3.1 Introduction and Transport-Layer Services

- A transport-layer protocol provides for logical communication between application processes running on different hosts.
- Application processes use the logical communication provided by the transport layer to send messages to each other.

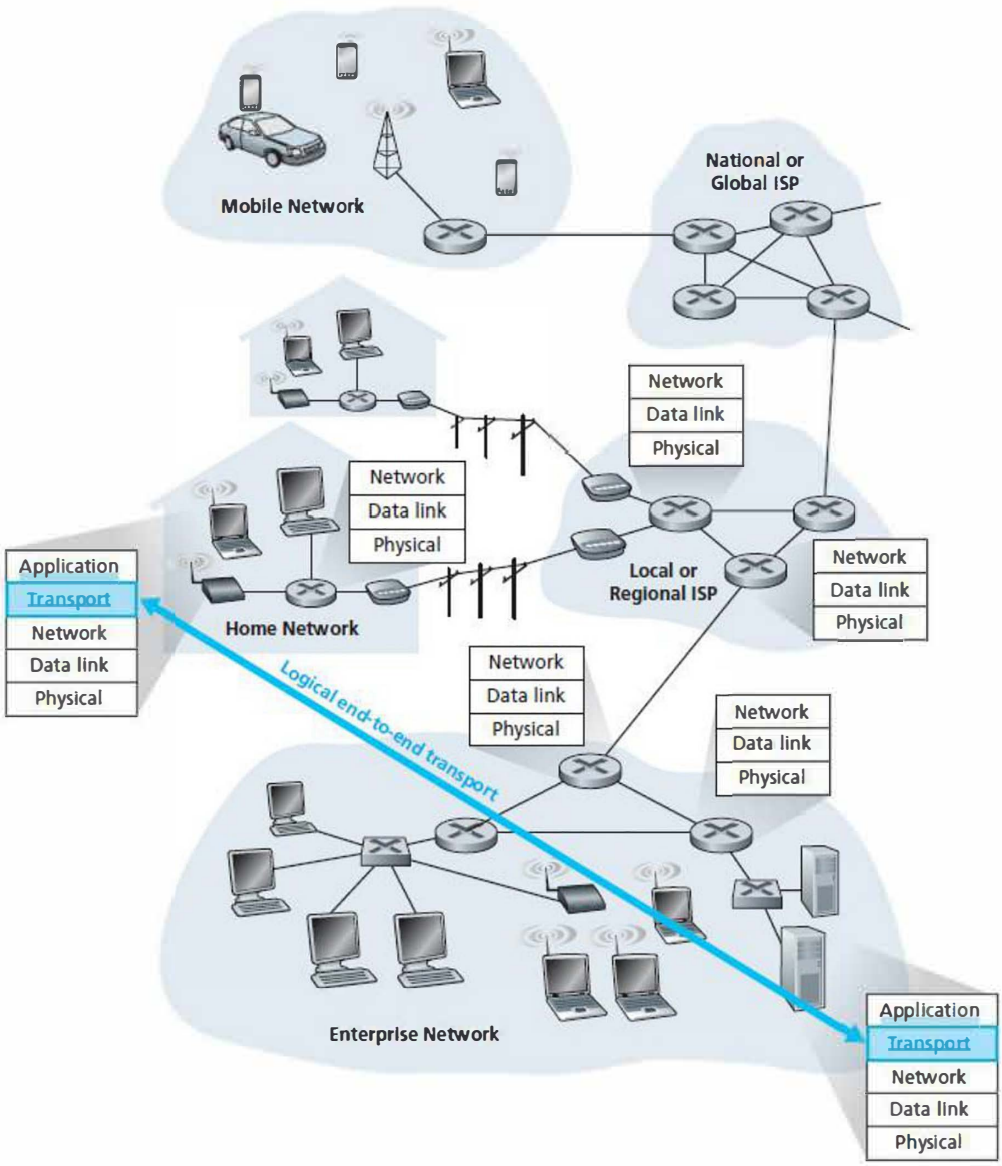


Figure 3.1 illustrates the notion of logical communication.

- The transport layer converts the application-layer messages into transport-layer segments.

- This is done by (possibly) breaking the application messages into smaller chunks and adding a transport-layer header to each chunk to create the transport-layer segment.
- The transport layer then passes the segment to the network layer at the sending end system, where the segment is encapsulated within a network-layer packet (a datagram in case of UDP) and sent to the destination.
- On the receiving side, the network layer extracts the transport-layer segment from the datagram and passes the segment up to the transport layer.
- The transport layer then processes the received segment, making the data in the segment available to the receiving application.
- More than one transport-layer protocol (i.e., TCP/ UDP) may be available to network applications.

3.1.1 Relationship between Transport and Network Layers

- A transport-layer protocol provides logical communication between processes running on different hosts, a network-layer protocol provides logical communication between hosts.
- Consider two houses with people staying there and let these houses be geographically apart.
- They do communicate via letters encapsulated within postal envelops. Also, suppose that they communication is one to one not many to one or one to many.
- This household example can become an analogy for understanding the relativity between transport layer & network layer. Hence following mappings are possible:

  Application messages = letters in envelopes

  Processes = people staying houses

  Hosts (also called end systems) = houses

  Transport-layer protocol = someone who collects/ dumps letters into letterbox

Network-layer protocol = postal service (including mail carriers)

- As per the above analogy, it can be noted that people responsible for collecting/ storing letters work within their respective homes; they are not involved in works other than that.
- Similarly, transport-layer protocols live in the end systems. Intermediate routers neither act on, nor recognize, any information that the transport layer may have added to the application messages.

3.1.2 Overview of the Transport Layer in the Internet

- 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*.
- This means that IP makes its "best effort" to deliver segments between communicating hosts, *but it makes no guarantees.* IP is said to be an unreliable service because it offers no guarantee about
  - Segment delivery,
  - Orderly delivery of segments,
  - Integrity of the data in the segments.
- As we know that transport layer's major protocols are TCP and UDP.
- The most fundamental responsibility of UDP and TCP is to extend IP's delivery service between two end systems to a delivery service between two processes running on the end systems.
- Extending host-to-host delivery to process-to-process delivery is called transport-layer multiplexing and de-multiplexing.
- UDP and TCP also provide integrity checking by including error detection fields in their segments' headers.

- In particular, like IP, UDP is an unreliable service—it does not guarantee that data sent by one process will arrive intact (or at all!) to the destination process.
- TCP provides reliable data transfer. Using flow control, sequence numbers, acknowledgments and timers like mechanisms, TCP ensures that data is delivered from sending process to receiving process, correctly and in order.
- TCP also provides congestion control.
- TCP strives to give each connection traversing a congested link an equal share of the link bandwidth by regulating the rate at which the sending sides of TCP connections can send traffic into the network.
- UDP traffic is unregulated that means an application using UDP transport can send at any rate it pleases, for as long as it pleases.

Probable Question: ***Brief on the relationship between transport and network layers***

3.2 Multiplexing and De-multiplexing

- The ultimate aim of any communication is process-to-process delivery service.
- Since multiple processes run on a computer, when the transport layer receives data from the network layer below, it needs to direct the received data to one of these four processes.
- A process (as part of a network application) can have one or more sockets for pushing

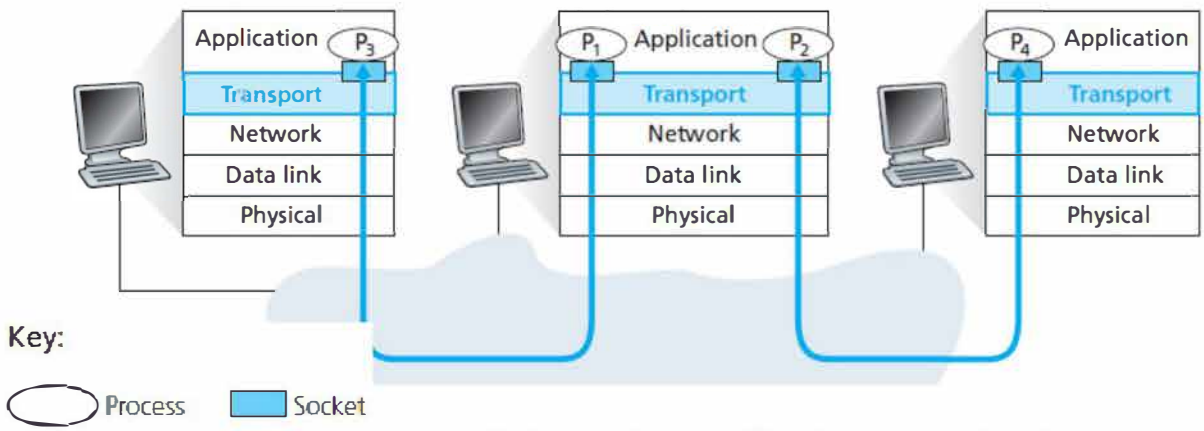


Figure 3.2: Transport layer multiplexing and de-multiplexing.

and retrieving data packets into/from sockets.

- Each socket has a unique identifier. The format of the identifier depends on whether the socket is a UDP or a TCP socket.
- A receiving host directs an incoming transport-layer segment to the appropriate socket by going through the fields of transport-layer segment. This job of delivering the data in a transport-layer segment to the correct socket is called *de-multiplexing*.
- The job of gathering data chunks at the source host from different sockets, encapsulating each data chunk with header information to create segments, and passing the segments to the network layer is called *multiplexing*.

Figure 3.3: Source and destination port-number fields in a transport-layer segment

- Transport-layer multiplexing requires

  (1) that sockets have unique identifiers, and

  (2) that each segment have special fields that indicate the socket to which the segment is to be delivered.

- These special fields are the source port number field and the destination port number field.
- Each port number is a 16-bit number, ranging from 0 to 65535. The port numbers ranging from 0 to 1023 are called well-known port numbers and are restricted, which means that they are reserved for use by well-known application protocols such as HTTP (which uses port number 80) and FTP (which uses port number 21).

Connectionless Multiplexing and De-multiplexing

- We can create a UDP socket by having the below instruction in Python language.

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

- Transport layer automatically assigns a port number in the range 1024 to 65535 to the socket that is currently not being used by any other UDP port in the host.
- Alternatively, the below instruction can be used to specify port number.

```
clientSocket.bind(( '' , 19157))
```

Read to know more…

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), and few other values. 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. Note that Host B could be running multiple processes, each with its own UDP socket and associated port number. As UDP segments arrive from the network, Host B directs (de-multiplexes) each segment to the appropriate socket by examining the segment's destination port number.

It is important to note that a UDP socket is fully identified by a two-tuple consisting of a destination IP address and a destination port number.

Connection-Oriented Multiplexing and De-multiplexing

- One subtle 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 (de-multiplex) the segment to the appropriate socket.

Let's us understand like this,

• 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))
```

• A connection-establishment request is nothing more than a TCP segment with destination port number 12000 and a special connection-establishment bit set in the TCP header.

• 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()
```

Also, 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.

- A newly created connection socket is identified by the above four values and all subsequently arriving segments whose source port, source IP address, destination port, and destination IP address match these four values will be de-multiplexed to this socket.

- 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. De-multiplexing is done on the basis of the four fields.

Read to know more…

The situation is illustrated in below figure, in which Host C initiates two HTTP sessions to server B, and Host A initiates one HTTP session to B. Hosts A and 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.

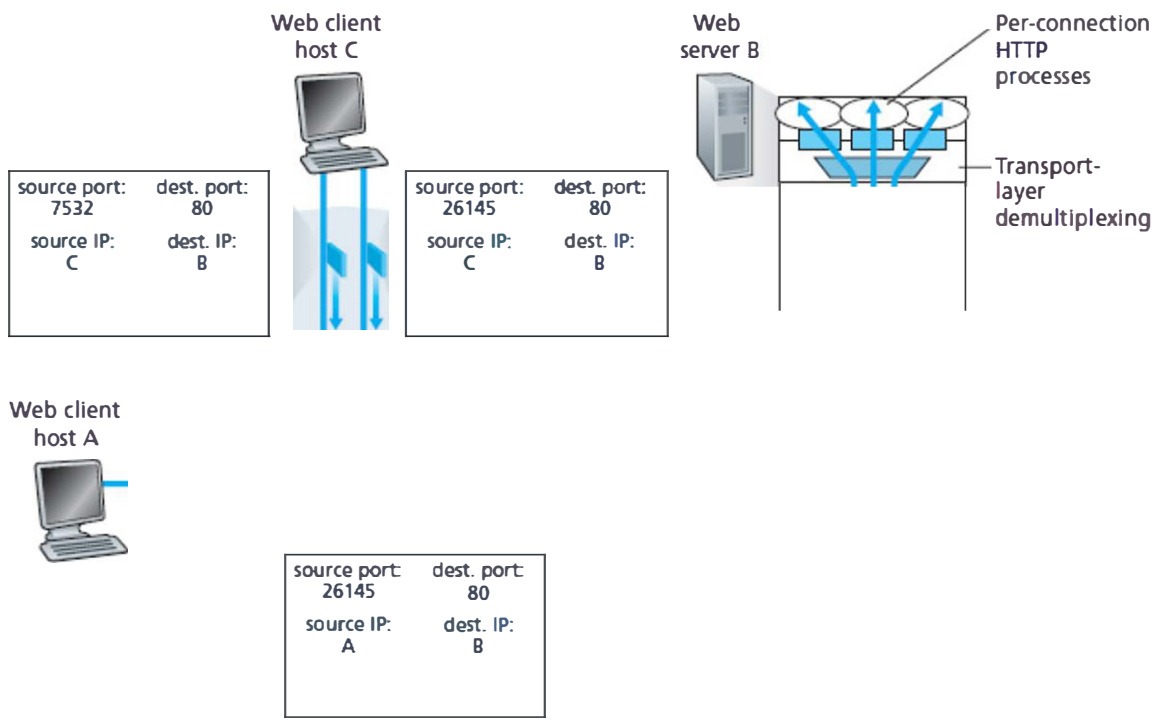


Figure 3. : Two clients, using the same destination port number (80) to communicate with the same Web server application

Because Host A is choosing source port numbers independently of C, it might also assign a source port of 26145 to its HTTP connection. But this is not a problem—server B will still be able to correctly de-multiplex the two connections having the same source port number, since the two connections have different source IP addresses.

Web Servers and TCP

- As we understood from the previous discussion that the server creates separate process per socket connection to handle the segments from various clients.
- Based on the four field values, de-multiplexing will be done so as to deliver the segments to respective processes of host server.
- Now, 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. So, web server manages the requests in a different way.
- They often use only one process, and create a new thread with a new connection socket for each new client connection.
- If the client and server are using persistent HTTP, then throughout the duration of the persistent connection the client and server exchange HTTP messages via the same server socket and if non-persistent, reverse is the mode.

Probable Question: ***Explain (1) Transport layer multiplexing and de-multiplexing***

(2) Transport layer segment

(3) Connectionless multiplexing and de-multiplexing

(4) Connection – oriented multiplexing and de-multiplexing

3.3 Connectionless Transport: UDP

- It's essential that the transport layer has to provide a multiplexing/de-multiplexing service in order to pass data between the network layer and the correct application-level process.
- But if the application developer chooses UDP instead of TCP, then the application is almost directly talking with IP which is connectionless.
- UDP takes messages from the application process, attaches source and destination port number fields for the multiplexing/de-multiplexing 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. But UDP is connectionless and unreliable.
- If querying host doesn't receive a reply (possibly because the underlying network lost the query or the reply), either it tries sending the query to another name server, or it informs the invoking application that it can't get a reply.

Now the question arises "Why UDP??".

UDP is preferred for the following reasons:

• *Finer application-level control over what data is sent, and when.* Since UDP doesn't involve handshaking, congestion control or any such mechanisms, there is a bigger possibility of high data transmission rate.

Since real-time applications often require an optimal sending rate and can tolerate some data loss, TCP's service model is not particularly well matched to these applications' needs.

• *No connection establishment.* UDP just blasts away without any formal preliminaries. Thus UDP does not introduce any delay to establish a connection.

• *No connection state.* UDP, on the other hand, does not maintain connection state and does not track any of these parameters, which seems to be a burden. For this reason, a server devoted to a particular application can typically support many more active clients when the application runs over UDP rather than TCP.

• *Small packet header overhead.* The TCP segment has 20 bytes of header overhead in every segment, whereas UDP has only 8 bytes of overhead.

- Both UDP and TCP are used today with multimedia applications, such as Internet phone, real-time video conferencing, and streaming of stored audio and video.
- As we know TCP has congestion control mechanism and is needed to prevent the network from entering a congested state in which very little useful work is done.

- If everyone were to start streaming high-bitrate video without using any congestion control, there would be so much packet overflow at routers that very few UDP packets would successfully traverse the source-to-destination path.

3.3.1 UDP Segment Structure

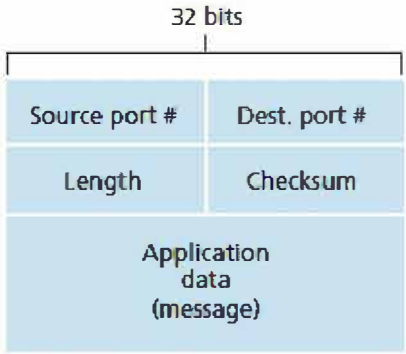


Figure 3. : The UDP segment structure

- Any application based on UDP will insert its data into the data field of the UDP segment.
- For example, for DNS, the data field contains either a query message or a response message. For a streaming audio application, audio samples fill the data field.

The UDP header has only four fields, each consisting of two bytes.

- Two *port number* fields allow the destination host to pass the application data to the correct process running on the destination end system
- The *length* field specifies the number of bytes in the UDP segment (header plus data).
- The *checksum* is used by the receiving host to check whether errors have been introduced into the segment.

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 (for example, by noise in the links or while stored in a router) as it moved from source to destination. UDP at the sender side performs the 1s complement of the sum of all the 16-bit words in the segment, with any overflow encountered

during the sum being wrapped around. This result is put in the checksum field of the UDP segment. As an 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

Note that this last addition had overflow, which was wrapped around. The 1s complement is obtained by converting all the 0s to 1s and converting all the 1s to 0s. Thus the 1s complement of the sum 0100101011000010 is 1011010100111101, which becomes the checksum.

At the receiver, all four 16-bit words are added, including the checksum. If no errors are introduced into the packet, then clearly the sum at the receiver will be 1111111111111111. If one of the bits is a 0, then we know that errors have been introduced into the packet.

- The main reason behind UDP employing this mechanism is that 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.
- Furthermore, 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.

Probable Question: ***Explain the significance of UDP.***

Explain UDP segment structure

A problem on UDP checksum calculation

3.4 Principles of Reliable Data Transfer

- Reliable data transfer in any context is all about integrity based transfer.
- This is precisely the service model offered by TCP to the Internet applications that invoke it.
- This task is made difficult by the fact that the layer below the reliable data transfer protocol may be unreliable.
- Below figure illustrates reliable data transfer along with service model and service implementation.
- It is the responsibility of a reliable data transfer protocol to implement schemes to deliver each bit as it is to the destination.
- Ex: TCP is a reliable data transfer protocol that is implemented on top of an unreliable (IP) end-to-end network layer
- Let us assume that we have method as a support: `rdt_send()` at the sender's end. (Here rdt stands for **reliable data transfer protocol** and **_send** indicates that the sending side of rdt is being called).
- The sending side of the data transfer protocol will be invoked from above by a call to `rdt_send()`. It will pass the data to be delivered to the upper layer at the receiving side.
- On the receiving side, `rdt_rcv()` will be called when a packet arrives from the receiving side of the channel. When the rdt protocol wants to deliver data to the upper layer, it will do so by calling `deliver_data()`.

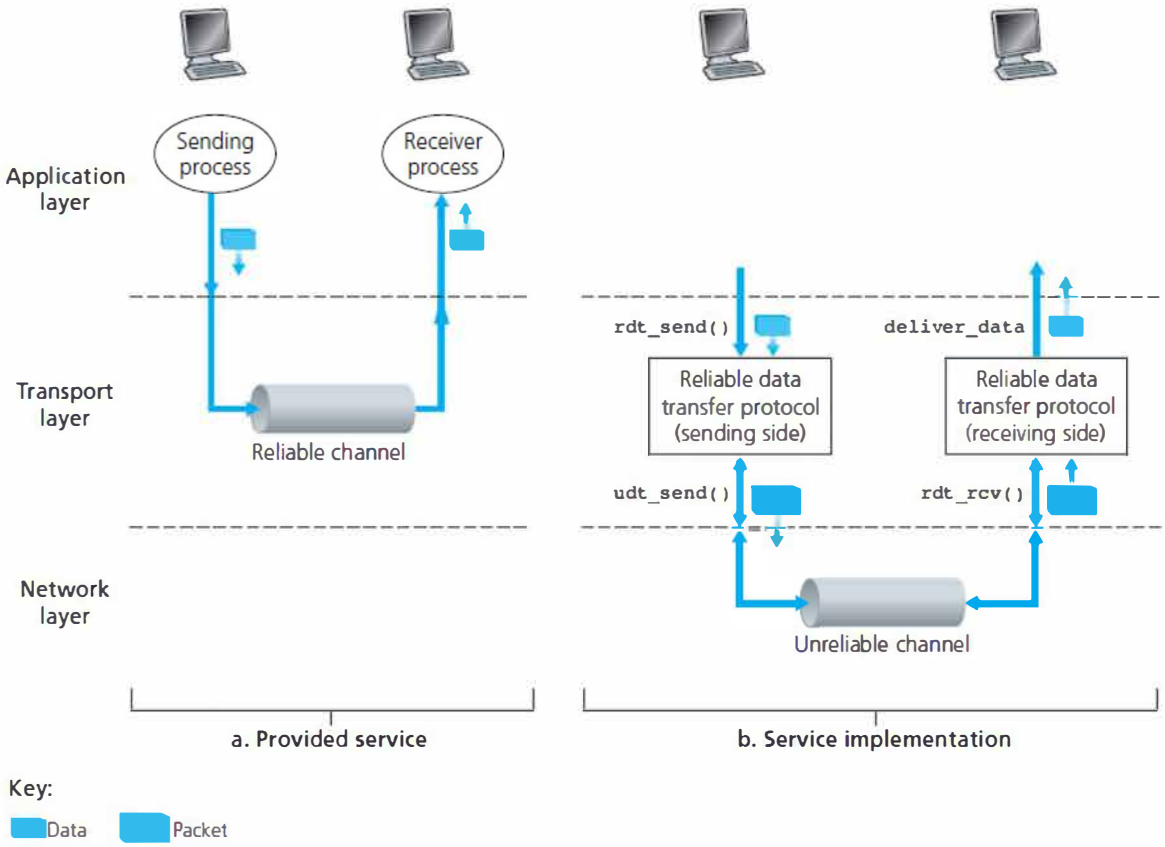


Figure 3. : Reliable data transfer: Service model and service implementation

- In this section we consider only the case of unidirectional data transfer, that is, data transfer from the sending to the receiving side.
- Further exploration will incrementally develop the sender and receiver sides of a reliable data transfer protocol, considering increasingly complex models of the underlying channel
- Both the send and receive sides of rdt send packets to the other side by a call to `udt_send()` (where udt stands for **unreliable data transfer**).

3.4.1 Building a Reliable Data Transfer Protocol

- Let us now step through a series of protocols, each one becoming more complex, arriving at a flawless, reliable data transfer protocol.

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

- We first consider the simplest case, in which the underlying channel is completely reliable.
- The finite-state machine (FSM) definitions for the rdt1.0 sender and receiver are shown in below figure. The FSM in figure (a) defines the operation of the sender, while the FSM in figure (b) defines the operation of the receiver.
- The arrows in the FSM description indicate the transition of the protocol from one state to another. 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.

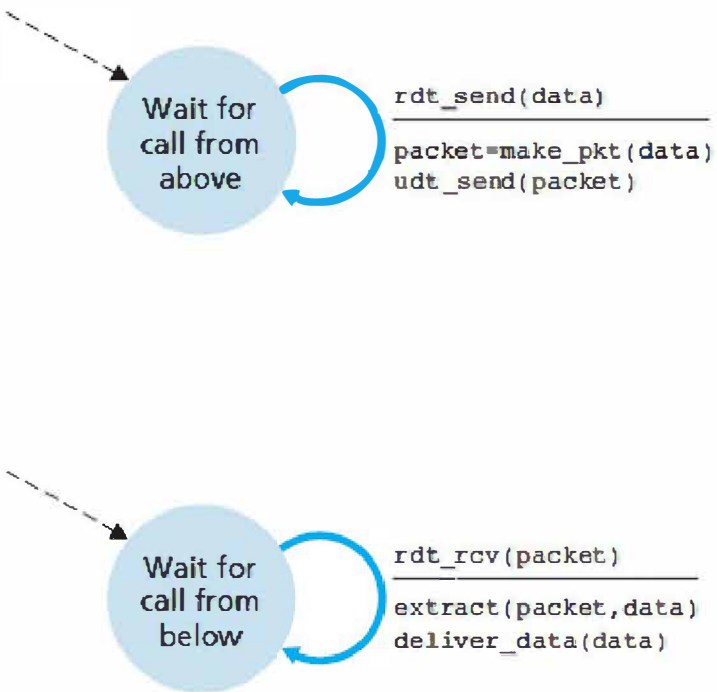


Figure 3. : rdt1.0 – A protocol for a completely reliable channel

- 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.

- 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)`).
- Also, all packet flow is from the sender to receiver; with a perfectly reliable channel there is no need for the receiver side to provide any feedback to the sender since nothing can go wrong.

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

- When a data packet under propagation on a certain channel, there are possibilities of bit errors typically occur in the physical components of a network as a packet is transmitted, propagates, or is buffered.
- Let's continue to assume for the moment that all transmitted packets are received (although their bits may be corrupted) in the order in which they were sent.

As an analogy, consider your telephonic conversation would be acknowledged by your friend saying "OK" after each sentence has been heard, understood, and recorded.

If the receiver hears a garbled sentence, you're asked to repeat the garbled sentence.

This message-dictation protocol uses both positive acknowledgments ("OK") and negative acknowledgments ("Please repeat that.").

- These control messages allow the receiver to let the sender know what has been received correctly, and what has been received in error and thus requires repeating.
- In a computer network setting, reliable data transfer protocols based on such retransmission are known as **ARQ (Automatic Repeat reQuest) protocols**.
- Fundamentally, three additional protocol capabilities are required in ARQ protocols to handle the presence of bit errors:

  • *Error detection*. First, a mechanism is needed to allow the receiver to detect when bit errors have occurred.

  • *Receiver feedback*. The only way for a sender to know that the receiver's receipt is those positive (ACK) and negative (NAK) acknowledgments. In principle, these packets need only

be 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.

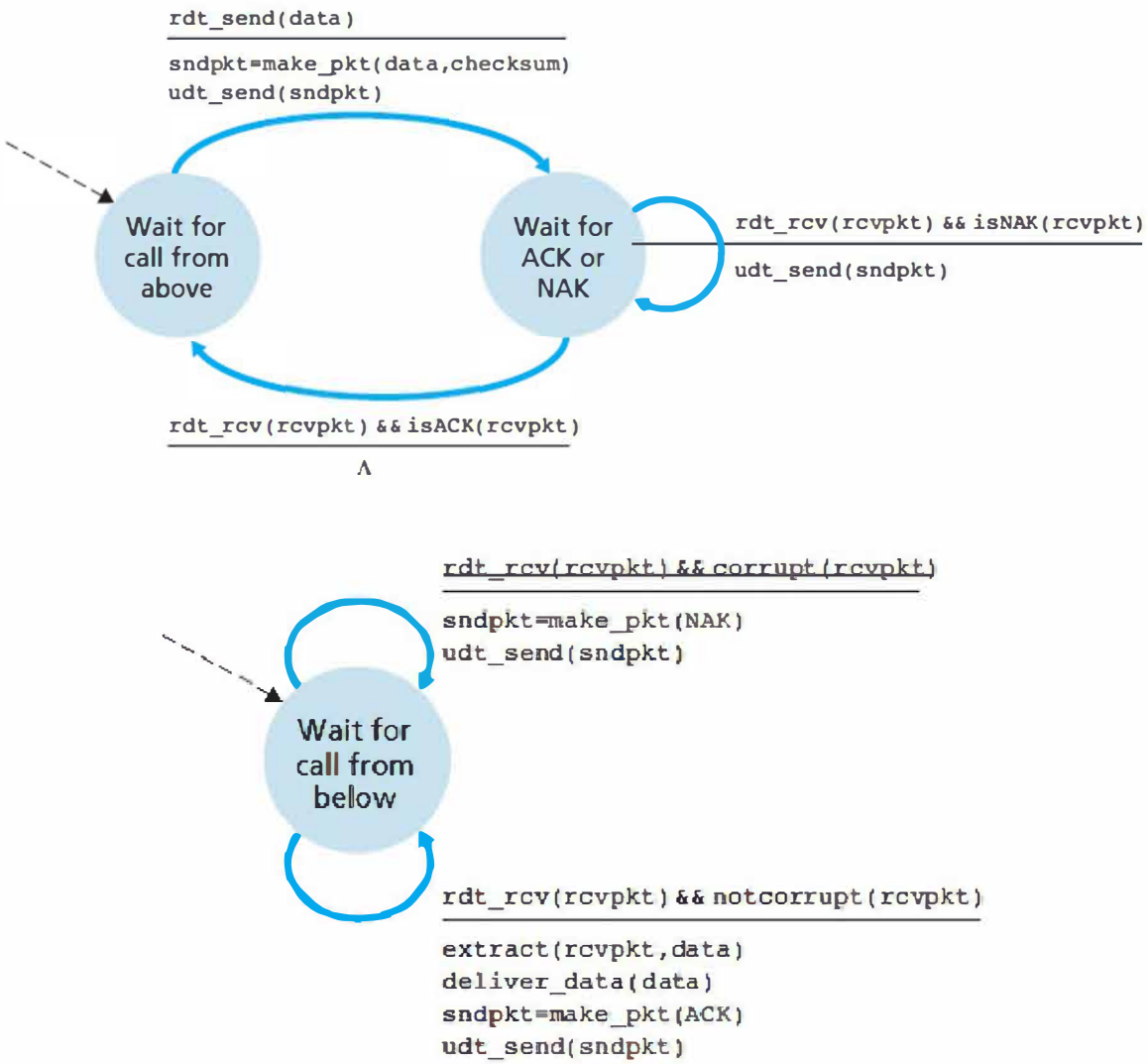


Figure 3.10: rdt2.0–A protocol for a channel with bit errors

- Above figure shows the FSM representation of rdt2.0, a data transfer protocol employing error detection, positive acknowledgments, and negative acknowledgments.

- The send side of rdt2.0 has two states: State describing 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.
- The sender protocol is waiting for an ACK or a NAK packet from the receiver.
- If an ACK packet is received, the sender knows that the most recently transmitted packet has been received correctly and thus the protocol returns to the state of waiting for data from the upper layer.
- If a NAK is received, the protocol retransmits the last packet and waits for an ACK or NAK to be returned by the receiver in response to the retransmitted data packet.
- It is important to note that when the sender is in the wait-for-ACK-or-NAK state, it cannot get more data from the upper layer unless the sender receives an ACK from receiver.
- 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 such as rdt2.0 are known as **Stop-and-Wait protocols.**
- 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.
- In the above figure, the notation `rdt_rcv(rcvpkt) && corrupt(rcvpkt)` corresponds to the event in which a packet is received and is found to be in error.
- Protocol rdt2.0 has a fatal flaw that the ACK or NAK packet could be corrupted!
- Consider three possibilities for handling corrupted ACKs or NAKs:

  • Introducing a new type of sender-to-receiver packet to our protocol something like sender asking “What did you say?” if the receiver’s ACK gets distorted.

  But if this itself gets corrupted?? The receiver, having no idea whether the garbled sentence was part of the dictation or a request to repeat the last reply, would probably then respond with

"What did you say?" And then, of course, that response might be garbled. Clearly, we're heading down a difficult path.

• 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, however, 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. Thus, it cannot know a priori whether an arriving packet contains new data or is a retransmission!
- A simple solution to this new problem is to add a new field i.e., packet ID to the data packet which sender sends. 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-and-wait protocol, a 1-bit sequence number will suffice.
- Since we are currently assuming a channel that 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.
- In the figures shown below, both the rdt2.1 sender and receiver has been portrayed using FSM states.
- The sender transits between its states with respect to transmission of packet 0 or 1 and waits for an ACK or NAK for the same.
- 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.
- 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.

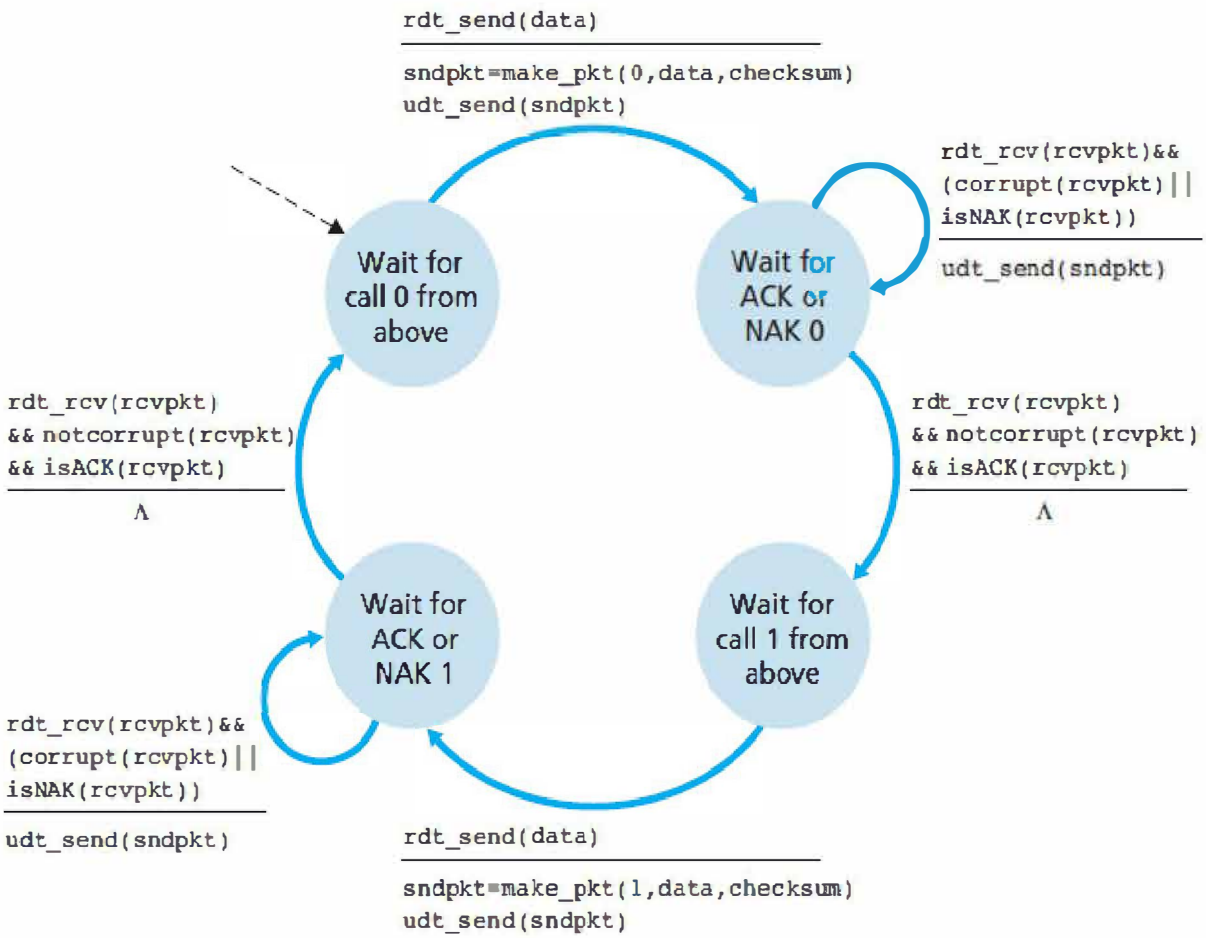


Figure 3. : rdt2.1 sender

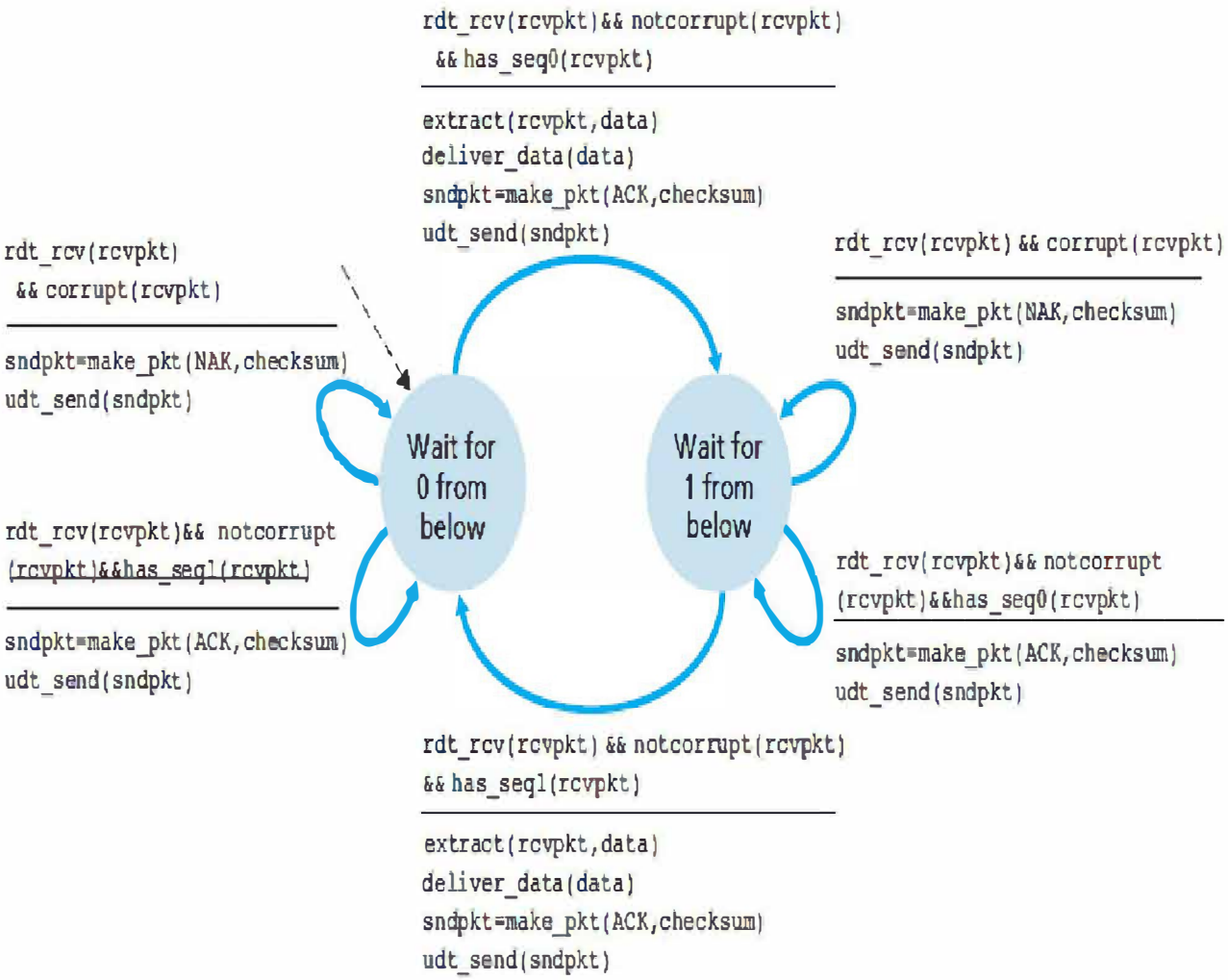


Figure 3. : rdt2.1 receiver

- Incorporating packet ID concept, the protocol gets updated to rdt2.2.
- One subtle change between rtdt2.1 and rdt2.2 is that the receiver must now include the sequence number of the packet being acknowledged by an ACK message (this is done by including the ACK,0 or ACK,1 argument in make_pkt() in the receiver FSM), and 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).

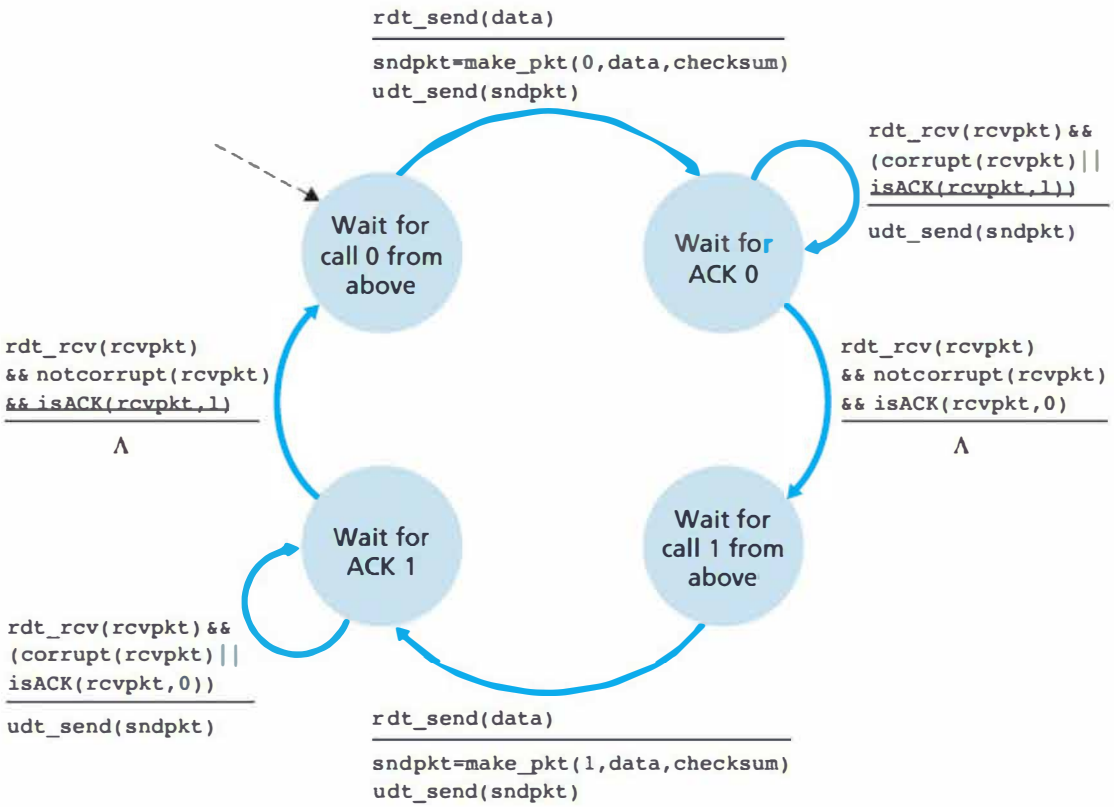


Figure 3. : rdt2.2 sender

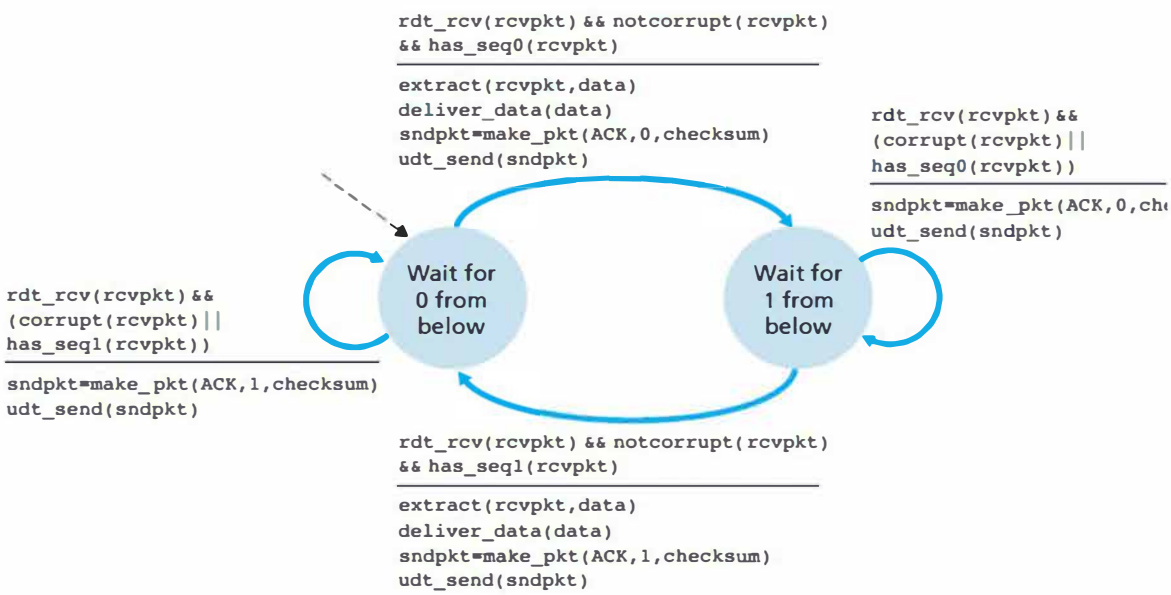


Figure 3. : rdt2.2 receiver

Probable Question: ***Elaborate on the principles of reliable data transfer technique***

Explain rdt1.0 along with Finite State Machine models

Explain rdt2.0 along with Finite State Machine models

What are ARQ protocols? Explain their protocol capabilities

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

- Suppose packet loss becomes the paranoid in addition to bit errors just because the underlying channel is unreliable, we must think about it.
- The use of check-summing, sequence numbers, ACK packets, and retransmissions—the techniques already developed in rdt2.2—will allow us to answer the latter concern.
- Suppose that the sender transmits a data packet and either that packet, or the receiver's ACK of that packet, gets lost. Hence no reply from the receiver.
- Sender can wait up to sometime and then retransmit. But how long? (A million dollar question)
- The sender must clearly wait at least as long as a round-trip delay between the sender and receiver (which may include buffering at intermediate routers) plus whatever amount of time is needed to process a packet at the receiver.
- The approach thus adopted in practice is for the sender to judiciously choose a time value such that packet loss is likely, although not guaranteed, to have happened.
- Whether packet loss or ACK lost, the solution is retransmit.
- Implementing a time-based retransmission mechanism requires a countdown timer that can interrupt the sender after a given amount of time has expired.
- The sender will thus need to be able to

  (1) start the timer each time a packet (either a first-time packet or a retransmission) is sent,

  (2) respond to a timer interrupt (taking appropriate actions), and

  (3) stop the timer.

- Below figure shows the sender FSM for rdt3.0, a fine tuned protocol that reliably transfers data over a channel that can corrupt or lose packets based on timer.

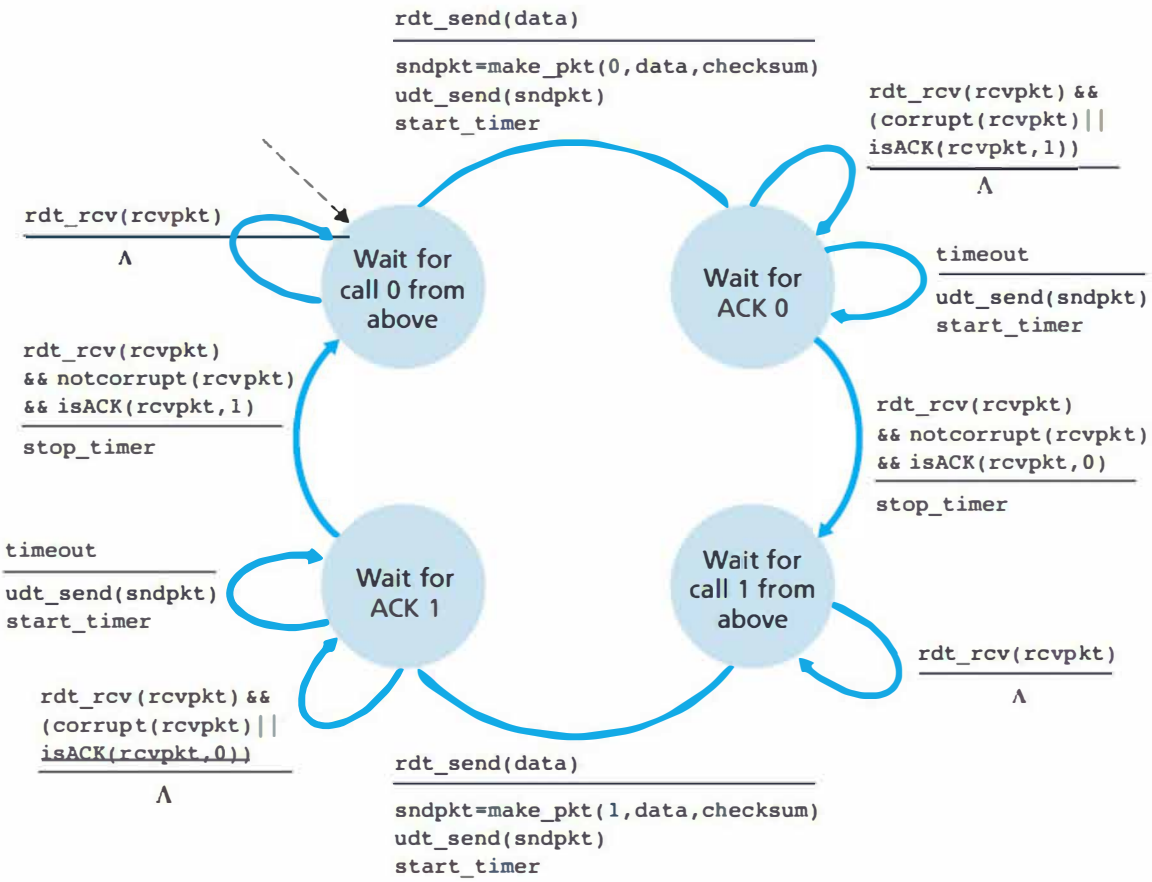


Figure 3. : rdt3.0 sender

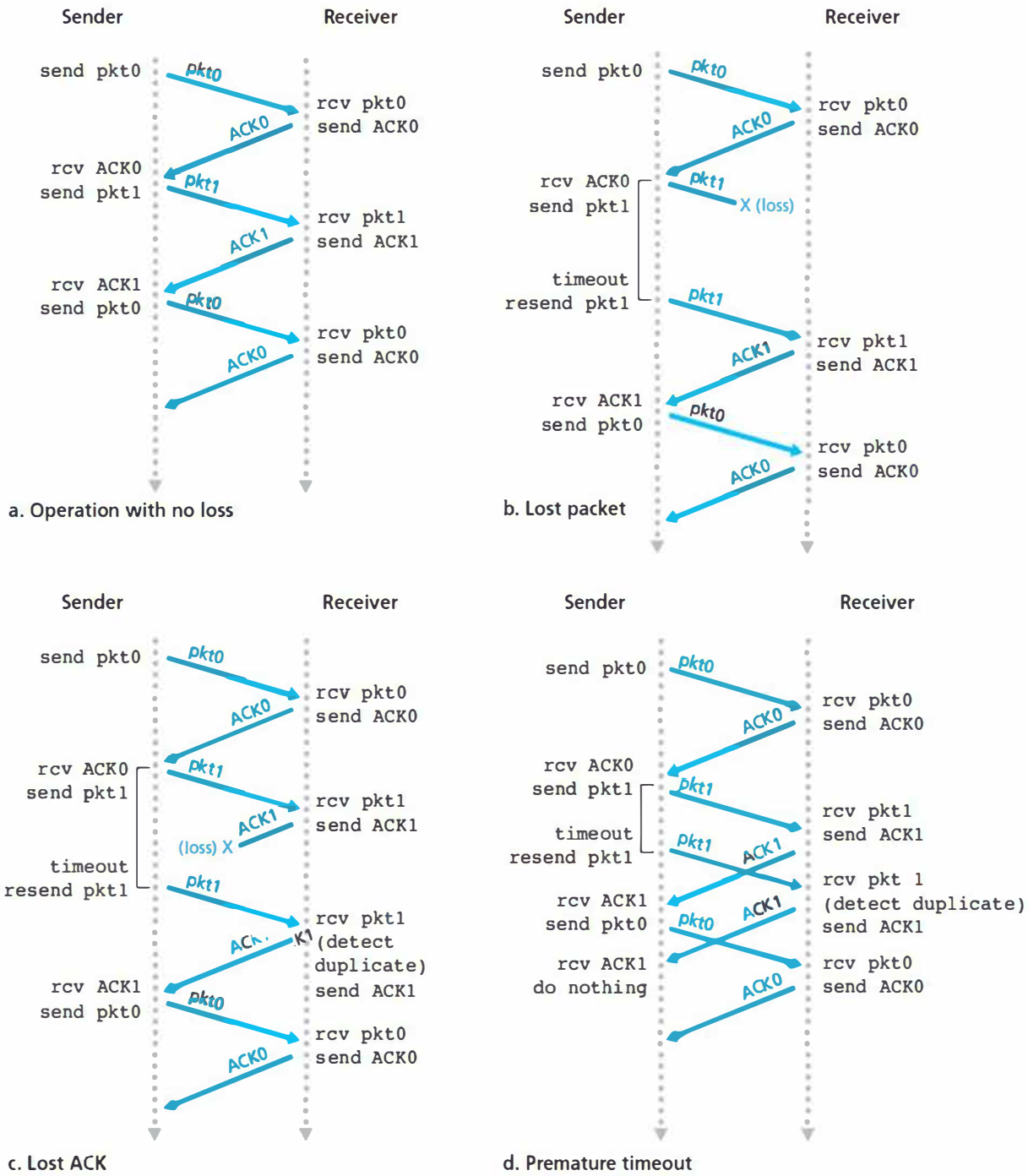


Figure 3. : Operation of rdt3.0, the alternating-bit protocol

Above figure shows how the protocol operates in four different contexts: operation with no loss, lost packet, lost ACK and premature timeout.

Probable Question: ***Explain Stop-and-wait ARQ protocol***

Explain rdt3.0 reliable data transfer along with FSMs

3.4.2 Pipelined Reliable Data Transfer Protocols

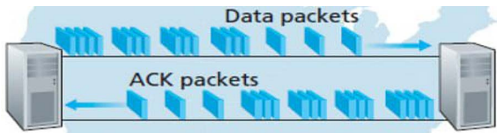
- Protocol rdt3.0 is a stop-and-wait protocol and is functionally correct.
- To appreciate the performance impact of this stop-and-wait behavior, let us consider a scenario. Two hosts, located geographically apart.
- Let the round-trip propagation delay between these two end systems, RTT, is approximately 30 milliseconds. Suppose that they are connected by a channel with a transmission rate, R, of 1 Gbps (10^9 bits per second). With a packet size, L, of 1,000 bytes, (8,000 bits) per packet, including both header fields and data, the time needed to actually transmit the packet into the 1 Gbps link is

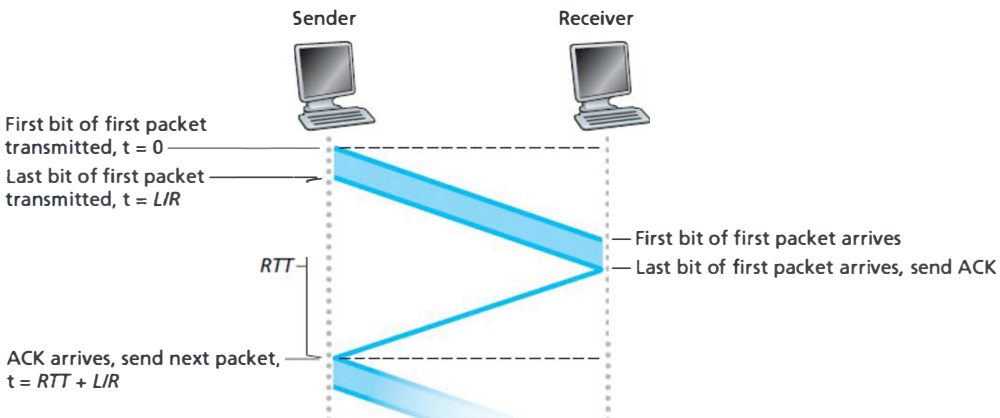
$$d_{trans} = L / R = 8000 \text{ bits per packet} / 10^9 \text{ bits per second} = 8 \text{ microseconds}$$

- If the sender sends a packet at t = 0, then at t=8 microsecond, the last bit enters into the communication channel. Since RTT is 30 milliseconds, the receiver receives the last bit at t = RTT/2 + 8 microseconds, i.e., at t = 15 milliseconds + 8 microseconds = 15.008 milliseconds.
- If the transmission time for ACK is negligible, then sender gets ACK at t = 30.008 milliseconds.
- Thus in 30.008 msec, sender was sending for 0.008 msec only.

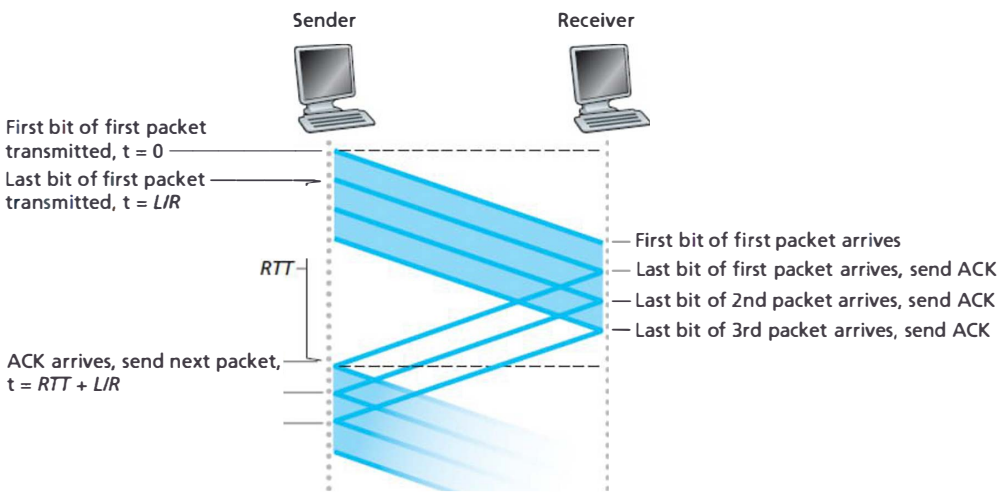
i.e., 0.008 / 30.008 = 0.00027 that means the sender was busy only 2.7 hundredths of one percent of the time!

- Viewed another way, the sender was able to send only 1,000 bytes in 30.008 milliseconds, an effective throughput of only 267 kbps—even though a 1 Gbps link was available!
- The solution to this particular performance problem is simple: Rather than operate in a stop-and-wait manner, the sender is allowed to send multiple packets without waiting for acknowledgments, as shown in below. Since the many in-transit sender-to-receiver packets can be visualized as filling a pipeline, this technique is known as pipelining.





a. Stop-and-wait operation



b. Pipelined operation

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

3.4.3 Go-Back-N (GBN)

- In this protocol, the sender is allowed to transmit multiple (not more than 'N') packets without waiting for an acknowledgment from the receiver.
- N is agreed upon value; unacknowledged packets in the pipeline.
- Sequence numbers are of vital interest, for packets to be transmitted. Let's understand like this:

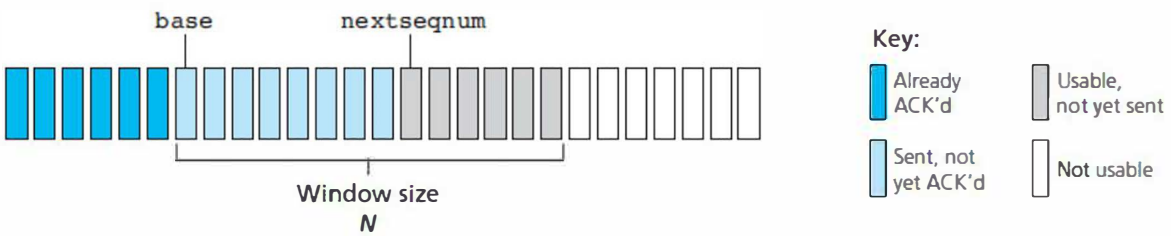


Figure 3. : Sender's view of sequence numbers in Go-Back-N

- If we define *base* to be the sequence number of the oldest unacknowledged packet and *nextseqnum* to be the smallest unused sequence number, then four intervals in the range of sequence numbers can be identified.

  1) Sequence numbers in the interval [0, base-1] correspond to packets that have already been transmitted and acknowledged.

  2) The interval [base, nextseqnum-1] corresponds to packets that have been sent but not yet acknowledged.

3) 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.

4) Finally, sequence numbers greater than or equal to base+N cannot be used until an unacknowledged packet currently in the pipeline (specifically, the packet with sequence number base) has been acknowledged.

- We must understand like this:

  Let base pointing Seq. number 6, Window size N be 14 and nextseqnum be 14 (as per the above figure)

  So, first interval [0, base-1] -> [0,5] that means first 6 sequence numbers.

  Second interval [base, nextseqnum-1] -> [6, 13]

  Third interval [nextseqnum, base+N-1] -> [14, 6+14-1] -> [14, 19] that means from seq. number 14 to 19.

  Fourth interval will start from base + N onwards, i.e., 20 onwards.

  From the above figure, 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 thereby updating base, nextseqnum values.

  In practice, some k-bits used to build sequence numbers thus the range of sequence numbers is $[0,2^{k-1}]$.

  A modulo 2^k arithmetic will be usually used to make it a ring for providing sequence numbers to new packets when upper bound is reached.

- Below figures give an extended FSM description of the sender and receiver sides of an ACK-based, NAK-free, GBN protocol.

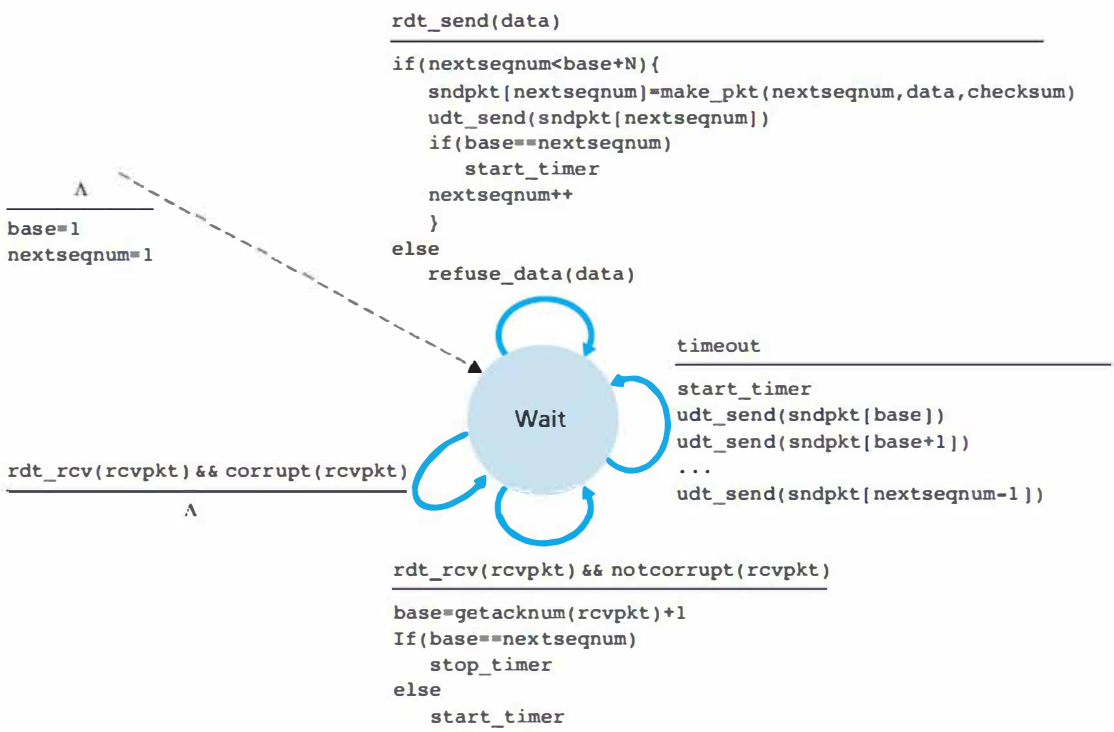


Figure 3. : Extended FSM description of GBN sender

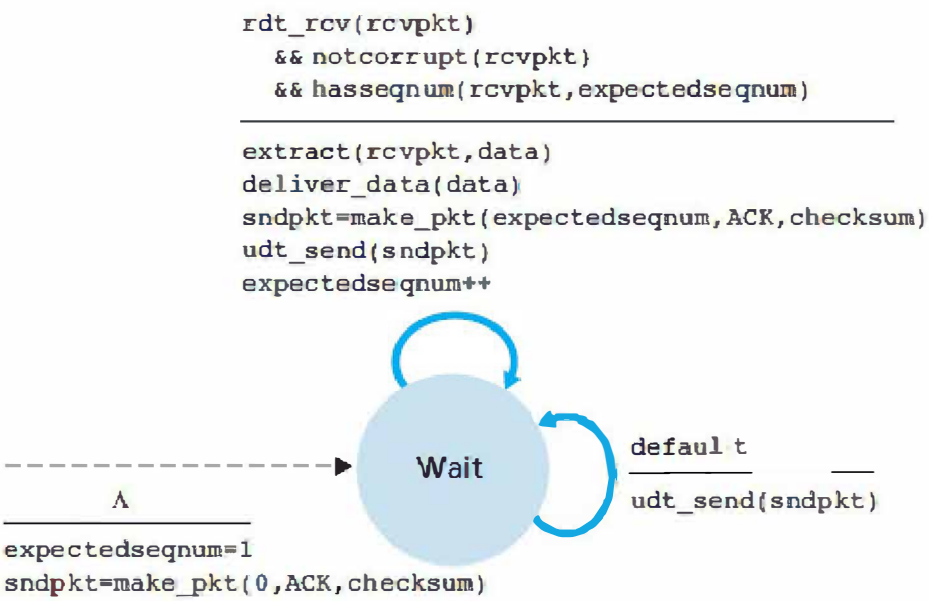


Figure 3. : Extended FSM description of GBN receiver

- The GBN sender must respond to three types of events:

  • *Invocation from above.* When rdt_send() is called from above, the sender first checks to see if the window is full, that is, whether there are *N* outstanding, unacknowledged packets. If 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.

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

  • *A timeout event.* The protocol's name, "Go-Back-N," is derived from the sender's behavior in the presence of lost or overly delayed packets.

- There will be a timer maintained 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.

- At the receiver's end, 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.

- The advantage of this approach is the simplicity of receiver buffering—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 `expectedseqnum`, shown in the receiver FSM in above figure.

- The disadvantage of throwing away 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.
- Below shown figure depicts 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).

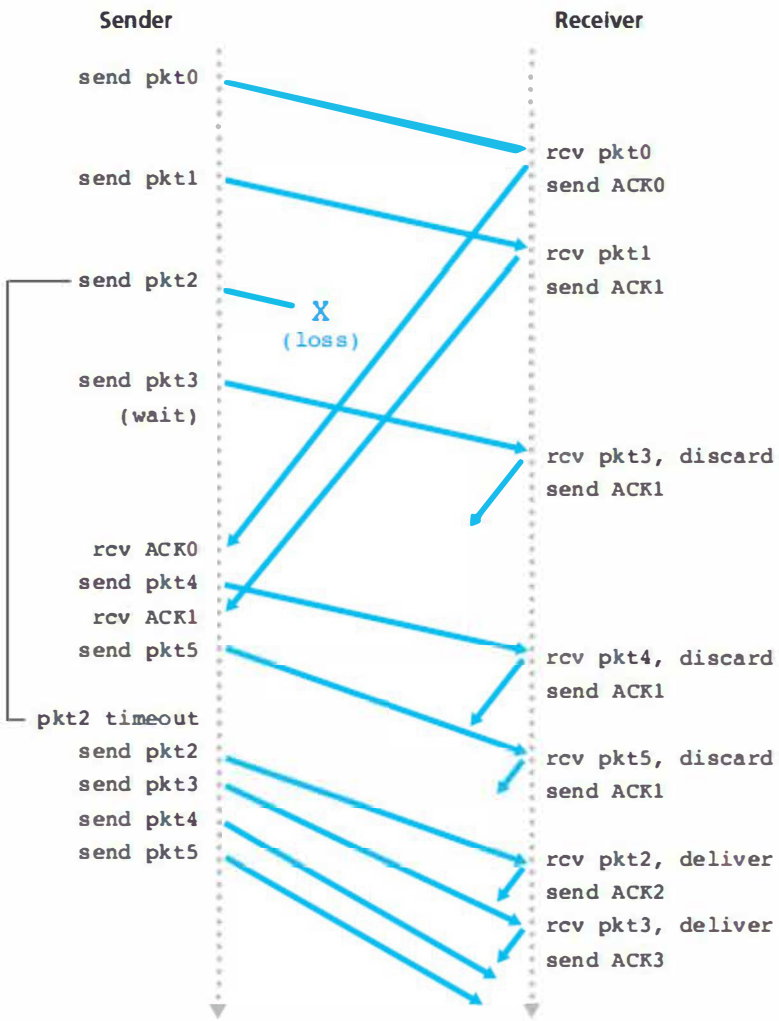
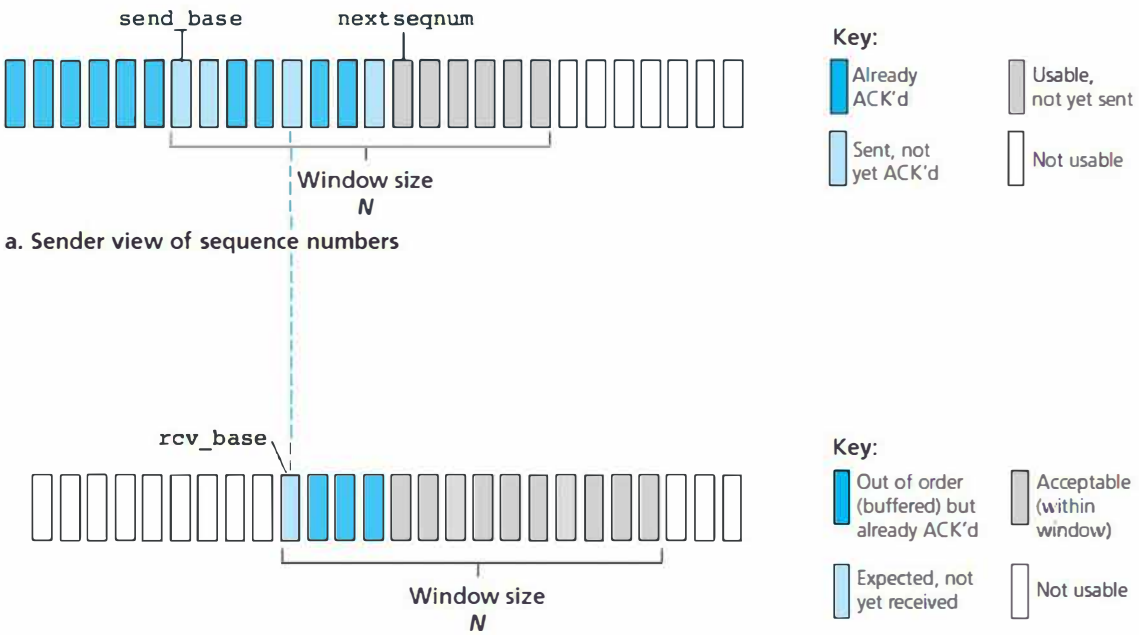


Figure 3. : Go-Back-N in operation

- 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.

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 we noted with stop-and-wait protocols.
- But, GBN also suffers from performance problems as it retransmits a large number of packets just because of a single packet error.
- As the name suggests, selective-repeat protocols avoid unnecessary retransmissions by having the sender retransmit only those packets that it suspects were received in error (that is, were lost or corrupted) at the receiver.
- Here too, a window size of N will be used to limit the number of outstanding, unacknowledged packets in the pipeline.
- However, unlike GBN, the sender will have already received ACKs for some of the packets in the window.



- There is a small difference between the window of GBN and SR that has been portrayed. Rest of the things remain same.
- Let us understand how it works. Sender transmits packets 0,1,2 but 2 is lost.(See below figure).
- Receiver acknowledges for packet 1 & 2 and continues to record further packets unlike GBN.
- For every packet, sender sets timeout for acknowledgement from the receiver as usual and acts accordingly.
- Since 2 is lost, no ACK, hence there is a timeout. This is Selective Repeat different from GBN.

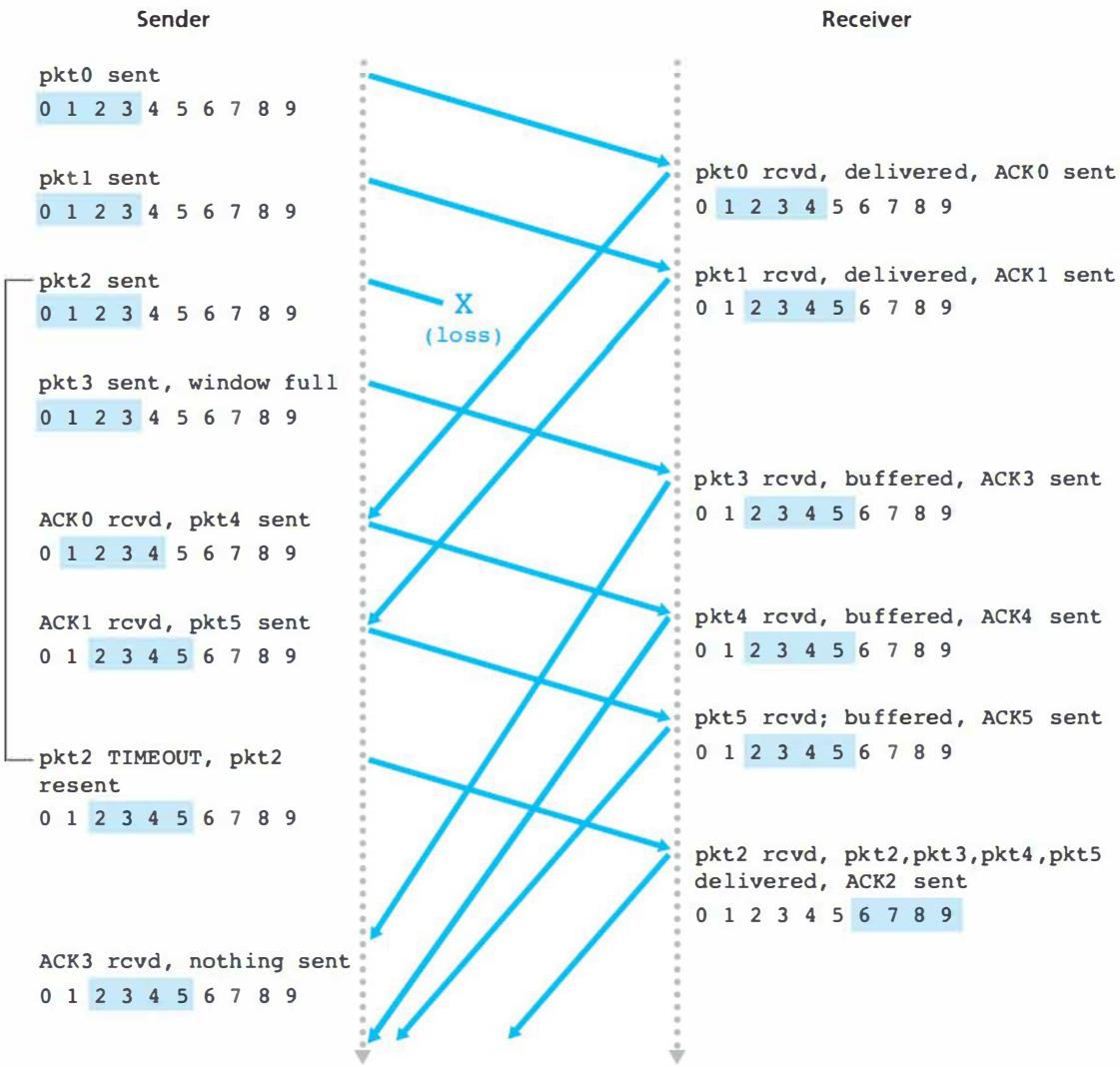


Figure 3. : SR Operation

Read to know more…

Consider what could happen, for example, with a finite range of four packet sequence numbers, 0, 1, 2, 3, and a window size of 3.

Suppose packets 0 through 2 are transmitted and correctly received and acknowledged at the receiver. At this point, the receiver's window is over the fourth, fifth, and sixth packets, which have sequence numbers 3, 0, and 1, respectively.

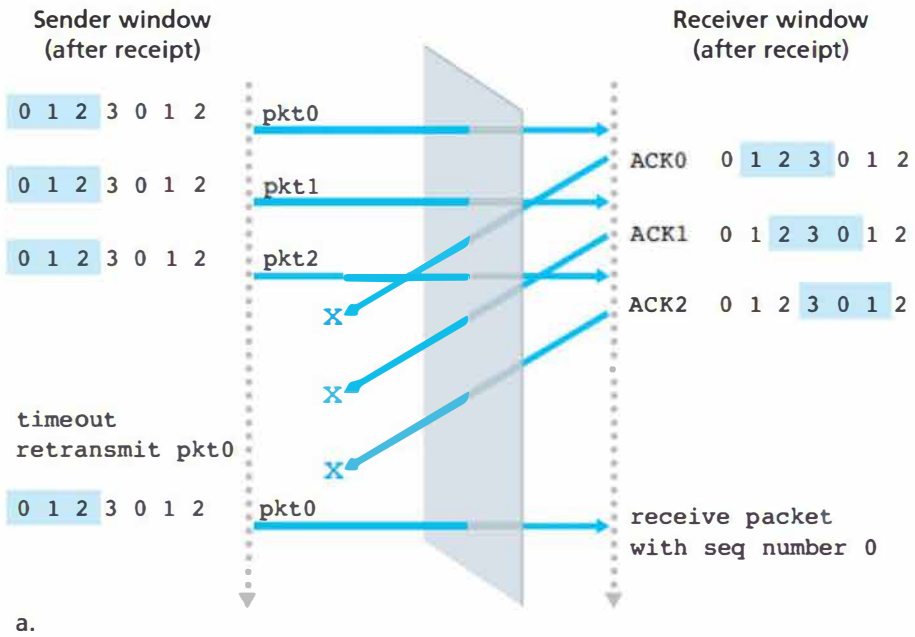
Now consider two scenarios. In the first scenario, shown in first figure (below), the ACKs for the first three packets are lost and the sender retransmits these packets. The receiver thus next receives a packet with sequence number 0—a copy of the first packet sent.

In the second scenario, shown in second figure (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.

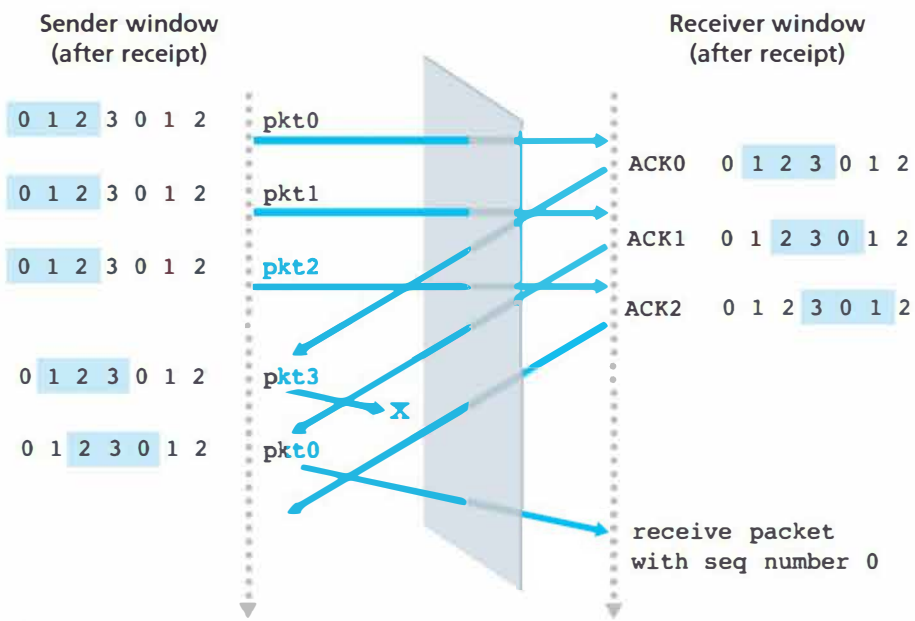
It is clear that there is no way of distinguishing the retransmission of the first packet from an original transmission of the fifth packet.

Clearly, a window size that is 1 less than the size of the sequence number space won't work. But how small must the window size be? A problem at the end of the chapter asks you to show that the

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



a.



b.

protocols

Probable Question: ***Explain Go-Back-N protocol with neat FSMs***

Explain Selective Repeat protocol with neat FSMs

3.5.1 The TCP Connection

- TCP is said to be **connection-oriented** because before one application process can begin to send data to another, the two processes must first "handshake".
- That is communicating parties must exchange control messages for connection establishment.
- As part of TCP connection establishment, both sides of the connection will initialize many TCP state variables associated with the TCP connection.
- The TCP protocol runs only in the end systems and hence the intermediate network elements do not maintain TCP connection state or related variables.
- A TCP connection provides a **full-duplex service**.
- A TCP connection is also always **point-to-point**, that is, between a single sender and a single receiver.
- Overall, to establish connection, the 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 which eventually constitutes a "Three-Way Handshake".
- The first two segments carry no payload, that is, no application-layer data.
- Once a TCP connection is established, the two application processes can send data to each other.
- Over the time, sender collects the data from send-buffer and hands it over to the network layer protocol (i.e., IP). At the receiver's end, IP puts the packets into receive-buffer and receiver collects from it.

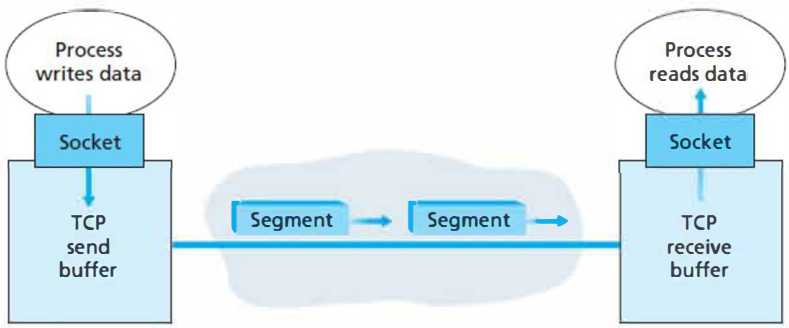
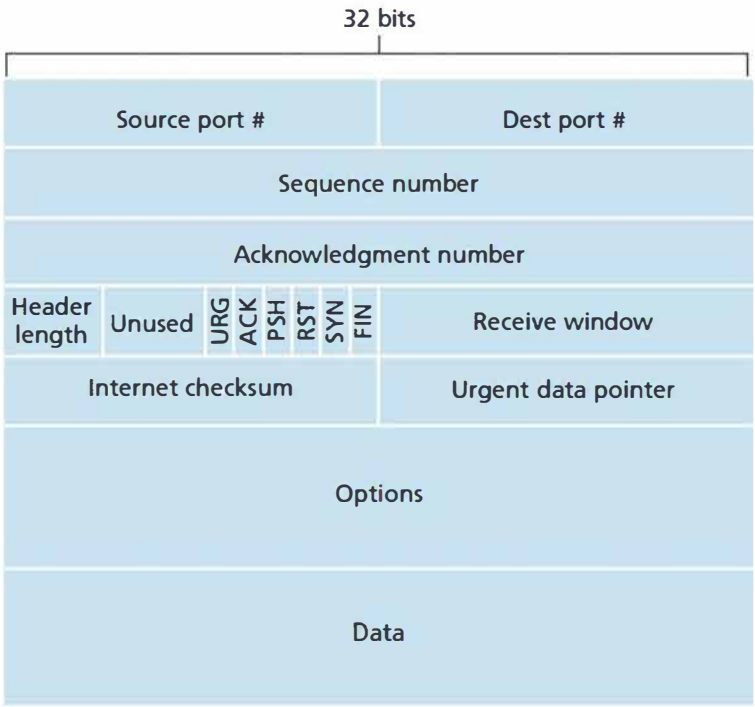


Figure 3. : TCP send and receive buffers

- Above diagram depicts the TCP based communication and also you may notice the role of sockets (discussed earlier).

3.5.2 TCP Segment Structure



- The TCP segment consists of header fields and a data field. The data field contains a chunk of application data.
- The MSS (Maximum Segment Size) limits the maximum size of a segment's data field.
- When TCP sends a large file, such as an image as part of a Web page, it typically breaks the file into chunks of size MSS (except for the last chunk, which will often be less than the MSS).
- As with UDP, the header includes source and destination port numbers, which are used for multiplexing/de-multiplexing data from/to upper-layer applications. Also, as with UDP, the header includes a checksum field. 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.

(Typically, the options field is empty, so that the length of the typical TCP header is 20 bytes.)

• 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. See RFC 854 and RFC 1323 for additional details.

• 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 bit is used to indicate whether the sender is attempting a connection for the right process in receiver or not.
- The SYN bit is used in connection establishment messages.
- The FIN bit is used for connection teardown.
- The PSH bit indicates that the receiver should pass the data to the upper layer immediately.
- The URG bit is used to indicate that there is data in this segment that the sending-side upper-layer entity has marked as "urgent." The segment with URG flag set will have high priority.

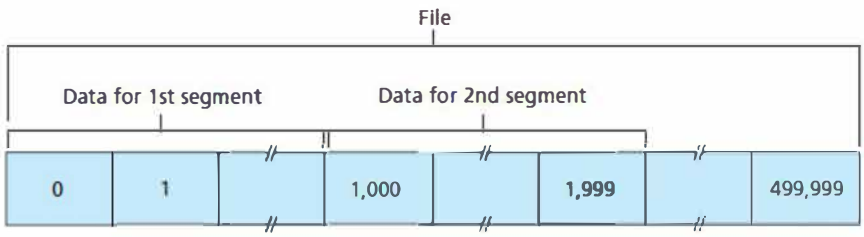
Sequence Numbers and Acknowledgment Numbers

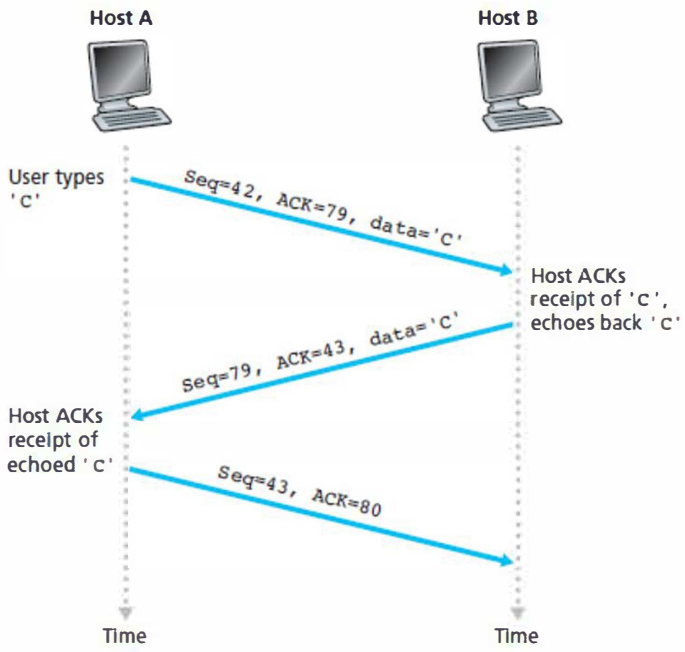
- Two of the most important fields in the TCP segment header are the sequence number field and the acknowledgment number field.
- TCP's sequence numbers are over the stream of transmitted bytes and not over the series of transmitted segments.

For example, suppose that a process in Host A wants to send a stream of data to a process in Host B over a TCP connection. The TCP in Host A will implicitly number each byte in the data stream. Suppose that the data stream consists of a file consisting of 500,000 bytes, that the MSS is 1,000 bytes, and that the first byte of the data stream is numbered 0. As shown in Figure 3.30, TCP constructs 500 segments out of the data stream. The sequence numbers will start like 0, 1000, 2000, and so on.

- Each sequence number is inserted in the sequence number field in the header of the appropriate TCP segment.
- Acknowledgment numbers are a little trickier than sequence numbers. Each of the segments that arrive from Host B has a sequence number for the data flowing from B to A.
- The acknowledgment number that Host A puts in its segment is the sequence number of the next byte Host A is expecting from Host B.

As an example, suppose that Host A has received all bytes numbered 0 through 535 from B and suppose that it is about to send a segment to Host B. Host A is waiting for byte 536 and all the subsequent bytes in Host B's data stream. So Host A puts 536 in the acknowledgment number field of the segment it sends to B. Logical view of a file split into segments with sequence numbers and sequence and acknowledgement number assignment is shown in below figures.





Suppose if few packets get lost during the communication, for the receiver, there are basically two choices: either

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

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

Probable Question: ***Explain the TCP connection***

Explain TCP Segment Structure with a neat diagram

3.5.3 Round-Trip Time Estimation and Timeout

- TCP also uses a timeout/retransmit mechanism to recover from lost segments.
- Perhaps the most obvious question is the length of the timeout intervals.
- Clearly, 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. Otherwise, unnecessary retransmissions would be sent. But how much larger?
- Let us have a variable 'SampleRTT', for recording RTT collected at time t.

- Obviously, the SampleRTT values will fluctuate from segment to segment due to congestion in the routers and to the varying load on the end systems.
- Because of this fluctuation, any given SampleRTT value may be atypical. 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:

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

- The recommended value of is $\alpha = 0.125$ (that is, 1/8). Note that EstimatedRTT is a weighted average of the SampleRTT values.
- In addition to having an estimate of the RTT, it is also valuable to have a measure of the variability of the RTT. Let us name it by 'DevRTT', as an estimate of how much SampleRTT typically deviates from EstimatedRTT:

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

- If the SampleRTT values have little fluctuation, then DevRTT will be small; on the other hand, if there is a lot of fluctuation, DevRTT will be large. The recommended value of β is 0.25.

Hence,

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

- Let's now carry out an exercise for implementing the above concept.
- Suppose that the five measured SampleRTT values are 106 ms, 120 ms, 140 ms, 90 ms, and 115 ms.

  i) Compute the EstimatedRTT after each of these SampleRTT values is obtained, Estimated RTT after each of these sample RTT value is obtained. Assume α : 0.125 and estimated RTT is 100 msec just before first of the samples obtained.

ii) Compute DevRTT , Assume $\beta = 0.25$ and DevRTT was 5 msec before first of these samples are obtained.

Calculate the EstimatedRTT after obtaining the first sample RTT=106ms**,**

EstimatedRTT = α * SampleRTT+(1- α) * EstimatedRTT

EstimatedRTT =0.125 * 106 + (1-0.125) * 100

=0.125* 106 + 0.875 * 100

=13.25 + 87.5

=100.75ms

Calculate the DevRTT after obtaining the first sample RTT:

DevRTT = β * | SampleRTT- EstimatedRTT|+(1- β)* DevRTT

=0.25 * |106-100.75| + (1-0.25) *5

=0.25 *5.25 + 0.75 * 5

=1.3125 + 3.75

=5.0625ms

Calculate the Timeout Interval after obtaining the first sample RTT:

TimeoutInterval = EstimatedRTT +4* DevRTT

= 100.75 + 4 *5.0625

=121ms

Calculate the EstimatedRTT after obtaining the second sample RTT=120ms**,**

EstimatedRTT = α * SampleRTT+(1- α) * EstimatedRTT

EstimatedRTT =0.125 * 120 + (1-0.125) * 100.75

=0.125* 120 + 0.875 * 100.75

=15 + 88.15625

=103.15625ms

Calculate the DevRTT after obtaining the second sample RTT:

DevRTT = β * | SampleRTT- EstimatedRTT|+(1- β)* DevRTT

=0.25 * |120-103.15625| + (1-0.25) *5.0625

=0.25 *16.84 + 0.75 * 5.0625

=4.21 + 3.79

=8ms

Calculate the Timeout Interval after obtaining the second sample RTT:

TimeoutInterval = EstimatedRTT +4* DevRTT

= 103.15 + 4 *8

=135.15ms

Calculate the EstimatedRTT after obtaining the third sample RTT=140ms**:**

EstimatedRTT = α * SampleRTT+(1- α) * EstimatedRTT

EstimatedRTT =0.125 * 140 + (1-0.125) * 103.15

=0.125* 140 + 0.875 * 103.15

=17.5 +90.26

=107.75ms

Calculate the DevRTT after obtaining the third sample RTT:

DevRTT = β * | SampleRTT- EstimatedRTT|+(1- β)* DevRTT

=0.25 * |140-107.75| + (1-0.25) *8

=0.25 *32.25 + 0.75 * 8

=8.06 + 6

=14.06ms

Calculate the Timeout Interval after obtaining the third sample RTT:

TimeoutInterval = EstimatedRTT +4* DevRTT

= 107.75 + 4 *14.06

=164ms

Calculate the EstimatedRTT after obtaining the fourth sample RTT=90ms**:**

EstimatedRTT = α * SampleRTT+(1- α) * EstimatedRTT

EstimatedRTT =0.125 * 90 + (1-0.125) * 107.75

=0.125* 90 + 0.875 * 107.75

=11.25 +94.28

=105.53ms

Calculate the DevRTT after obtaining the fourth sample RTT:

DevRTT = β * | SampleRTT- EstimatedRTT|+(1- β)* DevRTT

=0.25 * |90-105.53| + (1-0.25) *14.06

=0.25 *15.53 + 0.75 * 14.06

=3.88 + 10.545

=14.42ms

Calculate the Timeout Interval after obtaining the fourth sample RTT:

TimeoutInterval = EstimatedRTT +4* DevRTT

= 105.53 + 4 *14.42

=163.21ms

Calculate the EstimatedRTT after obtaining the fifth sample RTT=115ms:

EstimatedRTT = α * SampleRTT+(1- α) * EstimatedRTT

EstimatedRTT =0.125 * 115 + (1-0.125) * 105.53

=0.125* 115 + 0.875 * 105.53

=14.375 +92.34

=106.715ms

Calculate the DevRTT after obtaining the fifth sample RTT:

DevRTT = β * | SampleRTT- EstimatedRTT|+(1- β)* DevRTT

=0.25 * |115-106.715| + (1-0.25) *14.42

=0.25 *8.285 + 0.75 * 14.42

=2.07 + 10.815

=12.885ms

Calculate the Timeout Interval after obtaining the fifth sample RTT:

TimeoutInterval = EstimatedRTT +4* DevRTT

= 106.715 + 4 *12.885

=158.255ms

Probable Question: ***One problem can be expected in the examination***

3.5.4 Reliable Data Transfer

- Recall that the Internet's network-layer service (IP service) is unreliable. IP does not guarantee datagram delivery, in-order delivery of datagrams and also the integrity of the data in the datagrams.
- TCP creates a reliable data transfer service on top of IP's unreliable best effort service.

- TCP's reliable data transfer service ensures that the data stream that a process reads out of its TCP receive buffer is free from all types of errors.
- TCP accomplishes this with a single-timer based retransmission.

Let's 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.

Below pseudo code presents a highly simplified description of a TCP sender. There are three major events related to data transmission and retransmission in the TCP sender: data received from application above; timer timeout; and ACK receipt.

/* 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)

event1: 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;

event2: timer timeout

        retransmit not-yet-acknowledged segment with smallest sequence number

        start timer
```

```
        break;
event3: 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 */
```

The TCP state variable SendBase is the sequence number of the oldest unacknowledged byte. (Thus SendBase–1 is the sequence number of the last byte that is known to have been received correctly and in order at the receiver).

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.

The above shown sender module has undergone few modifications and are as follows:

Doubling the Timeout Interval

- The first concerns the length of the timeout interval after a timer expiration. In this modification, 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.

- This modification provides a limited form of congestion control. 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. Fortunately, the sender can often detect packet loss well before the timeout event occurs by noting so-called duplicate ACKs. 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 below (Only third event is modified).

```
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;
```

Probable Question: ***Explain TCP reliable data transfer. Give two methods to improvise it***

3.5.5 Flow Control

- When the TCP connection receives bytes that are correct and in sequence, it places the data in the receive buffer.
- The associated application process will read data either instantly or at a later stage.
- 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 provides flow control by having the ***sender*** maintain a variable called the **receive window**. Informally, the receive window is used to give the sender an idea of how much free buffer space is available at the receiver.
- Suppose that Host A is sending a large file to Host B over a TCP connection. Host B allocates a receive buffer to this connection; denote its size by **RcvBuffer**. From time to time, the application process in Host B reads from the buffer.

Receiver maintains 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, below equation must be satisfied

$$LastByteRcvd - LastByteRead <= RcvBuffer$$

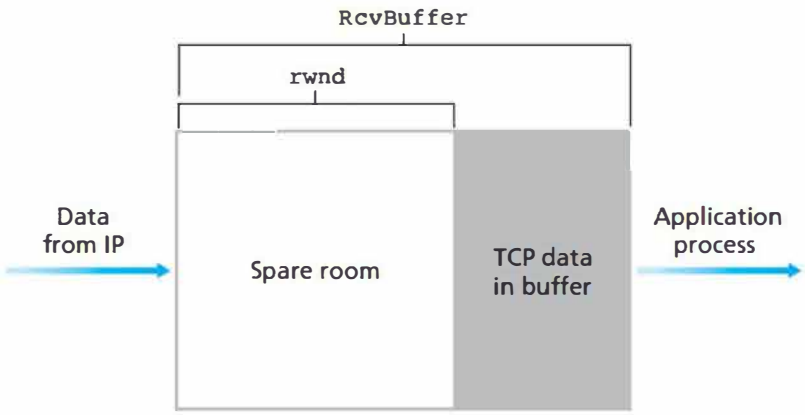
The receive window, denoted rwnd is set to the amount of spare room in the buffer:

$$rwnd = RcvBuffer - [LastByteRcvd - LastByteRead]$$

Because the spare room changes with time, rwnd is dynamic.

How does the connection use the variable rwnd to provide the flow-control service?

Take a look at the RcvBuffer:



- Host B tells Host A how much spare room it has in the connection buffer by placing its current value of **rwnd** in the receive window field of every segment it sends to A.

Initially, Host B sets rwnd = RcvBuffer.

- Host A in turn keeps track of two variables, **LastByteSent** and **LastByteAcked**, which have obvious meanings.
- Note that the difference between these two variables, LastByteSent – LastByteAcked, is the amount of unacknowledged data that A has sent into the connection.

- By keeping the amount of unacknowledged data less than the value of **rwnd**, Host A is assured that it is not overflowing the receive buffer at Host B. Thus, Host A makes sure throughout the connection's life that

$$\text{LastByteSent} - \text{LastByteAcked} <= \text{rwnd}$$

There is one minor technical 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. Now consider what happens. As the application process at B empties the buffer, TCP does not send new segments with new rwnd values to Host A; indeed, TCP sends a segment to Host A only if it has data to send or if it has an acknowledgment to send. Therefore, Host A is never informed that some space has opened up in Host B's receive buffer—Host A is blocked and can transmit no more data!

Solution: 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.

Probable Question: ***Explain Flow Control along with hurdles faced and resolution made***

3.5.6 TCP Connection Management

- Let us take a closer look at how a TCP connection is established and torn down.
- Let's first take a look at how a TCP connection is established. 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 by setting SYN flag bit (one of the flag bits in the segment's header) to 1. In addition, the client randomly chooses an initial sequence number (client_isn) and puts this number in the sequence number field of the initial TCP SYN segment.

• *Step 2.* Once the 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 contains three important pieces of information in the segment header.

a) SYN bit is set to 1.

b) Acknowledgment field of the TCP segment header is set to **client_isn+1.**

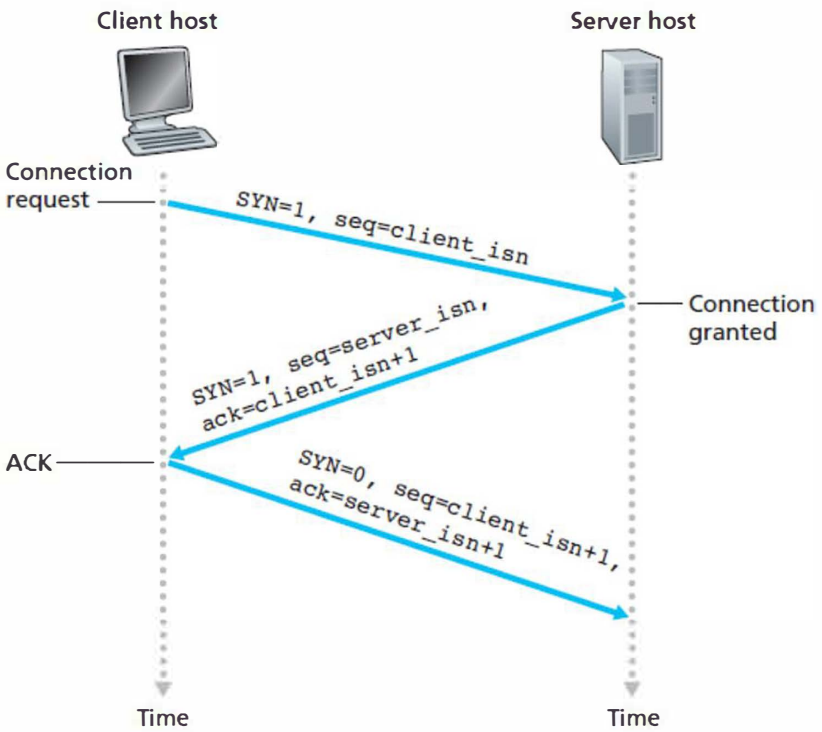
c) The server choosen its own initial sequence number **(server_isn)**

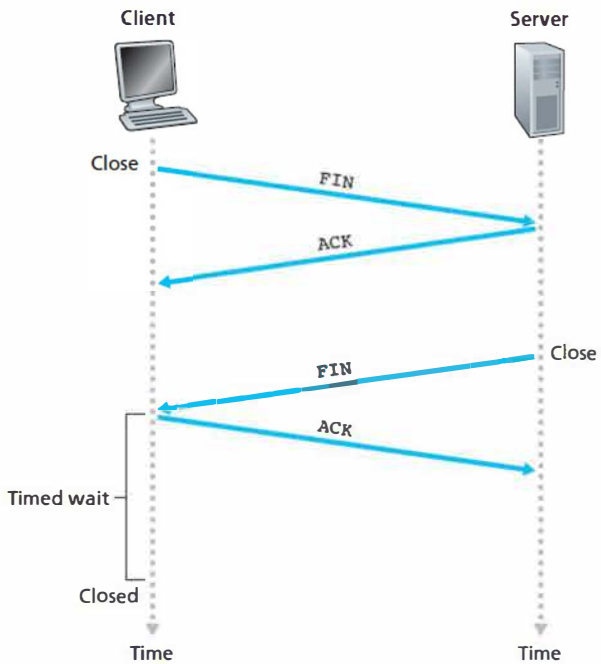
This connection granted segment is referred to as a **SYNACK segment**.

The above two segments do not contain application layer data.

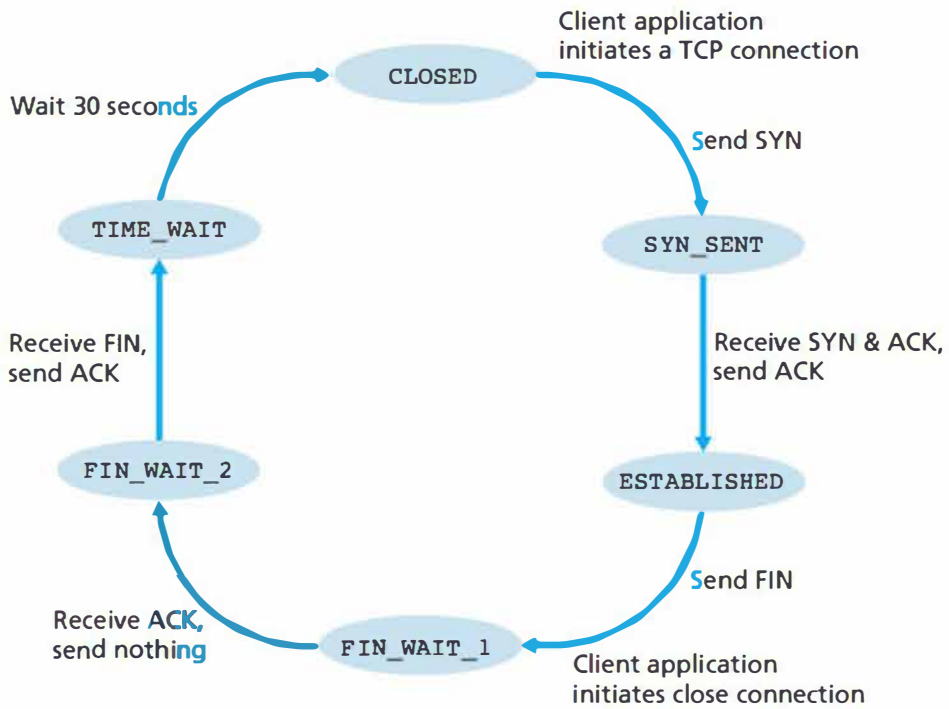
• *Step 3.* Upon receiving the SYNACK segment, the client also allocates buffers and variables to the connection. The client host then sends the server yet another segment; this last segment acknowledges the server's connection-granted segment with ACK = **server_isn+1** in the acknowledgment field of the TCP segment header.The SYN bit is set to zero, since the connection is established.

- Henceforth, the SYN bit will be set to zero.
- Either of the two processes participating in a TCP connection can end the connection by setting FIN bit to 1 in the TCP header.
- 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.
- The below shown two figures give a pictorial explanation of the process.





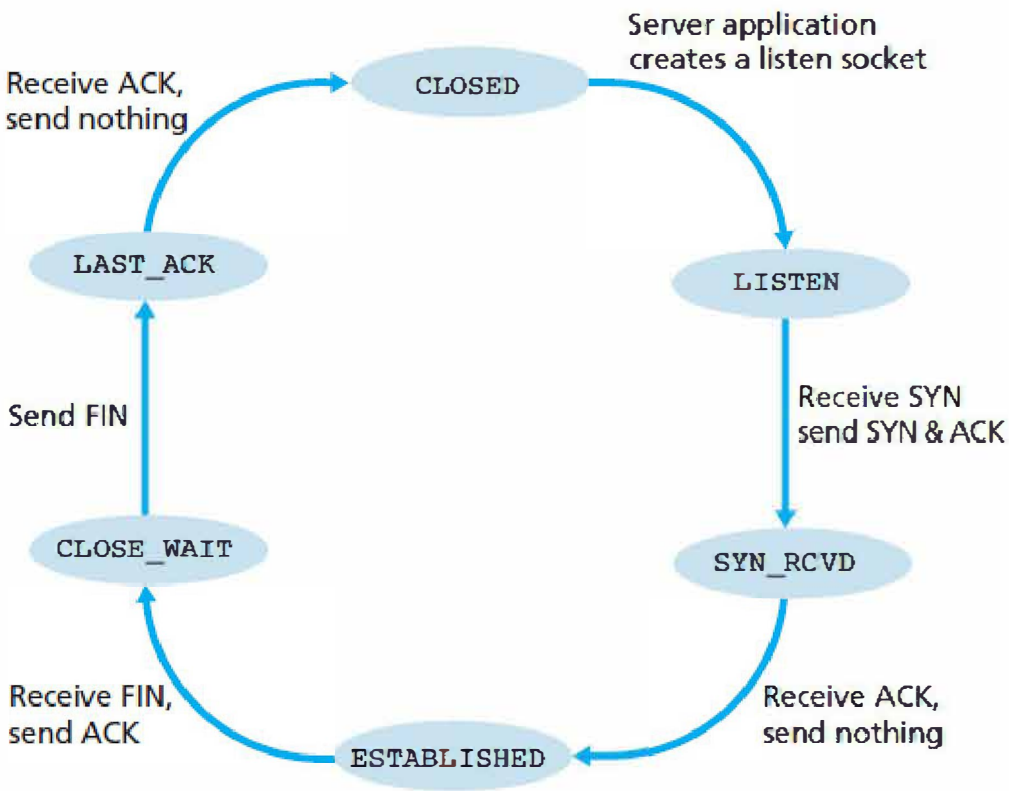
Below figure illustrates a typical sequence of TCP states that are visited by the **client** TCP.



- 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 and waits for an acknowledgement from the server for its previous segment.
- The client TCP enters the ESTABLISHED state after receiving the ACK. In this state, the TCP client can send and receive TCP segments containing payload (that is, application-generated) data
- The client sends a TCP segment by setting FIN bit to close the connection after it is done and enters the FIN_WAIT_1 state, waiting a segment from the server.
- When it receives this segment, the client TCP enters the FIN_WAIT_2 state and waits for another segment from the server with the FIN bit set to 1.

- Upon 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.

At last, the connection will get terminated by both the ends.



Above diagram shows how state transition takes place at the TCP server's end. I hope this is self-explanatory.

Let's consider what happens when 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 (that is, it is not running a Web server 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, it is telling the source "I don't have a socket for that segment. Please do not resend the segment."

When a host receives a UDP packet whose destination port number doesn't match with an ongoing UDP socket, the host sends a special ICMP datagram.

Probable Question: ***Explain how the TCP client establishes a TCP connection with a TCP Server***

3.6 Principles of Congestion Control

- Packet retransmission thus treats a symptom of network congestion (the loss of a specific transport-layer segment) but does not treat the cause of network congestion—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.
- In this section, we consider the problem of congestion control in a general context, seeking to understand why congestion is a bad thing, how network congestion is manifested in the performance received by upper-layer applications, and various approaches that can be taken to avoid, or react to, network congestion.

3.6.1 The Causes and the Costs of Congestion

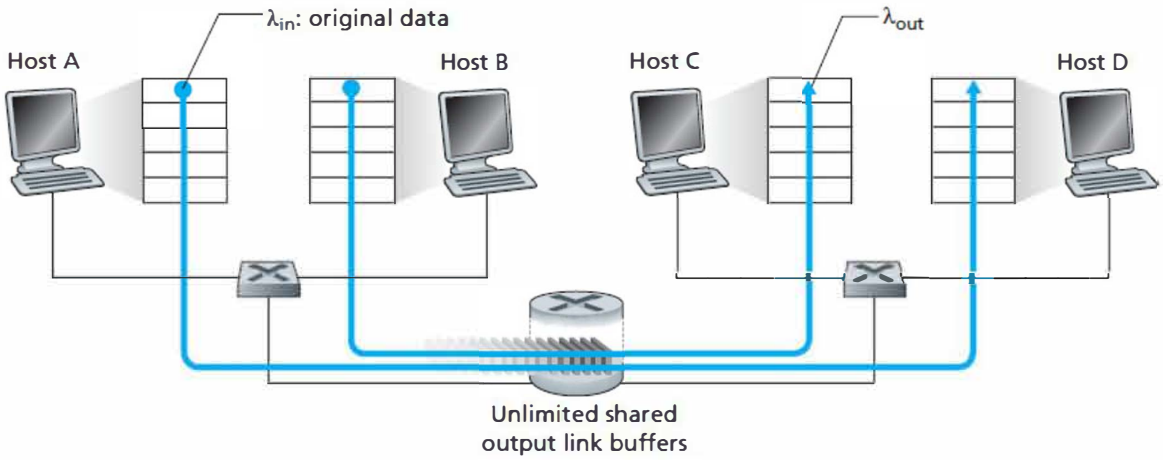
Let's begin our general study of congestion control by examining three increasingly complex scenarios in which congestion occurs.

Scenario 1: Two Senders, a Router with Infinite Buffers

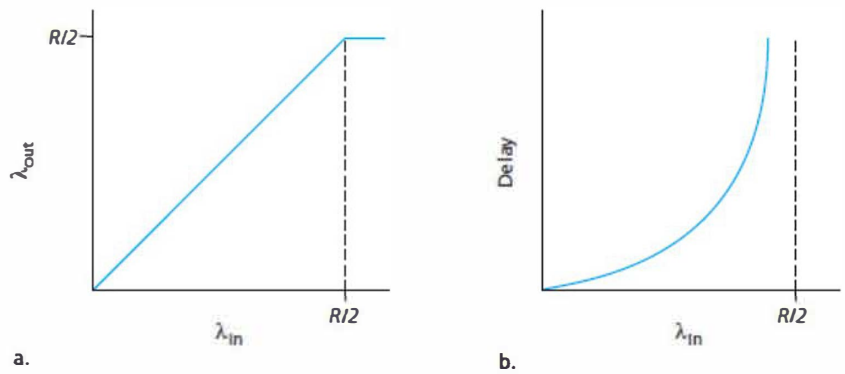
- We begin by considering perhaps the simplest congestion scenario possible: Two hosts (A and B) each have a connection that shares a single hop between source and destination, as shown in Figure below.
- Let's assume that the application in Host A is sending data into the connection (for example, passing data to the transport-level protocol via a socket) 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.



- 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 scenario, we assume that the router has an infinite amount of buffer space.
- Below given figure plots the performance of Host A's connection under this first scenario.

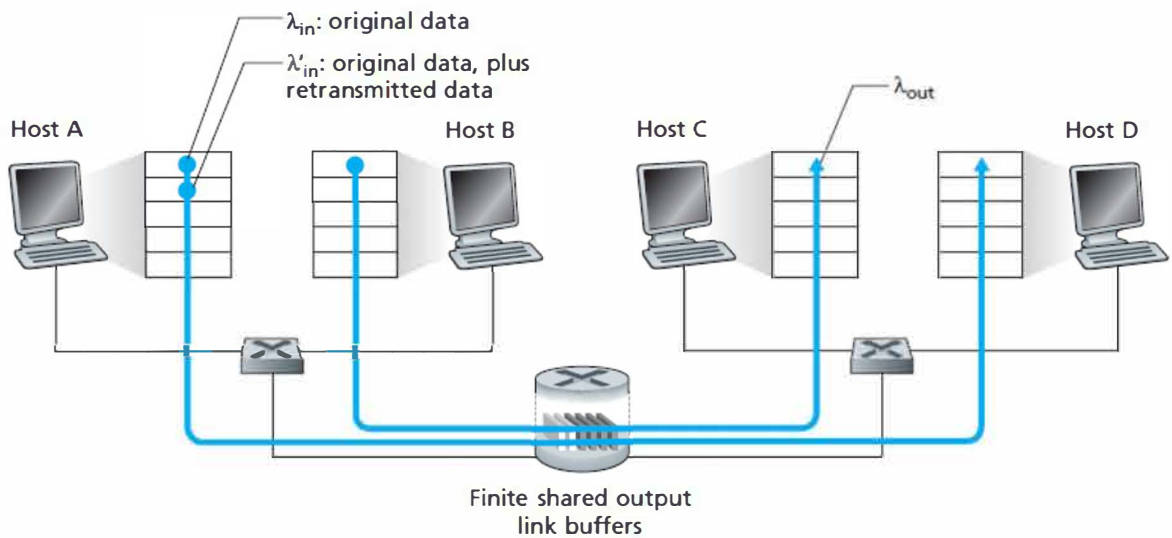


- The left graph plots the per-connection throughput (number of bytes per second at the receiver) as a function of the connection-sending rate.
- For a sending rate between 0 and $R/2$, the throughput at the receiver equals the sender's sending rate—everything sent by the sender is received at the receiver 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 $R/2$.
- No matter how high Hosts A and B set their sending rates, they will each never see a throughput higher than $R/2$.
- The right-hand graph in above figure, however, 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

Conclusion: *Even in this (extremely) idealized scenario, we've already found one cost of a congested network—large queuing delays are experienced as the packet arrival rate nears the link capacity.*

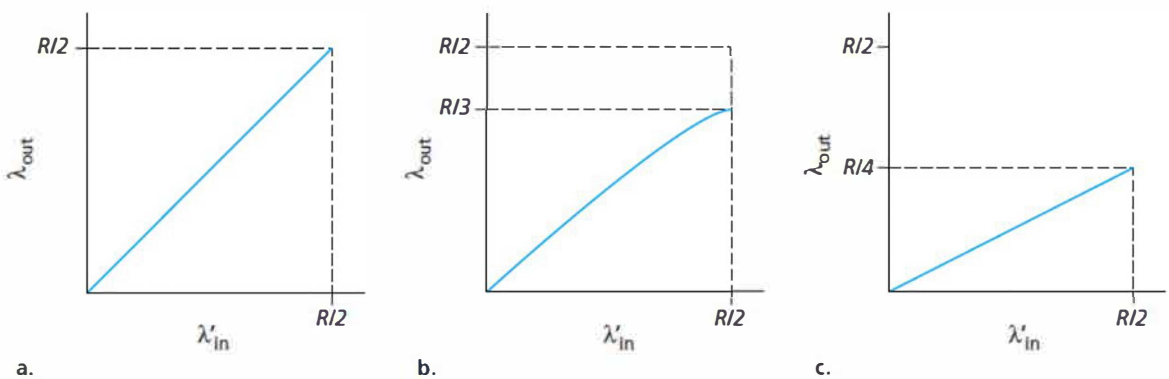
Scenario 2: Two Senders and a Router with Finite Buffers

- First, the amount of router buffering is assumed to be finite. A consequence of this real-world assumption is that packets will be dropped when arriving to an already full buffer.
- Second, we assume that each connection is reliable.
- If a packet containing a transport-level segment is dropped at the router, the sender will eventually retransmit it.
- Because packets can be retransmitted, we must now be more careful with our use of the term sending rate.
- Specifically, let us again denote the rate at which the application sends original data into the socket by λ_{in} bytes/sec.
- The rate at which the transport layer sends segments (containing original data and retransmitted data) into the network will be denoted λ'_{in} in bytes/sec.
- λ'_{in} in is sometimes referred to as the offered load to the network.
- First, consider the unrealistic case that Host A is able to somehow (magically!) 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 below figure a. From a throughput standpoint, performance is ideal—everything that is sent is received.
- Note that the average host sending rate cannot exceed R/2 under this scenario, since packet loss is assumed never to occur.
- Consider next the slightly more realistic case that the sender retransmits only when a packet is known for certain to be lost.
- In this case, the performance might look something like that shown in Figure b.
- Consider the case that the offered load, 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 0.5R units of data transmitted, 0.333R bytes/sec (on average) are original data and 0.166R bytes/sec (on average) are retransmitted data.
- We see here another cost of a congested network— the sender must perform retransmissions in order to compensate for dropped (lost) packets due to buffer overflow.
- Finally, let us 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.



- Of course, the receiver needs but one copy of this packet and will discard the retransmission.

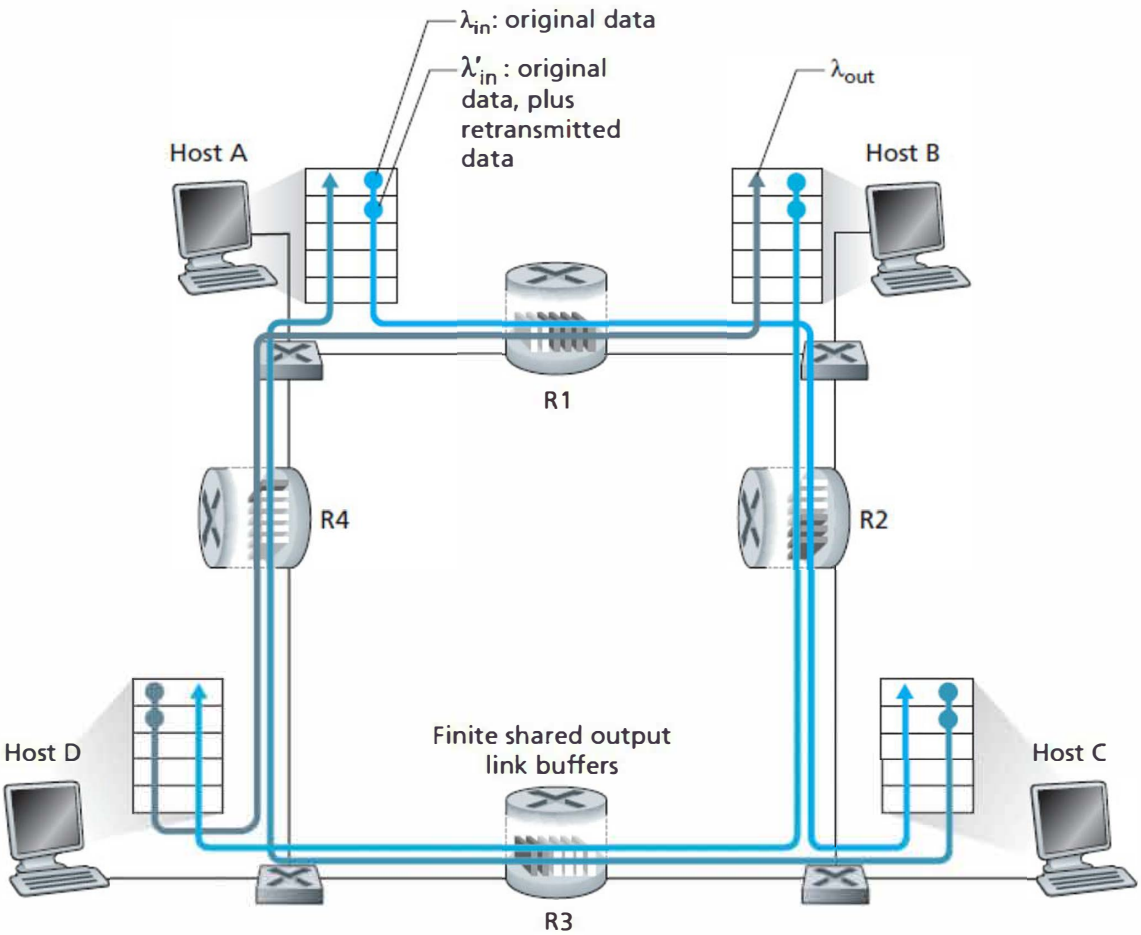
- In this case, the work done by the router in forwarding the retransmitted copy of the original packet was wasted, as the receiver will have already received the original copy of this packet.
- The router would have better used the link transmission capacity to send a different packet instead.
- Here then is yet another cost of a congested network—unneeded retransmissions by the sender in the face of large delays may cause a router to use its link bandwidth to forward unneeded copies of a packet.
- Figure (c) shows the throughput versus offered load when each packet is assumed to be forwarded (on average) twice by the router.

Conclusion: *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 our final congestion scenario, four hosts transmit packets, each over overlapping two-hop paths, as shown in Figure below.
- We again assume that each host uses a timeout/retransmission mechanism to implement a reliable data transfer service, that all hosts have the same value of in, and that all router links have capacity R bytes/sec.
- Let's 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 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 λ_{in}, an increase in in results in an increase in λ_{out}.

- Having considered the case of extremely low traffic, let's next examine the case that in (and hence in) 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 λ_{in}.



- If λ_{in} 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.
- In the high-traffic scenario outlined above, 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 network would have been equally well off (more accurately, equally bad off) if the first router had simply discarded that packet and remained idle.

Conclusion: *So here we see yet another cost of dropping a packet due to congestion— when a packet is dropped along a path, the transmission capacity that was used at each of the upstream links to forward that packet to the point at which it is dropped ends up having been wasted.*

3.6.2 Approaches to Congestion Control

- Here, we identify the two broad approaches to congestion control that are taken in practice and discuss specific network architectures and congestion-control protocols embodying these approaches.
- At the broadest level, we can distinguish among congestion-control approaches by whether the network layer provides any explicit assistance to the transport layer for congestion-control purposes:

  • *End-to-end congestion control*. In an end-to-end approach to congestion control, the network layer provides no explicit support to the transport layer for 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, packet loss and delay).

  TCP segment loss (as indicated by a timeout or a triple duplicate acknowledgment) is taken as an indication of network congestion and TCP decreases its window size accordingly.

• *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. 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.

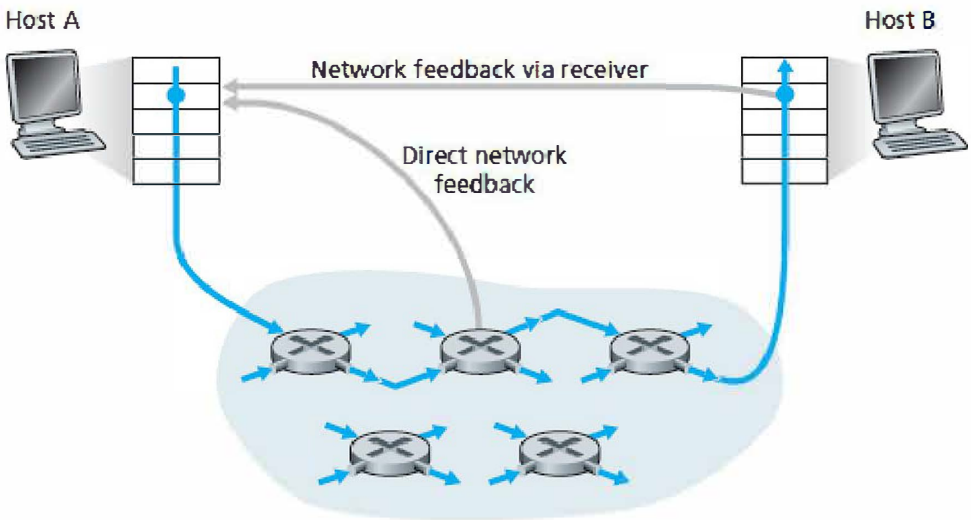
- Direct feedback may be sent from a network router to the sender. This form of notification typically takes the form of a choke packet (essentially saying, "I'm congested!").
- The second form of notification occurs when a router marks/updates a field in a packet flowing from sender to receiver to indicate congestion.
- Upon receipt of a marked packet, the receiver then notifies the sender of the congestion indication.

Probable Question: ***Explain TCP congestion control mechanism by taking any one relevant Scenario***

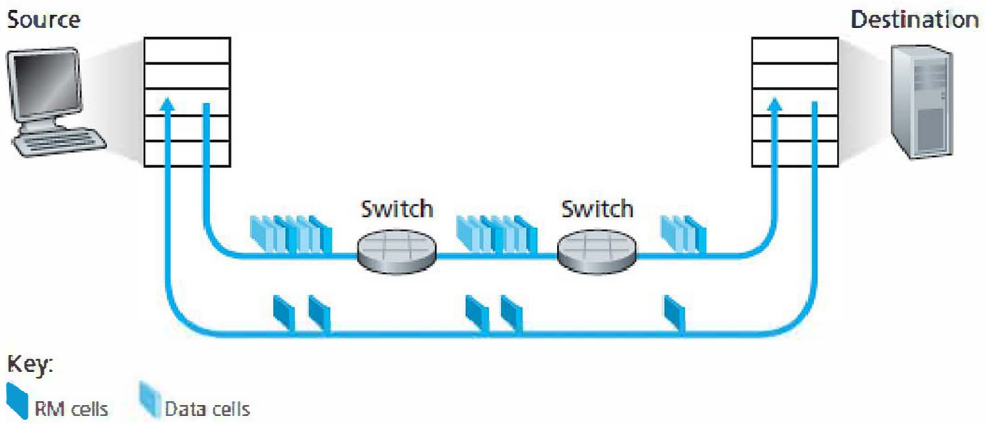
Explain various approaches to Congestion control under TCP

3.6.3 Network-Assisted Congestion-Control Example: ATM ABR Congestion Control

- Let us understand the congestion-control algorithm in ATM ABR—a protocol that takes a network-assisted approach toward congestion control.
- Fundamentally ATM takes a virtual-circuit (VC) oriented approach toward packet switching.
- This means that each switch on the source-to-destination path will maintain state about the source-to-destination VC.



- This per-VC state allows a switch to track the behavior of individual senders (e.g., tracking their average transmission rate) and to take source-specific congestion-control actions (such as explicitly signaling to the sender to reduce its rate when the switch becomes congested).
- This per-VC state at network switches makes ATM ideally suited to perform network-assisted congestion control.
- ABR has been designed as an elastic data transfer service in a manner reminiscent of TCP.
- When the network is underloaded, ABR service should be able to take advantage of the spare available bandwidth; when the network is congested, ABR service should throttle its transmission rate to some predetermined minimum transmission rate.
- Figure shows the framework for ATM ABR congestion control.
- With ATM ABR service, data cells are transmitted from a source to a destination through a series of intermediate switches.
- Interspersed with the data cells are resource-management cells (RM cells); these RM cells can be used to convey congestion-related information among the hosts and switches.



- When an RM cell arrives at a destination, it will be turned around and sent back to the sender (possibly after the destination has modified the contents of the RM cell).
- It is also possible for a switch to generate an RM cell itself and send this RM cell directly to a source.
- RM cells can thus be used to provide both direct network feedback and network feedback via the receiver.
- ATM ABR congestion control is a rate-based approach. That is, the sender explicitly computes a maximum rate at which it can send and regulates itself accordingly.
- ABR provides three mechanisms for signaling congestion-related information from the switches to the receiver:

  • *EFCI bit.* Each data cell contains an explicit forward congestion indication (EFCI) bit. A congested network switch can set the EFCI bit in a data cell to 1 to signal congestion to the destination host. The destination must check the EFCI bit in all received data cells. When an RM cell arrives at the destination, if the most recently received data cell had the EFCI bit set to 1, then the destination sets the congestion indication bit (the CI bit) of the RM cell to 1 and sends the RM cell back to the sender. Using the EFCI in data cells and the CI bit in RM cells, a sender can thus be notified about congestion at a network switch.

  • *CI and NI bits*. Sender-to-receiver RM cells are interspersed with data cells. The rate of RM cell interspersion is a tunable parameter, with the default value being one RM cell every 32

data cells. These RM cells have a congestion indication (CI) bit and a no increase (NI) bit that can be set by a congested network switch. Specifically, a switch can set the NI bit in a passing RM cell to 1 under mild congestion and can set the CI bit to 1 under severe congestion conditions. When a destination host receives an RM cell, it will send the RM cell back to the sender with its CI and NI bits intact.

• *ER setting*. Each RM cell also contains a 2-byte explicit rate (ER) field. A congested switch may lower the value contained in the ER field in a passing RM cell. In this manner, the ER field will be set to the minimum supportable rate of all switches on the source-to-destination path.

Probable Question: ***Explain ATM ABR Congestion Control approach***

3.7 TCP Congestion Control

- TCP implements end-to-end congestion control rather than network-assisted congestion control, since the IP layer provides no explicit feedback to the end systems regarding network congestion.
- TCP sender's behavior changes as the intensity of congestion varies i.e., if a TCP sender perceives that there is little congestion on the path between itself and the destination, then the TCP sender increases its send rate; if the sender perceives that there is congestion along the path, then the sender reduces its send rate.
- But this approach raises three questions.
  1) How does a TCP sender limit the rate at which it sends traffic into its connection?
  2) How does a TCP sender perceive that there is congestion on the path between itself and the destination?
  3) What algorithm should the sender use to change its send rate as a function of perceived end-to-end congestion?

Solutions

1) Answering the first, the TCP congestion-control mechanism operating at the sender keeps track of an additional variable, the **congestion window** denoted as **cwnd**, imposes a constraint on the rate at which a TCP sender can send traffic into the network.

Specifically, the amount of unacknowledged data at a sender may not exceed the minimum of **cwnd** and **rwnd**, that is:

LastByteSent – LastByteAcked <= min{cwnd, rwnd}

If rwnd is assumed as a large size buffer, then we simply have

LastByteSent – LastByteAcked <= cwnd

Consider a connection for which loss and packet transmission delays are negligible.

At the very beginning, sender sends data over a given RTT value. Thus the sender's send rate is roughly **cwnd/RTT** bytes/sec.

As cwnd varies, transmission rate varies.

2) Now, answering the second question.

Let us define a "loss event" at a TCP sender as the occurrence of either a timeout or the receipt of three duplicate ACKs from the receiver. When there is excessive congestion, then one (or more) router buffers along the path overflows, causing a datagram (containing a TCP segment) to be dropped. The dropped datagram, in turn, results in a loss event at the sender—either a timeout or the receipt of three duplicate ACKs—which is taken by the sender to be an indication of congestion on the sender-to-receiver path.

Because TCP uses acknowledgments to trigger (or clock) its increase in congestion window size, TCP is said to be **self-clocking**.

3) Finally, answering the third question.

If TCP senders are too cautious and send too slowly, they could under utilize the bandwidth in the network.

Following guiding principles can formulate an answer for this question:

• *A lost segment implies congestion, and hence, the TCP sender's rate should be decreased when a segment is lost.*

• *An acknowledged segment indicates that the network is delivering the sender's segments to the receiver, and hence, the sender's rate can be increased when an ACK arrives for a previously unacknowledged segment.*

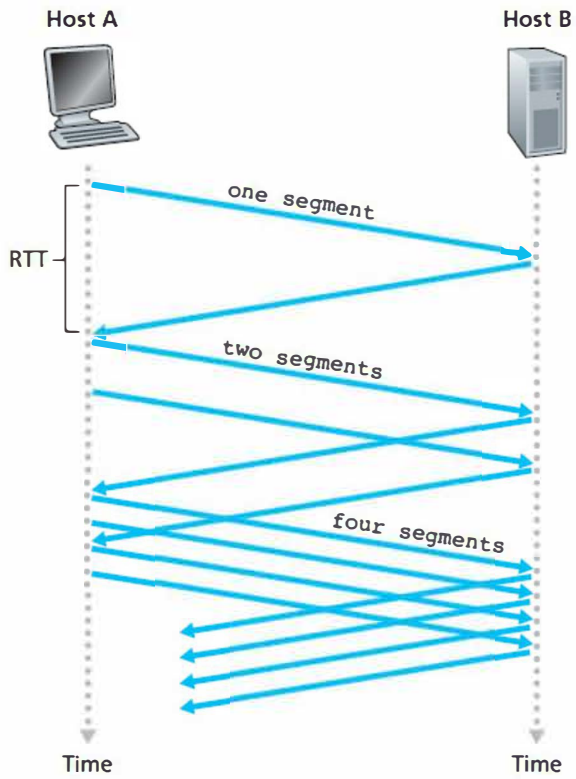
• *Bandwidth probing*. Segment/ ACK loss indicates a congested path and hassle-free delivery of segments/ ACK indicates congestion-free path. Bandwidth probing is checking the current status of bandwidth value and trying to increase transmission rate accordingly until there is a "loss event".

Given this overview of TCP congestion control, let us understand **TCP congestion-control algorithm**, a standard proposed in this regard. The algorithm has three major components: (1) slow start, (2) congestion avoidance, and (3) fast recovery.

Slow Start

When a TCP connection begins, the value of **cwnd** is typically initialized to a small value of 1 MSS, resulting in an initial sending rate of roughly MSS/RTT without applying the bandwidth probing. For example, if MSS = 500 bytes and RTT = 200 msec, the resulting initial sending rate is only about 20 kbps.

Thus, in the slow-start state, the value of cwnd begins at 1 MSS and increases by 1 MSS every time a transmitted segment is first acknowledged as shown in below figure.



When this acknowledgment arrives, the TCP sender increases the congestion window by one MSS and sends out two maximum-sized segments. These segments are then acknowledged, with the sender increasing the congestion window by 1 MSS for each of the acknowledged segments, giving a congestion window of 4 MSS, and so on. This process results in a doubling of the sending rate every RTT. Thus, the TCP send rate starts slow but grows exponentially during the slow start phase.

But when should this exponential growth end?

If there is a loss event (i.e., congestion) indicated by a timeout, the TCP sender sets the value of **cwnd** to 1 and begins the slow start process anew.

It also sets the value of a second state variable, **ssthresh** (shorthand for "slow start threshold") to cwnd/2—half of the value of the congestion window value when congestion was detected.

Thus when the value of **cwnd** equals **ssthresh,** slow start ends and TCP transitions into congestion avoidance mode.

Congestion Avoidance

On entry to the congestion-avoidance state, the value of cwnd is approximately half its value when congestion was last encountered—congestion could be just around the corner! Thus, TCP increases the value of **cwnd** by just a single MSS every RTT.

This can be accomplished in several ways. A common approach is for the TCP sender to increase **cwnd** by MSS bytes **(MSS/cwnd)** whenever a new acknowledgment arrives.

But when should congestion avoidance's linear increase (of 1 MSS per RTT) end?

TCP's congestion-avoidance algorithm behaves the same when a timeout occurs. The value of **cwnd** is set to 1 MSS, and the value of **ssthresh** is updated to half the value of **cwnd** when the loss event occurred. However, that a loss event also can be triggered by a triple duplicate ACK event.

Fast Recovery

In fast recovery, the value of **cwnd** is increased by 1 MSS for every duplicate ACK received for the missing segment that caused TCP to enter the fast-recovery state.

Eventually, either of the two things happen:

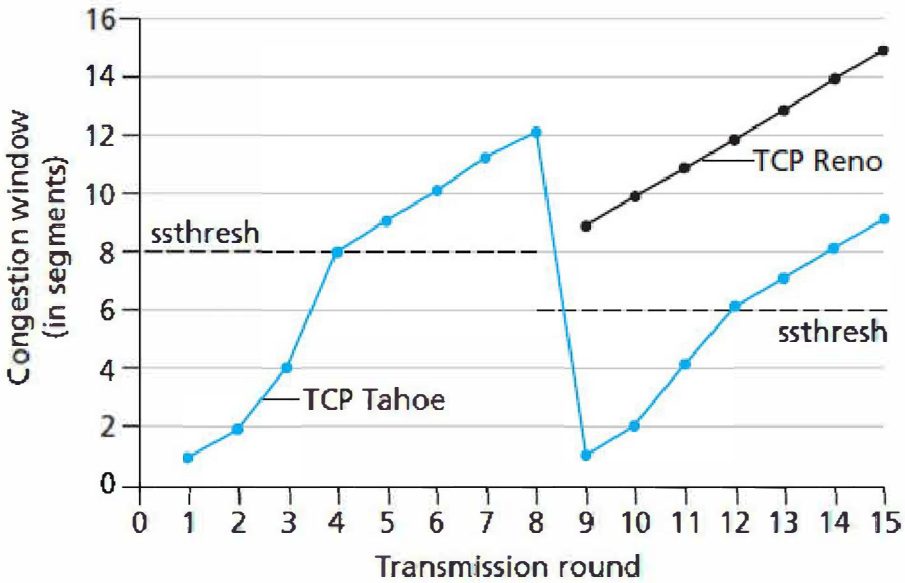
a) When an ACK arrives for the missing segment, TCP enters the congestion-avoidance state after deflating **cwnd**.

b) If a timeout event occurs, fast recovery transitions to the slow-start state after performing the same actions as in slow start and congestion avoidance: The value of cwnd is set to 1 MSS, and the value of ssthresh is set to half the value of cwnd when the loss event occurred.

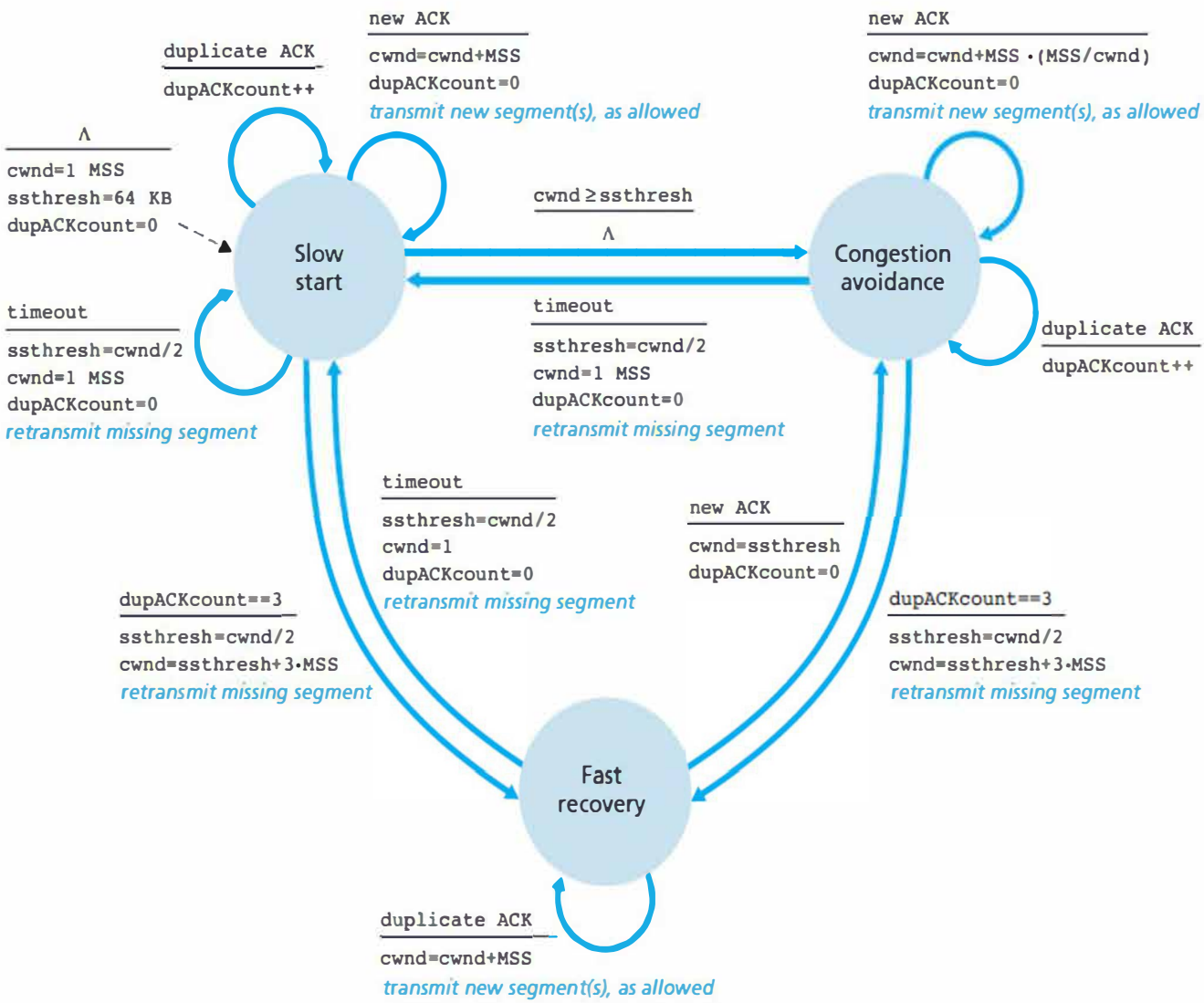
There two versions pertaining to the implementation of TCP: **TCP Tahoe & TCP Reno**

It is interesting that an early version of TCP, known as TCP Tahoe, unconditionally cut its congestion window to 1 MSS and entered the slow-start phase after either a timeout-indicated or triple-duplicate-ACK-indicated loss event. The other version, TCP Reno, incorporated fast recovery.

Below shown figure illustrates how these two versions carry out fast recovery phase.



The TCP Congestion Control algorithm has been modelled using FSM model and is shown as below.



Probable Question: ***Explain TCP congestion control algorithm along with its various phases***