Module – 2

Transport Layer : Introduction and Transport-Layer Services: Relationship Between Transport and Network Layers, Overview of the Transport Layer in the Internet, Multiplexing and Demultiplexing: Connectionless Transport: UDP,UDP Segment Structure, UDP Checksum, Principles of Reliable Data Transfer: Building a Reliable Data Transfer Protocol, Pipelined Reliable Data Transfer Protocols, Go-Back-N, Selective repeat, Connection-Oriented Transport TCP: The TCP Connection, TCP Segment Structure, Round-Trip Time Estimation and Timeout, Reliable Data Transfer, Flow Control, TCP Connection Management, Principles of Congestion Control: The Causes and the Costs of Congestion, Approaches to Congestion Control.

TRANSPORT LAYER

Transport layer resides in between Application layer and Network layer. It has the critical role of providing communication services directly to the application processes running on different hosts.

3.1 **Introduction and Transport-Layer Services**

- A transport-layer protocol provides for **logical communication** between application processes running on different hosts.
- *Logical communication* - from an application's perspective, it is assumed as ,the hosts running the processes were directly connected; in reality, the hosts may be in remote location, connected via numerous routers and a wide range of link types.
- Application processes use the logical communication provided by the transport layer to send messages to each other, without the knowledge of physical infrastructure used to carry these messages.

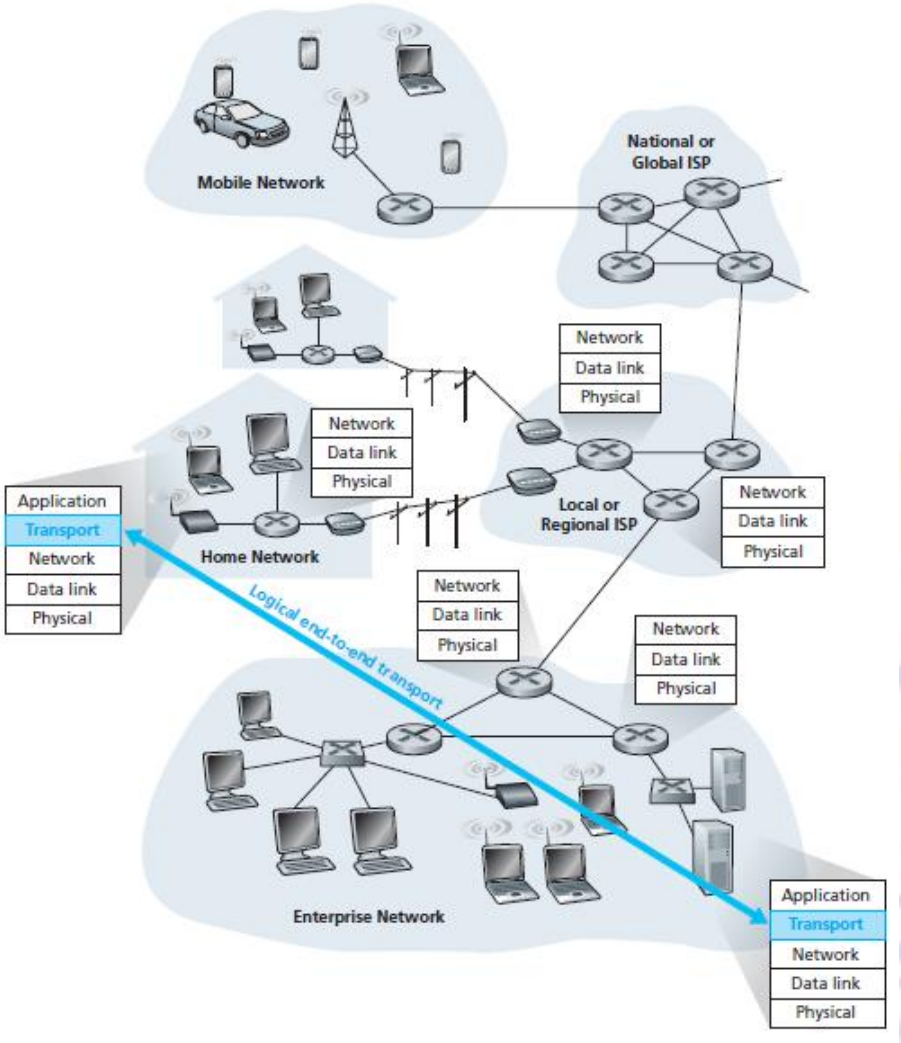


Figure 3.1 illustrates the notion of logical communication.

As shown in Figure 3.1, 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**.
- This is done by 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.
- Network routers act only on the network-layer fields of the datagram; that is, they do not examine the fields of the transport-layer segment encapsulated with the datagram.
- 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.

3.1.1 **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***.
- Transport-layer protocols live in the end systems.
- Within an end system, a transport protocol moves messages from application processes to the network layer and vice versa.
- Intermediate routers will not recognize, any information that the transport layer may have added to the application messages.
- Transport layer provides reliable service where as network layer provides unreliable service.

3.1.2 **Overview of the Transport Layer in the Internet**

Two protocols at transport layer are :

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

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

Transport layer packet is referred as a ***segment***. Transport-layer packet for TCP is referred as a **segment** and for UDP as a **datagram**.

Responsibility of IP at Network layer:

- The Internet's network-layer protocol has a name—IP, for Internet Protocol. IP provides logical communication between hosts.
- The IP service model is a **best-effort delivery service**. IP makes "best effort" to deliver segments between communicating hosts.
- IP does not guarantee segment delivery, orderly delivery of segments and the integrity of the data in the segments. Hence, IP is an **unreliable service**.
- Every host has at least one network-layer address, called IP address.

Responsibility of UDP and TCP at Transport layer:

UDP and TCP extends 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 provides integrity checking by including error detection fields in their segments' headers.

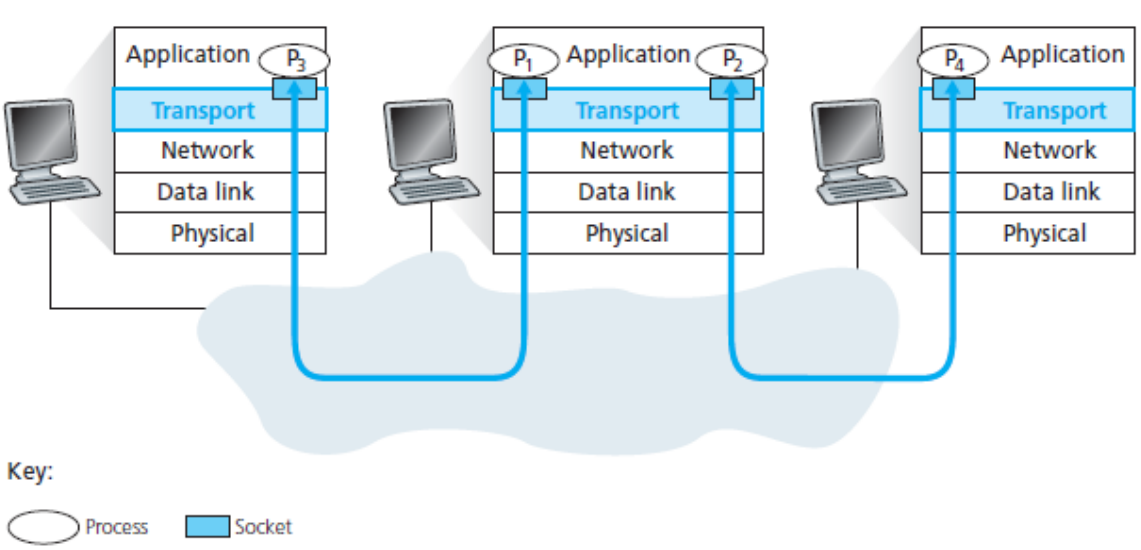
Services provided by UDP & TCP :

UDP is an unreliable service—it does not guarantee that data sent by one process will arrive intact to the destination process. UDP traffic is unregulated.

TCP Service :

- 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 converts IP's unreliable service between end systems into a reliable data transport service between processes.
- TCP 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.
- TCP strives to give each connection traversing a congested link an equal share of the link bandwidth.
- Regulates the rate at which the sending sides of TCP connections can send traffic into the network.

3.2 **Multiplexing and Demultiplexing**



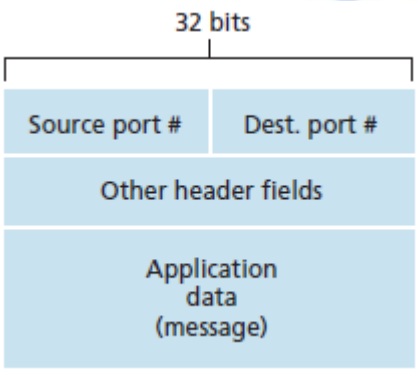
- When the transport layer in receiving host receives data from the network layer below, it needs to direct the received data to one of the four processes P1,P2,P3,P4.

- A process can have one or more **sockets**, through which data passes from the network to the process and through which data passes from the process to the network. Each socket has a unique identifier.
- Thus, as shown in Figure above, the transport layer in the receiving host does not deliver data directly to a process, but instead to an intermediary socket.
- Each transport-layer segment has a set of fields in the segment to redirect data to above layer. At the receiving end, the transport layer examines these fields to identify the receiving socket and then directs the segment to that socket.
- Delivering the data in a transport-layer segment to the correct socket is called **demultiplexing**.
- 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**.
- The transport layer in the middle host in Figure above must demultiplex segments arriving from the network layer below to either process P1 or P2 above;
- Demultiplexing is done by directing the arriving segment's data to the corresponding process's socket.
- The transport layer in the middle host must also gather outgoing data from these sockets, form transport-layer segments, and pass these segments down to the network layer

Transport-layer multiplexing requires :

(1) Unique identifiers for sockets
(2) Each segment must 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** They are reserved for use by well-known application protocols such as HTTP (80) and FTP (21).

Each socket in the host could be assigned a port number, and when a segment arrives at the host, the transport layer examines the destination port number in the segment and directs the segment to the corresponding socket. The segment's data then passes through the socket into the attached process.

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:

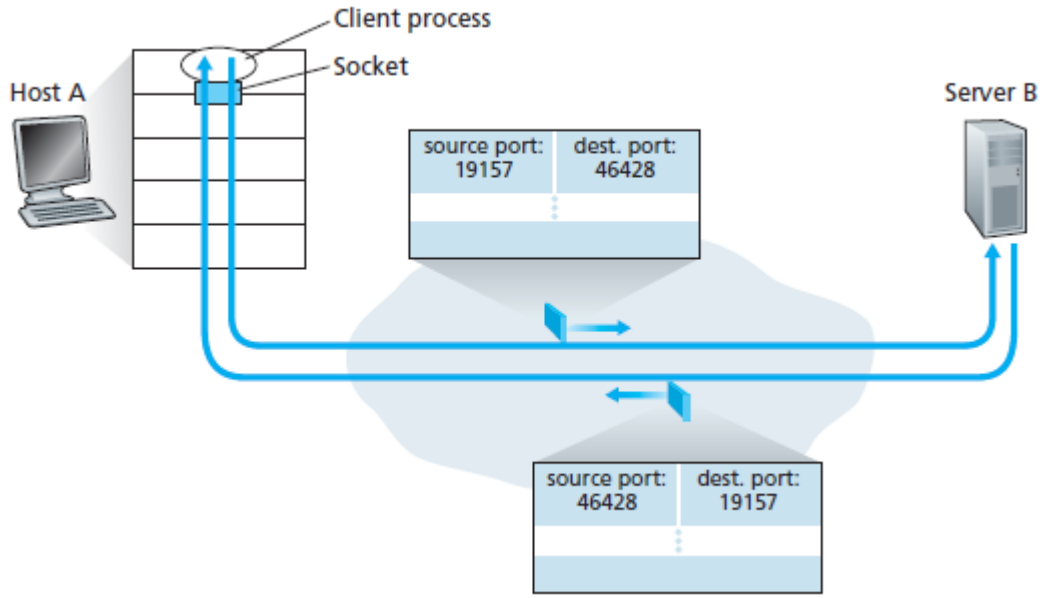
clientSocket = socket(AF_INET, SOCK_STREAM)

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

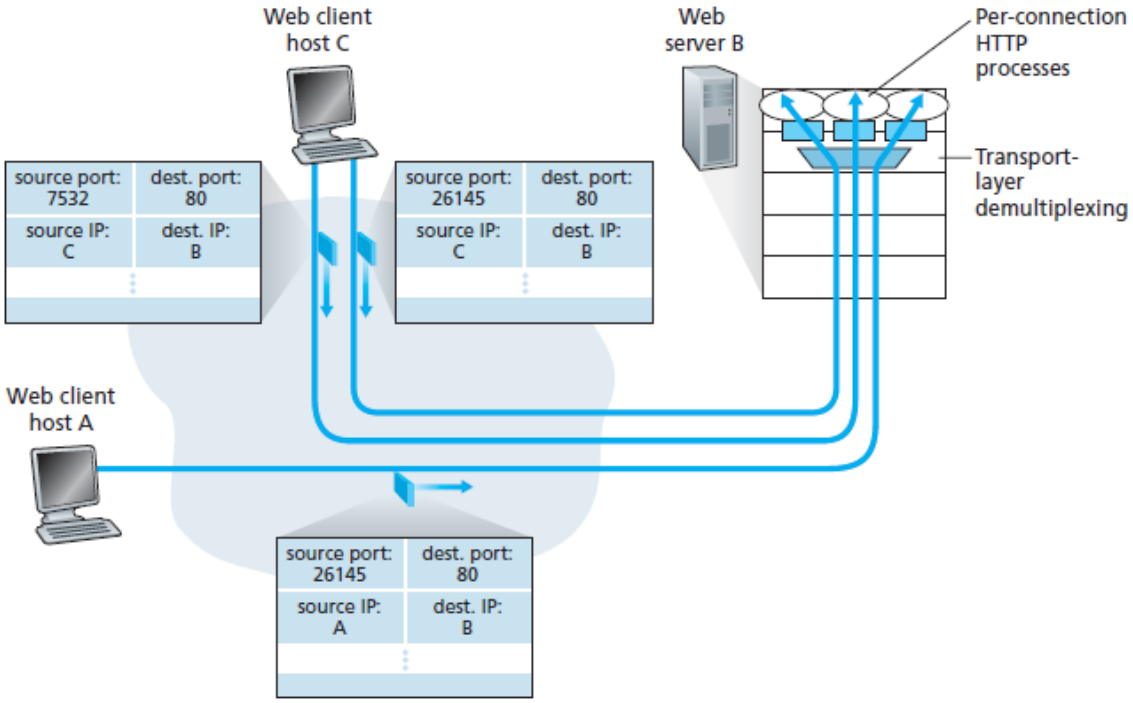
(4) its own IP address.

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.

3.3 **Connectionless Transport: UDP**

The transport layer has to provide a multiplexing/demultiplexing service in order to pass data between the network layer and the correct application-level process.

UDP, does the multiplexing/demultiplexing function , 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.

UDP has no handshaking between sending and receiving transport-layer entities before sending a segment. Hence, UDP is said to be *connectionless.*

UDP is best suited for many applications for the following reasons:

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 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.

No connection establishment.

TCP uses a three-way handshake before it starts to transfer data. UDP just sends data away without any formal preliminaries.

Thus UDP does not introduce any delay to establish a connection.

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, 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.

E-mail, remote terminal access, the Web, and file transfer run over TCP—all these applications need the reliable data transfer service of TCP.

UDP and TCP are used today with multimedia applications, such as Internet phone, real-time video conferencing, and streaming of stored audio and video.

The lack of congestion control in UDP can result in high loss rates between a UDP sender and receiver.

3.3.1 **UDP Segment Structure**

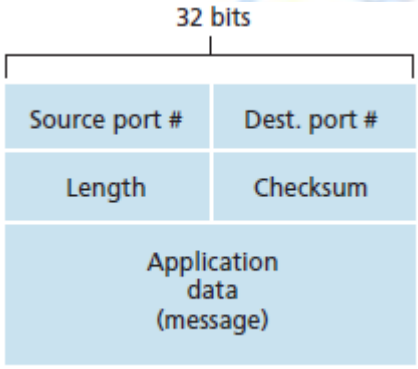


Figure :The UDP segment structure.

The application data occupies the data field of the UDP segment. For example, For a streaming audio application, audio samples fill the data field.

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.

The length field specifies the length of the UDP segment, including the header, in bytes.

3.3.2 **UDP Checksum**

The UDP checksum provides for error detection. That is, the checksum is used to determine whether bits within the UDP segment have been altered 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.

Example, suppose that we have the following three 16-bit words:

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

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 the sum at the receiver will be 1111111111111111.

If one of the bits is a 0, then errors have been introduced into the packet.

Take 1’s compliment of generated sum,which is all 0’s. Hence, no error has been detected.

UDP provides error checking because there is no guarantee that all the links between source and destination provide error checking; that is, one of the links may use a link-layer protocol that does not provide error checking.

Even if segments are correctly transferred across a link, it's possible that bit errors could be introduced when a segment is stored in a router's memory.

Neither link-by-link reliability nor in-memory error detection is guaranteed, UDP must provide error detection at the transport layer, *on an end-end basis*, if the end-end data transfer service is to provide error detection.

UDP has no error recovery method.

3.4 **Principles of Reliable Data Transfer**

With the reliable channel, no transferred data bits are corrupted or lost, and all are delivered in the order in which they were sent. This is the service model offered by TCP to the Internet applications that invoke it. It is the responsibility of a **reliable data transfer protocol** to implement this service abstraction.

The layer *below* the reliable data transfer protocol may be unreliable. For example, TCP is a reliable data transfer protocol that is implemented on top of an unreliable (IP) end-to-end network layer.

Figure 3.8(b) illustrates the interfaces for data transfer protocol.

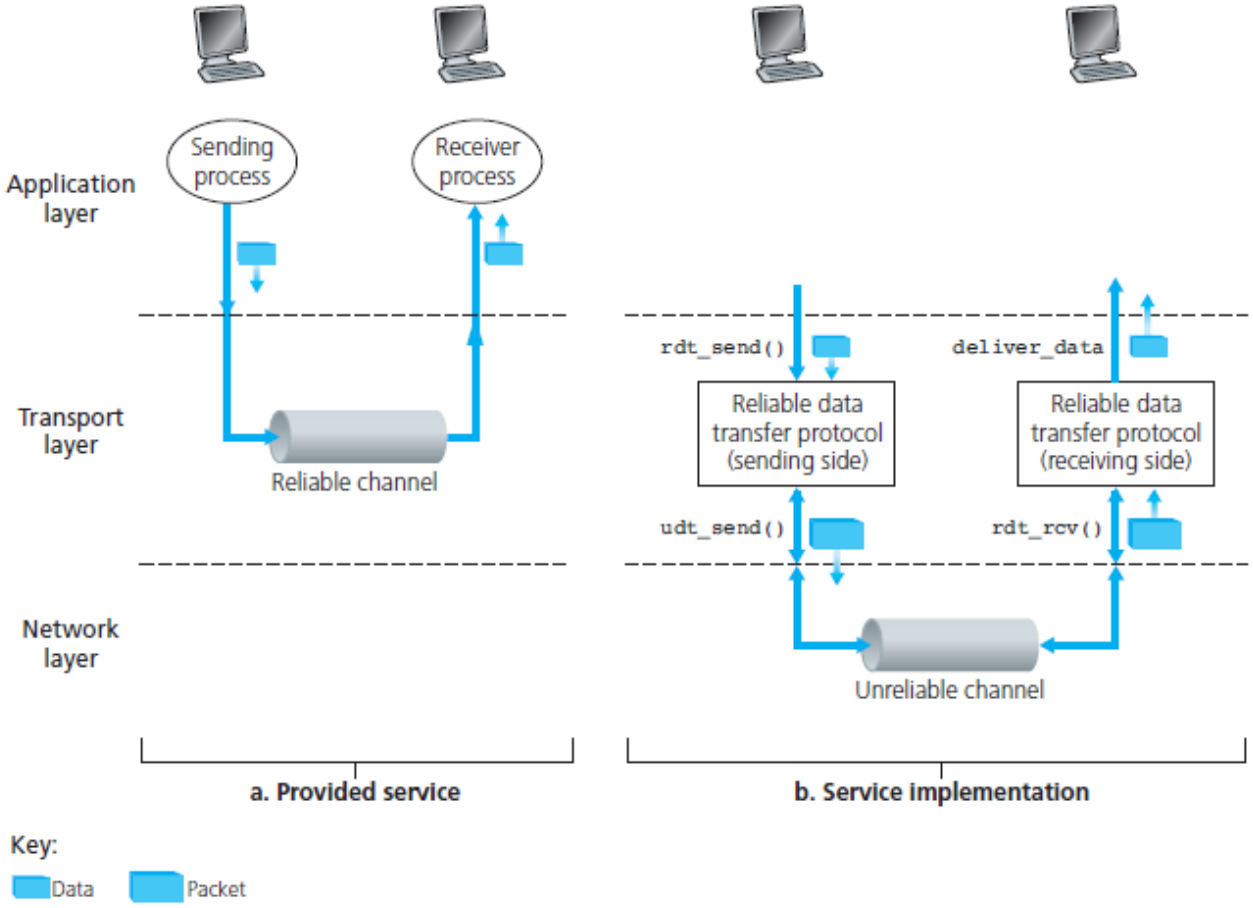
The sending side of the data transfer protocol will be invoked from above by a call to rdt_send().Here **rdt** stands for *reliable data transfer* protocol and **_send** indicates that the sending side of rdt is being called.

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 sending side of the channel.

When the rdt protocol wants to deliver data to the upper layer, it will do so by calling deliver_data().

Here **unidirectional data transfer is considered**, that is, data transfer from the sending to the receiving side.

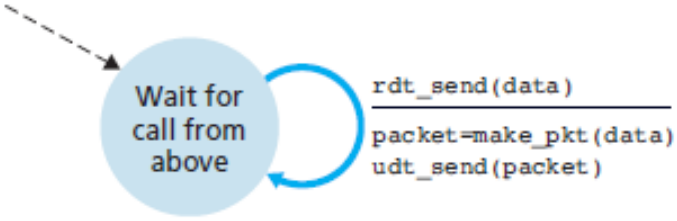


In addition to exchanging packets containing the data to be transferred, the sending and receiving sides of rdt will also need to exchange control packets back and forth.

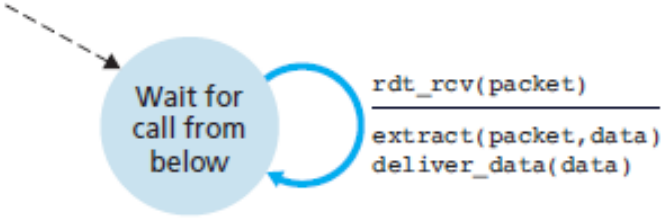
Both the send and receive sides of rdt send packets to the other side by a call to **udt_send( )**.

3.4.1 **Building a Reliable Data Transfer Protocol**

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



a. rdt1.0: sending side



b. rdt1.0: receiving side

Consider the simplest case, in which the underlying channel is completely reliable. The protocol is called **rdt1.0**. The **finite-state machine (FSM)** definitions for the rdt1.0 sender and receiver are shown in Figure above.

The FSM in Figure (a) defines the operation of the sender, while the FSM in Figure (b) defines the operation of the receiver.

There are *separate* FSMs for the sender and for the receiver. The sender and receiver FSMs in Figure each have just one state.

The arrows in the FSM description indicate the transition of the protocol from one state to another.

Since each FSM in Figure has just one state, a transition is necessarily from the one state back to itself;

The event causing the transition is shown above the horizontal line labeling the **transition**, and the **actions** taken when the event occurs are shown below the horizontal line.

When no action is taken on an event, or no event occurs and an action is taken, the symbol '**_**' below or above the horizontal is used . It explicitly denotes the lack of an action or event.

The initial state of the FSM is indicated by the dashed arrow.

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.

The rdt_send(data) event would result from a procedure call by the upper-layer application.

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)).

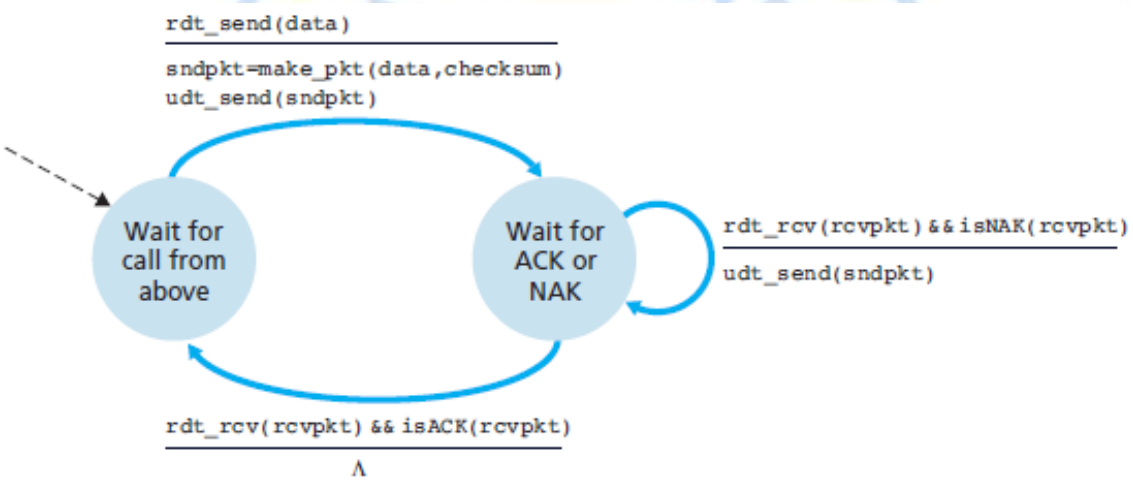
The rdt_rcv(packet) event would result from a procedure call (for example, to rdt_rcv()) from the lower layer protocol.

The packet flows 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

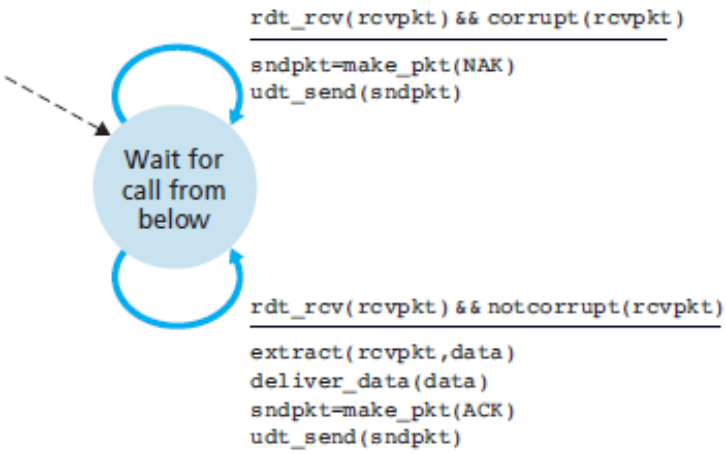
The receiver is able to receive data as fast as the sender happens to send data. Thus, there is no need for the sender to slow down its sending rate.

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.



a. rdt2.0: sending side



b. rdt2.0: receiving side

Such bit errors occur in the physical components of a packet is transmitted, propagates, or is buffered.

Assume that all transmitted packets are received (although their bits may be corrupted) in the order in which they were sent.

This message-dictation protocol uses both **positive acknowledgments** and **negative acknowledgments** . 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.

Reliable data transfer protocols based on 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***. A mechanism is needed to allow the receiver to detect bit errors if occurred. UDP uses the Internet checksum field for this purpose.

Error detection techniques allow the receiver to detect and possibly correct packet bit errors. These techniques require that extra bits(checksum) be sent from the sender to the receiver; these bits will be gathered into the packet checksum field of the rdt2.0 data packet.

• ***Receiver feedback***. Since the sender and receiver are typically executing on different end systems, it requires for the receiver to provide explicit feedback to the sender. The positive (ACK) and negative (NAK) acknowledgment replies in the message are examples of such feedback. Rdt2.0 protocol sends ACK and NAK packets back from the receiver to the sender. ACK packets is just one bit long; for example, a 0 value could indicate a NAK and a value of 1 could indicate an ACK.

• ***Retransmission***. A packet that is received in error at the receiver will be retransmitted by the sender. Figure shows the FSM representation of rdt2.0, a data transfer protocol employing error detection, positive acknowledgments, and negative acknowledgments.

The sender side of rdt2.0 has two states:

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) in Figure corresponds to this event), the sender knows that the most recently transmitted packet has been received correctly .

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 cannot 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 rdt2.0 is known as **stop-and-wait** protocol.

The **receiver-side** FSM for rdt2.0 still has a single state. On packet arrival, the receiver replies with either an ACK or a NAK, depending on whether or not the received packet is corrupted.

Notation **rdt_rcv(rcvpkt) && corrupt(rcvpkt)** corresponds to the event in which a packet is received and is found to be in error.

ACK or NAK packet could be corrupted and solution could be adding checksum bits to ACK/NAK packets in order to detect such errors.

The difficulty here is that if an ACK or NAK is corrupted, the sender has no way of knowing whether or not the receiver has correctly received the last piece of transmitted data.

Consider three possibilities for handling **corrupted ACKs or NAKs**:

• For the first possibility, if Sender receives garbled ACK/NAK , he creates a new packet – asking the receiver to resend ACK/NAK. But creation of such new packet leads to issues.

• A second alternative is to add enough checksum bits to allow the sender not only to detect, but also to recover from, bit errors. This solves the immediate problem for a channel that can corrupt packets but not lose them.

• A third approach is for the sender simply to resend the current data packet when it receives a garbled ACK or NAK packet. This approach, introduces **duplicate packets** into the sender-to-receiver channel. The fundamental difficulty with duplicate packets is that the receiver doesn't know whether the ACK or NAK it last sent was received correctly at the sender.

Solution :A simple solution to this new 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.

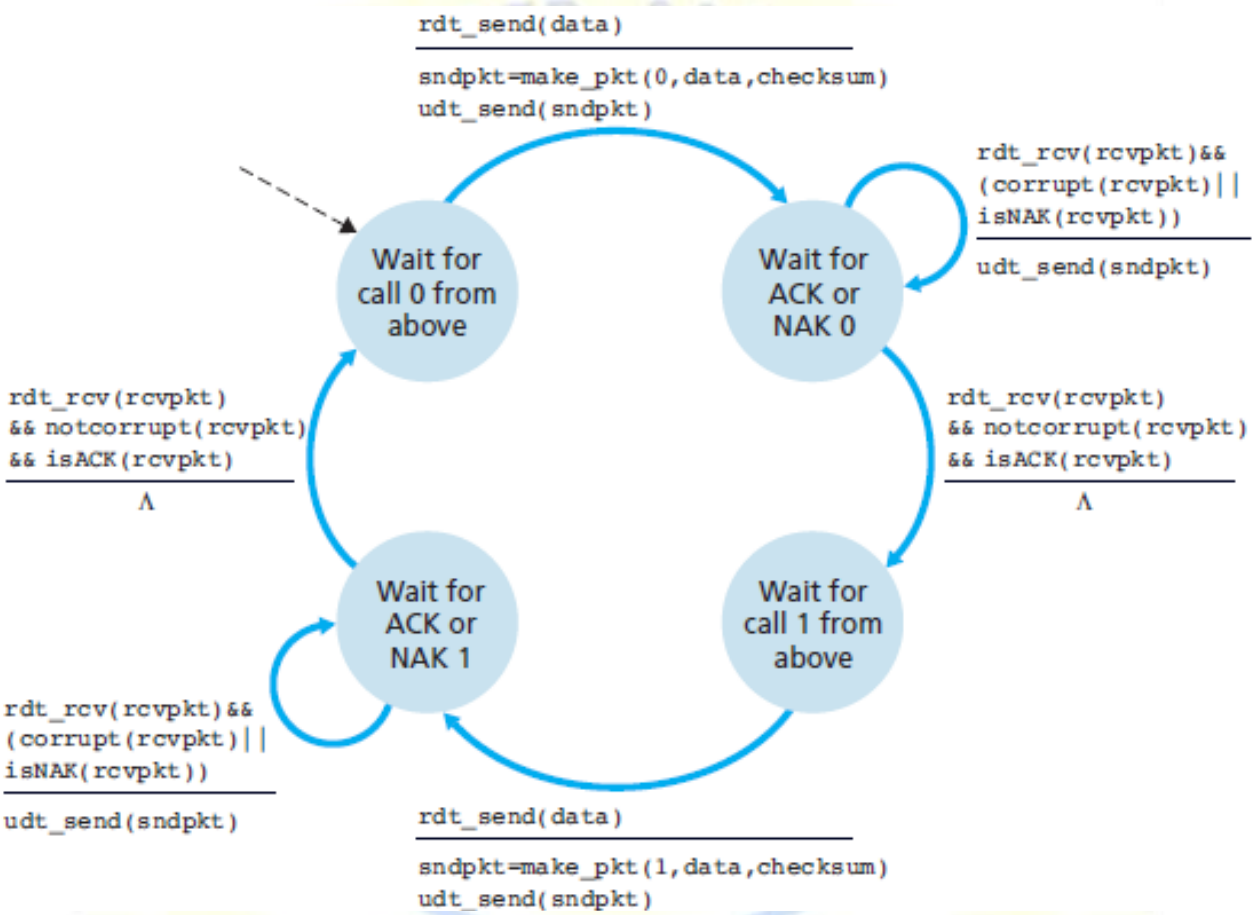
For this simple case of a stop-andwait protocol, a 1-bit sequence number will suffice, since it will allow the receiver to know whether the sender is resending the previously transmitted

packet (the sequence number of the received packet has the same sequence number as the most recently received packet) or a new packet.

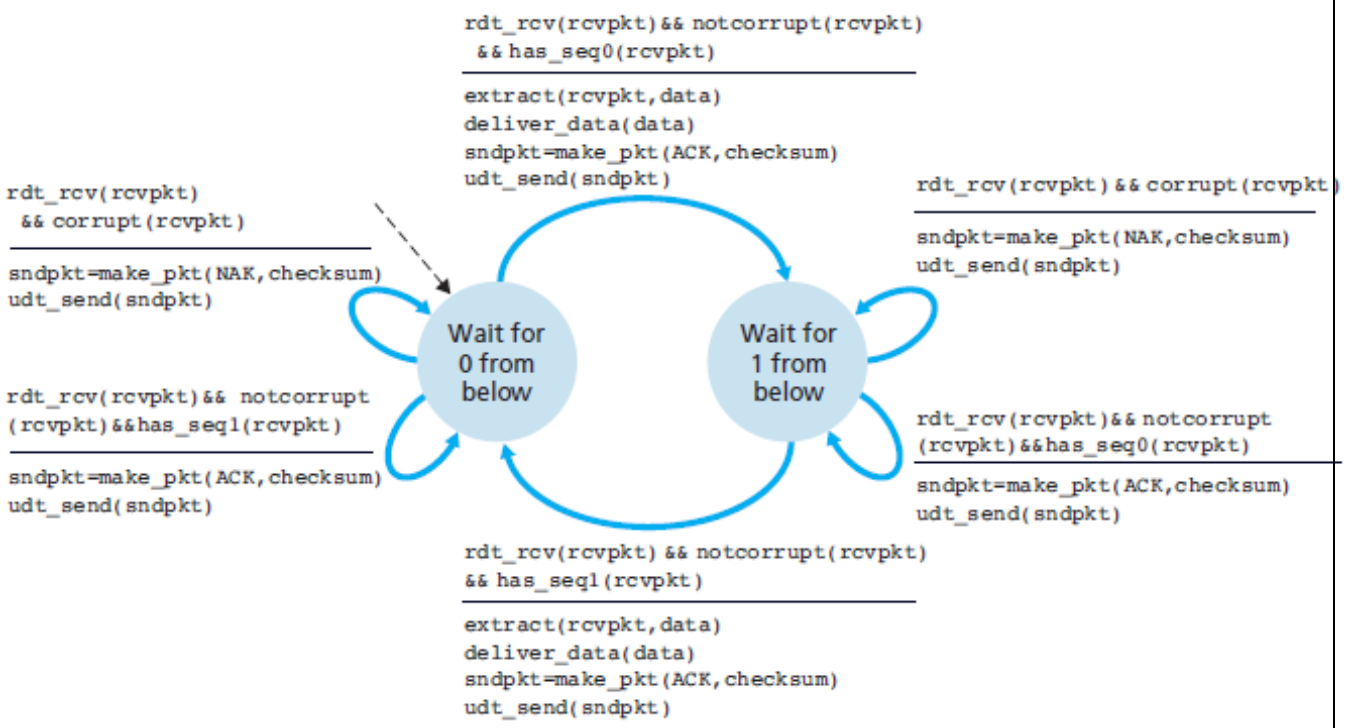
Since it is assumed that channel does not lose packets, ACK and NAK packets do not themselves need to indicate the sequence number of the packet they are acknowledging.

The sender knows that a received ACK or NAK packet (whether garbled or not) was generated in response to its most recently transmitted data packet.

RDT 2.1 sender(a) :



RDT 2.1 receiver(b) :



Figures (a) and (b) shows the FSM description for **rdt2.1.**

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.

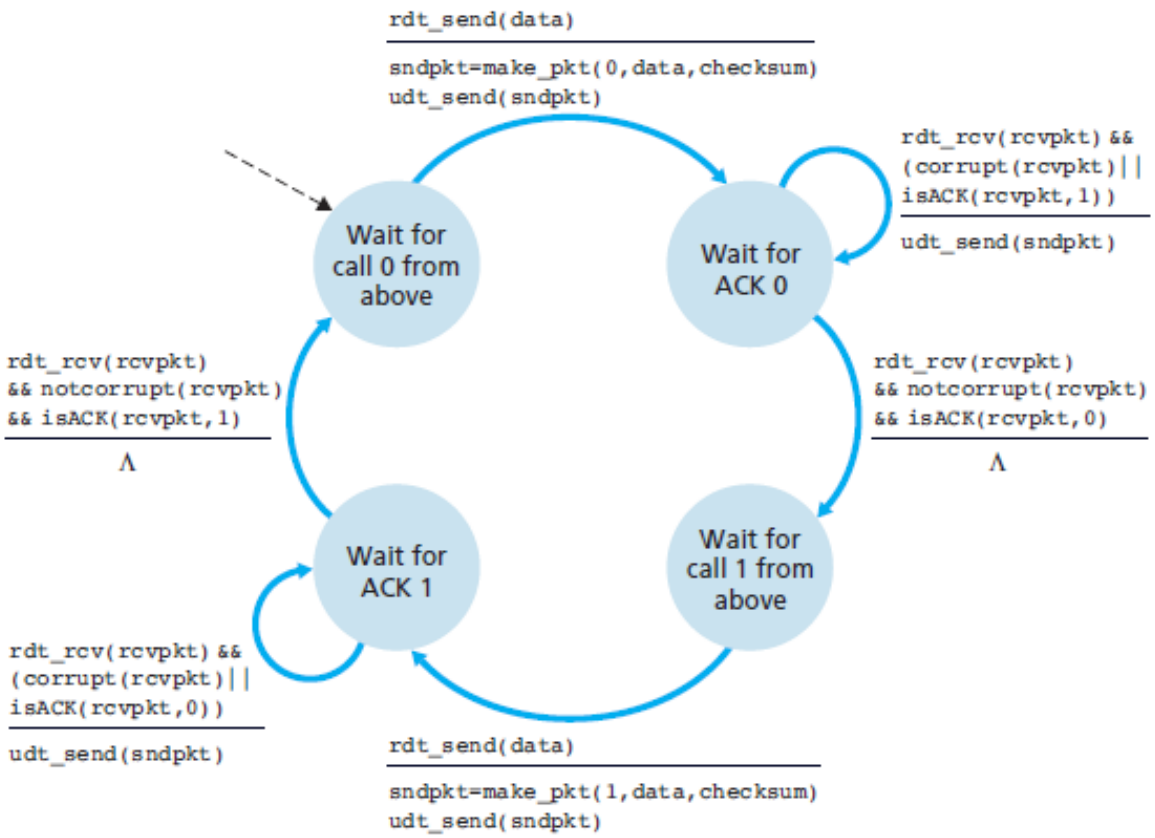
When a corrupted packet is received, the receiver sends a negative acknowledgment.

The same effect as a NAK could be accomplished if, instead of sending a NAK, we send an ACK for the last correctly received packet.

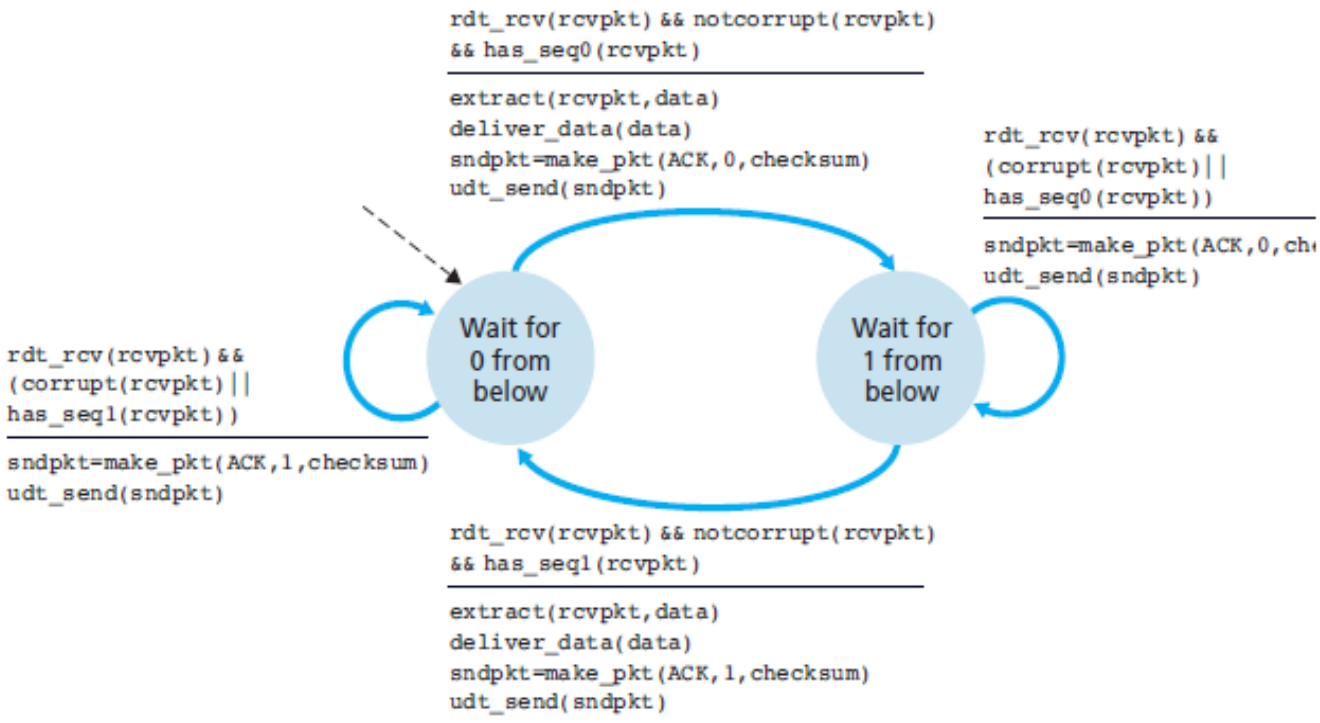
A sender that receives two ACKs for the same packet (that is, receives **duplicate ACKs**) knows that the receiver did not correctly receive the packet following the packet that is being ACKed twice.

RDT 2.2 sender :

NAK-free reliable data transfer protocol for a channel with bit errors is rdt2.2.



RDT 2.2 receiver :



The sequence number is included in the packet i.e being acknowledged by an ACK message (this is done by including the ACK,0 or ACK1 argument in make_pkt() in the receiver FSM)

The sender must now check the sequence number of the packet being acknowledged by a received ACK message (this is done by including the 0 or 1 argument in **isACK()** in the sender FSM).

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

The use of checksumming, sequence numbers, ACK packets, and retransmissions—are solutions for corrupted data , out of order data , lost packet problems.

Detecting and recovering from lost packets is the responsibility of sender.

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.

Solution is setting a timer by the sender.

The sender must clearly wait at least as long as a round-trip delay between the sender and receiver plus amount of time is needed to process a packet at the receiver.

If an ACK is not received within this timer, the packet is retransmitted. If a packet experiences a large delay, the sender may retransmit the packet even though neither the data packet nor its ACK have been lost.

This introduces the possibility of **duplicate data packets** in the sender-to-receiver channel. Sequence numbers can be used to handle the case of duplicate packets.

The sender does not know whether a data packet was lost, an ACK was lost, or if the packet or ACK was simply overly delayed. Retransmission is the solution for all these problems.

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)

(3) stop the timer.

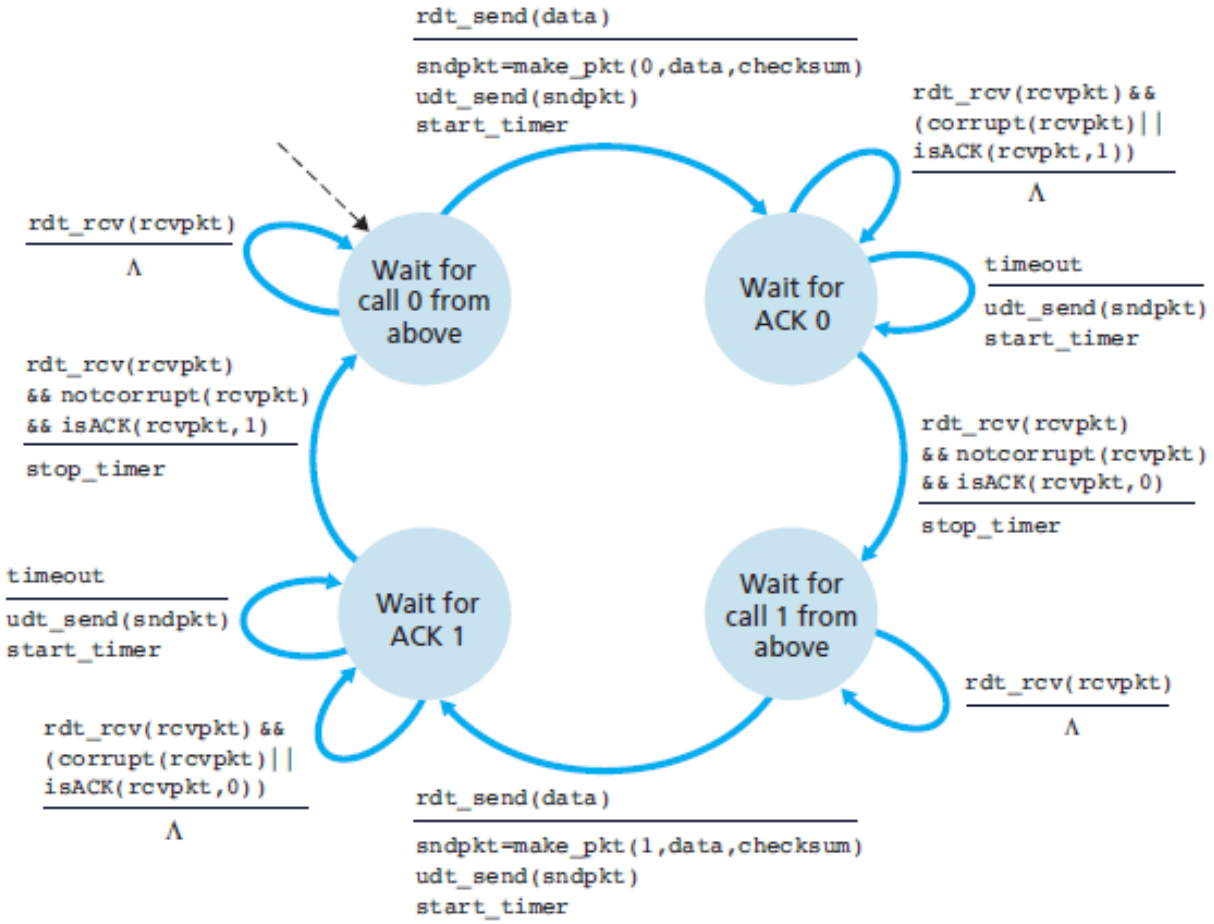
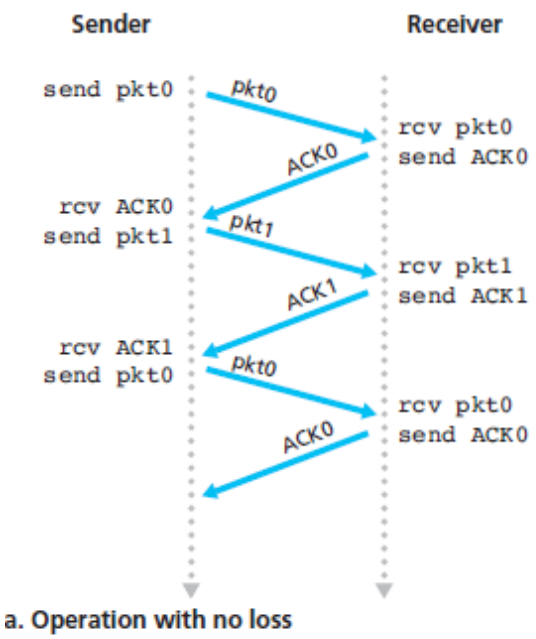
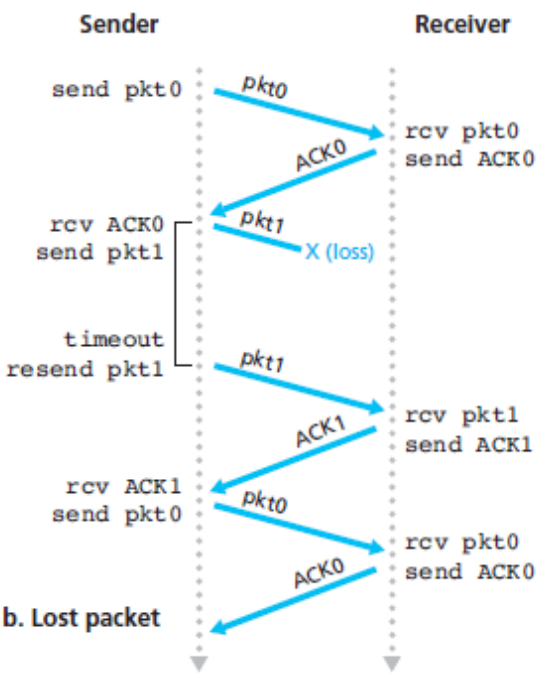


Figure shows the sender FSM for rdt3.0, a protocol that reliably transfers data over a channel that can corrupt or lose packets.

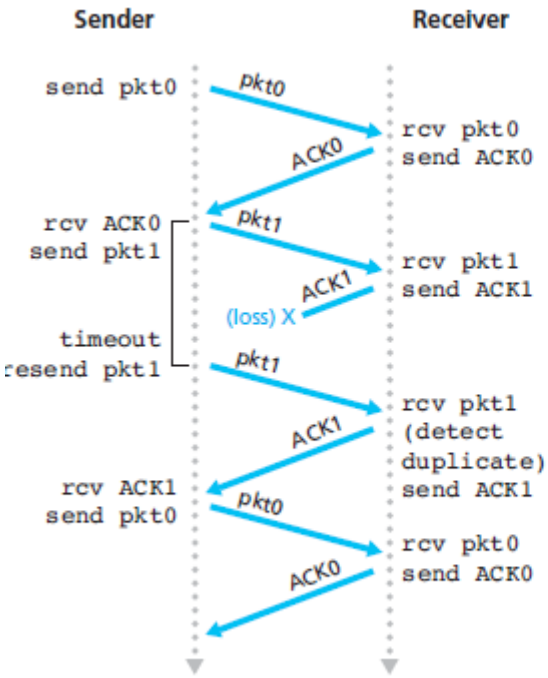
In Figure below, time moves forward from the top of the diagram toward the bottom of the diagram; Receive time for a packet is necessarily later than the send time for a packet as a result of transmission and propagation delays.



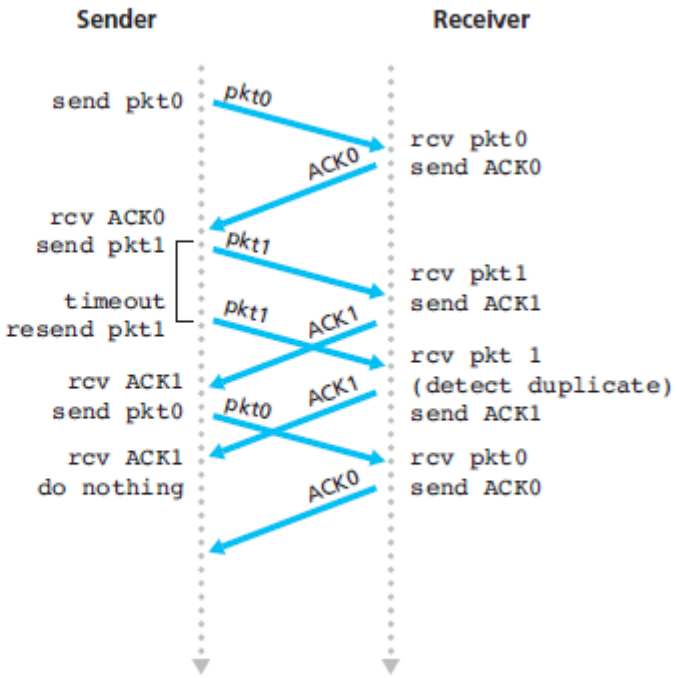
a. Operation with no loss



b. Lost packet



c. Lost ACK



d. Premature timeout

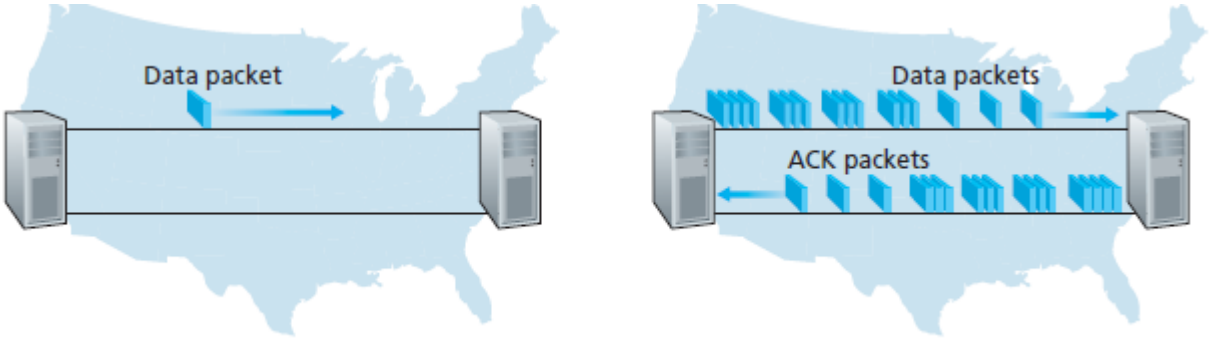
In Figures (b)–(d), the send-side brackets indicate the times at which a timer is set and later times out.

Because packet sequence numbers alternate between 0 and 1, protocol rdt3.0 is also known as the **alternating-bit protocol**.

3.4.2 **Pipelined Reliable Data Transfer Protocols**

Consider an idealized case of two hosts, one located on the West Coast of the United States

and the other located on the East Coast, as shown in Figure below.



a. A stop-and-wait protocol in operation

b. A pipelined protocol in operation

The round-trip propagation delay between these two end systems, RTT, is approximately 30 milliseconds. RTT=30ms

Suppose that they are connected by a channel with a transmission rate, *R*, of 1 Gbps (10^9 bits per second). R=1Gbps

With a packet size, *L*, of 1,000 bytes (8,000 bits) per packet, including both header fields and data. L=8000bitsps

The time needed to actually transmit the packet into the 1 Gbps link is

dtrans =L/R = 8000 bits/packet / 10^9 bits/sec = 8 microseconds

Figure (a) shows that with stop-and-wait protocol, if the sender begins sending the packet at ,

t = 0, then at $t = L/R$ = 8 microseconds;

The packet then makes its 15-msec journey from sender to receiver,RTT/2=15ms.

The last bit of the packet is emerging at the receiver at , t = RTT/2 + L/R =15.008 msec.

The ACK emerges back at the sender at t = RTT + L/R = 30.008 msec.(RTT= 15+15=30ms). 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

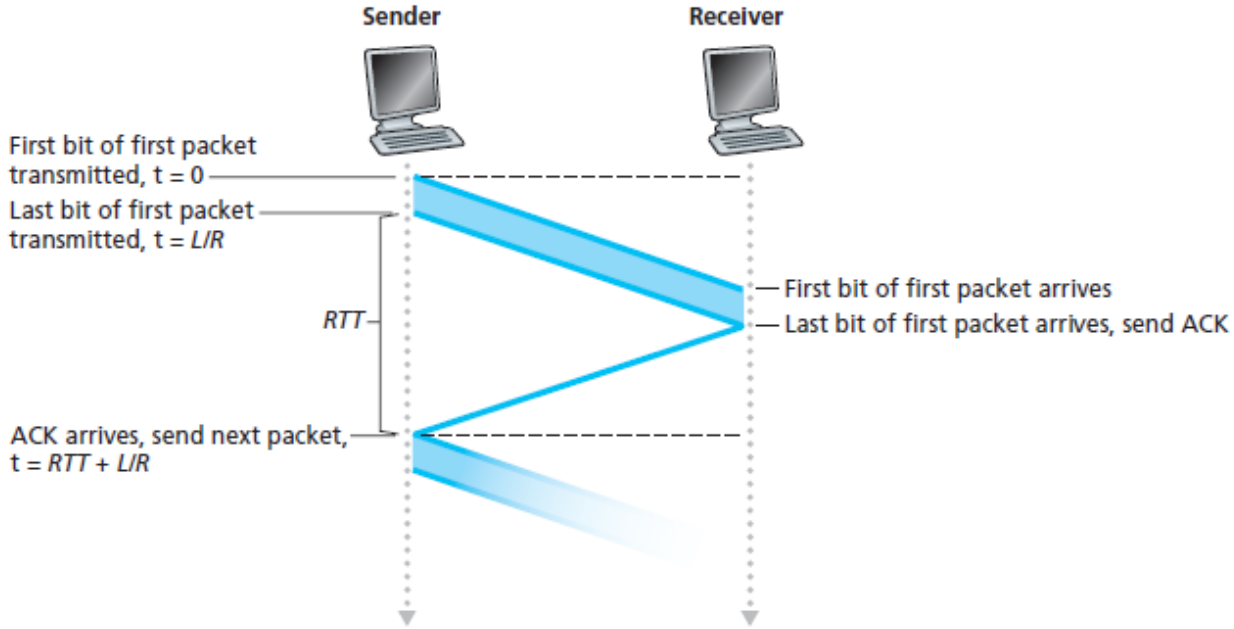
is actually busy sending bits into the channel, the analysis in below Figure (a) shows that the stop-and-wait protocol has sender utilization, *U*sender, of

*U*sender = L/R / ($RTT + L/R$) = .008 / 30.008 = 0.00027.

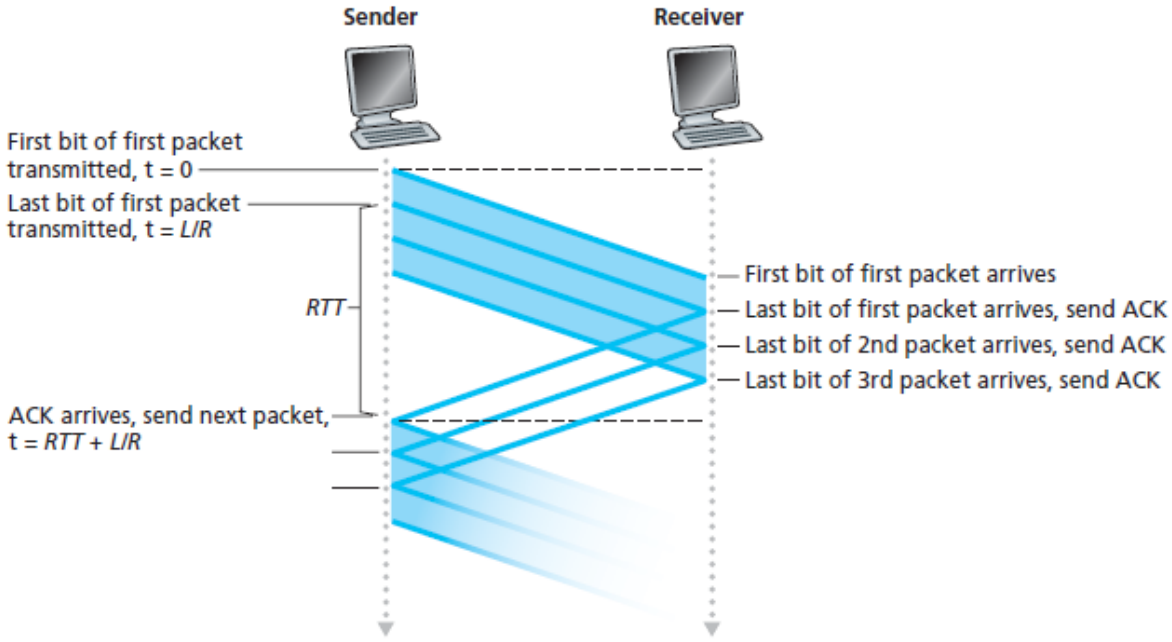
That is, the sender was busy only 2.7 hundredths of one percent of the time!

The solution to this performance problem is: Rather than operate in a stop-and-wait manner, the sender is allowed to send multiple packets without waiting for acknowledgments, as illustrated in below Figure (b).

Figure (b) shows that if the sender is allowed to transmit three packets before having to wait for acknowledgments, the utilization of the sender is essentially tripled.



a. Stop-and-wait operation



b. Pipelined operation

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. 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.

• 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: **Go-Back-N** and **selective repeat**.

3.4.3 **Go-Back-N (GBN)**

In a **Go-Back-N (GBN) protocol**, the sender is allowed to transmit multiple packets without waiting for an acknowledgment, but is constrained to have no more than some maximum allowable number, *N*, of unacknowledged packets in the pipeline.

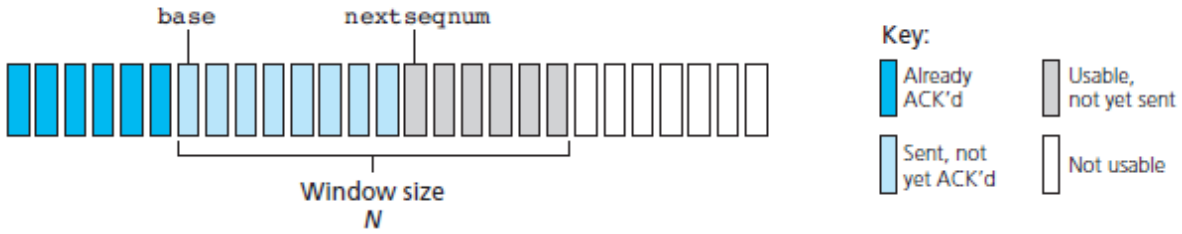


Figure above shows the sender's view of the range of sequence numbers in a GBN protocol.

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).

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 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.

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 as a **sliding-window protocol**.

A packet's sequence number is carried in a fixed-length field in the packet header. If *k* is the number of bits in the packet sequence number field, the range of sequence numbers is thus $[0,2^k - 1]$.

The sequence number space can be thought of as a ring of size 2^k, where sequence number $2^k - 1$ is immediately followed by sequence number 0.

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.

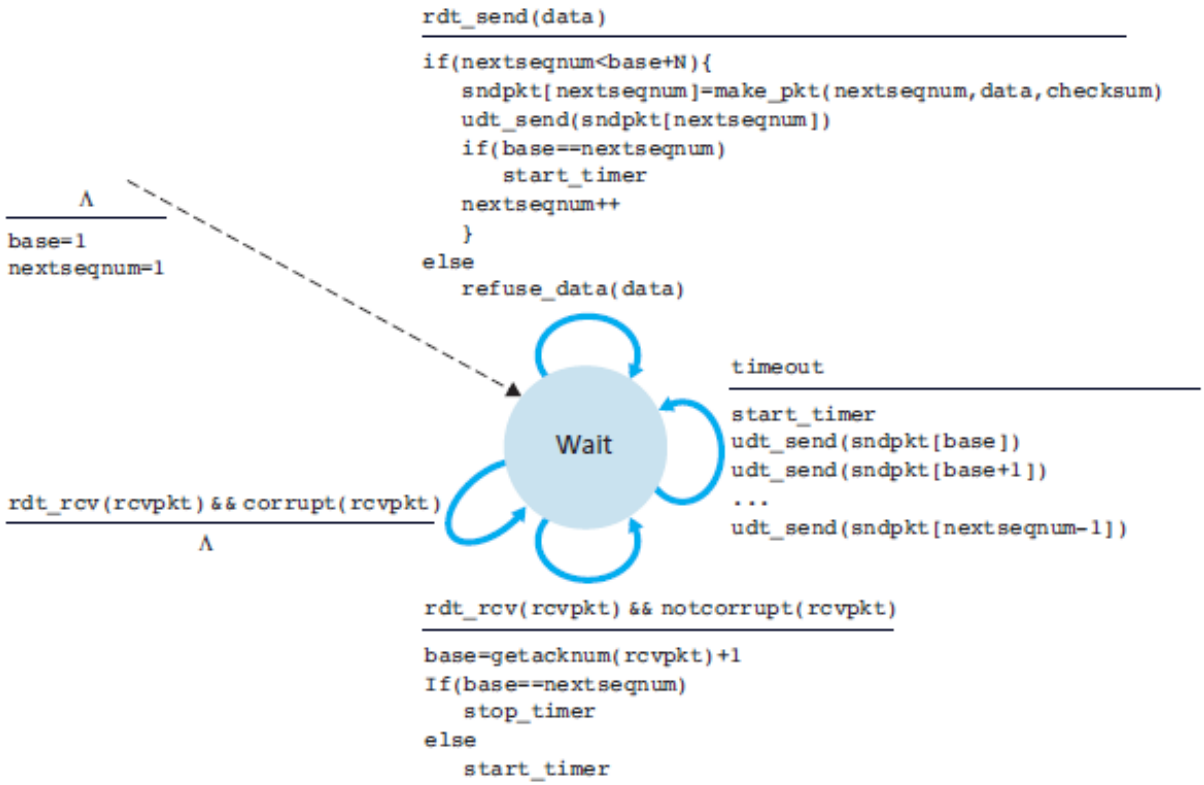
If the window is not full, a packet is created and sent, and variables are appropriately updated.

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 have to try again.

• ***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***. *A* timer will 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.



Sender in above Figure uses only a single timer, which can be thought of as a timer for the oldest transmitted but not yet acknowledged packet.

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 action:** 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.

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, cumulative acknowledgments is used for GBN.

In GBN protocol, the **receiver discards** out-of-order packets because the receiver must deliver data **in order** to the upper layer.

Suppose the 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*.

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.

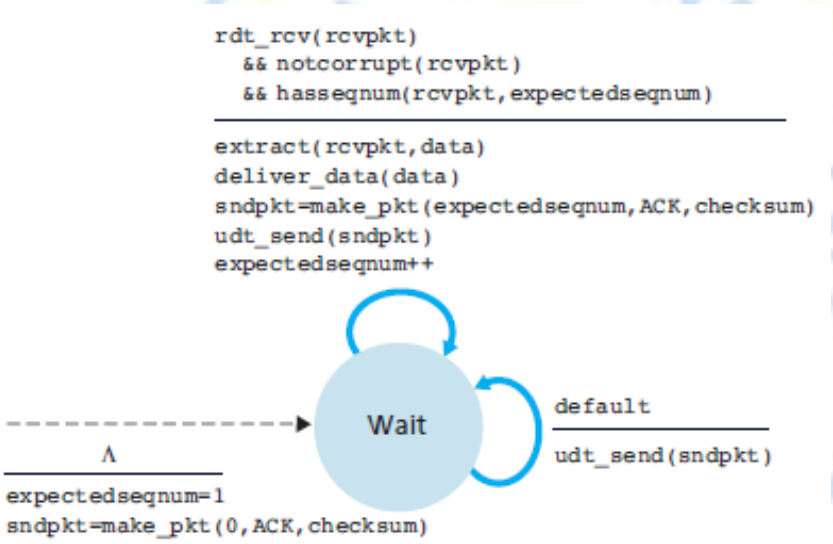
Thus, the receiver can simply discard packet *n* + 1.

The advantage of this approach is the receiver need not buffer *any* out-of-order packets.

Thus, while the sender must maintain the upper and lower bounds of its window and the position of nextseqnum within this window.

The only piece of information the receiver need maintain is the sequence number of the next in-order packet.

This value is held in the variable expected seqnum, shown in the receiver FSM in Figure below.



The disadvantage of discarding a correctly received packet is that the subsequent retransmission of that packet might be lost or garbled and thus even more retransmissions would be required.

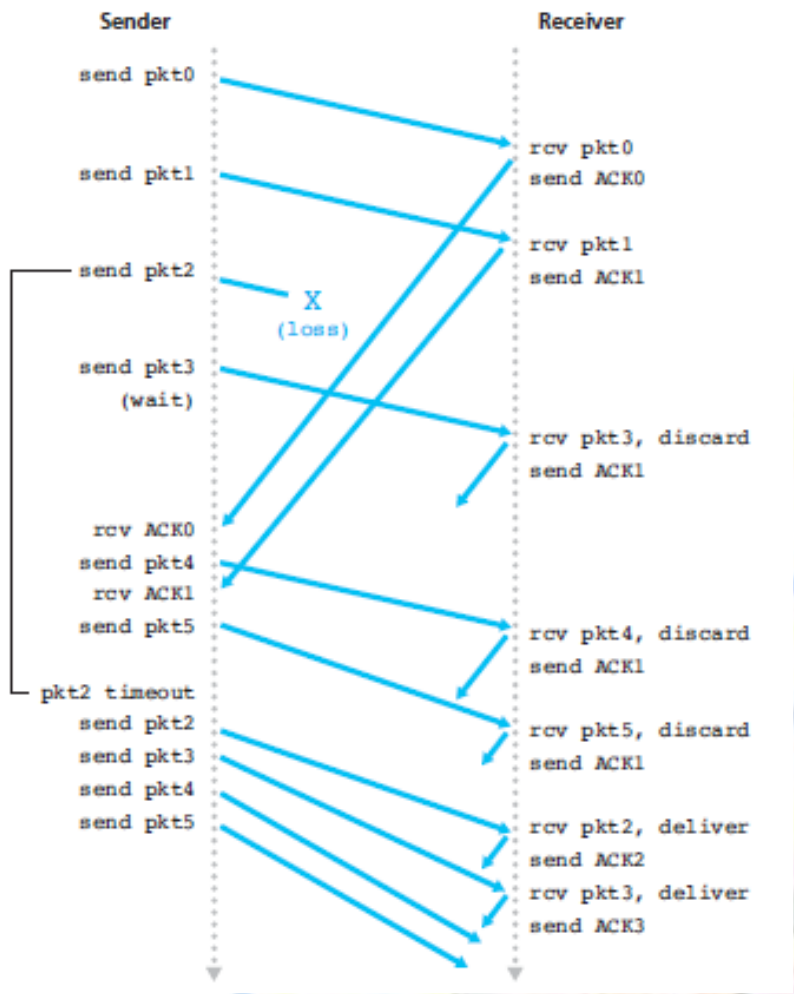


Figure above shows the operation of the GBN protocol for the case of a window size of four packets.

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.

In the sender, the events would be :

(1) a call from the upper-layer entity to invoke rdt_send(),

(2) a timer interrupt,

(3) a call from the lower layer to invoke rdt_rcv() when a packet arrives.

3.4.4 **Selective Repeat (SR)**

The GBN protocol allows the sender to potentially "fill the pipeline" with packets, thus avoiding the channel utilization problems.

GBN has 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.

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 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 sender will have already received ACKs for some of the packets in the window. Figure shows the SR sender's view of the sequence number space.

Figure details the various actions taken by the SR sender.

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.

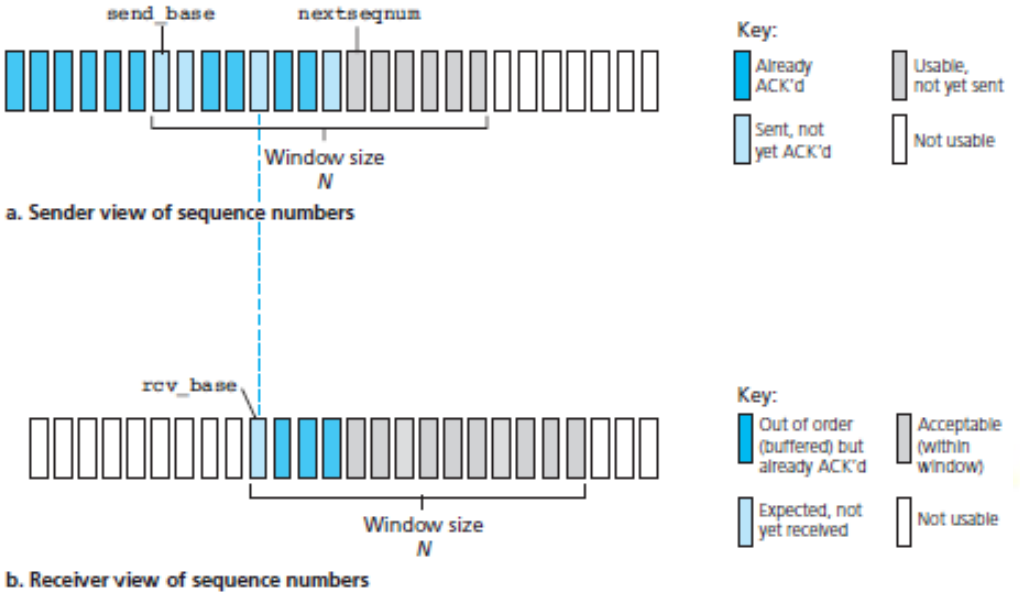
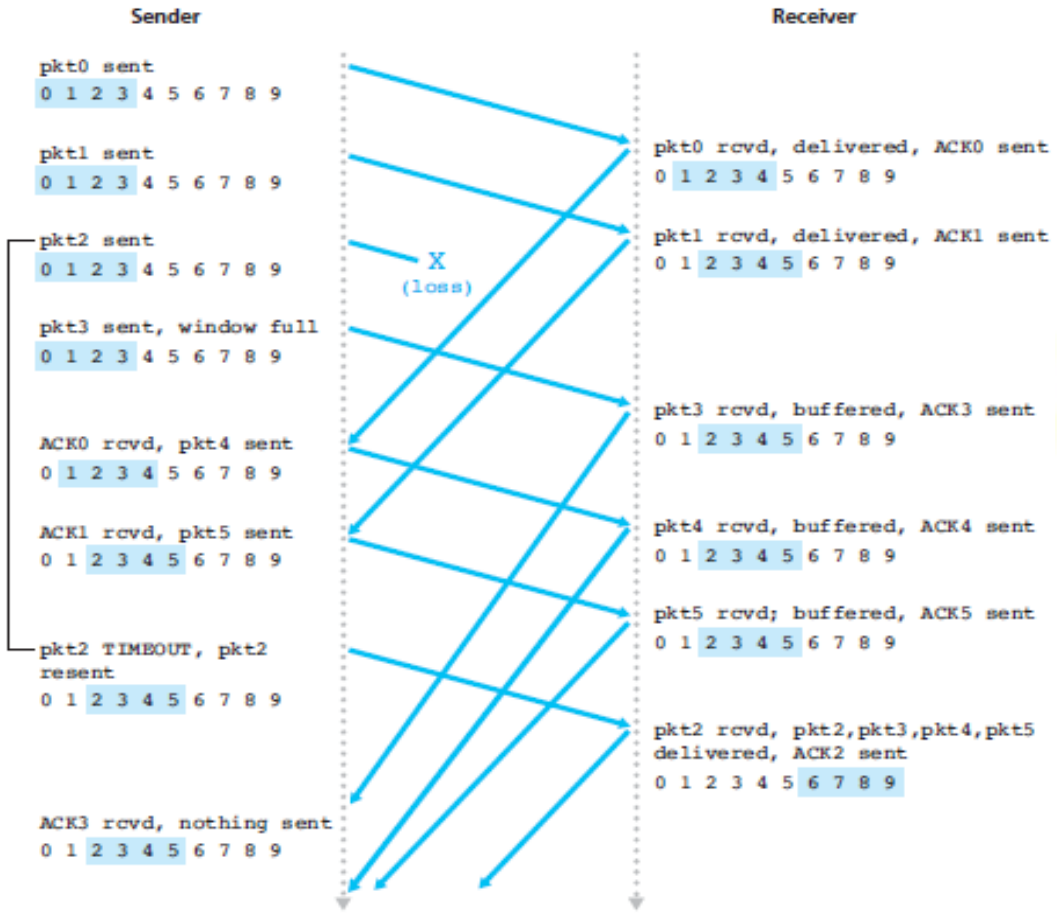


Figure above itemizes the various actions taken by the SR receiver.

In Step 2 in Figure above, the receiver reacknowledges (rather than ignores) already received packets with certain sequence numbers *below* the current window base.



Above Figure shows an example of SR operation in the presence of lost packets.

Here, the receiver initially buffers packets 3, 4, and 5, and delivers them together with packet 2 to the upper layer when packet 2 is finally received.

For example, if there is no ACK for packet send_base propagating from the receiver to the sender, the sender will eventually retransmit packet send_base, even though it is clear that the receiver has already received.

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 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 :* 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) 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 above Figure.

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. that packet. If the receiver were not to acknowledge this packet, the sender's window would never move forward.

The sender and receiver will not always have an identical view of what has been received correctly and what has not. For SR protocols, this means that the sender and receiver windows will not always coincide.

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.

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.

Consider two scenarios :

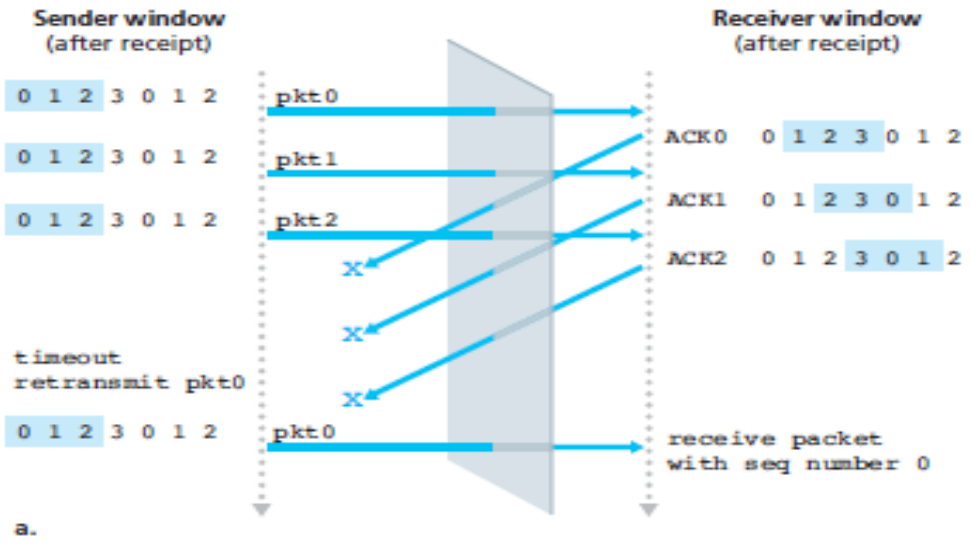
In the first scenario, shown in Figure (a) below, the ACKs for the first three packets are lost and the sender retransmits these packets. The receiver thus next receives a packet with sequence number 0—a copy of the first packet sent.

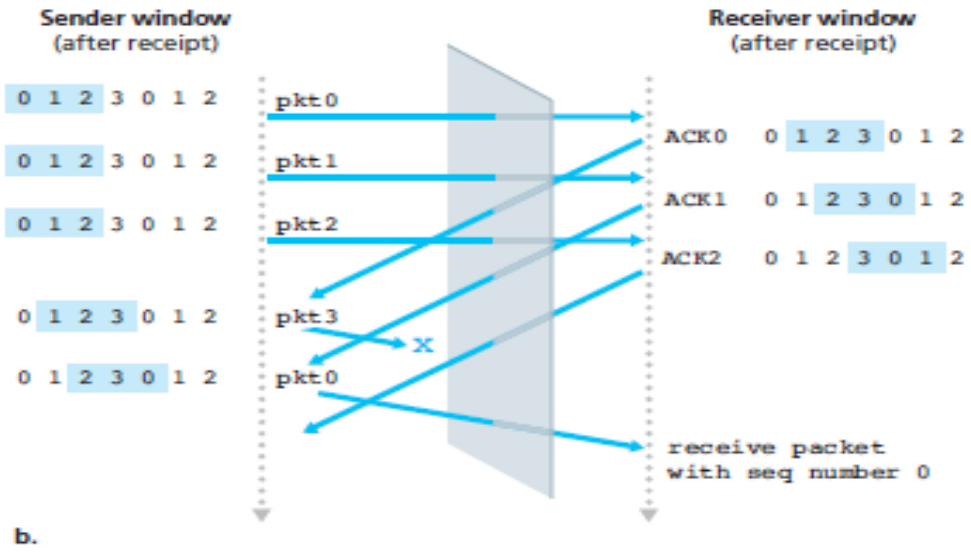
In the second scenario, shown in Figure (b) below, the ACKs for the first three packets are all delivered correctly. The sender thus moves its window forward and sends the fourth, fifth, and sixth packets, with sequence numbers 3, 0, and 1, respectively.

The packet with sequence number 3 is lost, but the packet with sequence number 0 arrives—a packet containing *new* data.

Consider the receiver's viewpoint in Figure below, since the receiver cannot "see" the actions taken by the sender. All the receiver observes is the sequence of messages it receives from the channel and sends into the channel.

The two scenarios in Figure are *identical.*





There is no way of distinguishing the retransmission of the first packet from an original transmission of the fifth packet.

Solution is the window size must be less than or equal to half the size of the sequence number space for SR protocols.

Table below summarizes these mechanisms.

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). • When a packet has been received by the receiver • but the receiver-to-sender ACK has been lost. When 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 send 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.

Table : Summary of reliable data transfer mechanisms and their use

3.5 Connection Oriented TCP :

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.

TCP connection establishment, both sides of the connection will initialize many TCP state variables associated with the TCP connection.

TCP protocol runs only in the end systems and not in the intermediate network elements (routers and link-layer switches), the intermediate network elements do not maintain TCP connection state.

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. So-called **"multicasting"—**the transfer of data from one sender to many receivers in a single send operation—is not possible with TCP.

Python client program command :

clientSocket.connect((serverName,serverPort))

where **serverName** is the name of the server and **serverPort** identifies the process on the server. TCP in the client then proceeds to establish a TCP connection with TCP in the server.

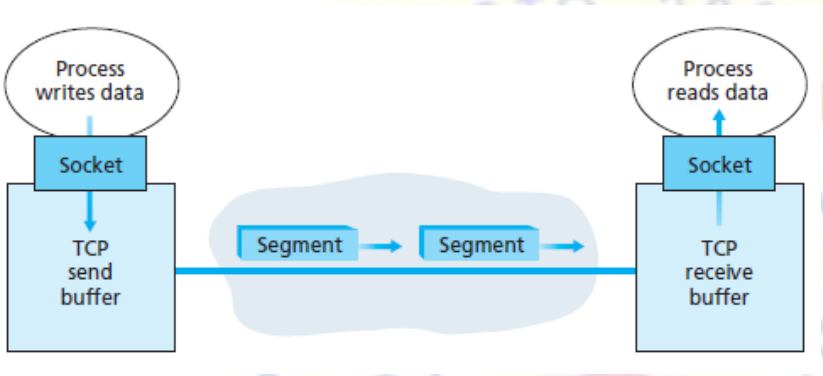
Client first sends a special TCP segment; the server responds with a second special TCP segment and finally the client responds again with a third special segment.

The first two segments carry no payload, that is, no application-layer data; the third of these segments may carry a payload. Because three segments are sent between the two hosts, this connection- establishment procedure is often referred to as a **three-way handshake**.

Once a TCP connection is established, the two application processes can send data to each other. 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 socket, the data is in the hands of TCP running in the client. As Figure shows, 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 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 (when encapsulated in an IP datagram) plus the TCP/IP header length (typically 40 bytes) 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, as shown in Figure above.

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.

Thus, TCP connection consists of buffers, variables and a socket connection to a process in one host, and another set of buffers, variables and a socket connection to a process in another host.

TCP Segment Structure :

The TCP segment consists of header fields and a data field. The data field contains a chunk of application data. 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.

Interactive applications , often transmit data chunks that are smaller than the MSS;

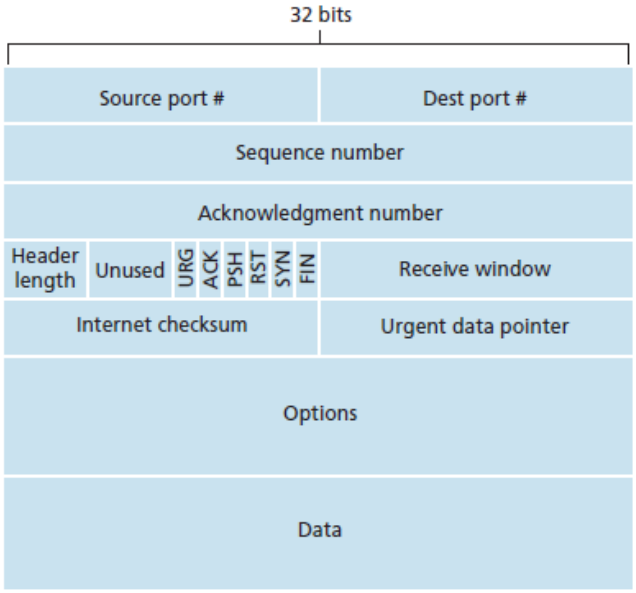


Figure above shows the structure of the TCP segment. The header includes **source and destination port numbers**, which are used for multiplexing / demultiplexing data from/to upper-layer applications.

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.

• 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.

• 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.

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.

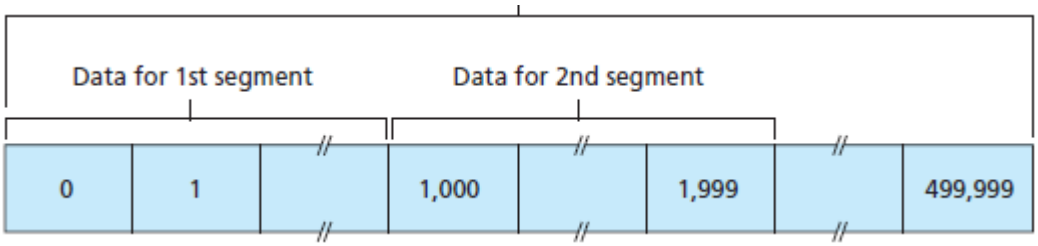
Sequence Numbers and Acknowledgment Numbers

Two of the most important fields in the TCP segment header are the sequence number field and the acknowledgment number field. These fields are a critical part of TCP's reliable data transfer service.

TCP views data as an unstructured, but ordered, stream of bytes. 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.

Example. Suppose 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.



As shown in Figure above, 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.

Consider acknowledgment numbers.

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).

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 aboutto send a segment to Host B. Host A is waiting for byte 536 and all the subsequentbytes in Host B's data stream. So Host A puts 536 in the acknowledgment numberfield of the segment it sends to B.

Suppose 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**.

Host A received the third segment (bytes 900 through 1,000) before receiving the second segment (bytes 536 through 899). Thus, the third segment arrived out of order. The issue is: Host A receives out-of-order segments in a TCP connection.

There are two choices:

(1) the receiver immediately discards out-of-order segments

(2) the receiver keeps the out-of-order bytes and waits for the missing bytes to fill in the gaps.

In Figure above, we assumed that the initial sequence number was zero. Both sides of a TCP connection randomly choose an initial sequence number.

Telnet: A Case Study for Sequence and Acknowledgment Numbers

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.

"Echo back" is used to ensure that characters seen by the Telnet user have already been received and processed at the remote site.

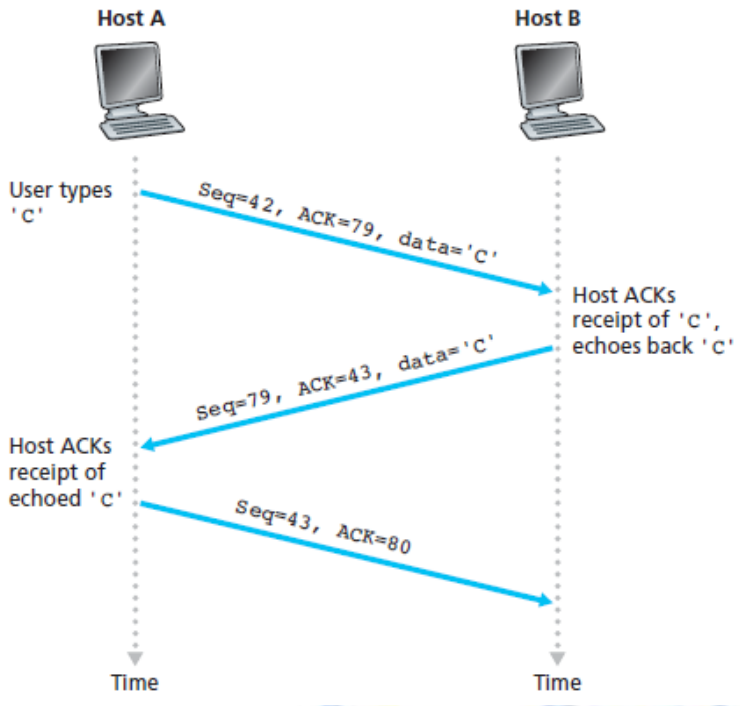
Each character thus traverses the network twice between the time the user hits the key and the time the character is displayed on the user's monitor.

Suppose the user types a single letter, 'C'.

As shown in Figure below, the starting sequence numbers are 42 and 79 for the client and server, respectively.

The sequence number of a segment is the sequence number of the first byte in the data field. Thus, the first segment sent from the client will have sequence number 42; the first segment sent from the server will have sequence number 79.

The acknowledgment number is the sequence number of the next byte of data that the host is waiting for.



After the TCP connection is established but before any data is sent, the client is waiting for byte 79 and the server is waiting for byte 42.

As shown in Figure above, three segments are sent.

- The first segment is sent from the client to the server, containing the 1-byte ASCII representation of the letter 'C' in its data field. This first segment also has 42 in its sequence number field.

  Because the client has not yet received any data from the server, this first segment will have 79 in its acknowledgment number field.

- The second segment is sent from the server to the client. It serves a dual purpose.

  First it provides an acknowledgment of the data the server has received. By putting 43 in the acknowledgment field, the server is telling the client that it has successfully received everything up through byte 42 and is now waiting for bytes 43 onward.

  The second purpose of this segment is to echo back the letter 'C.' Thus, the second segment has the ASCII representation of 'C' in its data field. This second segment has the sequence number 79, the initial sequence number of the server-to client data flow of this TCP connection, as this is the very first byte of data that the server is sending. This acknowledgment is said to be **piggybacked** on the server-to-client data segment.

- The third segment is sent from the client to the server. Its purpose is to acknowledge the data it has received from the server. This segment has an empty data field .

  The segment has 80 in the acknowledgment number field because the client has received the stream of bytes up through byte sequence number 79 and it is now waiting for bytes 80 .

Round-Trip Time Estimation and Timeout

TCP, uses a timeout/retransmit mechanism to recover from lost segments. The timeout should be larger than the connection's round-trip time (RTT), that is, the time from when a segment is sent until it is acknowledged.

Estimating the Round-Trip Time

TCP estimates the round-trip time between sender and receiver.

This is accomplished as follows:

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.

Also, TCP never computes a SampleRTT for a segment that has been retransmitted; it only measures SampleRTT for segments that have been transmitted once .

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. 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.

Upon obtaining a new SampleRTT, TCP updates EstimatedRTT according to the following formula:

EstimatedRTT = (1 – α) • EstimatedRTT + α • SampleRTT

The formula above is written in the form of a programming-language statement— the 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 (that is, 1/8)

in which case the formula above becomes:

EstimatedRTT = 0.875 • EstimatedRTT + 0.125 • SampleRTT

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.

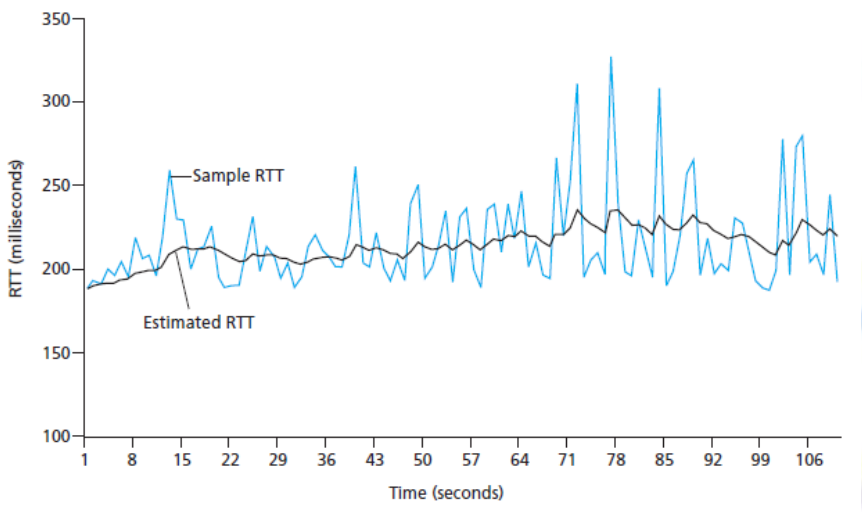


Figure shows the SampleRTT values and EstimatedRTT for a value of α = 1/8 for a TCP connection between say gaia.cs.umass.edu(site 1) to fantasia.eurecom.fr(site 2).

Clearly, the variations in the SampleRTT are smoothed out in the computation of the EstimatedRTT.

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

The RTT variation, DevRTT, as an estimate of how much SampleRTT typically deviates from EstimatedRTT:

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

DevRTT is an EWMA of the difference between SampleRTT and EstimatedRTT. If the SampleRTT values have little fluctuation, then DevRTT will be small; 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

Given values of EstimatedRTT and DevRTT, 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 segmentis 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 margin should be large when there is a lot of fluctuation in the SampleRTT values; it should be small when there is little fluctuation. The value of DevRTT should thus come into play here. All of these considerations are taken into account in TCP's method for determining the retransmission timeout interval:

TimeoutInterval = EstimatedRTT + 4 • DevRTT

When a timeout occurs, the value of TimeoutInterval is doubled to avoid a premature timeout occurring for a subsequent segment that will soon be acknowledged.

As soon as a segment is received and EstimatedRTT is updated, the TimeoutInterval is again computed using the formula above.

Reliable Data Transfer

The Internet's network-layer service (IP service) is unreliable. IP does not guarantee datagram delivery, does not guarantee in-order delivery of datagrams and does not guarantee the integrity of the data in the datagrams. With IP service, datagrams can overflow router

buffers and never reach their destination, datagrams can arrive out of order, and bits in the datagram can get corrupted (flipped from 0 to 1 and vice versa).

Transport-layer segments are carried across the network by IP datagrams, transport-layer segments can suffer from these problems as well.

TCP creates a **reliable data transfer service** on top of IP's unreliable besteffort 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.

The recommended TCP timer management procedures use only a *single* retransmission timer, even if there are multiple transmitted but not yet acknowledged segments.

Suppose that data is being sent in only one direction, from Host A to Host B, and that Host A is sending a large file.

Three major events related to data transmission and retransmission in the TCP sender:

- data received from application above;
- timer timeout;
- ACK receipt.

1.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.

Each segment includes a sequence number that is the byte-stream number of the first data byte in the segment.

If the timer is already not running for some other 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.

Simplified TCP sender :

/* Assume sender is not constrained by TCP flow or congestion control, that data from above is less than MSS in size, and that data transfer is in one direction only. */

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 */

2.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.

3.The third major event that must be handled by the TCP sender is the arrival of an acknowledgment segment (ACK) from the receiver .

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.

TCP uses cumulative acknowledgments, so that y acknowledges the receipt of all bytes before byte number y.

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

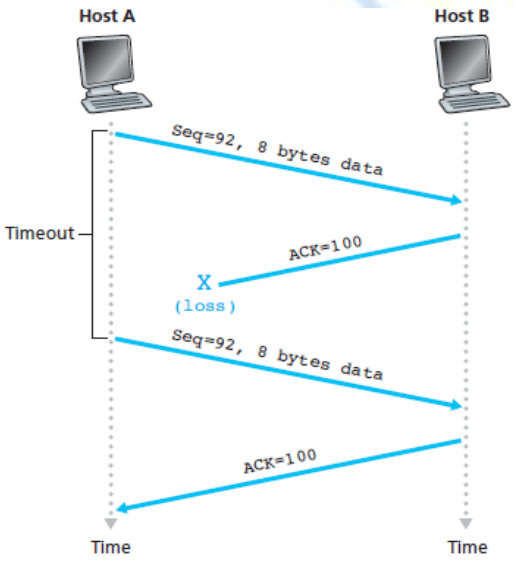
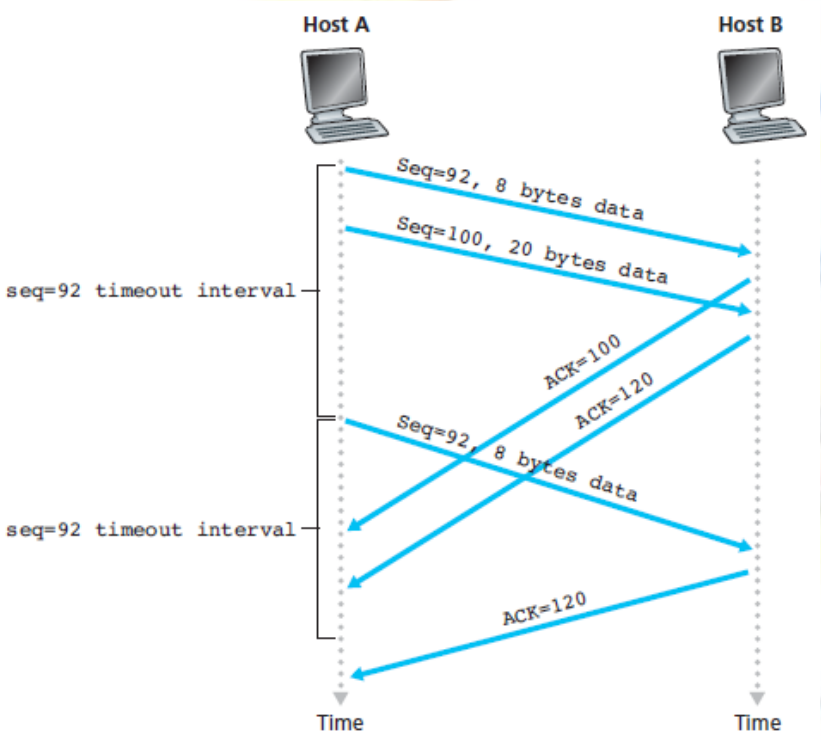


Figure above depicts the first scenario, in which Host A sends one segment to Host B. Suppose that this segment has sequence number 92 and contains 8 bytes of data. After sending this segment, Host A waits for a segment from B with acknowledgment number 100. Although the segment from A is received at B, the acknowledgment from B to A gets lost.

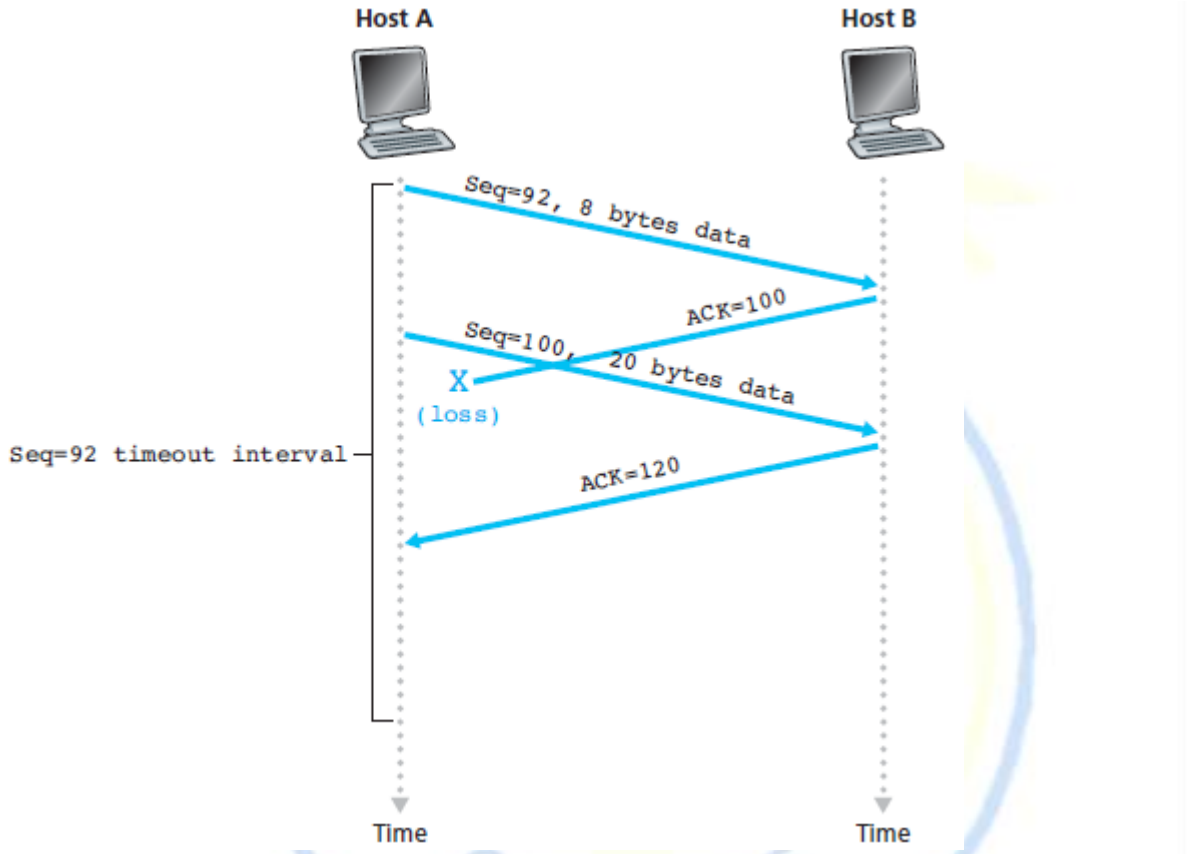
In this case, the timeout event occurs, and Host A retransmits the same segment. When Host B receives the retransmission, it observes from the sequence number that the segment contains data that has already been received. Thus, TCP in Host B will discard the bytes in the retransmitted segment.



In a second scenario, shown in Figure, 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.

In a third and final scenario, suppose 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. This scenario is illustrated in Figure below.



Doubling the Timeout Interval

The length of the timeout interval after a timer expiration. In this modification, whenever the timeout event occurs, TCP retransmits the not-yet acknowledged segment with the smallest sequence number, as described above. But 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 TimeoutInterval associated with the oldest not yet acknowledged segment is .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. Thus the intervals grow exponentially after each retransmission. However, whenever the timer is started after either of

the two other events (that is, data received from application above, and ACK received), the TimeoutInterval is derived from the most recent values of EstimatedRTT and DevRTT.

This modification provides a limited form of congestion control.

The timer expiration is most likely caused by congestion in the network, that is, too many packets arriving at one (or more) router queues in the path between the source and destination, causing packets to be dropped and/or long queuing delays. In times of congestion, if the sources continue to retransmit packets persistently, the congestion may get worse. Instead, TCP acts more politely, with each sender retransmitting after longer and longer intervals.

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.

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.

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.

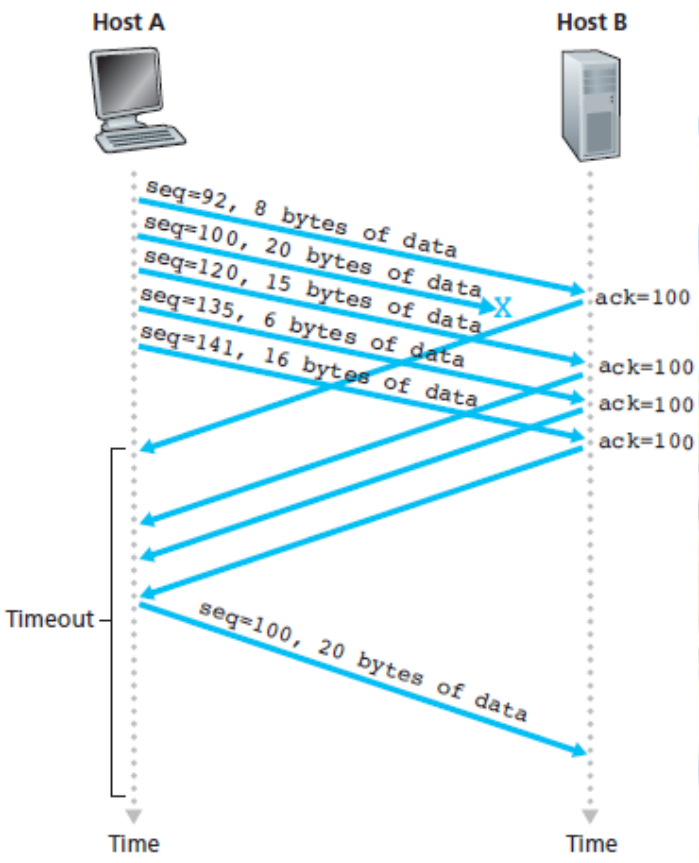
This gap could be the result of lost or reordered segments within the network.

Table :

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.

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.



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. This is shown in Figure above, where the second segment is lost, then retransmitted before its timer expires. For TCP with fast retransmit, the following code snippet replaces the ACK received event in Figure 3.33:

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;
```

Go-Back-N or Selective Repeat

TCP sender need only maintain the smallest sequence number of a transmitted but unacknowledged byte (SendBase) and the sequence number of the next byte to be sent (NextSeqNum). TCP looks a lot like a GBN-style protocol. But there are some striking differences between TCP and Go-Back-N.

Many TCP implementations will buffer correctly received but out-of-order .Suppose that the acknowledgment for packet $n < N$ gets lost, but the remaining $N – 1$ acknowledgments arrive at the sender before their respective timeouts. In this example, GBN would retransmit not only packet n, but also all of the subsequent packets $n + 1, n + 2, \ldots, N$. TCP, on the other hand, would retransmit at most one segment, namely, segment n.

Moreover, TCP would not even retransmit segment n if the acknowledgment for segment $n + 1$ arrived before the timeout for segment n. A proposed modification to TCP, the so-called

Selective acknowledgment

TCP receiver acknowledges out-of-order segments selectively rather than just cumulatively acknowledging the last correctly received, inorder segment. When combined with selective retransmission—skipping the retransmission of segments that have already been selectively acknowledged by the receiver.

Flow Control

Hosts on each side of a TCP connection sets aside a receive buffer for the connection. When the TCP connection receives bytes that are correct and insequence, it places the data in the receive buffer.

The associated application process will read data from this buffer, but not necessarily at the instant the data arrives.

The receiving application may be busy with some other task and may not even attempt to read the data until long after it has arrived. If the application is relatively slow at reading the data, the sender can very easily overflow the connection's receive buffer by sending too much data too quickly.

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 sender can also be throttled due to congestion within the IP network; this form of sender control is referred to as **congestion control**.

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.

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.

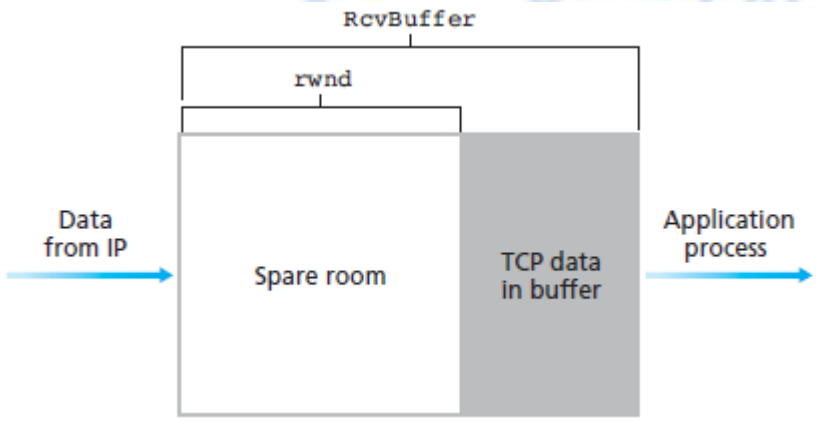
• LastByteRcvd: the number of the last byte in the data stream that has arrived from the network and has been placed in the receive buffer at B Because TCP is not permitted to overflow the allocated buffer, we must have

LastByteRcvd – LastByteRead <= RcvBuffer

The receive window, denoted rwnd is set to the amount of spare room in the buffer:

rwnd = RcvBuffer – [LastByteRcvd – LastByteRead]

Because the spare room changes with time, rwnd is dynamic. The variable rwnd is illustrated in Figure below.



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 B must keep track of several connection-specific variables.

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

Problem with this scheme: 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. 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

Suppose a process running in one host (client) wants to initiate a connection with another process in another host (server). 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 (refer TCP segment Figure), 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, 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. It contains three important pieces of information in the

segment header :

1) The SYN bit is set to 1.
2) The acknowledgment field of the TCP segment header is set to client_isn+1.
3) 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, "Server received clients SYN packet to start a connection with clients initial sequence number, client_isn.

Server is agree to establish this connection.

Servers initial sequence number is server_isn." The connection granted 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;

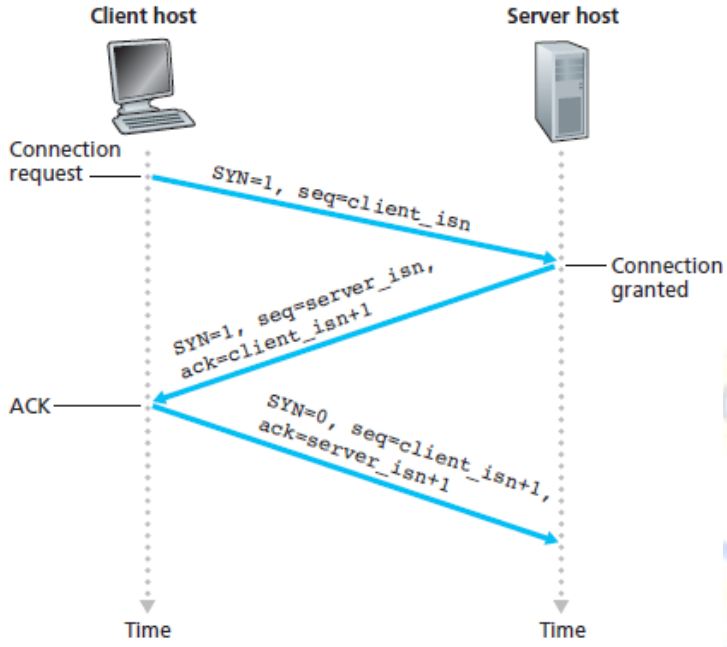
This last segment acknowledges the server's connection-granted segment (the client does so by putting the value server_isn+1 in the acknowledgment field of the TCP segment header). The SYN bit is set to zero, since the connection is established.

This third stage of the three-way handshake may carry client-to-server data in the segment payload.

Once these three steps have been completed, the client and server hosts can send segments containing data to each other. In each of these future segments, the SYN bit will be set to zero.

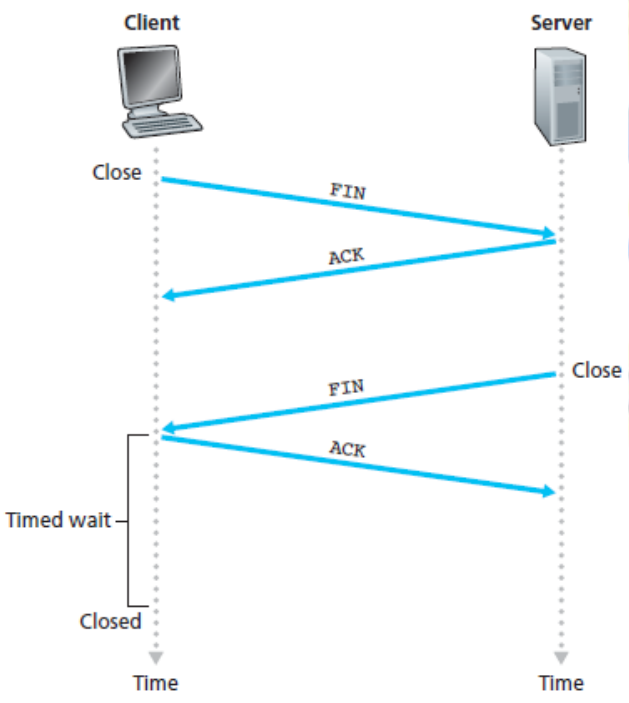
In order to establish the connection, three packets are sent between the two hosts, as illustrated in Figure.

For this reason, this connection establishment procedure is often referred to as a **three-way handshake**.



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.

Example, suppose the client decides to close the connection, as shown in Figure below. The client application process issues a close command. This causes the client TCP to send a special TCP segment to the server process.



This special segment has a flag bit in the segment's header, the FIN bit, 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.

During the life of a TCP connection, the TCP protocol running in each host makes transitions through various **TCP states**.

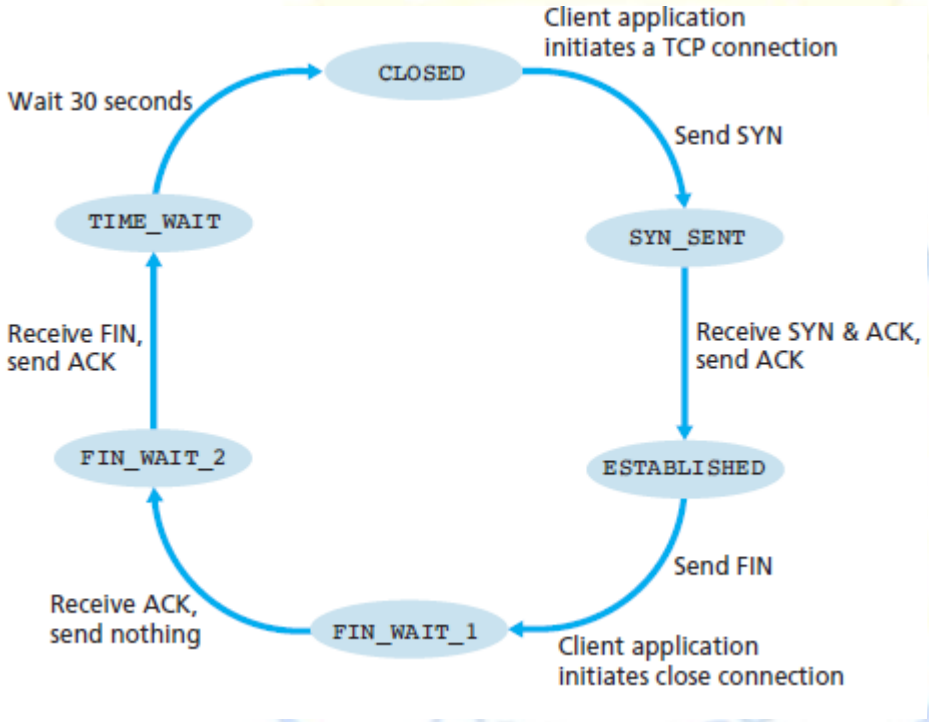


Figure above illustrates a typical sequence of TCP states that are visited by the *client* TCP. The client TCP begins in the CLOSED state. The application on the client side initiates a new TCP connection.

This causes TCP in the client to send a SYN segment to TCP in the server. After having sent the SYN segment, the client TCP enters the SYN_SENT state. While in the SYN_SENT state, the client TCP waits for a segment from the server TCP that includes an acknowledgment for the client's previous segment and has the SYN bit set to 1.

Having received such a segment, the client TCP enters the ESTABLISHED state. While in the ESTABLISHED state, the TCP client can send and receive TCP segments containing payload (message) data.

Suppose that the client application decides it wants to close the connection. This causes the client TCP to send a TCP segment with the FIN bit set to 1 and to enter the FIN_WAIT_1 state.

While in the FIN_WAIT_1 state, the client TCP waits for a TCP segment from the server with an acknowledgment. When it receives this segment, the client TCP enters the FIN_WAIT_2 state.

While in the FIN_WAIT_2 state, the client waits for another segment from the server with the FIN bit set to 1; after receiving this segment, the client TCP acknowledges the server's segment and enters the TIME_WAIT state.

The TIME_WAIT state lets the TCP client resend the final acknowledgment in case the ACK is lost. The time spent in the TIME_WAIT state is implementation-dependent, but typical values are 30 seconds, 1 minute, and 2 minutes.

After the wait, the connection formally closes and all resources on the client side (including port numbers) are released.

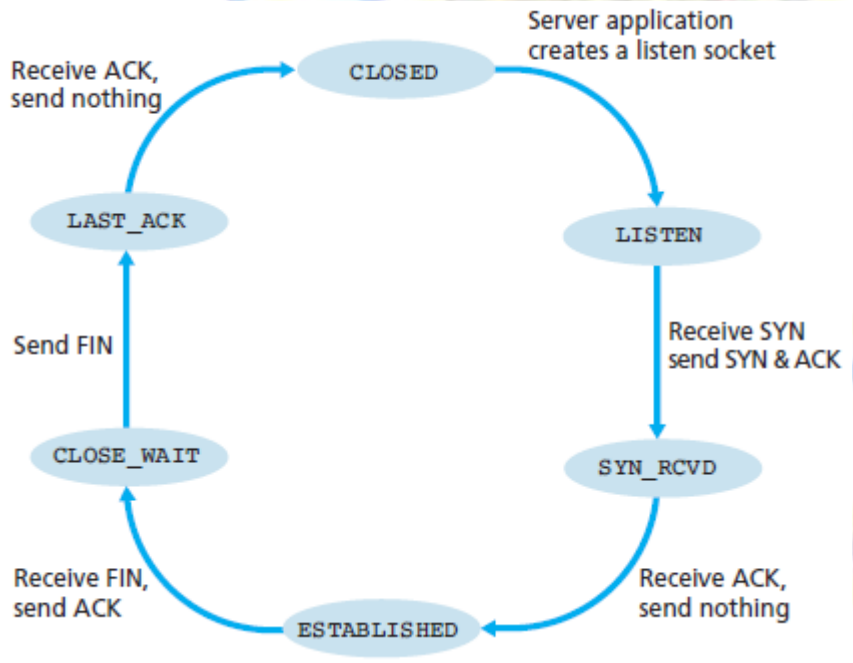


Figure above illustrates the series of states typically visited by the server-side TCP, assuming the client begins connection teardown.

If a host receives a TCP segment whose port numbers or source IP address do not match with any of the ongoing sockets in the host.

For example, suppose a host receives a TCP SYN packet with destination port 80, but the host is not accepting connections on port 80.

Then the host will send a special reset segment to the source. This TCP segment has the RST flag bit set to 1. Thus, when a host sends a reset segment,

Source sends TCP SYN segment with destination port 6789 to target host. There are three possible outcomes:

• *The source host receives a TCP SYNACK segment from the target host*. Since this means that an application is running with TCP port 6789 on the target post, returns "open."

• *The source host receives a TCP RST segment from the target host*. This means that the SYN segment reached the target host, but the target host is not running an application with TCP port 6789.

But the attacker at least knows that the segments destined to the host at port 6789 are not blocked by any firewall on the path between source and target hosts.

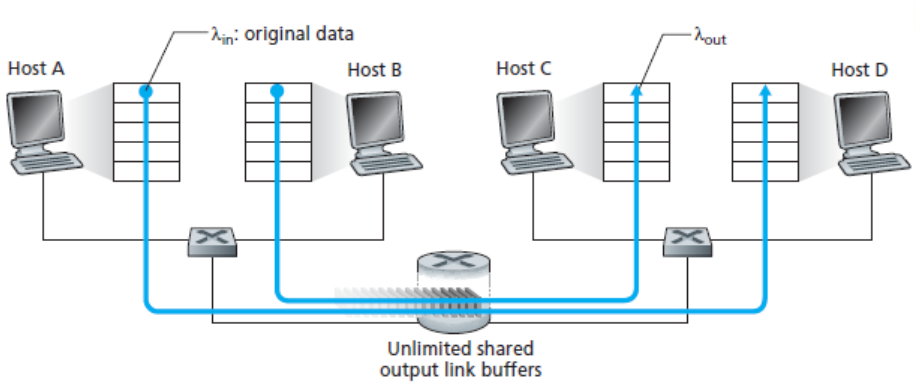
• *The source receives nothing*. SYN segment was blocked by an intervening firewall and never reached the target host.

Principles of Congestion Control

Cause of network congestion is too many sources attempting to send data at too high a rate. To treat the cause of network congestion, mechanisms are needed to throttle senders in the face of network congestion.

The Causes and the Costs of Congestion

Scenario 1: Two Senders, a Router with Infinite Buffers



The simplest congestion scenario possible: Two hosts (A and B) each have a connection that shares a single hop between source and destination, as shown in Figure above.

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.

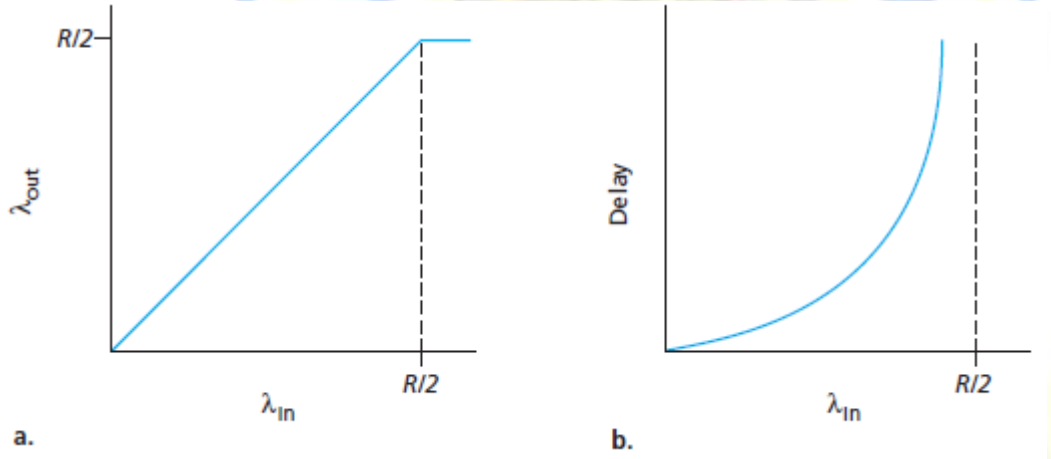


Figure above plots the performance of Host A's connection under this first scenario. The left graph plots the **per-connection throughput** (number of bytes per second at the receiver) as a function of the connection-sending rate. For a sending rate between 0 and *R*/2, the throughput at the receiver equals the sender's sending rate—everything sent by the sender is received at the receiver with a finite delay.

When the sending rate is above *R*/2, however, the throughput is only *R*/2. This upper limit on throughput is a consequence of the sharing of link capacity between two connections. The link simply cannot deliver packets to a receiver at a steady-state rate that exceeds *R*/2.

Even if Hosts A and B set their sending rates high, they will each never see a throughput higher than *R*/2.

Achieving a per-connection throughput of *R*/2 is good because the link is fully utilized in delivering packets to their destinations.

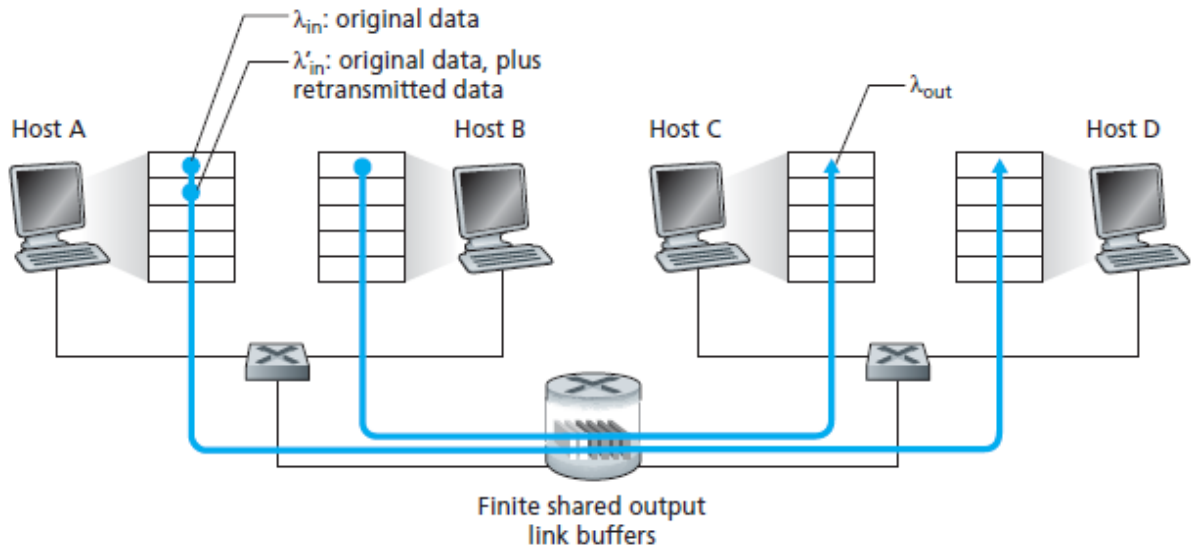
The right-hand graph in Figure a above, shows the consequence of operating near link capacity. As the sending rate approaches *R*/2 (from the left), the average delay becomes larger and larger.

When the sending rate exceeds *R*/2, the average number of queued packets in the router is unbounded and the average delay between source and destination becomes infinite (assuming that the connections operate at these sending rates for an infinite period of time and there is an infinite amount of buffering available).

Thus, while operating at an aggregate throughput of near *R* may be ideal from a throughput standpoint, it is far from ideal from a delay standpoint.

Thus, Large queuing delays are experienced as the packetarrival rate nears the link capacity.

Scenario 2: Two Senders and a Router with Finite Buffers



Scenario 1 is modified in the following two ways : (Figure).

1.The amount of router buffering is assumed to be finite. A consequence of this assumption is that packets will be dropped when arriving to an already full buffer.

2. 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.

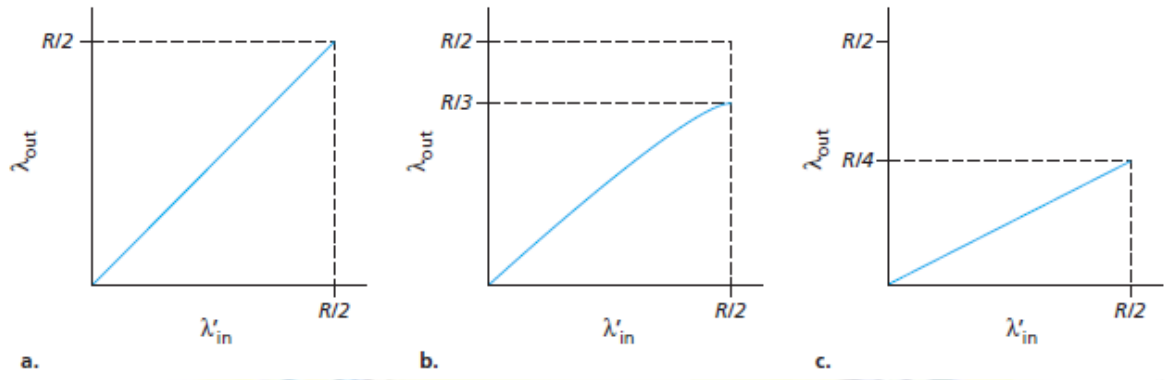
The rate at which the application sends original data into the socket by in bytes/sec.

The rate at which the transport layer sends segments (containing original data *and* retransmitted data) into the network will be denoted λ_{in} bytes/sec. λ_{in} is referred to as the **offered load** to the network.

The performance realized under scenario 2 will now depend strongly on how retransmission is performed.

1.Consider the unrealistic case that Host A is able to somehow determine whether or not a buffer is free in the router and thus sends a packet only when a buffer is free.

In this case, no loss would occur, λ_{in} would be equal to λ_{in}, and the throughput of the connection would be equal to λ_{in}.



This case is shown in Figure (a).

From a throughput standpoint, performance is ideal—everything that is sent is received.

The average host sending rate cannot exceed *R*/2 under this scenario, since packet loss is assumed never to occur.

2.Consider a realistic case in which the sender retransmits only when a packet is known for certain to be lost.

In this case, the performance is shown in Figure (b).

Consider the case that the offered load, λ_{in} (the rate of original data transmission plus retransmissions), equals *R/2*. According to Figure (b), at this value of the offered load, the rate at which data are delivered to the receiver application is *R*/3.

Thus, out of the 0.5*R* units of data transmitted, 0.333*R* bytes/sec (on average) are original data and 0.166*R* bytes/sec (on average) are retransmitted data.

The sender must perform retransmissions in order to compensate for dropped (lost) packets due to buffer overflow.

Finally, consider the case that the sender may time out prematurely and retransmit a packet that has been delayed in the queue but not yet lost.

In this case, both the original data packet and the retransmission may reach the receiver. The receiver needs but one copy of this packet and will discard the retransmission.

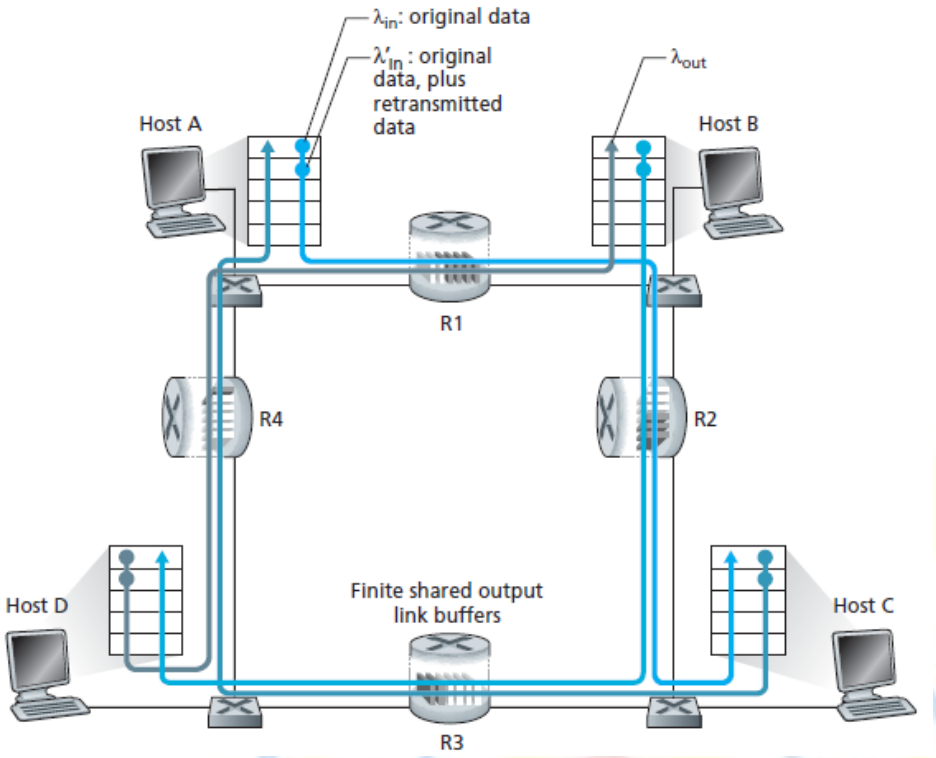
In this case, the work done by the router in forwarding the retransmitted copy of the original packet was wasted, as the receiver will have already received the original copy of this packet. The router would have better used the link transmission capacity to send a different packet instead.

The 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.

Figure (c) shows the throughput versus offered load when each packet is assumed to be forwarded (on average) twice by the router. Since each packet is forwarded twice, the throughput will have an asymptotic value of *R*/4 as the offered load approaches *R*/2.

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

In final congestion scenario, four hosts transmit packets, each over overlapping two-hop paths, as shown in Figure below.



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}, and that all router links have capacity *R* bytes/sec.

Consider the connection from Host A to Host C, passing through routers R1 and R2. The A–C connection shares router R1 with the D–B connection and shares router R2 with the B–D connection.

For extremely small values of λ_{in}, buffer overflows are rare (as in congestion scenarios 1 and 2), and the throughput approximately equals the offered load.

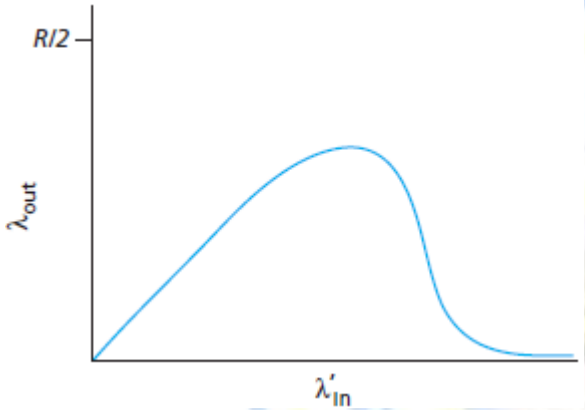
For slightly larger values of λ_{in}, the corresponding throughput is also larger, since more original data is being transmitted into the network and delivered to the destination, and overflows are still rare.

Thus, for small values of λin, an increase in λin results in an increase in λout. Having considered the case of extremely low traffic, consider the case that λin (and hence λin) is extremely large.

Consider router R2. The A–C traffic arriving to router R2 (which arrives at R2 after being forwarded from R1) can have an arrival rate at R2 that is at most *R*, the capacity of the link

from R1 to R2, regardless of the value of λin. If λin is extremely large for all connections (including the B–D connection), then the arrival rate of B–D traffic at R2 can be much larger than that of the A–C traffic. Because the A–C and B–D traffic must compete at router R2 for the limited amount of buffer space, the amount of A–C traffic that successfully gets through R2 (that is, is not lost due to buffer overflow) becomes smaller and smaller as the offered load from B–D gets larger and larger.

In the limit, as the offered load approaches infinity, an empty buffer at R2 is immediately filled by a B–D packet, and the throughput of the A–C connection at R2 goes to zero. This, in turn, *implies that the A–C end-to-end throughput goes to zero* in the limit of heavy traffic.



The offered load versus throughput tradeoff shown in Figure 3.48.

The reason for the eventual decrease in throughput with increasing offered load is evident when one considers the amount of wasted work done by the network.

In the high-traffic scenario, whenever a packet is dropped at a second-hop router, the work done by the first-hop router in forwarding a packet to the second-hop router ends up being "wasted."

The transmission capacity used at the first router to forward the packet to the second router could have been used to transmit a different packet.

Example, when selecting a packet for transmission, it might be better for a router to give priority to packets that have already traversed some number of upstream routers.

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.

Approaches to Congestion Control

The two broad approaches to congestion control that are taken in practice and discuss specific network architectures and congestion-control protocols embodying these approaches.

• **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.

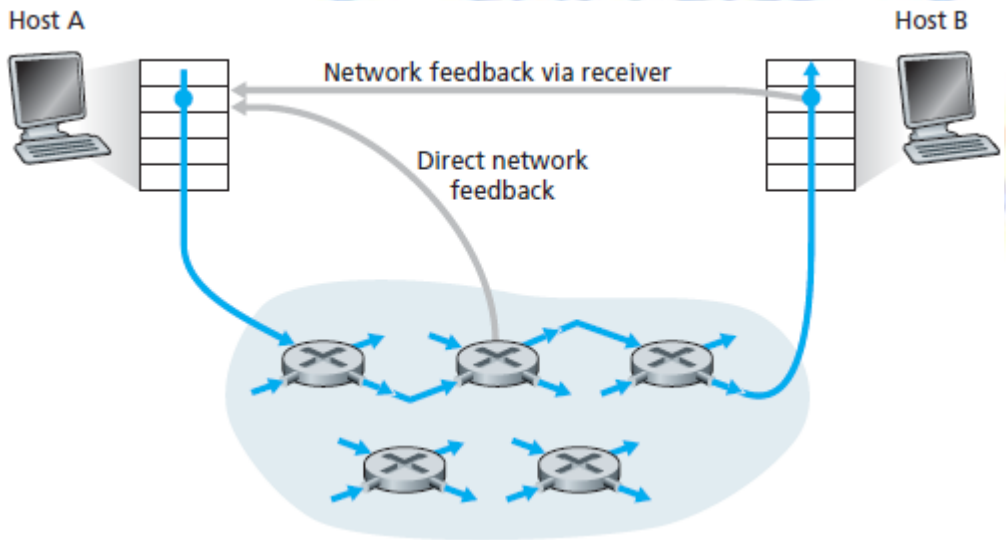
Even the presence of congestion in the network must be inferred by the end systems based only on observed network behavior (for example, packetloss and delay).

• *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 is used in ATM available bit-rate (ABR) congestion control.More sophisticated network feedback is also possible.

For example, one form of ATM ABR congestion control allows a router to inform the sender explicitly of the transmission rate it (the router) can support on an outgoing link.

For network-assisted congestion control, congestion information is typically fed back from the network to the sender in one of two ways, as shown in Figure below.



1.Direct feedback may be sent from a network router to the sender. This form of notification

typically takes the form of a **choke packet** .

2.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.