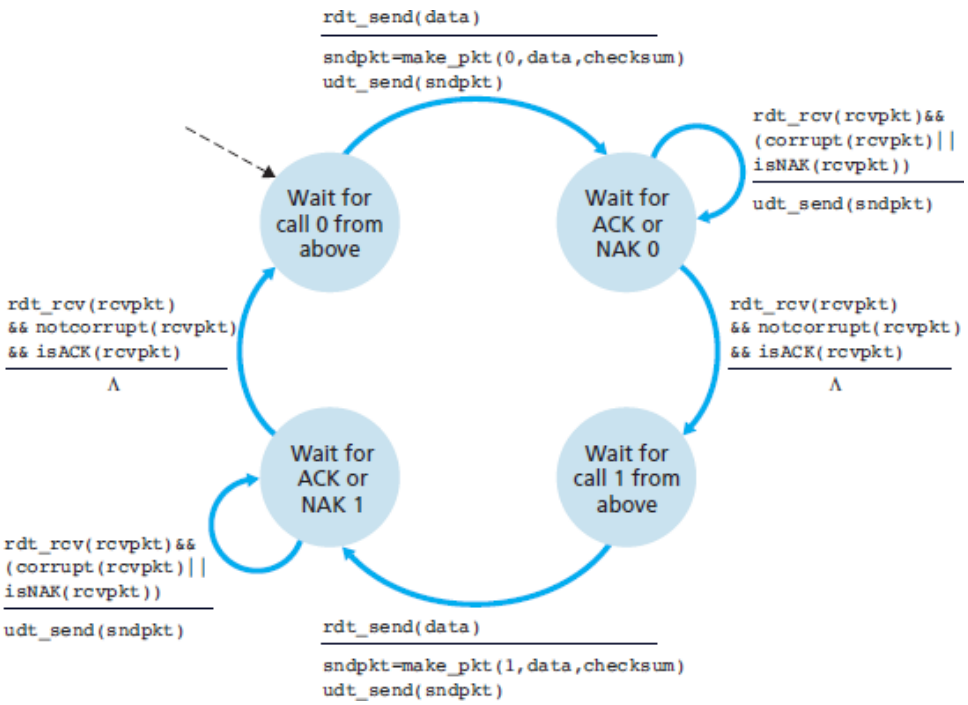
- The sender side has two states. In the leftmost state, the send-side protocol is waiting for data to be passed down from the upper layer.
- When the rdt_send(data) event occurs, the sender will create a packet (sndpkt) containing the data to be sent, along with a packet checksum and then send the packet via the udt_send(sndpkt) operation.
- In the rightmost state, the sender protocol is waiting for an ACK or a NAK packet from the receiver. If an ACK packet is received (the notation rdt_rcv(rcvpkt) && isACK (rcvpkt)), the sender knows that the most recently transmitted packet has been received correctly and thus the protocol returns to the state of waiting for data from the upper layer.
- If a NAK is received, the protocol retransmits the last packet and waits for an ACK or NAK to be returned by the receiver in response to the retransmitted data packet.
- When the sender is in the wait-for-ACK-or-NAK state, it cannot get more data from the upper layer; that is, the rdt_send() event can not occur; that will happen only after the sender receives an ACK and leaves this state. Thus, the sender will not send a new piece of data

until it is sure that the receiver has correctly received the current packet. Because of this behavior, protocols are known as stop-and-wait protocols.

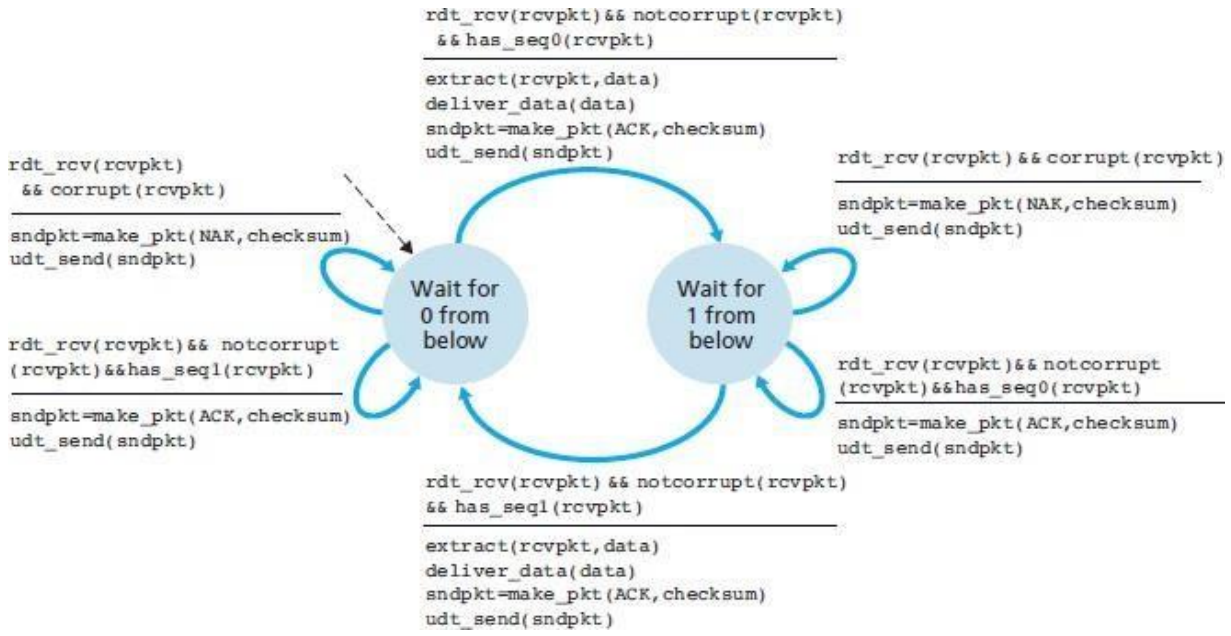
If ACK or NAK is corrupted (Duplicate Packet):

- In this case the sender resends the current data packet when it receives a garbled ACK or NAK packet. This approach, however, introduces duplicate packets into the sender-to-receiver channel.
- A simple solution to this problem is to add a new field to the data packet and have the sender number its data packets by putting a sequence number into this field. The receiver then need only check this sequence number to determine whether or not the received packet is a retransmission.
- For stop-and wait protocol, a 1-bit sequence number will suffice, since it will allow the receiver to know whether the sender is resending the previously transmitted packet or a new packet.

Sender: **RDT 2.1 sender(a) :**



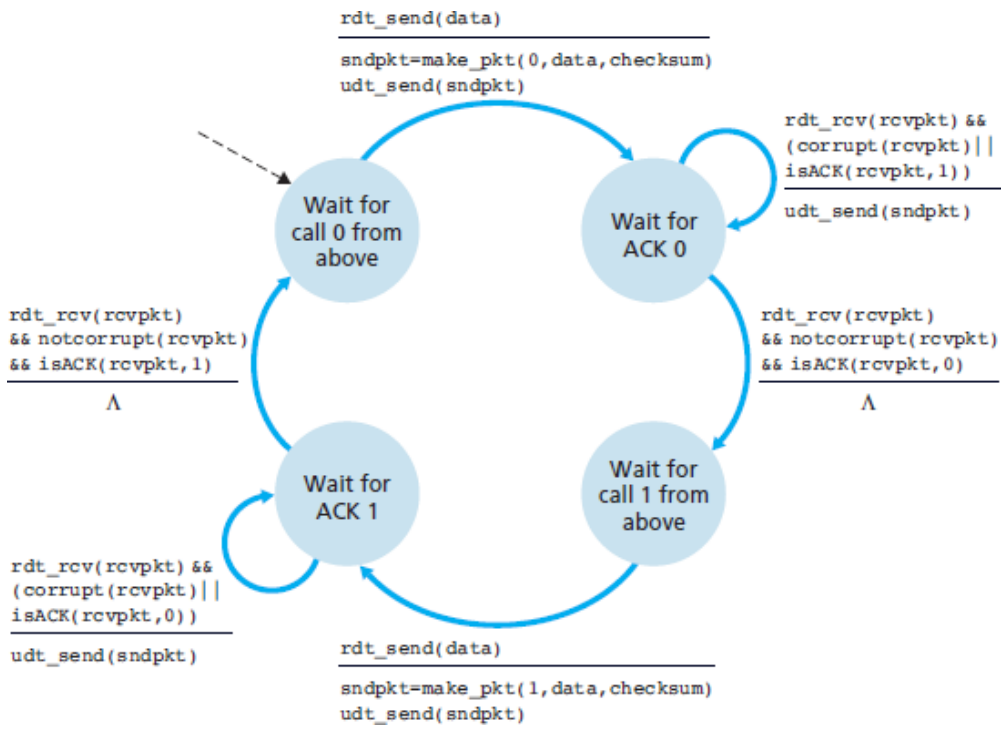
Receiver: RDT 2.1 receiver(b) :



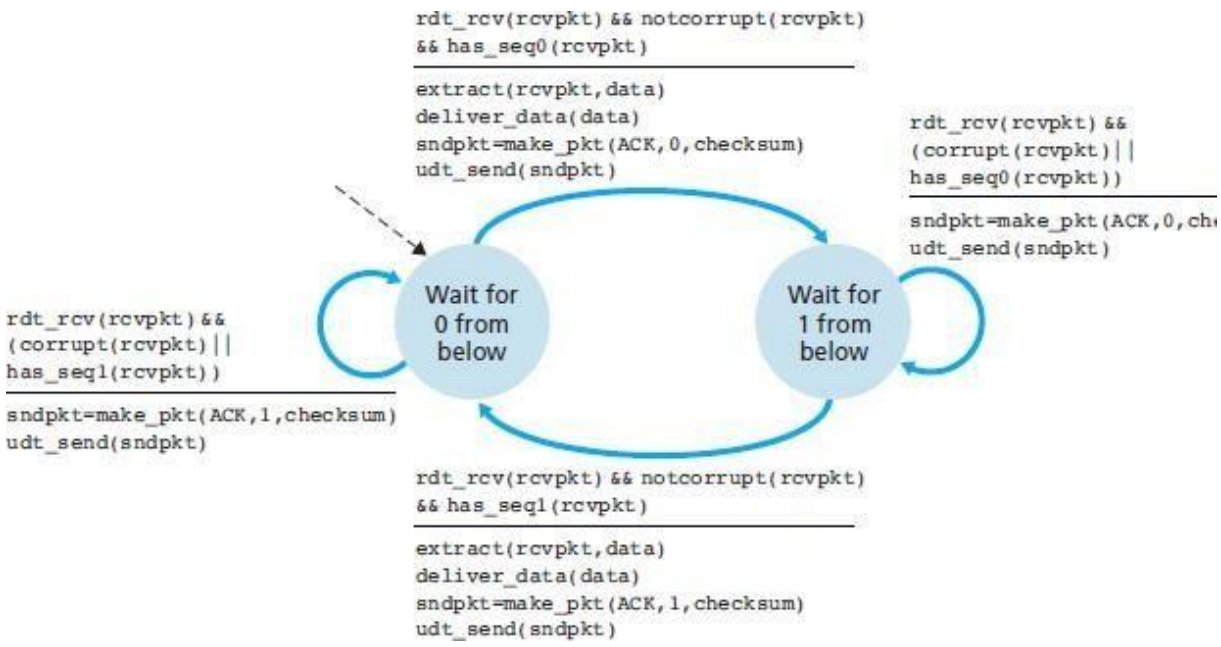
Duplicate ACK

- 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.

Sender: RDT 2.2 sender :

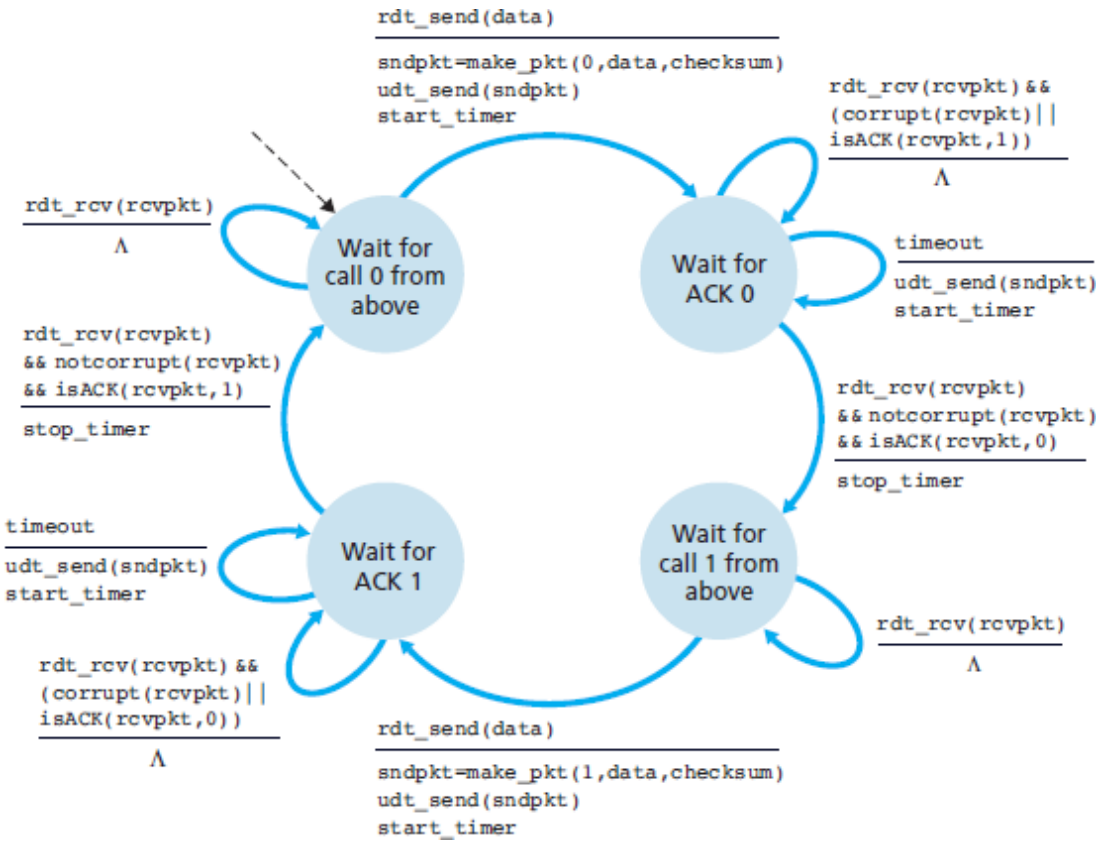


Receiver: RDT 2.2 receiver :

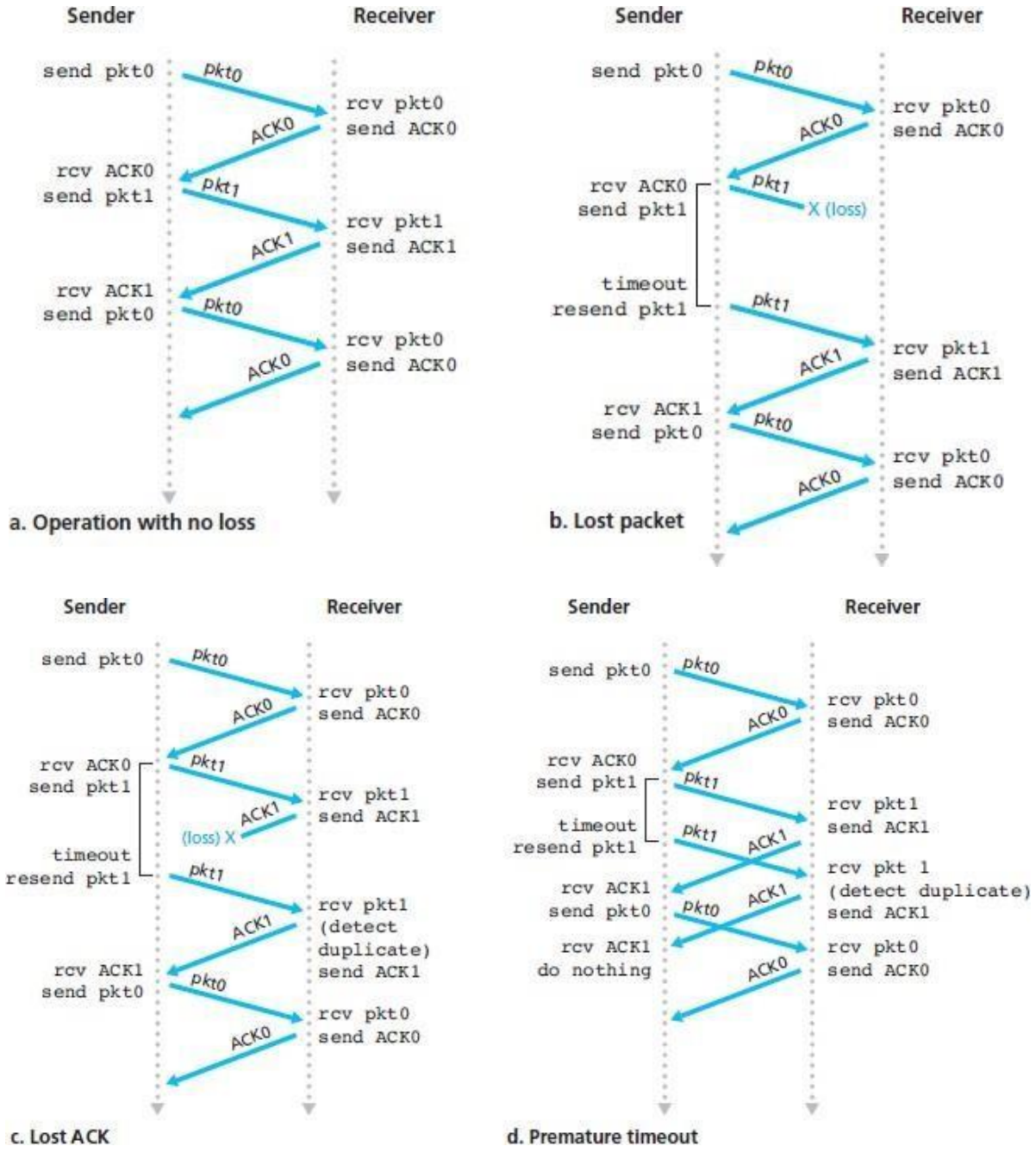


Reliable Data Transfer over a Lossy Channel with Bit Errors: rdt3.0

- Suppose now that in addition to corrupting bits, the underlying channel can lose packets as well.
- Two additional concerns must now be addressed by the protocol: how to detect packet loss and what to do when packet loss occurs.
- Suppose that the sender transmits a data packet and either that packet, or the receiver's ACK of that packet, gets lost. In either case, no reply is forthcoming at the sender from the receiver. If the sender is willing to wait long enough so that it is certain that a packet has been lost, it can simply retransmit the data packet.
- But how long must the sender wait to be certain that something has been lost? The sender must clearly wait at least as long as a round-trip delay between the sender and receiver plus whatever amount of time is needed to process a packet at the receiver.
- The approach thus adopted in practice is for the sender to judiciously choose a time value such that packet loss is likely, although not guaranteed, to have happened. If an ACK is not received within this time, the packet is retransmitted.



Following figures depicts various scenarios:



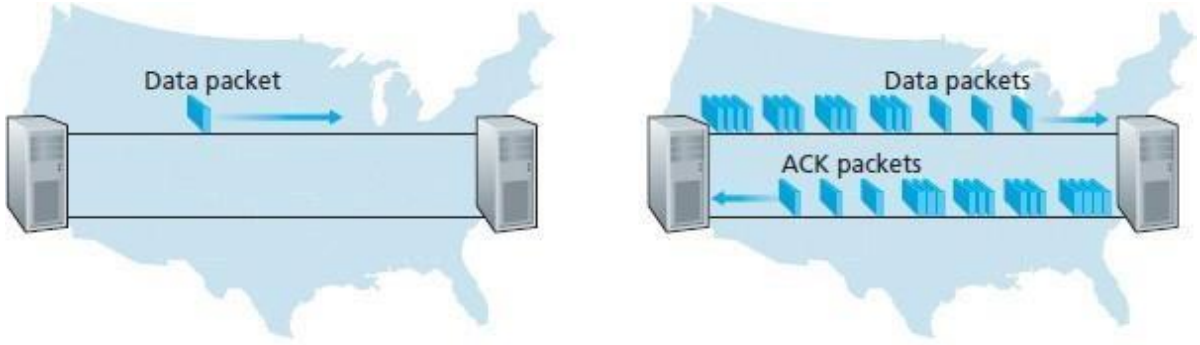
2.4.2 Pipelined Reliable Data Transfer Protocols

Let us consider two hosts located at two different locations.

The speed-of-light round-trip propagation delay between these two end systems, RTT, is approximately 30 milliseconds. Suppose that they are connected by a channel with a

transmission rate, R, of 1 Gbps (10^9 bits per second). With a packet size, L, of 1,000 bytes (8,000 bits) per packet, including both header fields and data, the time needed to
actually transmit the packet into the 1 Gbps link is

$$d_{trans} = \frac{L}{R} = \frac{8000 \text{ bits/packet}}{10^9 \text{ bits/sec}} = 8 \text{ microseconds}$$



a. A stop-and-wait protocol in operation

b. A pipelined protocol in operation

Above Figure (a) shows that with our stop-and-wait protocol, if the sender begins sending the packet at t = 0, then at t = L/R = 8 microseconds, the last bit enters the channel at the sender side. The packet then makes its 15-msec cross-country journey, with the last bit of the packet emerging at the receiver at t = RTT/2 + L/R = **15.008 msec.**

Assuming for simplicity that ACK packets are extremely small (so that we can ignore their transmission time) and that the receiver can send an ACK as soon as the last bit of a data packet is received, the ACK emerges back at the sender at t = RTT + L/R = **30.008 msec.**

At this point, the sender can now transmit the next message. Thus, in **30.008 msec**, the sender was sending for only 0.008 msec. If we define the utilization of the sender (or the channel) as the fraction of time the sender is actually busy sending bits into the channel, the analysis shows that the stop-and-wait protocol has a rather dismal sender utilization, Usender, of

$$U_{sender} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.00027$$

That is, the sender was busy only 2.7 hundredths of one percent of the time!

The solution to this particular performance problem is simple: Rather than operate in a stop-and-wait manner, the sender is allowed to send multiple packets without waiting for acknowledgments. if the sender is allowed to transmit three packets before having to wait

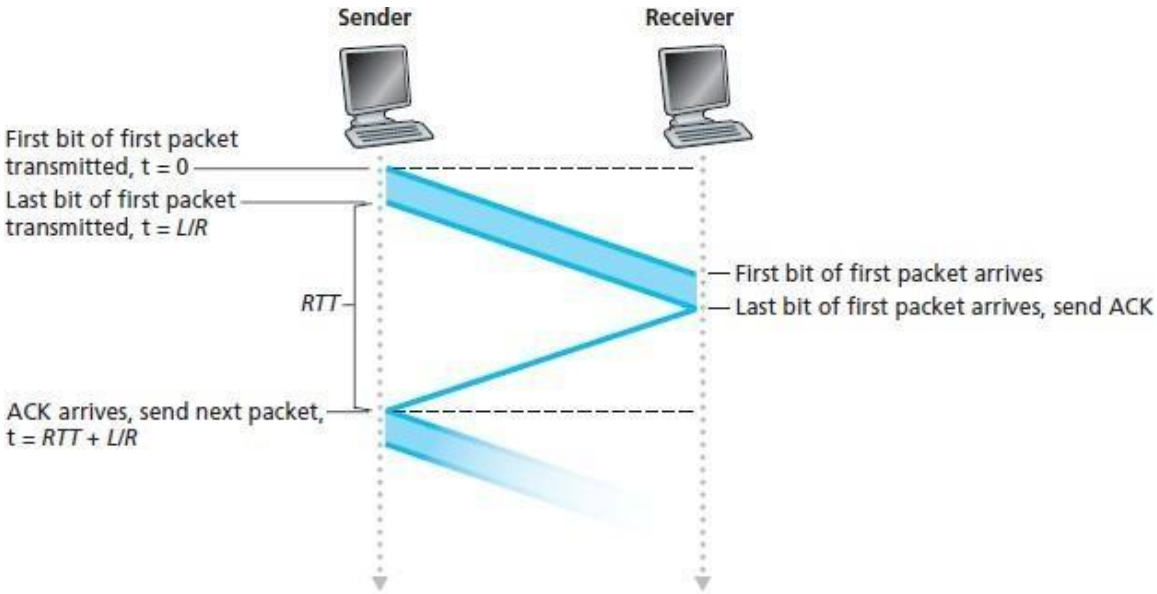
for acknowledgments.

Since the many in-transit sender-to-receiver packets can be visualized as filling a pipeline, this technique is known as pipelining. Pipelining has the following consequences for reliable data transfer protocols:

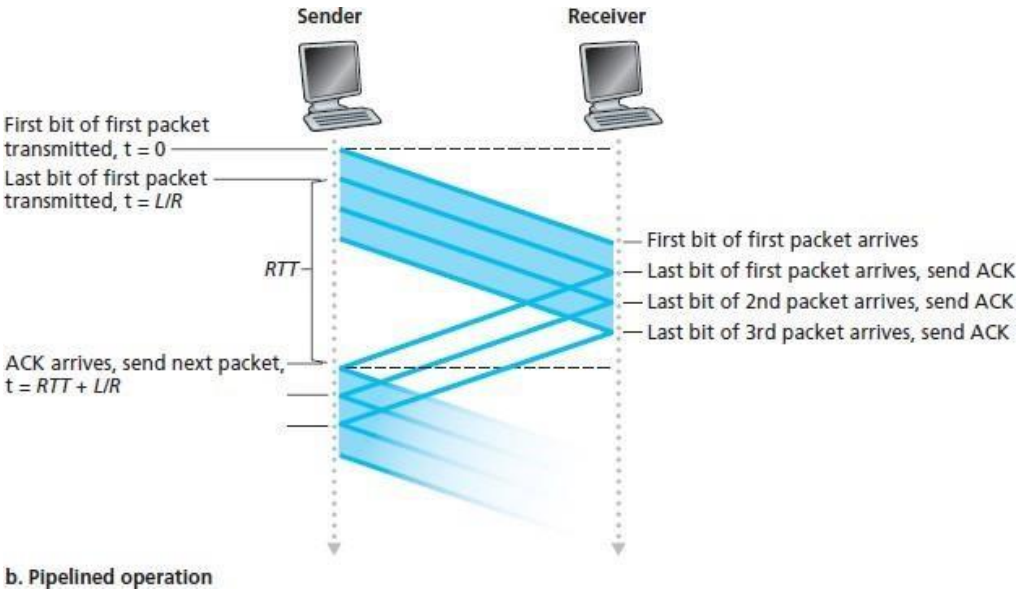
• The range of sequence numbers must be increased, since each in-transit packet (not counting retransmissions) must have a unique sequence number and there may be multiple, in-transit, unacknowledged packets.

• The sender and receiver sides of the protocols may have to buffer more than one packet. Minimally, the sender will have to buffer packets that have been transmitted but not yet acknowledged. Buffering of correctly received packets may also be needed at the receiver, as discussed below.

• The range of sequence numbers needed and the buffering requirements will depend on the manner in which a data transfer protocol responds to lost, corrupted, and overly delayed packets. Two basic approaches toward pipelined error recovery can be identified: Go-Back-N and selective repeat.



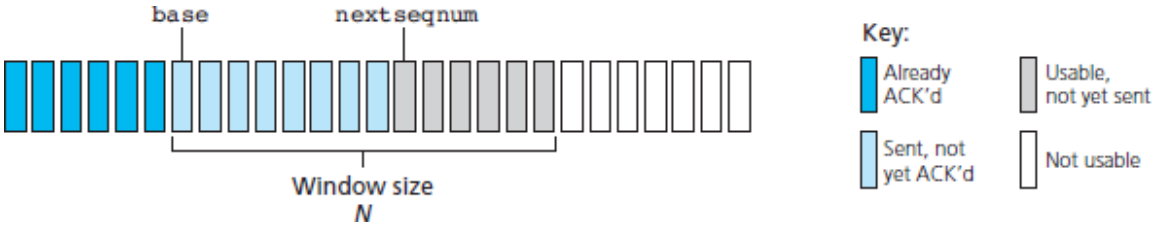
a. Stop-and-wait operation



b. Pipelined operation

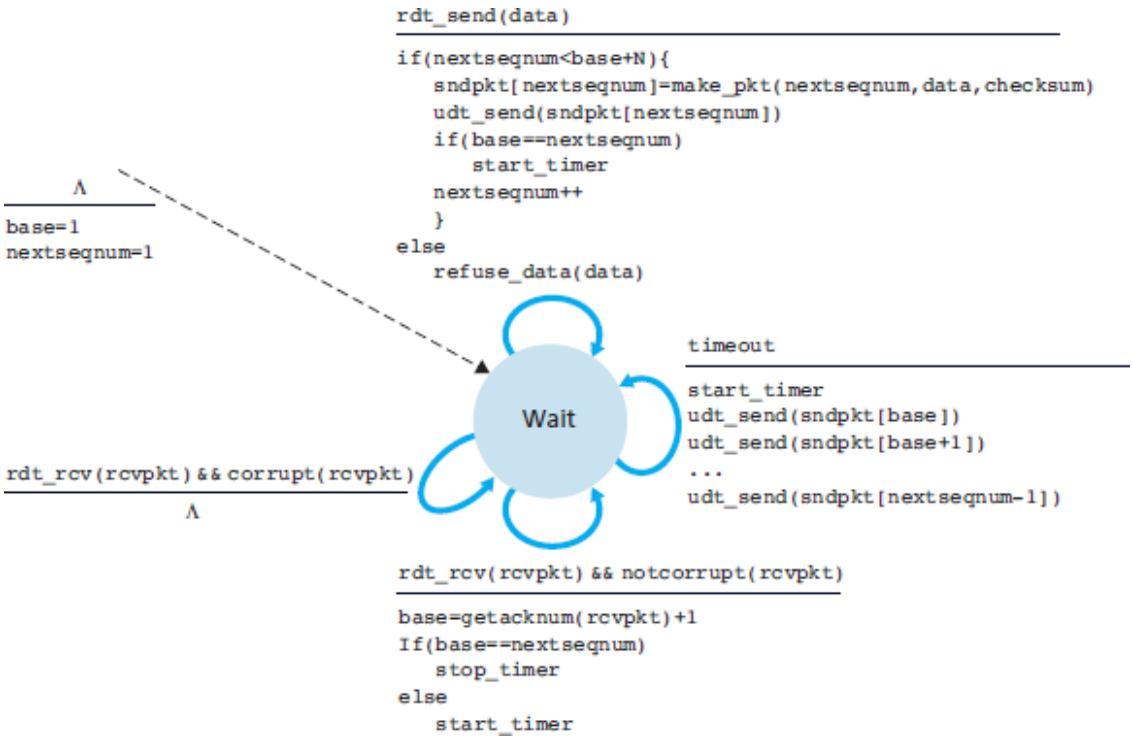
2.4.3 Go-Back-N (GBN)

- 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 base is the sequence number of the oldest unacknowledged packet and nextseqnum is the smallest unused sequence number (that is, the sequence number of the next packet to be sent). 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.

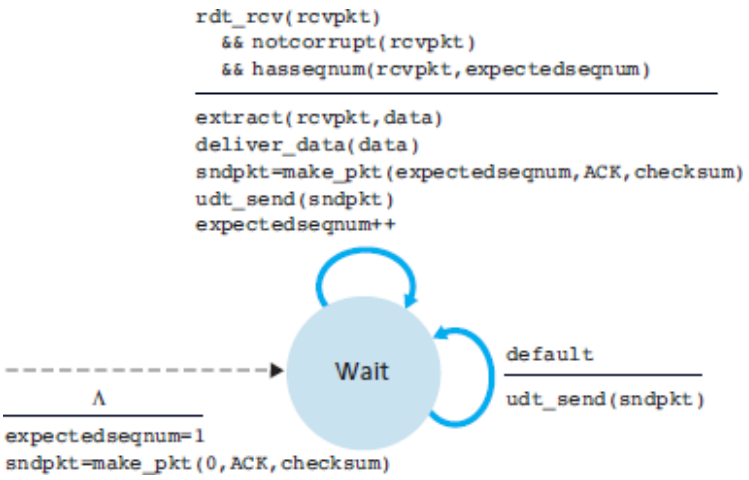


- Sequence numbers greater than or equal to base+N cannot be used until an unacknowledged packet currently in the pipeline has been acknowledged.

- The range of permissible sequence numbers for transmitted but not yet acknowledged packets can be viewed as a window of size N over the range of sequence numbers. 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 [0,2k – 1]. With a finite range of sequence numbers, all arithmetic involving sequence numbers must then be done using modulo 2k arithmetic.



GBN Sender



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 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. We'll come back to this issue shortly when we examine the receiver side of GBN.
- **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.

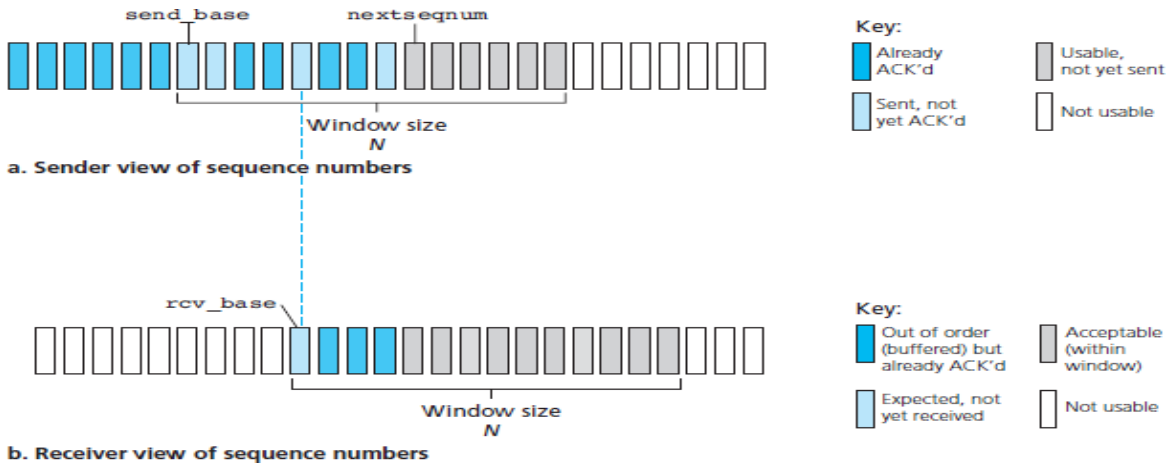
Receiver:

- If a packet with sequence number n is received correctly and is in order, 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.
- In our GBN protocol, the receiver discards out-of-order packets. GBN discard a correctly received but out-of-order packet.
- 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.

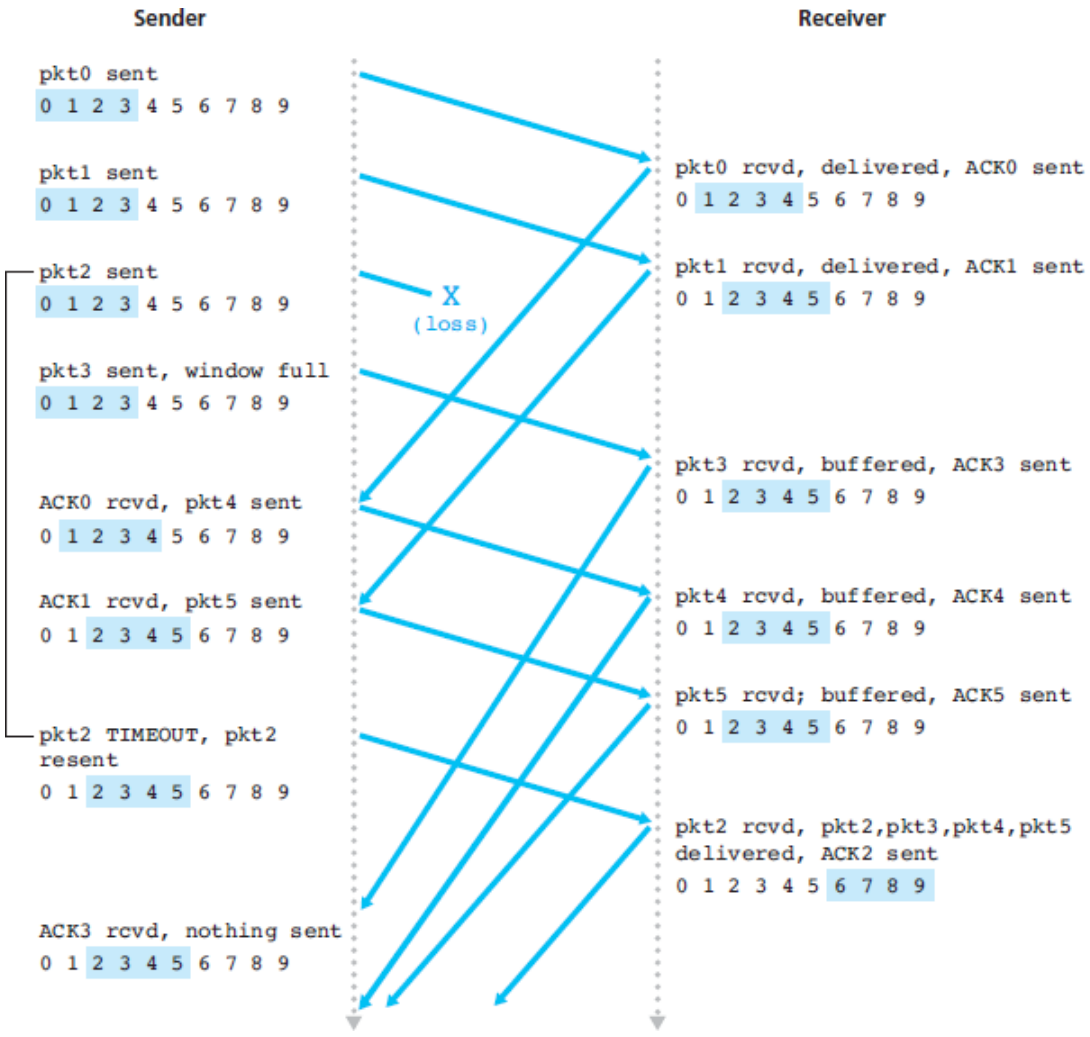
2.4.4 Selective Repeat (SR)

- Limitation of GBN: GBN itself suffers from performance problems. In particular, 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.
- As the name suggests, 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.
- This individual, as needed, retransmission will require that the receiver individually acknowledge correctly received packets.
- A window size of N will again be used to limit the number of outstanding, unacknowledged packets in the pipeline.
- 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.



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 [Varghese 1997].
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 sender 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. As an example, consider Figure 3.26. When a packet with a sequence number of `rcv_base=2` is received, it and packets 3, 4, and 5 can be 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.

SR receiver events and actions

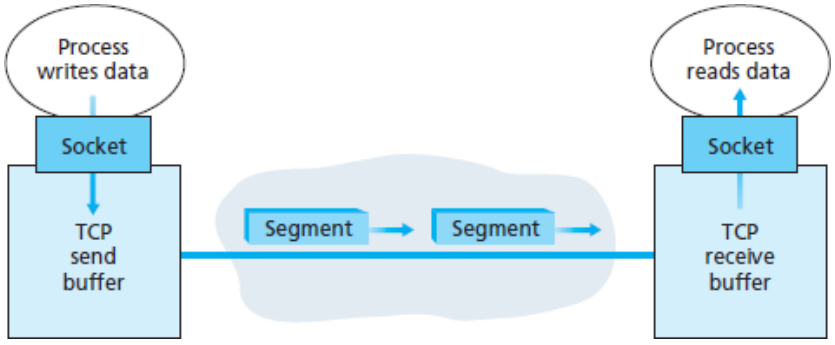
Summary of reliable data transfer mechanisms and their use

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.

2.5 Connection-Oriented Transport: TCP

2.5.1 The TCP Connection

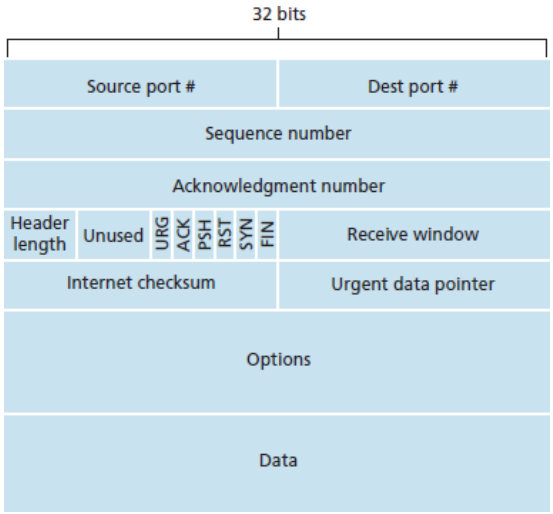
- TCP is said to be connection-oriented because connection has to be established between two application processes before they start transmitting data.
- As part of TCP connection establishment, both sides of the connection will initialize many TCP state variables associated with the TCP connection.
- 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. Multicasting is not allowed.
- Once a TCP connection is established, the two application processes can send data to each other.
- Let's consider the sending of data from the client process to the server process. The client process passes a stream of data through the socket
- Once the data passes through the door, the data is in the hands of TCP running in the client.



- 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 (maximum transmission unit, MTU), and then setting the MSS to ensure that a TCP segment plus the TCP/IP header length will fit into a single link-layer frame.
- 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.
- The application reads the stream of data from this buffer.
- Each side of the connection has its own send buffer and its own receive buffer.

2.5.2 TCP Segment Structure



- The TCP segment consists of header fields and a data field.
- The data field contains a chunk of application data.
- The minimum length of TCP header is 20 bytes.
- The header includes **source and destination port numbers**, which are used for multiplexing/demultiplexing data from/to upper-layer applications.
- The header includes a **checksum field** for error detection.

- 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.
  - The 16-bit **receive window** field is used for flow control. It is 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.
  - 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.
  - 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.

Sequence Numbers and Acknowledgment Numbers

The sequence number for a segment is the byte-stream number of the first byte in the segment.

Example: Suppose that a process in Host A wants to send a stream of data to a process in Host B over a TCP connection. The TCP in Host A will implicitly number each byte in the data stream. Suppose that the data stream consists of a file consisting of 500,000 bytes, that the MSS is 1,000 bytes, and that the first byte of the data stream is numbered 0. TCP constructs 500 segments out of the data stream. The first segment gets assigned sequence number 0, the second segment gets assigned sequence number 1,000, the third segment gets assigned sequence number 2,000, and so

on. Each sequence number is inserted in the sequence number field in the header of the appropriate TCP segment.

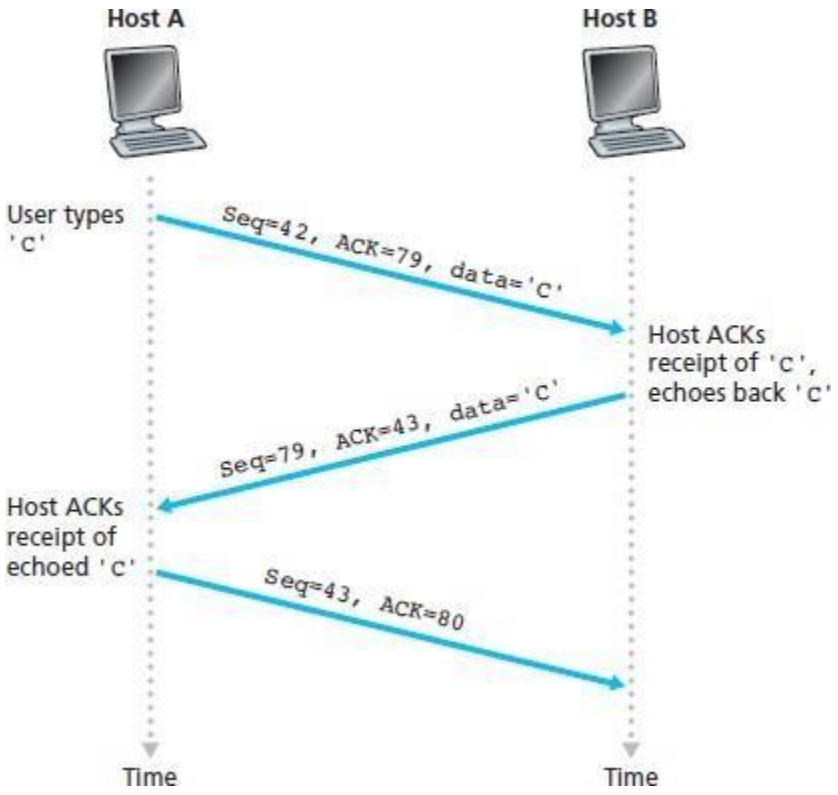
The **acknowledgment number** that Host A puts in its segment is the sequence number of the next byte Host A is expecting from Host B.

Example: 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.

Telnet: A Case Study for Sequence and Acknowledgment Numbers

Telnet is a popular application-layer protocol used for remote login. It runs over TCP and is designed to work between any pair of hosts.

Suppose Host A initiates a Telnet session with Host B. Because Host A initiates the session, it is labeled the client, and Host B is labeled the server. Each character typed by the user (at the client) will be sent to the remote host; the remote host will send back a copy of each character, which will be displayed on the Telnet user's screen. This "echo back" is used to ensure that characters seen by the Telnet user have already been received and processed at the remote site. Each character thus traverses the network twice between the time the user hits the key and the time the character is displayed on the user's monitor.



2.5.3 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.

The SampleRTT values will fluctuate from segment to segment due to congestion in the routers and to the varying load on the end systems.

In order to estimate a typical RTT, it is therefore natural to take some sort of average of the SampleRTT values. TCP maintains an average, called EstimatedRTT, of the SampleRTT values.

$$\text{EstimatedRTT} = (1 - \alpha) \cdot \text{EstimatedRTT} + \alpha \cdot \text{SampleRTT}$$

Here $\alpha = 0.125$

In statistics, This kind of average is called an exponential weighted moving average (EWMA).

In addition to having an estimate of the RTT, it is also valuable to have a measure of the variability of the RTT : DevRTT.

DevRTT, is an estimate of how much SampleRTT typically deviates from EstimatedRTT:

$$DevRTT = (1 - \beta) \cdot DevRTT + \beta \cdot |SampleRTT - EstimatedRTT|$$

Here $\beta = 0.25$

Now Timeout can be calculated as:

$$TimeoutInterval = EstimatedRTT + 4 \cdot DevRTT$$

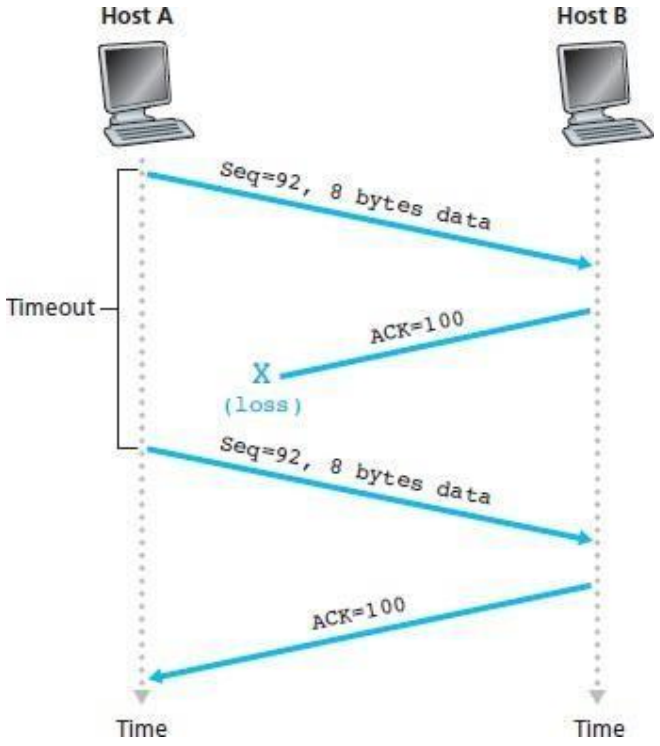
2.5.4 Reliable Data Transfer

- 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.

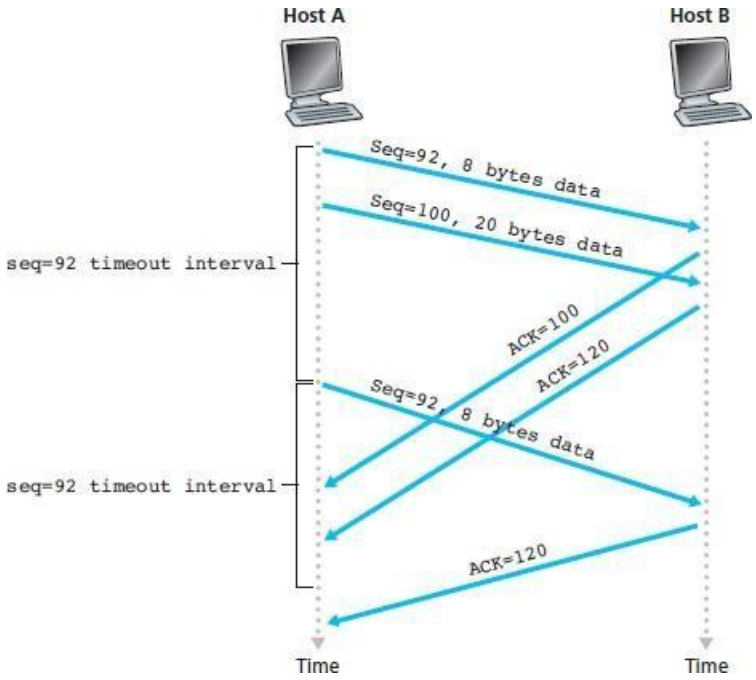
```
NextSeqNum=InitialSeqNumber
SendBase=InitialSeqNumber

loop (forever) {
    switch(event)

        event: data received from application above
            create TCP segment with sequence number NextSeqNum
            if (timer currently not running)
                start timer
            pass segment to IP
            NextSeqNum=NextSeqNum+length(data)
            break;

        event: timer timeout
            retransmit not-yet-acknowledged segment with
                smallest sequence number
            start timer
            break;

        event: ACK received, with ACK field value of y
            if (y > SendBase) {
                SendBase=y
                if (there are currently any not-yet-acknowledged segments)
                    start timer
                }
            break;

    } /* end of loop forever */
```

A Few Interesting Scenarios:

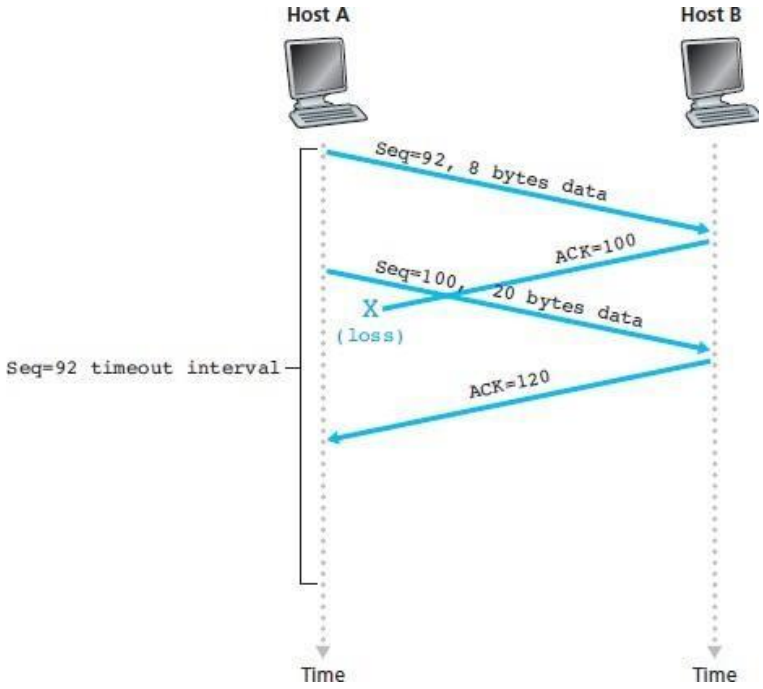
1. Host A sends one segment to Host B. Suppose that this segment has sequence number 92 and contains 8 bytes of data. After sending this segment, Host A waits for a segment from B with acknowledgment number 100. Although the segment from A is received at B, the acknowledgment from B to A gets lost. In this case, the timeout event occurs, and Host A retransmits the same segment. Of course, when Host B receives the retransmission, it observes from the sequence number that the segment contains data that has already been received. Thus, TCP in Host B will discard the bytes in the retransmitted segment.



2. Host A sends two segments back to back. The first segment has sequence number 92 and 8 bytes of data, and the second segment has sequence number 100 and 20 bytes of data. Suppose that both segments arrive intact at B, and B sends two separate acknowledgments for each of these segments. The first of these acknowledgments has acknowledgment number 100; the second has acknowledgment number 120. Suppose now that neither of the acknowledgments arrives at Host A before the timeout. When the timeout event occurs, Host A resends the first segment with sequence number 92 and restarts the timer. As long as the ACK for the second segment arrives before the new timeout, the second segment will not be retransmitted.



3. Host A sends the two segments, exactly as in the second example. The acknowledgment of the first segment is lost in the network, but just before the timeout event, Host A receives an acknowledgment with acknowledgment number 120. Host A therefore knows that Host B has received everything up through byte 119; so Host A does not resend either of the two segments.



Doubling the Timeout Interval

Each time TCP retransmits, it sets the next timeout interval to twice the previous value, rather than deriving it from the last EstimatedRTT and DevRTT.

For example, suppose Timeout Interval associated with the oldest not yet acknowledged segment is 0.75 sec when the timer first expires. TCP will then retransmit this segment and set the new expiration time to 1.5 sec. If the timer expires again 1.5 sec later, TCP will again retransmit this segment, now setting the expiration time to 3.0 sec.

Fast Retransmit

One of the problems with timeout-triggered retransmissions is that the timeout period can be relatively long. 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.

Event	TCP Receiver Action
Arrival of in-order segment with expected sequence number. All data up to expected sequence number already acknowledged.	Delayed ACK. Wait up to 500 msec for arrival of another in-order segment. If next in-order segment does not arrive in this interval, send an ACK.
Arrival of in-order segment with expected sequence number. One other in-order segment waiting for ACK transmission.	Immediately send single cumulative ACK, ACKing both in-order segments.
Arrival of out-of-order segment with higher-than-expected sequence number. Gap detected.	Immediately send duplicate ACK, indicating sequence number of next expected byte (which is the lower end of the gap).
Arrival of segment that partially or completely fills in gap in received data.	Immediately send ACK, provided that segment starts at the lower end of gap.

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.

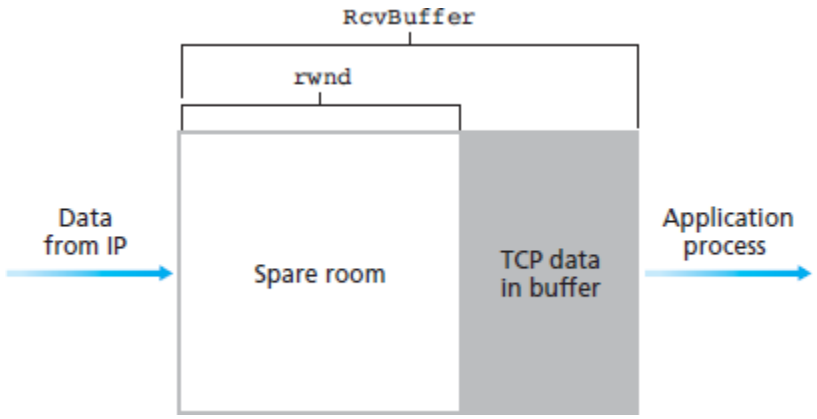
Because a sender often sends a large number of segments back to back, if one segment is lost, there will likely be many back-to-back duplicate ACKs. If the TCP sender receives three duplicate ACKs for the same data, it takes this as an indication that the segment following the segment that has been ACKed three times has been lost. In the case that three duplicate ACKs

are received, the TCP sender performs a fast retransmit, retransmitting the missing segment before that segment's timer expires.

```
event: ACK received, with ACK field value of y
            if (y > SendBase) {
                    SendBase=y
                    if (there are currently any not yet
                                acknowledged segments)
                        start timer
                    }
            else { /* a duplicate ACK for already ACKed
                    segment */
                increment number of duplicate ACKs
                    received for y
                if (number of duplicate ACKS received
                    for y==3)
                    /* TCP fast retransmit */
                    resend segment with sequence number y
                }
            break;
```

2.5.5 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 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.
- Suppose that Host A is 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]

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.

Host A in turn keeps track of two variables, LastByteSent and Last- ByteAcked. 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

2.5.6 TCP Connection Management

TCP has 3 phases

1) Connection Establishment phase

2) Data transmission phase

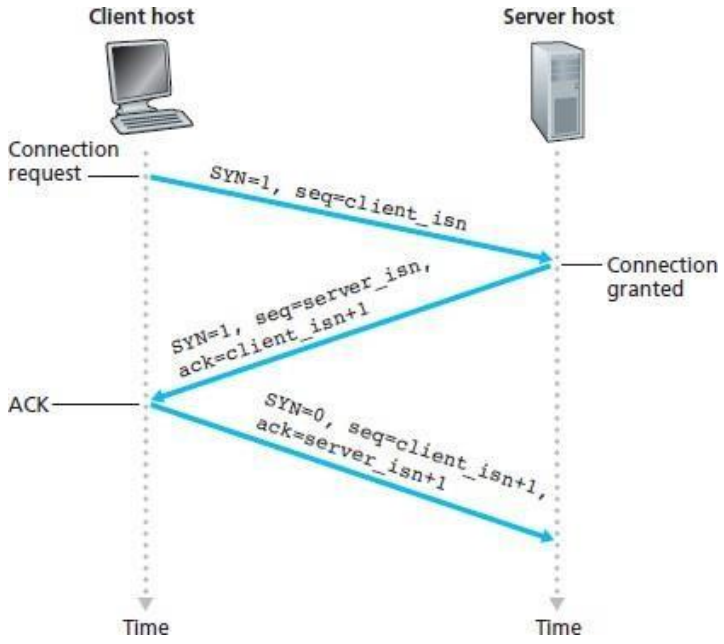
3) Connection Termination phase

Connection establishment phase:

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.

Step 2: Once the IP datagram containing the TCP SYN segment arrives at the server host 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. This connection-granted segment also contains no application layer data. However, it does contain three important pieces of information in the segment header. 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 is referred as 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 SYN bit is set to zero, since the connection is established. This third stage of the three-way handshake may carry client-to-server data in the segment payload.

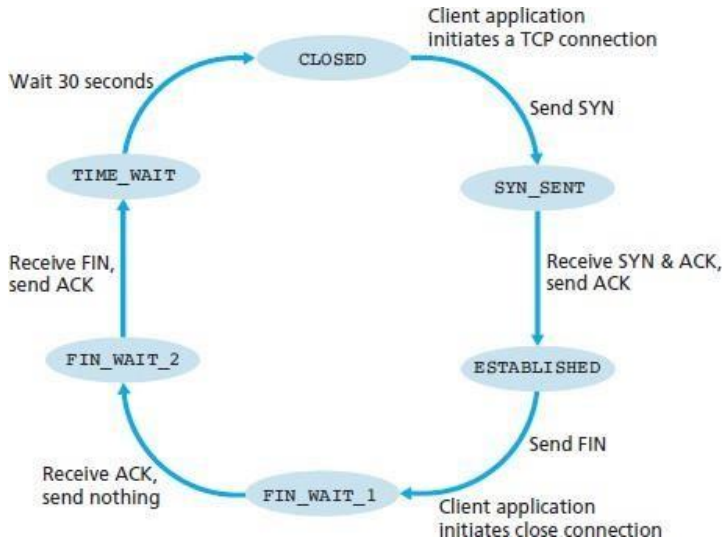


Connection Termination phase:

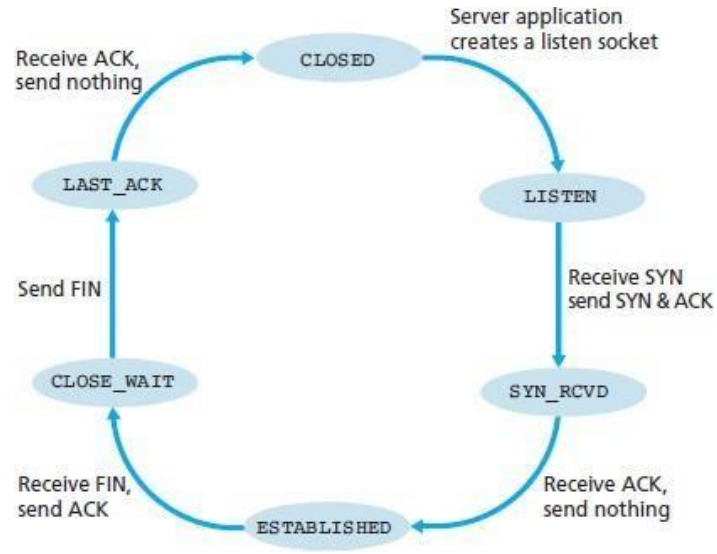
Either of the two processes participating in a TCP connection can end the connection. When a connection ends, the "resources" (that is, the buffers and variables) in the hosts are deallocated. For connection termination TCP sends segment with FIN flag set to 1. When the server receives this segment, it sends the client an acknowledgment segment in return. The server then sends its own shutdown segment, which has the FIN bit set to 1. Finally, the client acknowledges the server's shutdown segment. At this point, all the resources in the two hosts are now deallocated.

State transition diagram:

Client



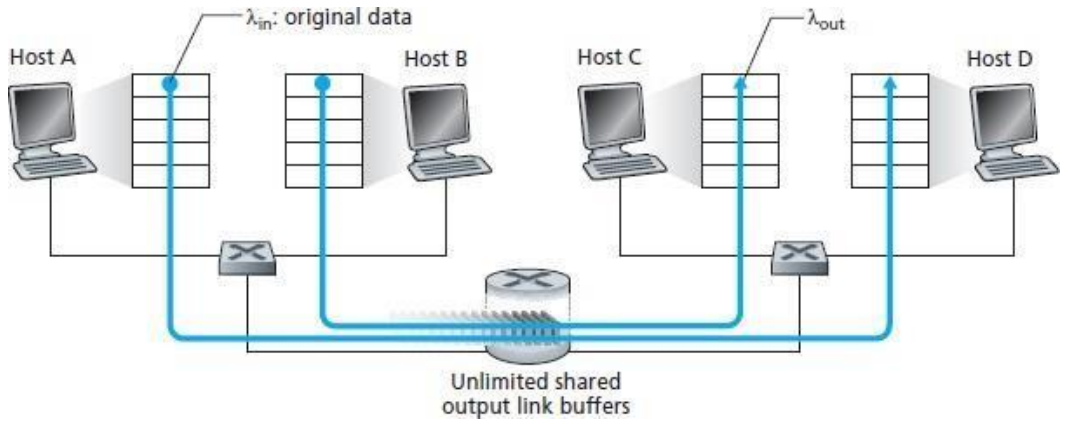
Server:



2.6 Principles of Congestion Control

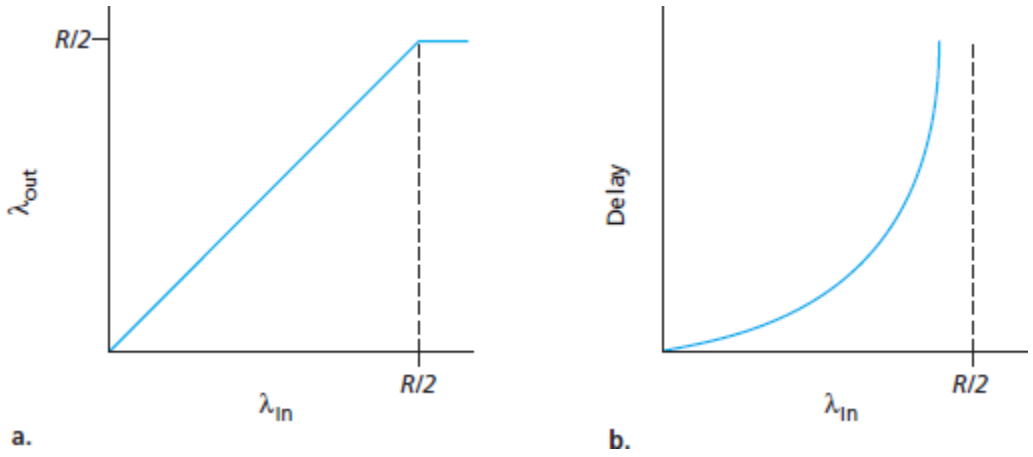
2.61The Causes and the Costs of Congestion

Scenario 1: Two Senders, a Router with Infinite Buffers

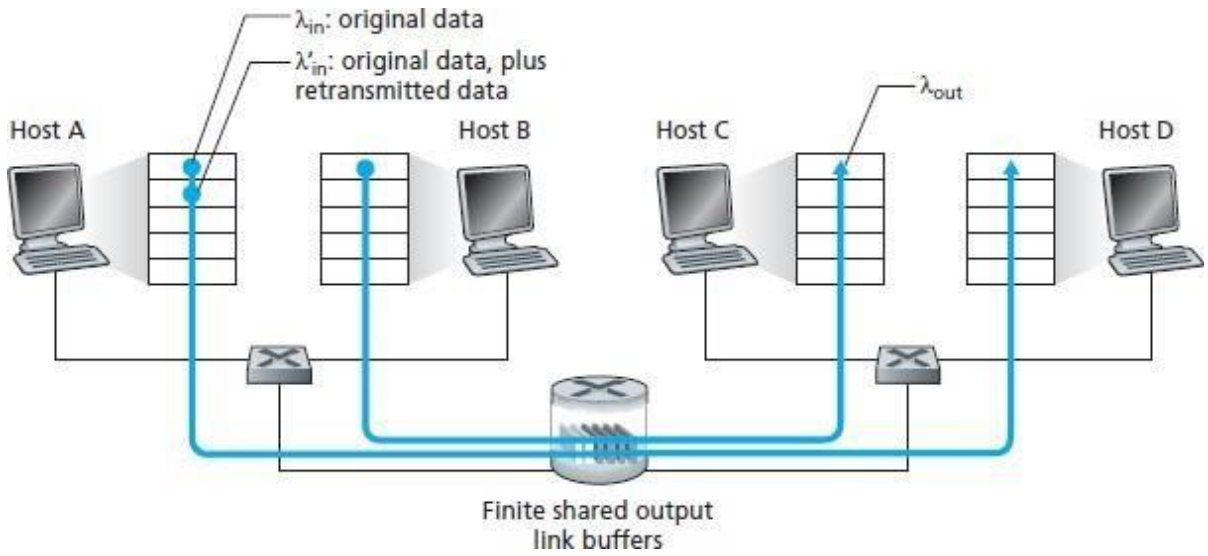


- Two hosts (A and B) each have a connection that shares a single hop between source and destination.
- Let's assume that the application in Host A is sending data into the connection at an average rate of λ_{in} bytes/sec.
- These data are original in the sense that each unit of data is sent into the socket only once. The underlying transport-level protocol is a simple one.
- Data is encapsulated and sent; no error recovery (for example, retransmission), flow control, or congestion control is performed.
- Ignoring the additional overhead due to adding transport- and lower-layer header information, the rate at which Host A offers traffic to the router in this first scenario is thus λ_{in} bytes/sec.
- Host B operates in a similar manner, and we assume for simplicity that it too is sending at a rate of λ_{in} bytes/sec.
- Packets from Hosts A and B pass through a router and over a shared outgoing link of capacity R. The router has buffers that allow it to store incoming packets when the packet-arrival rate exceeds the outgoing link's capacity. In this first scenario, we assume that the router has an infinite amount of buffer space.

- Below figure plots the performance of Host A's connection under this first scenario. The left graph plots the per-connection throughput (number of bytes per second at the receiver) as a function of the connection-sending rate.

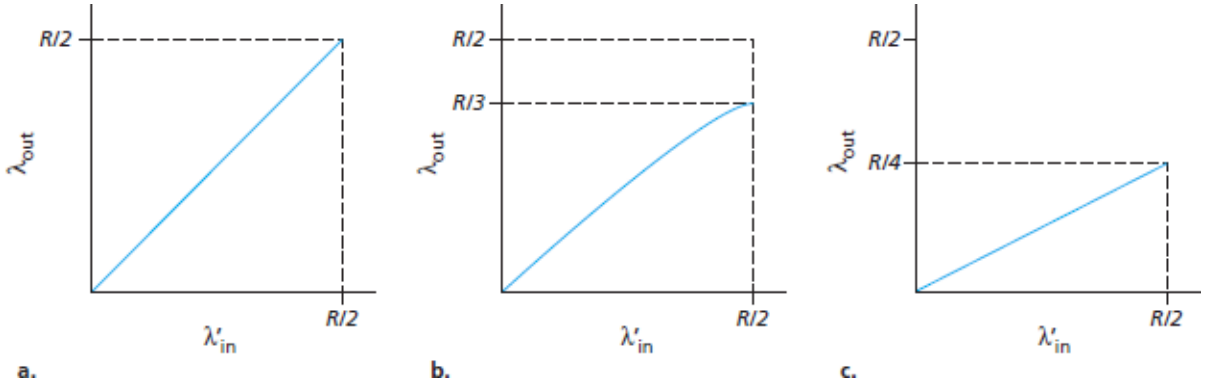


Scenario 2: Two Senders and a Router with Finite Buffers



- First, the amount of router buffering is assumed to be finite. A consequence of this real-world assumption is that packets will be dropped when arriving to an already full buffer.
- Second, we assume that each connection is reliable. If a packet containing a transport-level segment is dropped at the router, the sender will eventually retransmit it.
- Because packets can be retransmitted, we must now be more careful with our use of the term sending rate.

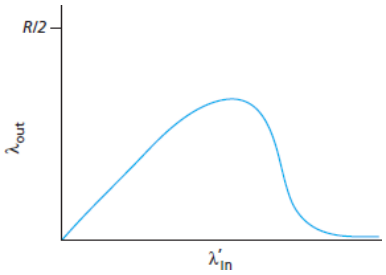
- Specifically, let us again denote the rate at which the application sends original data into the socket by λ_{in} in bytes/sec. The rate at which the transport layer sends segments (containing original data and retransmitted data) into the network will be denoted λ'_{in} in bytes/sec. λ'_{in} in is sometimes referred to as the offered load to the network.

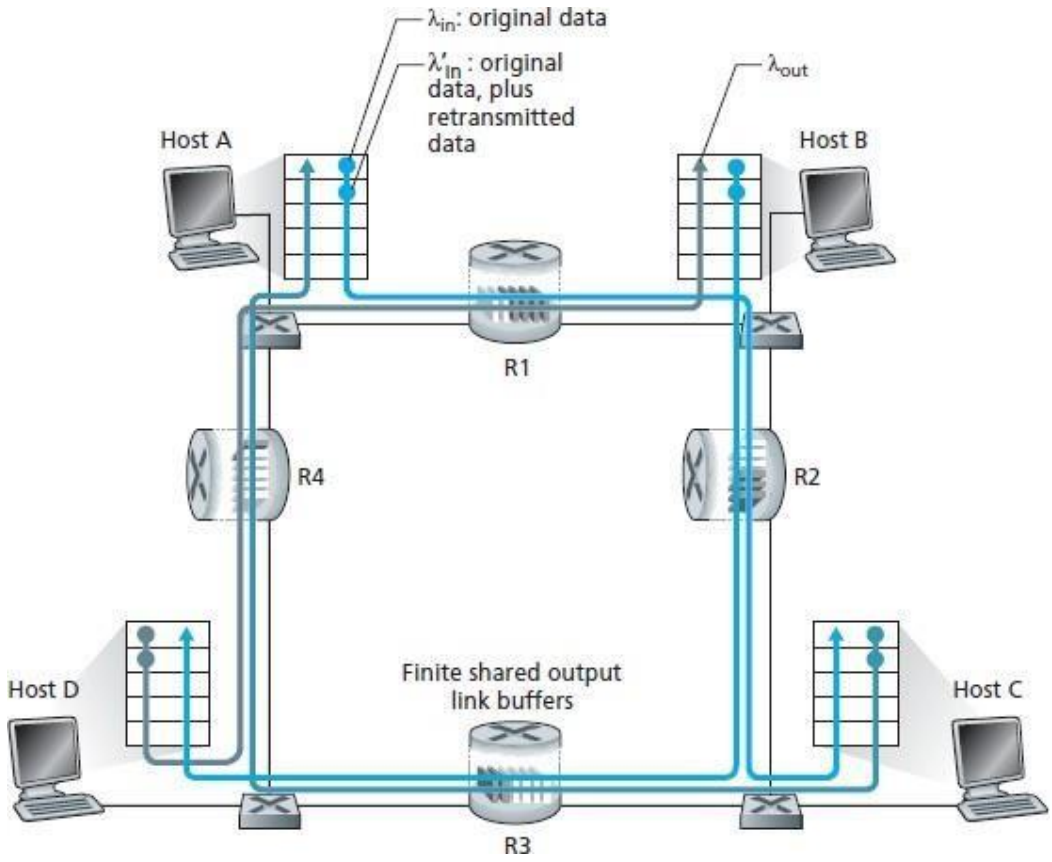


- We see here another cost of a congested network— the sender must perform retransmissions in order to compensate for dropped (lost) packets due to buffer overflow.
- Here then is yet another cost of a congested network—unneeded retransmissions by the sender in the face of large delays may cause a router to use its link bandwidth to forward unneeded copies of a packet.

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

- We again assume that each host uses a timeout/retransmission mechanism to implement a reliable data transfer service, that all hosts have the same value of λ_{in} in, and that all router links have capacity R bytes/sec.
- Here we see yet another cost of dropping a packet due to congestion— when a packet is dropped along a path, the transmission capacity that was used at each of the upstream links to forward that packet to the point at which it is dropped ends up having been wasted.





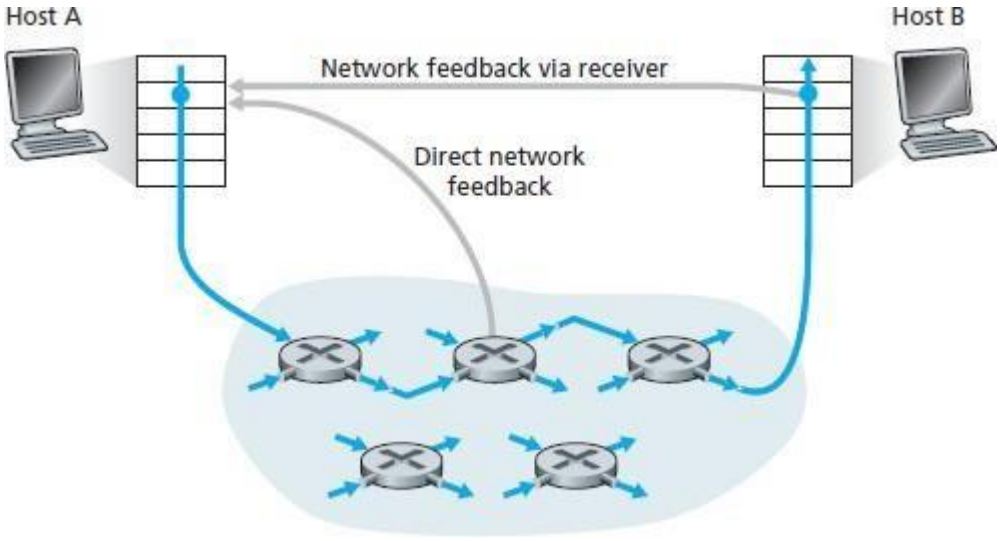
2.6.2 Approaches to Congestion Control

- **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 congestion control purposes. Even the presence of congestion in the network must be inferred by the end systems based only on observed network behavior
- **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.

For network-assisted congestion control, congestion information is typically fed back from the network to the sender in one of two ways:

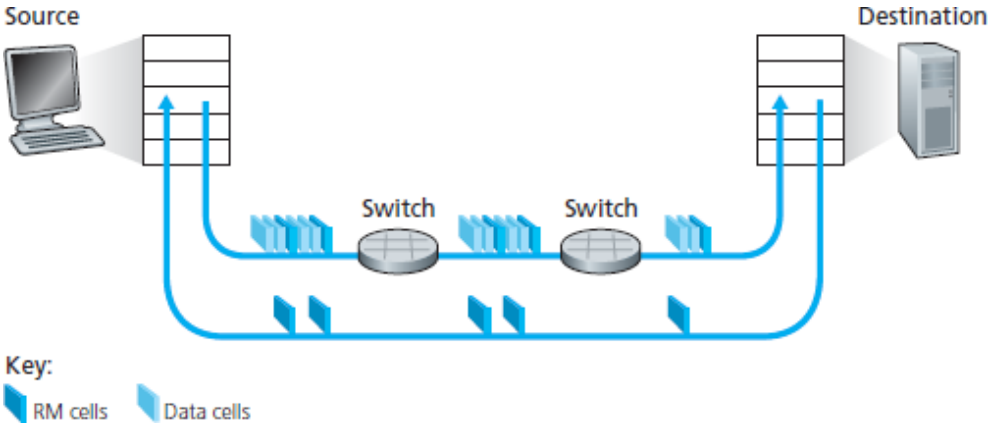
Direct feedback may be sent from a network router to the sender. This form of notification typically takes the form of a **choke packet.**

The second form of notification occurs when a router marks/updates a field in a packet flowing from sender to receiver to indicate congestion. Upon receipt of a marked packet, the receiver then notifies the sender of the congestion indication.



2.6.3 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.
- 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
  - 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.

2.7 TCP Congestion Control

- 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:

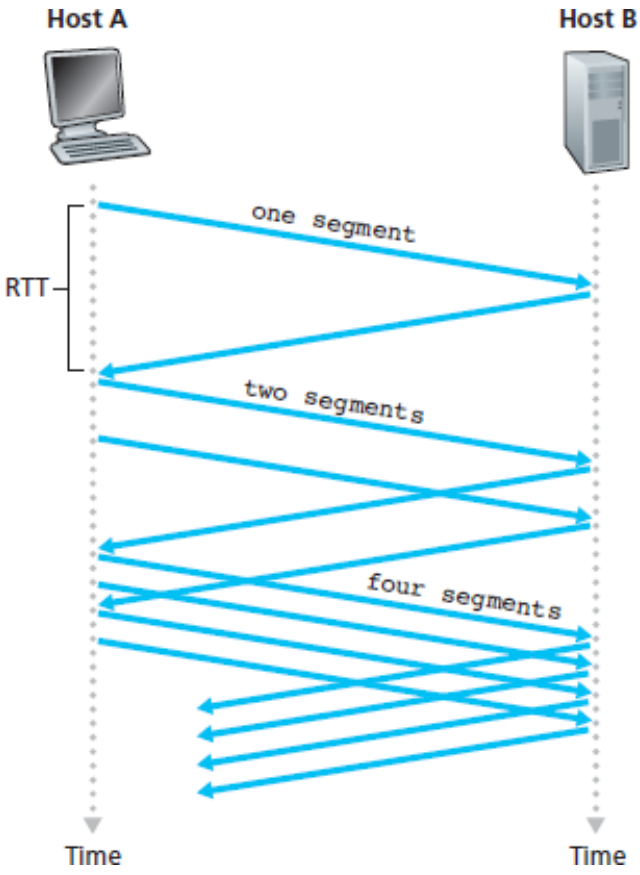
$$LastByteSent - LastByteAcked \leq \min\{cwnd, rwnd\}$$

- 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.
- Congestion window can be adjusted with following 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.

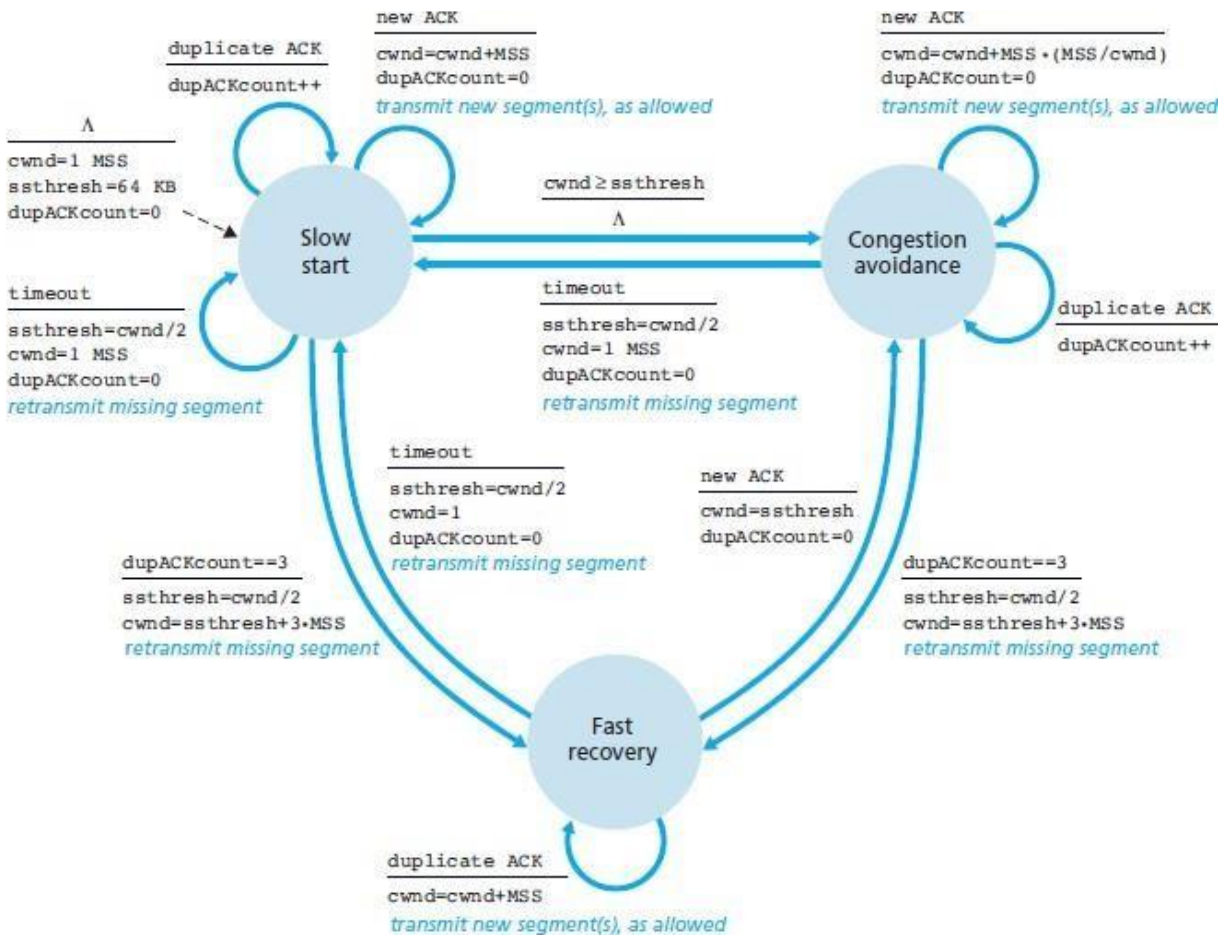
TCP Congestion control mechanisms:

1) Slow Start

- When a TCP connection begins, the value of cwnd is typically initialized to a small value of 1 MSS, resulting in an initial sending rate of roughly MSS/RTT.
- 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.
- In the below figure 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.



- 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 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. Since ssthresh is half the value of cwnd when congestion was last detected, it might be a bit reckless to keep doubling cwnd when it reaches or surpasses the value of ssthresh. Thus, when the value of cwnd equals ssthresh, slow start ends and TCP transitions into congestion avoidance mode.
- If three duplicate ACKs are detected, in which case TCP performs a fast retransmit and enters the fast recovery state.



2) 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.
- 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.

3) 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.

Macroscopic Description of TCP Throughput:

$$\text{average throughput of a connection} = \frac{0.75 \cdot W}{RTT}$$

Where W is window size

TCP Over High-Bandwidth Paths:

$$\text{average throughput of a connection} = \frac{1.22 \cdot MSS}{RTT\sqrt{L}}$$

L is loss rate L

RTT is the round-trip time

MSS is maximum segment size