MODULE 2: TRANSPORT LAYER

MODULE 2: TRANSPORT LAYER

2.1 Introduction and Transport Layer Services

- A transport-layer protocol provides logical-communication b/w application-processes running on different hosts.
- Transport-layer protocols are implemented in the end-systems but not in network-routers.
- On the sender, the transport-layer
  - → receives messages from an application-process
  - → converts the messages into the segments and
  - → passes the segment to the network-layer.
- On the receiver, the transport-layer
  - → receives the segment from the network-layer
  - → converts the segments into to the messages and
  - → passes the messages to the application-process.
- The Internet has 2 transport-layer protocols: TCP and UDP

2.1.1 Relationship between Transport and Network Layers

- A transport-layer protocol provides logical-communication b/w processes running on different hosts.

  Whereas, a network-layer protocol provides logical-communication between hosts.
- Transport-layer protocols are implemented in the end-systems but not in network-routers.
- Within an end-system, a transport protocol
  - → moves messages from application-processes to the network-layer and vice versa.
  - → but doesn't say anything about how the messages are moved within the network-core.
- The routers do not recognize any info. which is appended to the messages by the transport-layer.

2.1.2 Overview of the Transport Layer in the Internet

- When designing a network-application, we must choose either TCP or UDP as transport protocol.

  1) UDP (User Datagram Protocol)
  - ➢ UDP provides a connectionless service to the invoking application.
  - ➢ The UDP provides following 2 services:
    - **i)** Process-to-process data delivery and
    - **ii)** Error checking.
  - ➢ UDP is an unreliable service i.e. it doesn‘t guarantee data will arrive to destination-process.

  2) TCP (Transmission Control Protocol)
  - ➢ TCP provides a connection-oriented service to the invoking application.
  - ➢ The TCP provides following 3 services:
    1) Reliable data transfer i.e. guarantees data will arrive to destination-process correctly.
    2) Congestion control and
    3) Error checking.

2.2 Multiplexing and Demultiplexing

- A process can have one or more sockets.
- The sockets are used to pass data from the network to the process and vice versa.

  - ➢ At the sender, the transport-layer
    - → gathers data-chunks at the source-host from different sockets
    - → encapsulates data-chunk with header to create segments and
    - → passes the segments to the network-layer.
  - ➢ The job of combining the data-chunks from different sockets to create a segment is called multiplexing.

  2) Demultiplexing
  - ➢ At the receiver, the transport-layer
    - → examines the fields in the segments to identify the receiving-socket and
    - → directs the segment to the receiving-socket.
  - ➢ The job of delivering the data in a segment to the correct socket is called demultiplexing.
- In Figure 2.1,
  - ➢ In the middle host, the transport-layer must demultiplex segments arriving from the network- layer to either process P1 or P2.

➢ The arriving segment's data is directed to the corresponding process's socket.

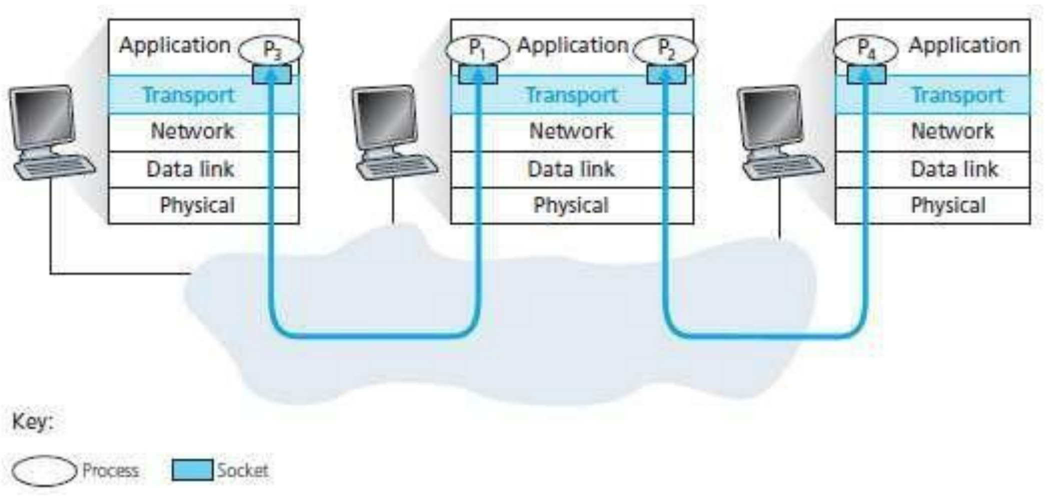


Figure 2.1: Transport-layer multiplexing and demultiplexing

2.2.1 Endpoint Identification

• Each socket must have a unique identifier.
• Each segment must include 2 header-fields to identify the socket (Figure 2.2):
  1) Source-port-number field and
  2) Destination-port-number field.
• Each port-number is a 16-bit number: 0 to 65535.
• The port-numbers ranging from 0 to 1023 are called well-known port-numbers and are restricted. For example: HTTP uses port-no 80
  FTP uses port-no 21
• When we develop a new application, we must assign the application a port-number, which are known as ephemeral ports (49152–65535).

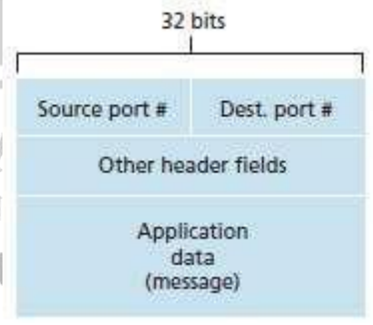
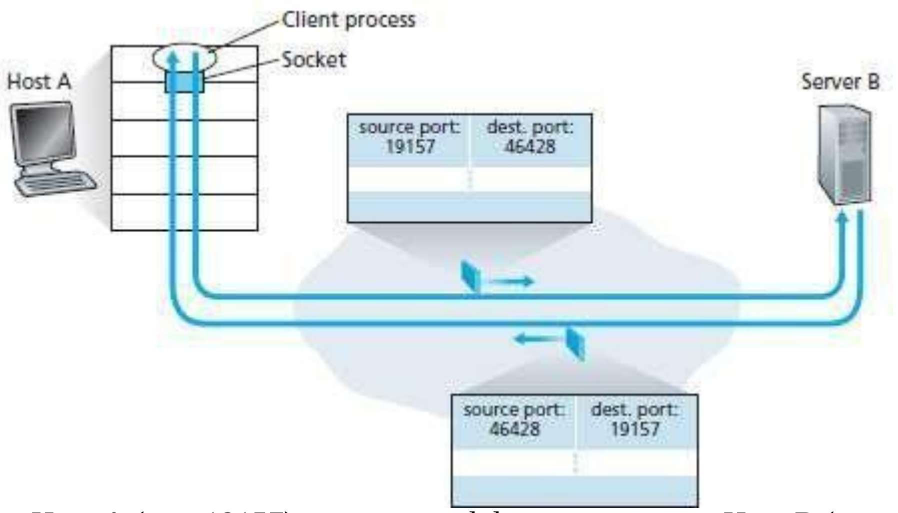


Figure 2.2: Source and destination-port-no fields in a transport-layer segment

• How the transport-layer implements the demultiplexing service?
• Answer:
  ➢ Each socket in the host will be assigned a port-number.
  ➢ When a segment arrives at the host, the transport-layer
    → examines the destination-port-no in the segment
    → directs the segment to the corresponding socket and
    → passes then the segment to the attached process.

2.2.2 Connectionless Multiplexing and Demultiplexing

• At client side of the application, the transport-layer automatically assigns the port- number. Whereas, at the server side, the application assigns a specific port-number.



• Suppose process on Host-A (port 19157) wants to send data to process on Host-B (port 46428).

Figure 2.3: The inversion of source and destination-port-nos

• At the sender A, the transport-layer
  → creates a segment containing source-port 19157, destination-port 46428 & data and
  → passes then the resulting segment to the network-layer.
• At the receiver B, the transport-layer
  → examines the destination-port field in the segment and
  → delivers the segment to the socket identified by port 46428.
• A UDP socket is identified by a two-tuple:
  1) Destination IP address &
  2) Destination-port-no.
• As shown in Figure 2.3,
  Source-port-no from Host-A is used at Host-B as "return address" i.e. when B wants to send a segment back to A.

2.2.3 Connection Oriented Multiplexing and Demultiplexing

• Each TCP connection has exactly 2 end-points. (Figure 2.4).
• Thus, 2 arriving TCP segments with different source-port-nos will be directed to 2 different sockets, even if they have the same destination-port-no.
• A TCP socket is identified by a four-tuple:
  1) Source IP address
  2) Source-port-no
  3) Destination IP address &
  4) Destination-port-no.

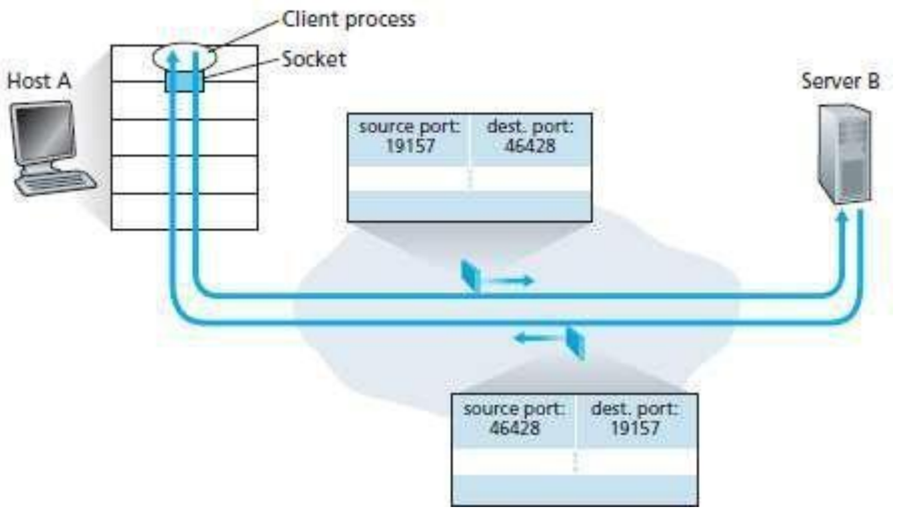


Figure 2.4: The inversion of source and destination-port-nos

- The server-host may support many simultaneous connection-sockets.
- Each socket will be
  - → attached to a process.
  - → identified by its own four tuple.
- When a segment arrives at the host, all 4 fields are used to direct the segment to the appropriate socket. (i.e. Demultiplexing).

2.2.4 Web Servers and TCP

- Consider a host running a Web-server (ex: Apache) on port 80.
- When clients (ex: browsers) send segments to the server, all segments will have destination-port 80.
- The server distinguishes the segments from the different clients using two-tuple:
  1) Source IP addresses &
  2) Source-port-nos.
- Figure 2.5 shows a Web-server that creates a new process for each connection.
- The server can use either i) persistent HTTP or ii) non-persistent HTTP

  i) Persistent HTTP
  - ➢ Throughout the duration of the persistent connection the client and server exchange HTTP messages via the same server socket.

  ii) Non-persistent HTTP
  - ➢ A new TCP connection is created and closed for every request/response.
  - ➢ Hence, a new socket is created and closed for every request/response.
  - ➢ This can severely impact the performance of a busy Web-server.

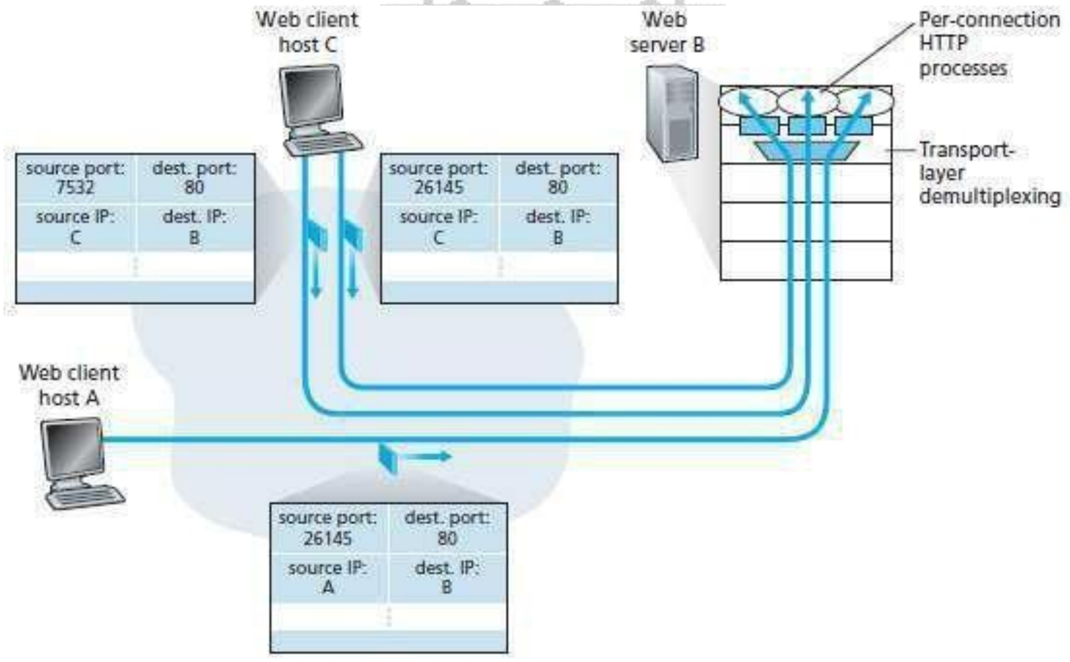


Figure 2.5: Two clients, using the same destination-port-no (80) to communicate with the same Web-server application

2.3 Connectionless Transport: UDP

- UDP is an unreliable, connectionless protocol.
  - ➢ Unreliable service means UDP doesn't guarantee data will arrive to destination-process.
  - ➢ Connectionless means there is no handshaking b/w sender & receiver before sending data.
- It provides following 2 services:
  - i) Process-to-process data delivery and
  - ii) Error checking.
- It does not provide flow, error, or congestion control.
- At the sender, UDP
  - → takes messages from the application-process
  - → attaches source- & destination-port-nos and

→ passes the resulting segment to the network-layer.

- At the receiver, UDP
  → examines the destination-port-no in the segment and
  → delivers the segment to the correct application-process.
- It is suitable for application program that
  → needs to send short messages &
  → cannot afford the retransmission.
- UDP is suitable for many applications for the following reasons:

1) Finer Application Level Control over what Data is Sent, and when.

- ➢ When an application-process passes data to UDP, the UDP
  → packs the data inside a segment and
  → passes immediately the segment to the network-layer.
- ➢ On the other hand,
  In TCP, a congestion-control mechanism throttles the sender when the n/w is congested

- ➢ TCP uses a three-way handshake before it starts to transfer data.
- ➢ UDP just immediately passes the data without any formal preliminaries.
- ➢ Thus, UDP does not introduce any delay to establish a connection.
- ➢ That‘s why, DNS runs over UDP rather than TCP.

- ➢ TCP maintains connection-state in the end-systems.
- ➢ This connection-state includes
  → receive and send buffers
  → congestion-control parameters and
  → sequence- and acknowledgment-number parameters.
- ➢ On the other hand,

4) Small Packet Header verhead.

- ➢ The TCP segment has 20 bytes of header overhead in every segment.
- ➢ On the other hand, UDP has only 8 bytes of overhead.

Table 2.1: Popular Internet applications and their underlying transport protocols

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

2.3.1 UDP Segment Structure

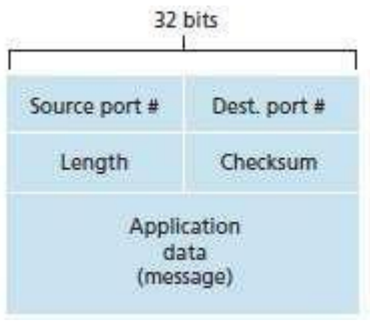


Figure 2.6: UDP segment structure

- UDP Segment contains following fields (Figure 2.6):
  1) **Application Data:** This field occupies the data-field of the segment.
  2) **Destination Port No:** This field is used to deliver the data to correct process running on the destination-host. (i.e. demultiplexing function).
  3) **Length:** This field specifies the number of bytes in the segment (header plus data).
  4) **Checksum:** This field is used for error-detection.

2.3.2 UDP Checksum
- The checksum is used for error-detection.
- The checksum is used to determine whether bits within the segment have been altered.
- How to calculate checksum on the sender:
  1) All the 16-bit words in the segment are added to get a sum.
  2) Then, the 1's complement of the sum is obtained to get a result.
  3) Finally, the result is added to the checksum-field inside the segment.
- How to check for error on the receiver:
  1) All the 16-bit words in the segment (including the checksum) are added to get a sum.
     i) For no errors: In the sum, all the bits are 1. (x: 1111111)
     ii) For any error: In the sum, at least one of the bits is a 0. (Ex: 1011111)

Example:
- On the sender:
  - ➢ Suppose that we have the following three 16-bit words:

```
0110011001100000
0101010101010101          → three 16 bits words
1000111100001100
```

  - ➢ The sum of first two 16-bit words is:

```
0110011001100000
0101010101010101
1011101110110101
```

  - ➢ Adding the third word to the above sum gives:

```
1011101110110101          → sum of 1st two 16 bit words
1000111100001100          → third 16 bit word
0100101011000010          → sum of all three 16 bit words
```

  - ➢ aking 1‘s complement for the final sum:

```
0100101011000010          → sum of all three 16 bit words
1011010100111101          → 1‘s complement for the final sum
```

- The 1‘s complement value is called as checksum which is added inside the segment.
- On the receiver
  - ➢ All four 16-bit words are added, including the checksum.
     i) If no errors are introduced into the packet, then clearly the sum will be 1111111111111111.
     ii) If one of the bits is a 0, then errors have been introduced into the packet.

2.4 Principles of Reliable Data Transfer
- Figure 2.7 illustrates the framework of reliable data transfer protocol.

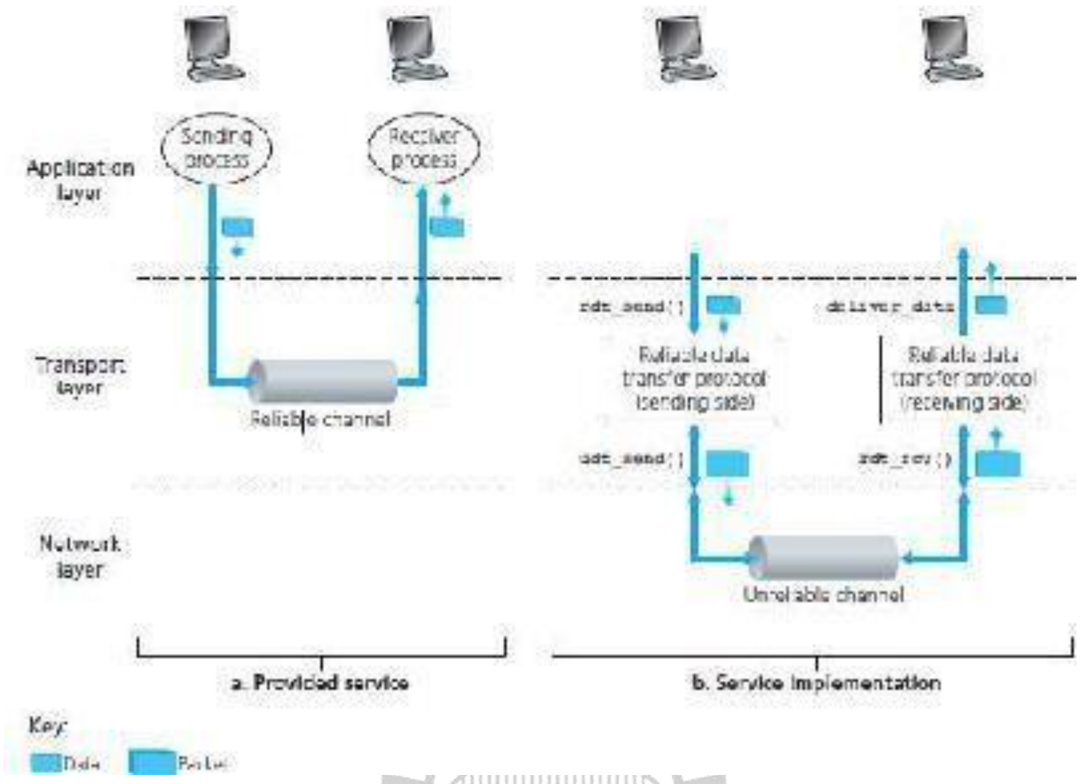


Figure 2.7: Reliable data transfer: Service model and service implementation

- On the sender, rdt_send() will be called when a packet has to be sent on the channel.
- On the receiver,
  - i) rdt_rcv()will be called when a packet has to be recieved on the channel.
  - ii) deliver data() will be called when the data has to be delivered to the upper layer

2.4.1 Building a Reliable Data Transfer Protocol

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

- Consider data transfer over a perfectly reliable channel.
- We call this protocol as rdt1.0.

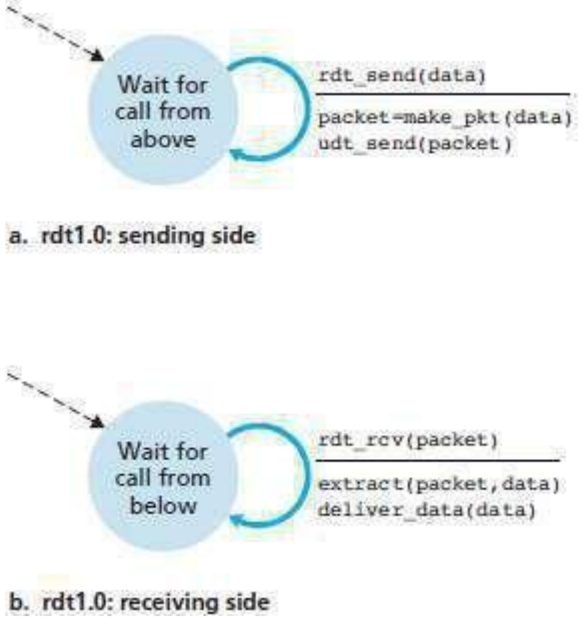


Figure 2.8: rdt1.0 – A protocol for a completely reliable channel

- The finite-state machine (FSM) definitions for the rdt1.0 sender and receiver are shown in Figure 2.8.
- The sender and receiver FSMs have only one state.
- In FSM, following notations are used:
  - i) The arrows indicate the transition of the protocol from one state to another.
  - ii) The event causing the transition is shown above the horizontal line labelling the transition.
  - iii) The action taken when the event occurs is shown below the horizontal line.
  - iv) The dashed arrow indicates the initial state.
- On the sender, rdt
  - → 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 receiver, 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)).

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

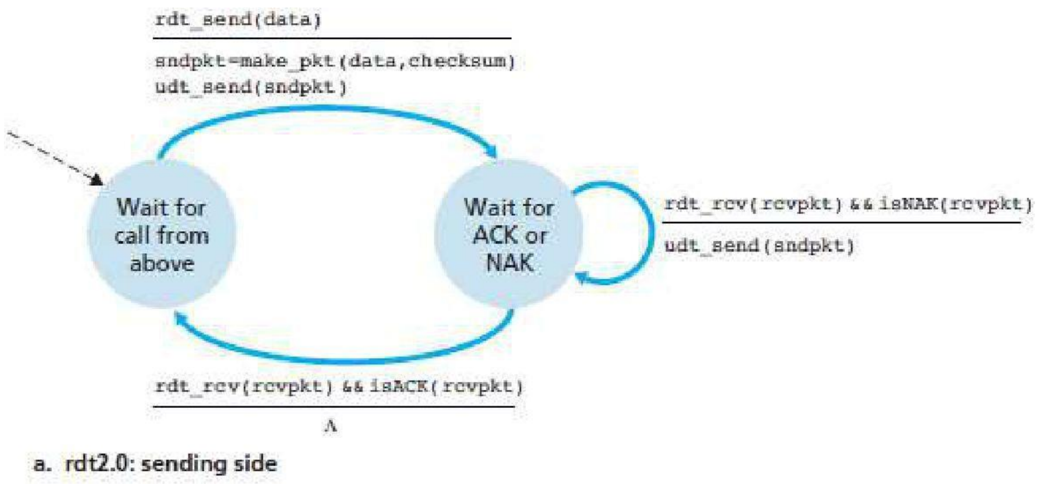
- Consider data transfer over an unreliable channel in which bits in a packet may be corrupted.
- We call this protocol as rdt2.0.
- The message dictation protocol uses both
  - → positive acknowledgements (ACK) and
  - → negative acknowledgements (NAK).
- The receiver uses these control messages to inform the sender about
  - → what has been received correctly and
  - → what has been received in error and thus requires retransmission.
- Reliable data transfer protocols based on the retransmission are known as ARQ protocols.
- Three additional protocol capabilities are required in ARQ protocols:

  - ➢ A mechanism is needed to allow the receiver to detect when bit-errors have occurred.
  - ➢ UDP uses the checksum field for error-detection.
  - ➢ Error-correction techniques allow the receiver to detect and correct packet bit-errors.

  2) Receiver Feedback

  - ➢ Since the sender and receiver are typically executing on different end-systems.
  - ➢ The only way for the sender to learn about status of the receiver is by the receiver providing explicit feedback to the sender.
  - ➢ For example: ACK & NAK

  - ➢ A packet that is received in error at the receiver will be retransmitted by the sender.
- Figure 2.9 shows the FSM representation of rdt2.0.



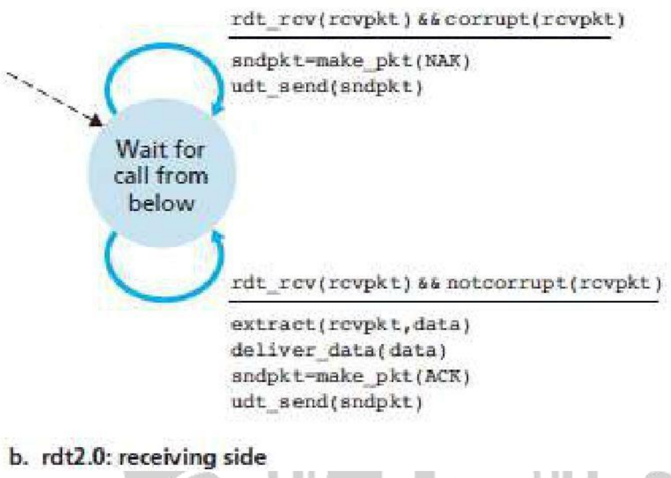


Figure 2.9: rdt2.0–A protocol for a channel with bit-errors

Sender FSM
- The sender of rdt2.0 has 2 states:
  1) In one state, the protocol is waiting for data to be passed down from the upper layer.
  2) In other state, the protocol is waiting for an ACK or a NAK from the receiver.
     i) If an ACK is received, the protocol
        → knows that the most recently transmitted packet has been received correctly
        → returns to the state of waiting for data from the upper layer.
     ii) If a NAK is received, the protocol
        → retransmits the last packet and
        → waits for an ACK or NAK to be returned by the receiver.
- The sender will not send a new data until it is sure that the receiver has correctly received the current packet.
- Because of this behaviour, protocol rdt2.0 is known as stop-and-wait protocols.

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

2.4.1.2.1 Sender Handles Garbled ACK/NAKs: rdt2.1
- Problem with rdt2.0:

  If an ACK or NAK is corrupted, the sender cannot know whether the receiver has correctly received the data or not.
- Solution: The sender resends the current data packet when it receives garbled ACK or NAK packet.
  - ➢ Problem: This approach introduces duplicate packets into the channel.
  - ➢ Solution: Add sequence-number field to the data packet.

- ➢ The receiver has to only check the sequence-number to determine whether the received packet is a retransmission or not.
- For a stop-and-wait protocol, a 1-bit sequence-number will be sufficient.
- A 1-bit sequence-number allows the receiver to know whether the sender is sending
  → previously transmitted packet (0) or
  → new packet (1).
- We call this protocol as rdt2.1.
- Figure 2.10 and 2.11 shows the FSM description for rdt2.1.

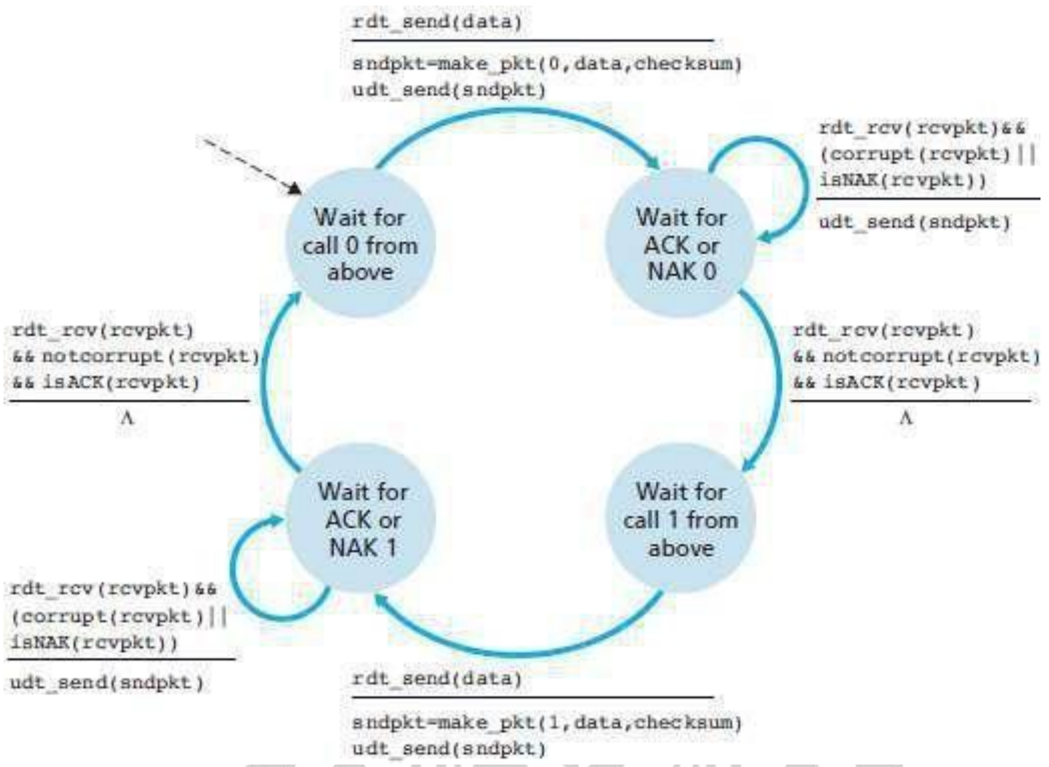
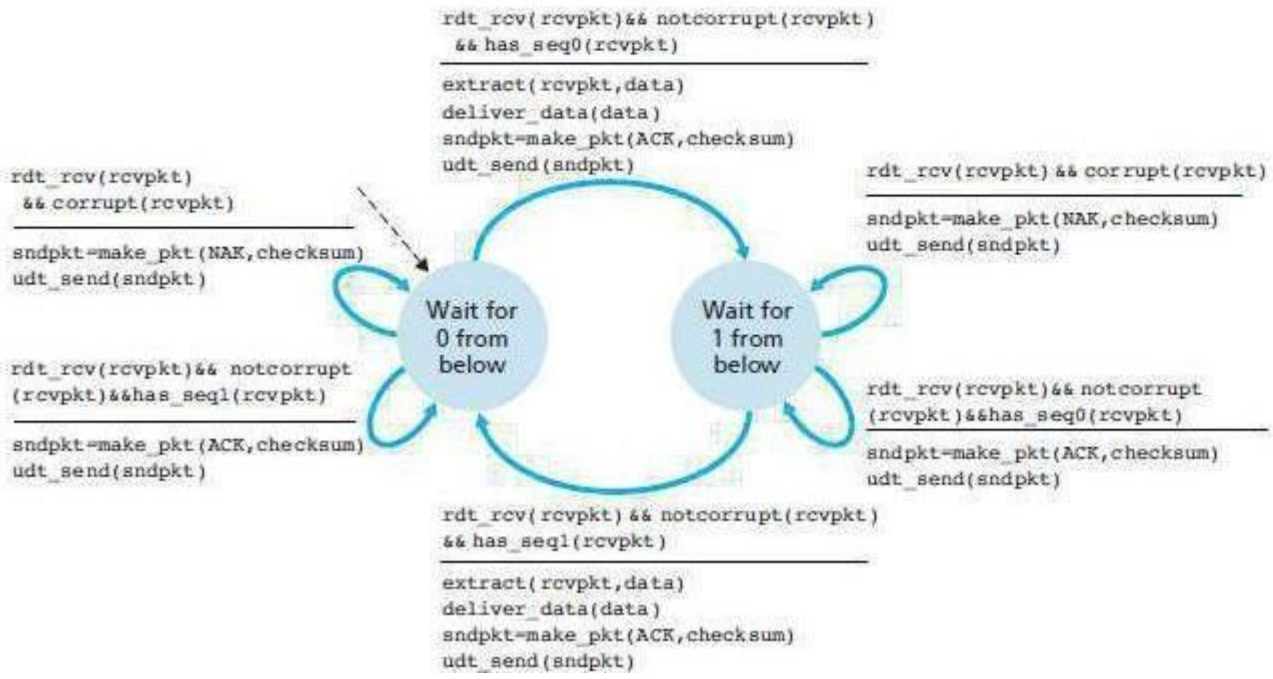


Figure 2.10: rdt2.1 sender



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

Figure 2.11: rdt2.1 receiver

2.4.1.2.2 Sender uses ACK/NAKs: rdt2.2

- Protocol rdt2.2 uses both positive and negative acknowledgments from the receiver to the sender.
  - i) When out-of-order packet is received, the receiver sends a positive acknowledgment (ACK).
  - ii) When a corrupted packet is received, the receiver sends a negative acknowledgment (NAK).
- We call this protocol as rdt2.2. (Figure 2.12 and 2.13).

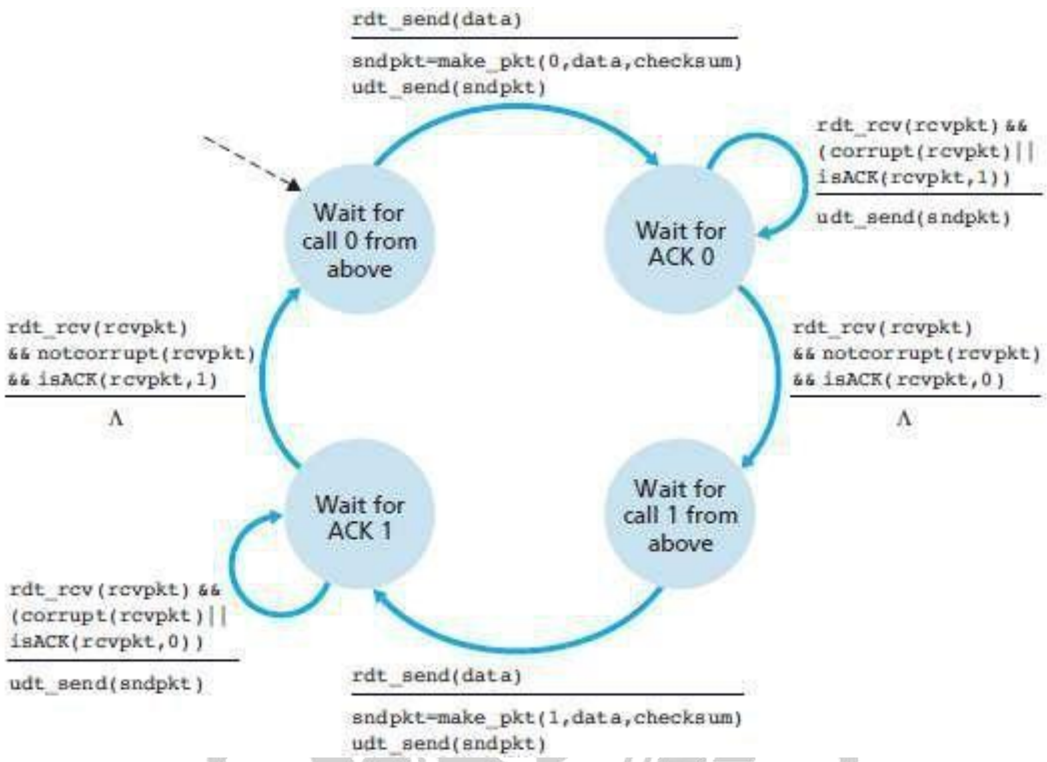

rdt_send(data)
sndpkt=make_pkt(0,data,checksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && (corrupt(rcvpkt)|| isACK(rcvpkt,1))
udt_send(sndpkt)
Wait for call 0 from above
Wait for ACK 0
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,1)
Λ
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,0)
Λ
Wait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

Figure 2.12: rdt2.2 sender

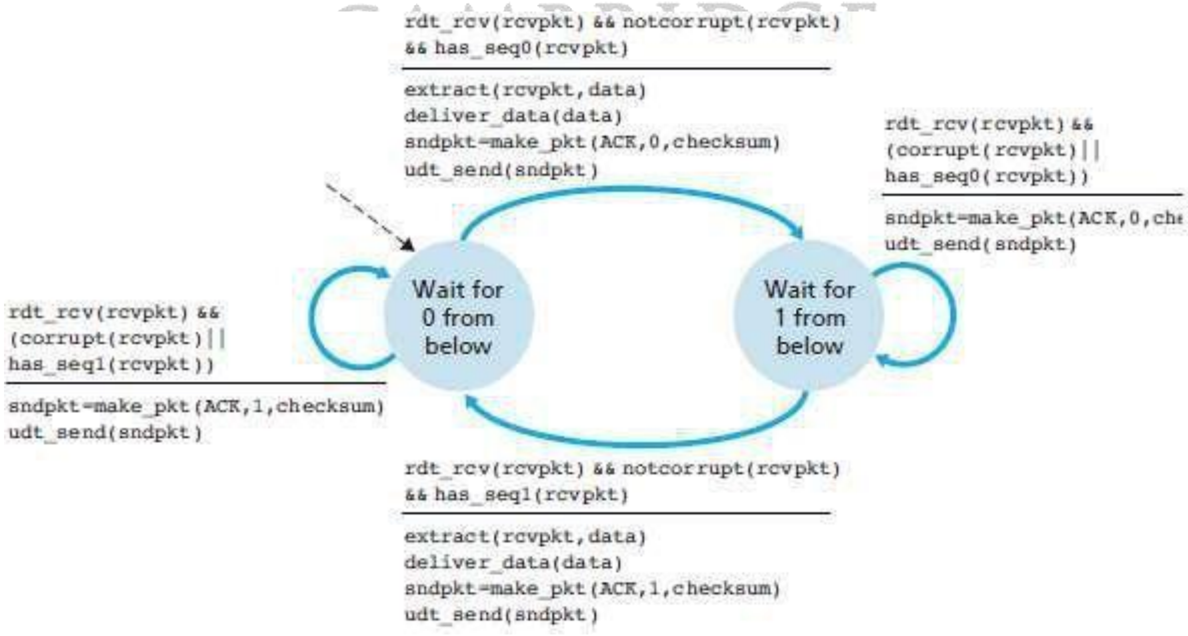

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq0(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
sndpkt=make_pkt(ACK,0,checksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && (corrupt(rcvpkt)|| has_seq0(rcvpkt))
sndpkt=make_pkt(ACK,0,ch
udt_send(sndpkt)
rdt_rcv(rcvpkt) && (corrupt(rcvpkt)|| has_seq1(rcvpkt))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

Figure 2.13: rdt2.2 receiver

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

- Consider data transfer over an unreliable channel in which packet lose may occur.

- We call this protocol as rdt3.0.
- Two problems must be solved by the rdt3.0:
  1) How to detect packet loss?
  2) What to do when packet loss occurs?
- Solution:
  - ➢ The sender
    - → sends one packet & starts a timer and
    - → waits for ACK from the receiver (okay to go ahead).
  - ➢ If the timer expires before ACK arrives, the sender retransmits the packet and restarts the timer.
- The sender must wait at least as long as
  1) A round-trip delay between the sender and receiver plus
  2) Amount of time needed to process a packet at the receiver.
- Implementing a time-based retransmission mechanism requires a countdown timer.
- The timer must interrupt the sender after a given amount of time has expired.
- Figure 2.14 shows the sender FSM for rdt3.0, a protocol that reliably transfers data over a channel that can corrupt or lose packets;
- Figure 2.15 shows how the protocol operates with no lost or delayed packets and how it handles lost data packets.
- Because sequence-numbers alternate b/w 0 & 1, protocol rdt3.0 is known as alternating-bit protocol.

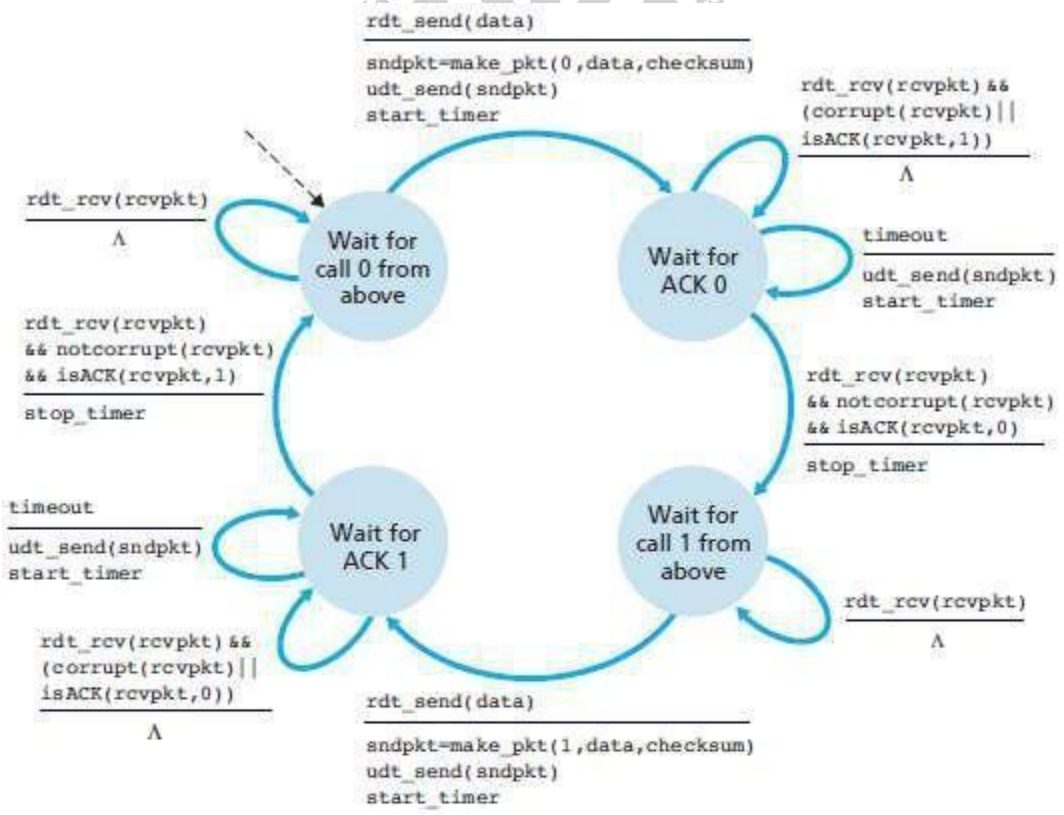

rdt_send(data)
sndpkt=make_pkt(0,data,checksum)
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt) && (corrupt(rcvpkt)|| isACK(rcvpkt,1))
Λ
rdt_rcv(rcvpkt)
Λ
Wait for call 0 from above
Wait for ACK 0
timeout
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,1)
stop_timer
rdt_rcv(rcvpkt)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

Figure 2.14: rdt3.0 sender

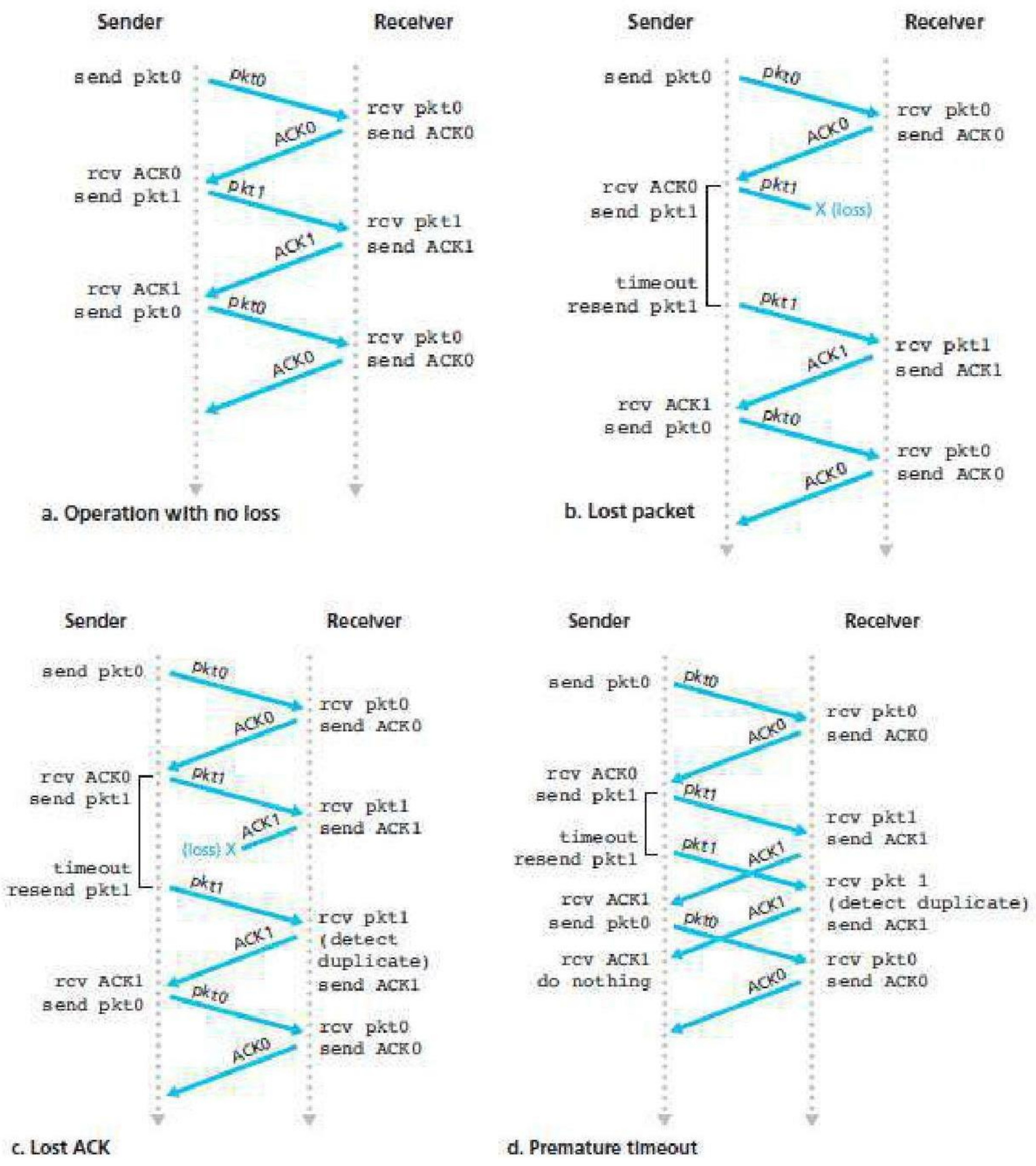


Figure 2.15: Operation of rdt3.0, the alternating-bit protocol

2.4.2 Pipelined Reliable Data Transfer Protocols

- The sender is allowed to send multiple packets without waiting for acknowledgments.
- This is illustrated in Figure 2.16 (b).
- Pipelining has the following consequences:
  1) The range of sequence-numbers must be increased.
  2) The sender and receiver may have to buffer more than one packet.
- Two basic approaches toward pipelined error recovery can be identified:
  1) Go-Back-N and 2) Selective repeat.

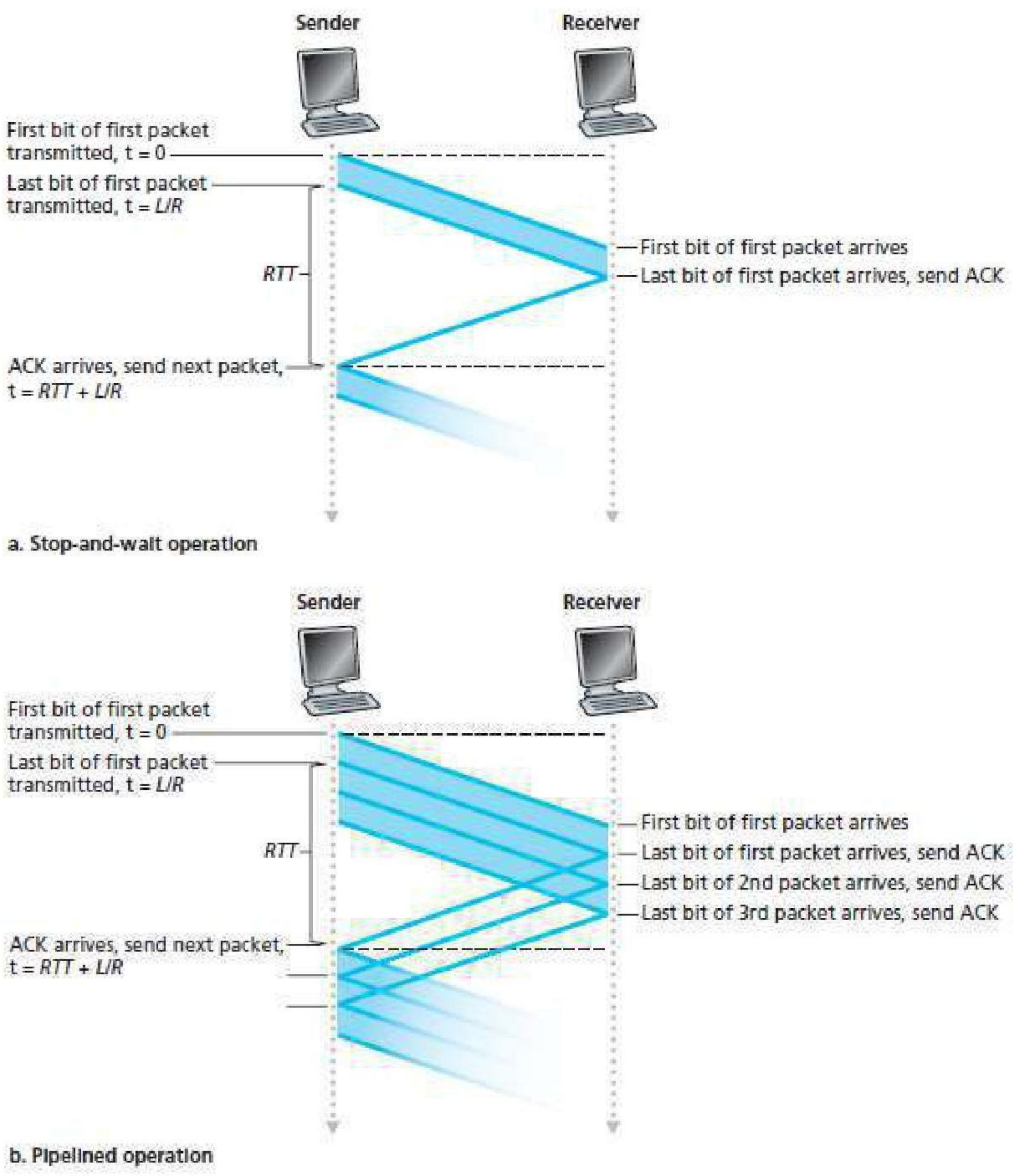

Sender
Receiver
First bit of first packet transmitted, t = 0
Last bit of first packet transmitted, t = L/R
First bit of first packet arrives
Last bit of first packet arrives, send ACK
RTT
ACK arrives, send next packet, t = RTT + L/R
a. Stop-and-wait operation
Sender
Receiver
First bit of first packet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

Figure 2.16: Stop-and-wait and pipelined sending

2.4.3 Go-Back-N (GBN)

- The sender is allowed to transmit multiple packets without waiting for an acknowledgment.
- But, the sender is constrained to have at most N unacknowledged packets in the pipeline.
  Where N = window-size which refers maximum no. of unacknowledged packets in the pipeline
- GBN protocol is called a sliding-window protocol.
- Figure 2.17 shows the sender‘s view of the range of sequence-numbers.

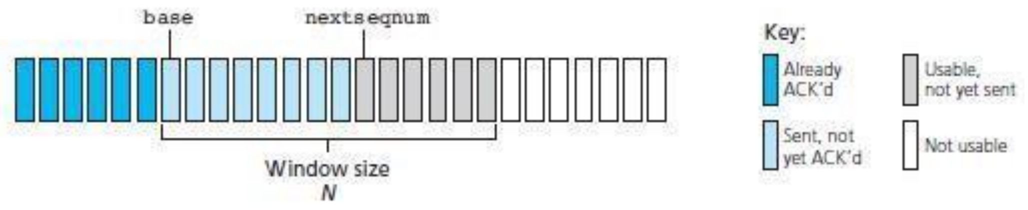


Figure 2.17: Sender‘s view of sequence-numbers in Go-Back-N

- Figure 2.18 and 2.19 give a FSM description of the sender and receivers of a GBN protocol.

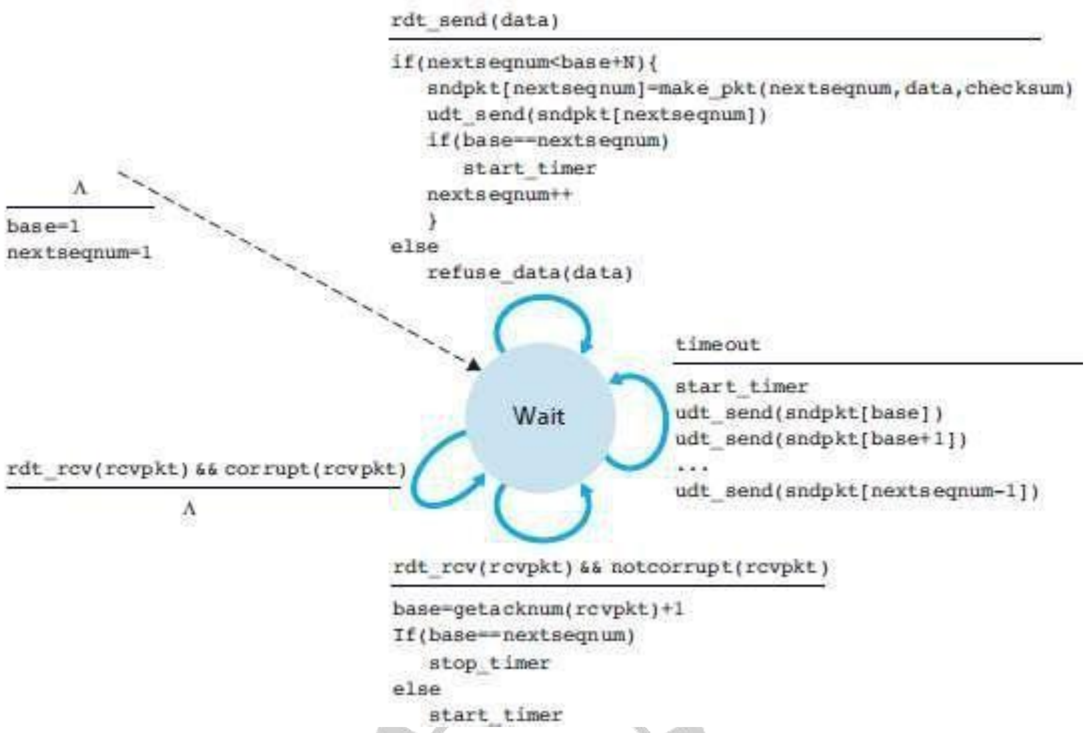


Figure 2.18: Extended FSM description of GBN sender

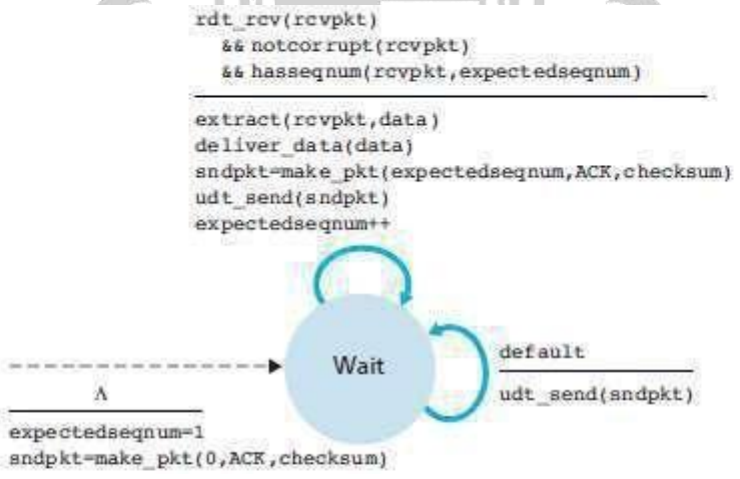


Figure 2.19: Extended FSM description of GBN receiver

2.4.3.1 GBN Sender

- The sender must respond to 3 types of events:
  - **1)** Invocation from above.
  - ➢ When rdt_send() is called from above, the sender first checks to see if the window is full i.e. whether there are N outstanding, unacknowledged packets.
    - i) If the window is not full, the sender creates and sends a packet.
    - ii) If the window is full, the sender simply returns the data back to the upper layer. This is an implicit indication that the window is full.

  - ➢ An acknowledgment for a packet with sequence-number n will be taken to be a cumulative acknowledgment.
  - ➢ All packets with a sequence-number up to n have been correctly received at the receiver.

  - ➢ A timer will be used to recover from lost data or acknowledgment packets.
    - i) If a timeout occurs, the sender resends all packets that have been previously sent but that

have not yet been acknowledged.

ii) If an ACK is received but there are still additional transmitted but not yet acknowledged packets, the timer is restarted.

iii) If there are no outstanding unacknowledged packets, the timer is stopped.

- 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 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.

2.4.3.3 Operation of the GBN Protocol

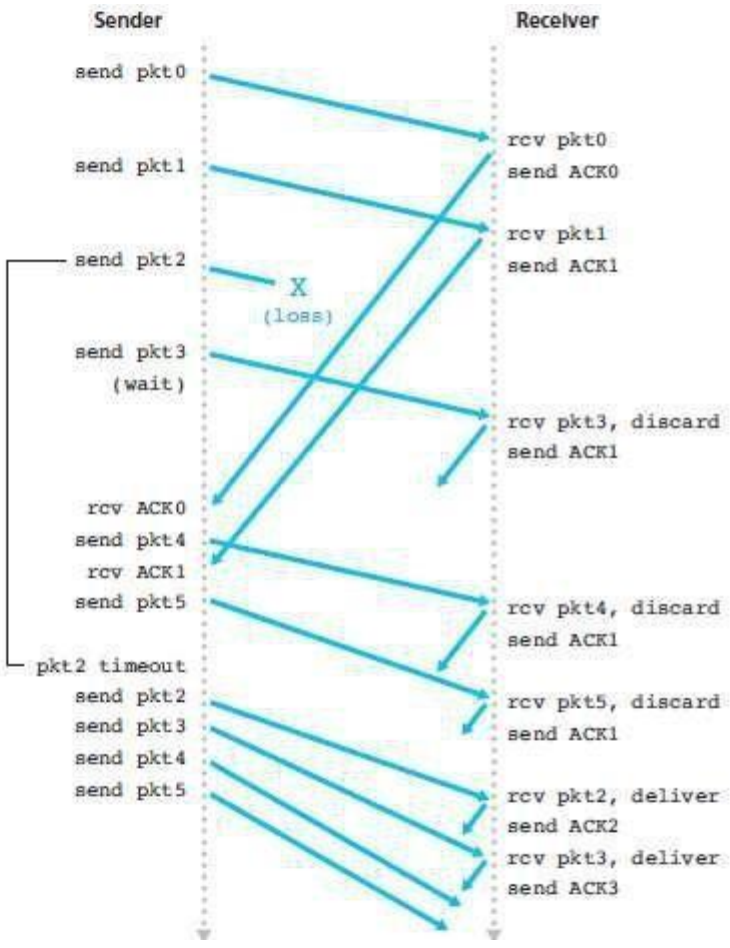


Figure 2.20: Go-Back-N in operation

- Figure 2.20 shows the operation of the GBN protocol for the case of a window-size of four packets.
- The sender sends packets 0 through 3.
- The sender then must wait for one or more of these packets to be acknowledged before proceeding.
- As each successive ACK (for ex, ACK0 and ACK1) is received, the window slides forward and the sender transmits one new packet (pkt4 and pkt5, respectively).
- On the receiver, packet 2 is lost and thus packets 3, 4, and 5 are found to be out of order and are discarded.

2.4.4 Selective Repeat (SR)

- Problem with GBN:
  - GBN suffers from performance problems.
  - When the window-size and bandwidth-delay product are both large, many packets can be in the

pipeline.
- ➢ Thus, a single packet error results in retransmission of a large number of packets.

- Solution: Use Selective Repeat (SR).

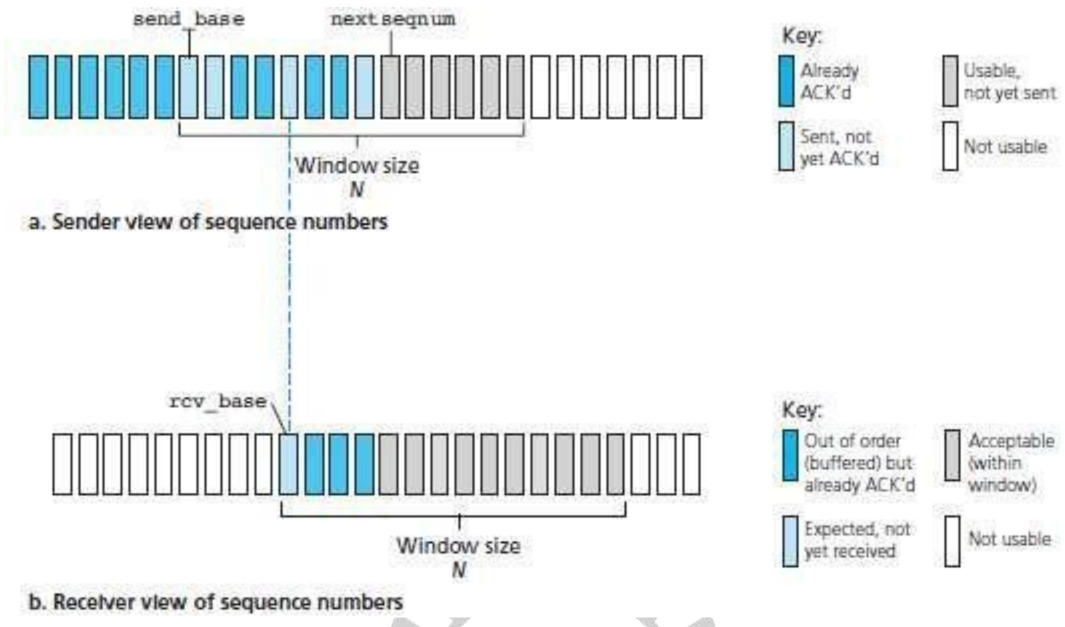

send_base
nextseqnum
Window size N
a. Sender view of sequence numbers
Key:
Already ACK'd
Usable, not yet sent
Sent, not yet ACK'd
Not usable
rcv_base
Window size N
b. Receiver view of sequence numbers
Key:
Out of order (buffered) but already ACK'd
Acceptable (within window)
Expected, not yet received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

Figure 2.21: Selective-repeat (SR) sender and receiver views of sequence-number space

- The sender retransmits only those packets that it suspects were erroneous.
- Thus, avoids unnecessary retransmissions. Hence, the name –selective-repeat‖.
- The receiver individually acknowledge correctly received packets.
- A window-size N is used to limit the no. of outstanding, unacknowledged packets in the pipeline.
- Figure 2.21 shows the SR sender‗s view of the sequence-number space.

2.4.4.1 SR Sender
- The various actions taken by the SR sender are as follows:

1) Data Received from above.

➢ When data is received from above, the sender checks the next available sequence-number for the packet.

➢ If the sequence-number is within the sender‗s window;

➢ Timers are used to protect against lost packets.

➢ Each packet must have its own logical timer. This is because

→ only a single packet will be transmitted on timeout.

3) ACK Received.

➢ If an ACK is received, the sender marks that packet as having been received.

➢ If the packet‗s sequence-number is equal to send base, the window base is increased by the smallest sequence-number.

➢ If there are untransmitted packets with sequence-numbers that fall within the window, these packets are transmitted.

2.4.4.2 SR Receiver
- The various actions taken by the SR receiver are as follows:

1) Packet with sequence-number in [rcv_base, rcv_base+N-1] is correctly received.

➢ In this case,

→ received packet falls within the receiver‗s window and

→ selective ACK packet is returned to the sender.

➢ If the packet was not previously received, it is buffered.

➢ If this packet has a sequence-number equal to rcv_base, then this packet, and any previously buffered and consecutively numbered packets are delivered to the upper layer.

➢ The receive-window is then moved forward by the no. of packets delivered to the upper layer.

➢ For example: consider Figure 2.22.

¤ When a packet with a sequence-number of rcv_base=2 is received, it and packets 3, 4, and 5 can be delivered to the upper layer.

2) Packet with sequence-number in [rcv_base-N, rcv base-1] is correctly received.

➢ In this case, an ACK must be generated, even though this is a packet that the receiver has previously acknowledged.

3) Otherwise.

➢ Ignore the packet.

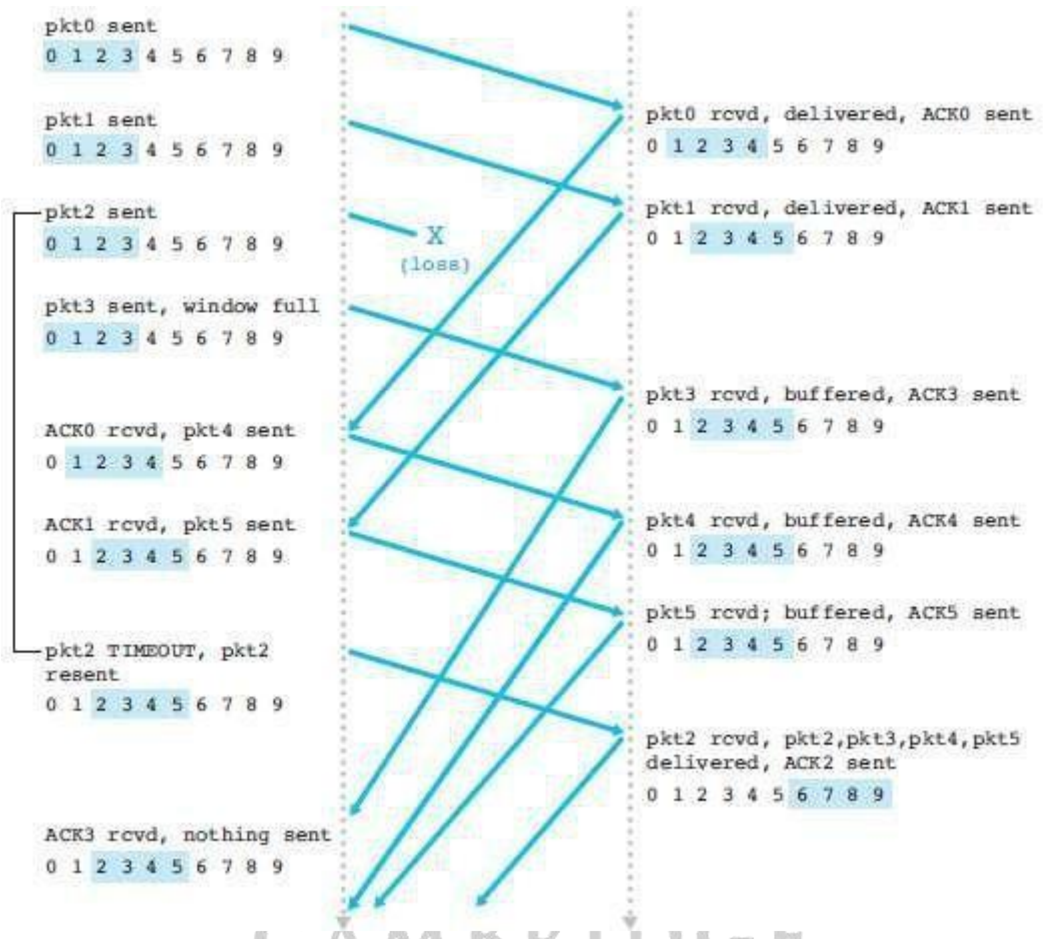


Figure 2.22: SR operation

2.4.5 Summary of Reliable Data Transfer Mechanisms and their Use

Table 2.2: Summary of reliable data transfer mechanisms and their use

Mechanism	Use, Comments
Checksum	Used to detect bit errors in a transmitted packet.
Timer	Used to timeout/retransmit a packet because the packet (or its ACK) was lost. Because timeouts can occur when a packet is delayed but not lost, duplicate copies of a packet may be received by a receiver.
Sequence-number	Used for sequential numbering of packets of data flowing from sender to receiver. Gaps in the sequence-numbers of received packets allow the receiver to detect a lost packet. Packets with duplicate sequence-numbers allow the receiver to detect duplicate copies of a packet.

Acknowledgment	Used by the receiver to tell the sender that a packet or set of packets has been received correctly. Acknowledgments will typically carry the sequence-number of the packet or packets being acknowledged. Acknowledgments may be individual or cumulative, depending on the protocol.
Negative acknowledgment	Used by the receiver to tell the sender that a packet has not been received correctly. Negative acknowledgments will typically carry the sequence-number of the packet that was not received correctly.
Window, pipelining	The sender may be restricted to sending only packets with sequence- numbers that fall within a given range. By allowing multiple packets to be transmitted but not yet acknowledged, sender utilization can be increased over a stop-and-wait mode of operation.

2.5 Connection-Oriented Transport: TCP

• TCP is a reliable connection-oriented protocol.
 ➢ Connection-oriented means a connection is established b/w sender & receiver before sending the data.
 ➢ Reliable service means TCP guarantees that the data will arrive to destination-process correctly.

• TCP provides flow-control, error-control and congestion-control.

2.5.1 The TCP Connection

• The features of TCP are as follows:

1) Connection Oriented
 ➢ TCP is said to be connection-oriented. This is because
  The 2 application-processes must first establish connection with each other before they begin communication.
 ➢ Both application-processes will initialize many state-variables associated with the connection.

2) Runs in the End Systems
 ➢ TCP runs only in the end-systems but not in the intermediate routers.
 ➢ The routers do not maintain any state-variables associated with the connection.

 ➢ TCP connection provides a full-duplex service.
 ➢ Both application-processes can transmit and receive the data at the same time.

4) Point-to-Point
 ➢ A TCP connection is point-to-point i.e. only 2 devices are connected by a dedicated-link
 ➢ So, multicasting is not possible.

 ➢ Connection-establishment process is referred to as a three-way handshake. This is because 3 segments are sent between the two hosts:
  i) The client sends a first-segment.
  ii) The server responds with a second-segment and
  iii) Finally, the client responds again with a third segment containing payload (or data).

6) Maximum Segment Size (MSS)
 ➢ MSS limits the maximum amount of data that can be placed in a segment.

 ➢ As shown in Figure 2.23, consider sending data from the client-process to the server-process.

  At Sender
  i) The client-process passes a stream-of-data through the socket.
  ii) Then, TCP forwards the data to the send-buffer.
  iii) Each chunk-of-data is appended with a header to form a segment.
  iv) The segments are sent into the network.

  i) The segment‘s data is placed in the receive-buffer.
  ii) The application reads the stream-of-data from the receive-buffer.

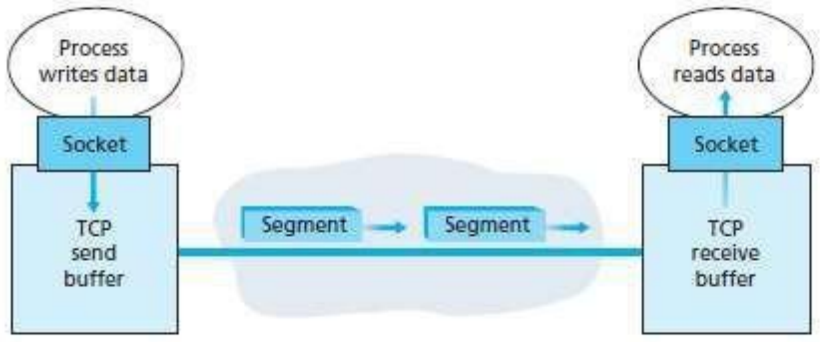


Figure 2.23: TCP send and receive-buffers

2.5.2 TCP Segment Structure

- The segment consists of header-fields and a data-field.
- The data-field contains a chunk-of-data.
- When TCP sends a large file, it breaks the file into chunks of size MSS.
- Figure 2.24 shows the structure of the TCP segment.

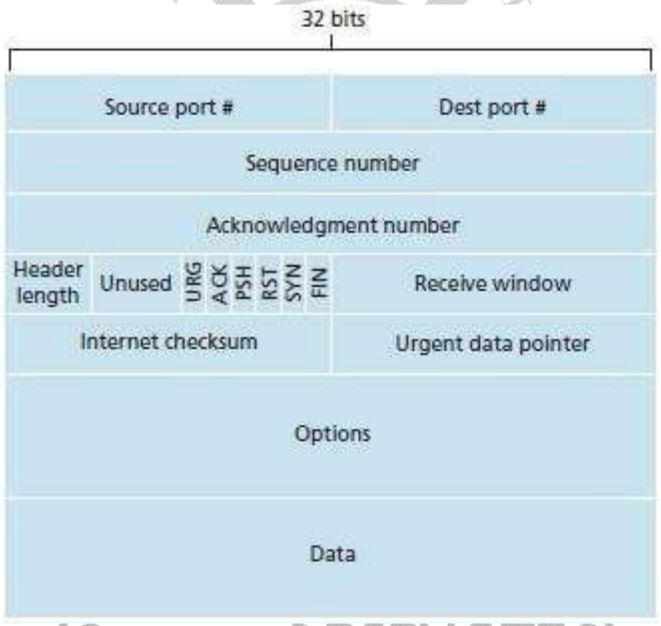


Figure 2.24: TCP segment structure

- The fields of TCP segment are as follows:
  1) Source and Destination Port Numbers
  ➢ These fields are used for multiplexing/demultiplexing data from/to upper-layer applications.
  2) Sequence Number & Acknowledgment Number
  ➢ These fields are used by sender & receiver in implementing a reliable data-transfer-service.
  3) Header Length
  ➢ This field specifies the length of the CP header.

  ➢ This field contains 6 bits.
  i) ACK
  ¤ This bit indicates that value of acknowledgment field is valid. **ii) RST, SYN & FIN**
  ¤ These bits are used for connection setup and teardown.
  iii) PSH
  ¤ This bit indicates the sender has invoked the push operation.
  iv) RG
  ¤ This bit indicates the segment contains urgent-data.

5) Receive Window
- ➢ his field defines receiver's window size
- ➢ his field is used for flow control.

6) Checksum
- ➢ his field is used for error-detection.

7) Urgent Data Pointer
- ➢ This field indicates the location of the last byte of the urgent data.

8) Options
- ➢ This field is used when a sender & receiver negotiate the MSS for use in high-speed networks.

2.5.2.1 Sequence Numbers and Acknowledgment Numbers

Sequence Numbers

- The sequence-number is used for sequential numbering of packets of data flowing from sender to receiver.
- Applications:
  1) Gaps in the sequence-numbers of received packets allow the receiver to detect a lost packet.
  2) Packets with duplicate sequence-numbers allow the receiver to detect duplicate copies of a packet.

- The acknowledgment-number is used by the receiver to tell the sender that a packet has been received correctly.
- Acknowledgments will typically carry the sequence-number of the packet being acknowledged.

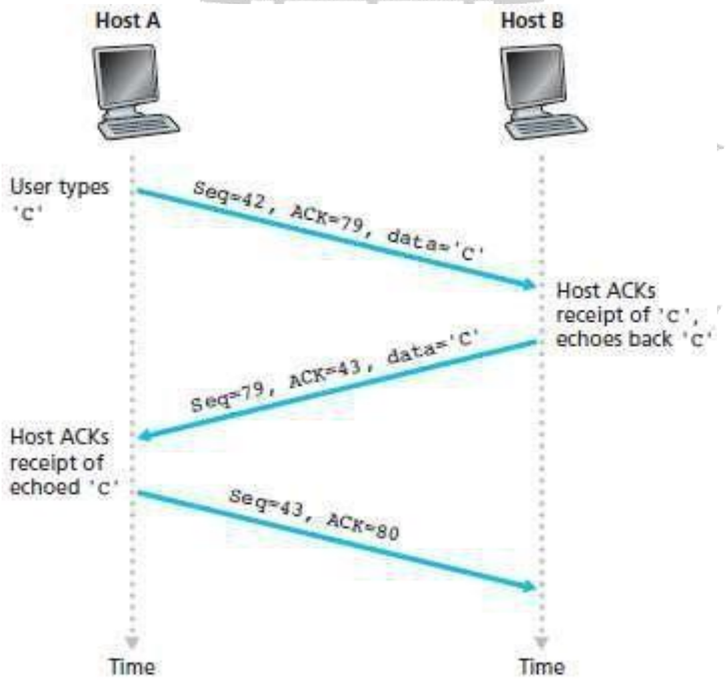


Figure 2.25: Sequence and acknowledgment-numbers for a simple Telnet application over TCP

- Consider an example (Figure 2.25):
  - ➢ A process in Host-A wants to send a stream-of-data to a process in Host-B.
  - ➢ In Host-A, each byte in the data-stream is numbered as shown in Figure 2.26.

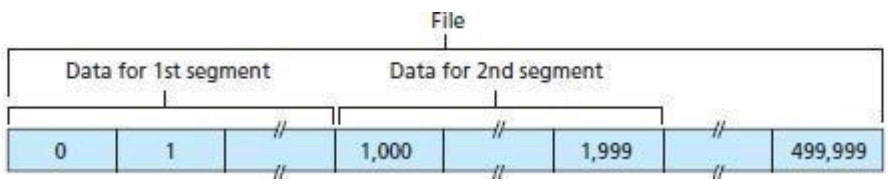


Figure 2.26: Dividing file data into TCP segments

- The first segment from A to B has a sequence-number 42 i.e. Seq=42.
- The second segment from B to A has a sequence-number 79 i.e. Seq=79.
- The second segment from B to A has acknowledgment-number 43, which is the sequence-number of the next byte, Host-B is expecting from Host-A. (i.e. ACK=43).
- What does a host do when it receives out-of-order bytes?

  Answer: There are two choices:

  1) The receiver immediately discards out-of-order bytes.
  2) The receiver

     → keeps the out-of-order bytes and

     → waits for the missing bytes to fill in the gaps.

2.5.2.2 Telnet: A Case Study for Sequence and Acknowledgment Numbers

- Telnet is a popular application-layer protocol used for remote-login.
- Telnet runs over TCP.
- Telnet is designed to work between any pair of hosts.
- As shown in Figure 2.27, suppose client initiates a Telnet session with server.
- Now suppose the user types a single letter, ‗C‘.
- Three segments are sent between client & server:

- The first-segment is sent from the client to the server.
- The segment contains

  → letter ‗C‘

  → sequence-number 42

  → acknowledgment-number 79

- The second-segment is sent from the server to the client.
- Two purpose of the segment:

  i) It provides an acknowledgment of the data the server has received.

  ii) It is used to echo back the letter ‗C‘.
- The acknowledgment for client-to-server data is carried in a segment carrying server-to-client data.
- This acknowledgment is said to be piggybacked on the server-to-client data-segment.

- The third segment is sent from the client to the server.
- One purpose of the segment:

  i) It acknowledges the data it has received from the server.

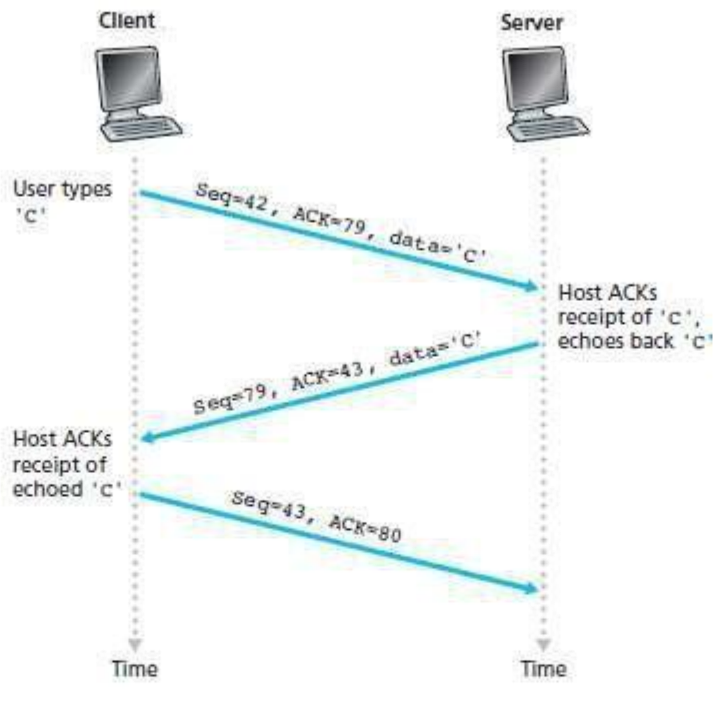


Figure 2.27: Sequence and acknowledgment-numbers for a simple Telnet application over TCP

2.5.3 Round Trip Time Estimation and Timeout

- TCP uses a timeout/retransmit mechanism to recover from lost segments.
- Clearly, the timeout should be larger than the round-trip-time (RTT) of the connection.

2.5.3.1 Estimating the Round Trip Time

- SampleRTT is defined as
  ―The amount of time b/w when the segment is sent and when an acknowledgment is received.‖
- Obviously, the SampleRTT values will fluctuate from segment to segment due to congestion.
- TCP maintains an average of the SampleRTT values, which is referred to as EstimatedRTT.

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

- DevRTT is defined as
  ―An estimate of how much SampleRTT typically deviates from EstimatedRTT.‖

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

- If the SampleRTT values have little fluctuation, then DevRTT will be small.
  If the SampleRTT values have huge fluctuation, then DevRTT will be large.

2.5.3.2 Setting and Managing the Retransmission Timeout Interval

- What value should be used for timeout interval?
- Clearly, the interval should be greater than or equal to EstimatedRTT.
- Timeout interval is given by:

$$\text{TimeoutInterval} = \text{EstimatedRTT} + 4 \cdot \text{DevRTT}$$

2.5.4 Reliable Data Transfer

- IP is unreliable i.e. IP does not guarantee data delivery.
  IP does not guarantee in-order delivery of data. IP
  does not guarantee the integrity of the data.
- TCP creates a reliable data-transfer-service on top of IP‘s unreliable-service.
- At the receiver, reliable-service means
  → data-stream is uncorrupted
  → data-stream is without duplication and
  → data-stream is in sequence.

2.5.4.1 A Few Interesting Scenarios

2.5.4.1.1 First Scenario

- As shown in Figure 2.28, Host-A sends one segment to Host-B.
- Assume the acknowledgment from B to A gets lost.
- In this case, the timeout event occurs, and Host-A retransmits the same segment.
- When Host-B receives retransmission, it observes that the sequence-no has already been received.
- Thus, Host-B will discard the retransmitted-segment.

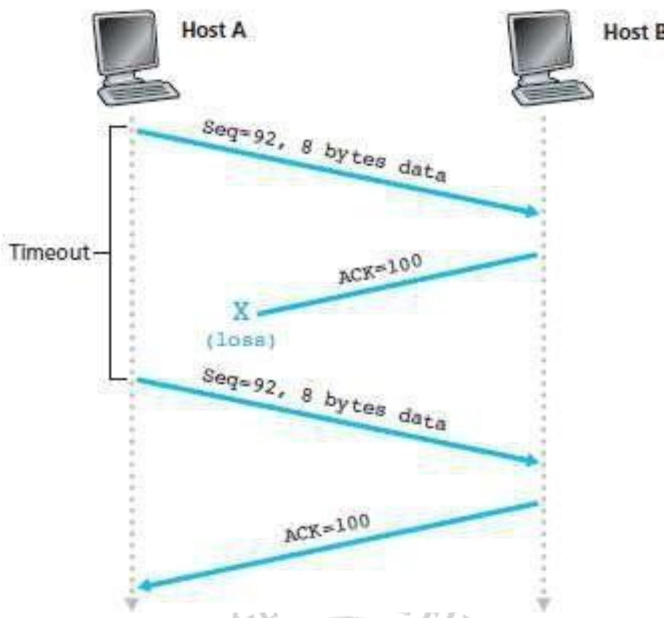


Figure 2.28: Retransmission due to a lost acknowledgment

2.5.4.1.2 Second Scenario

• As shown in Figure 2.29, Host-A sends two segments back-to-back.
• Host-B sends two separate acknowledgments.
• Suppose neither of the acknowledgments arrives at Host-A before the timeout.
• When the timeout event occurs, Host-A resends the first-segment and restarts the timer.
• The second-segment will not be retransmitted until ACK for the second-segment arrives before the new timeout.

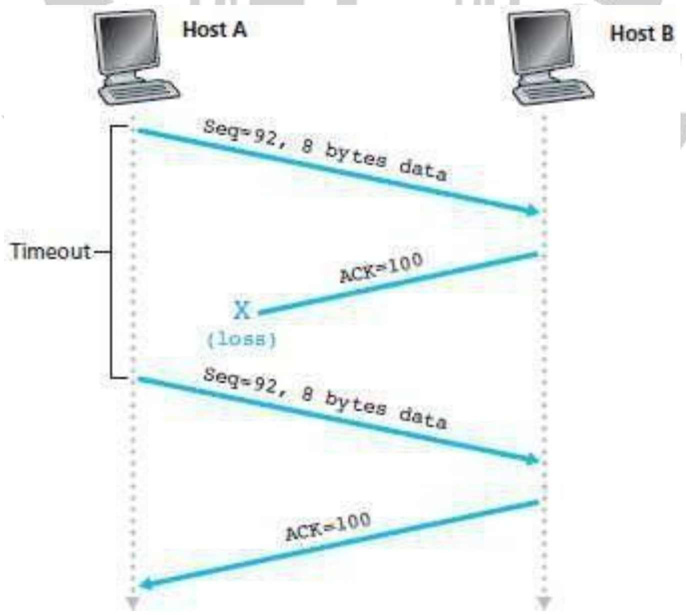


Figure 2.29: Segment 100 not retransmitted

2.5.4.1.3 Third Scenario

• As shown in Figure 2.30, Host-A sends the two segments.
• The acknowledgment of the first-segment is lost.
• But just before the timeout event, Host-A receives an acknowledgment-no 120.
• Therefore, Host-A knows that Host-B has received all the bytes up to 119.
• So, Host-A does not resend either of the two segments.

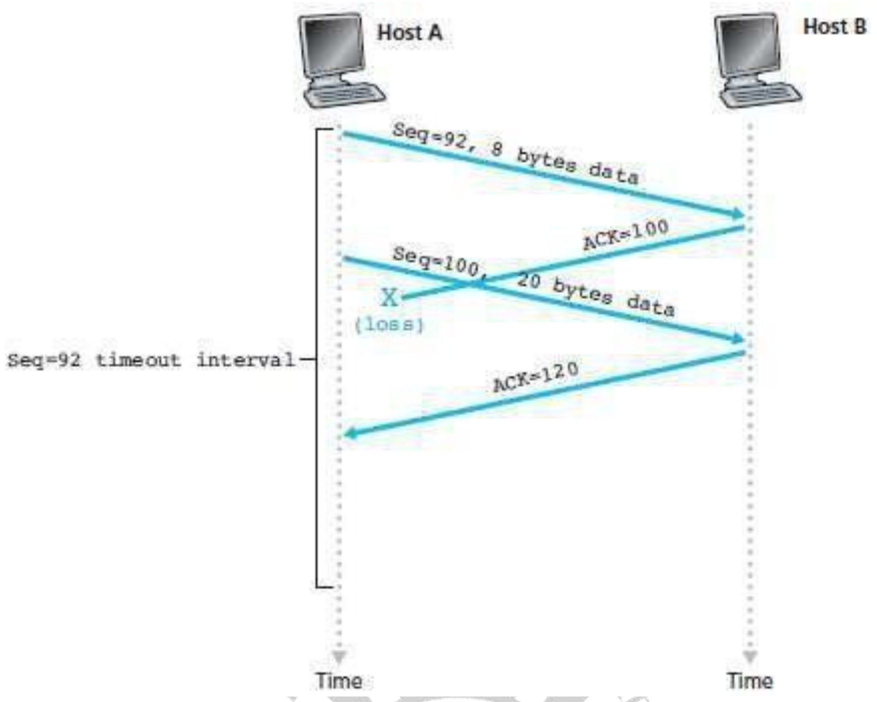


Figure 2.30: A cumulative acknowledgment avoids retransmission of the first-segment

2.5.4.2 Fast Retransmit

- The timeout period can be relatively long.
- The sender can often detect packet-loss well before the timeout occurs by noting duplicate ACKs.
- A duplicate ACK refers to ACK the sender receives for the second time. (Figure 2.31).

Table 2.3: TCP ACK Generation Recommendation

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

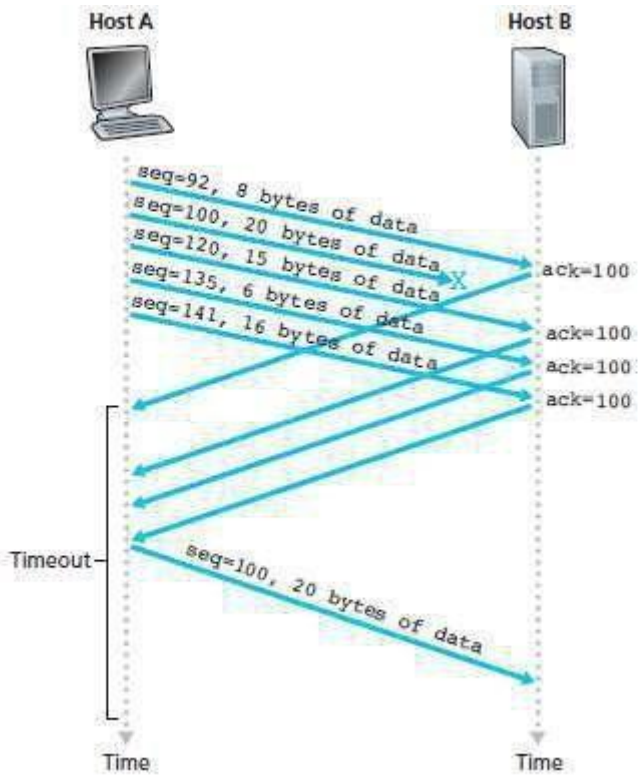


Figure 2.31: Fast retransmit: retransmitting the missing segment before the segment's timer expires

2.5.5 Flow Control

- TCP provides a flow-control service to its applications.
- A flow-control service eliminates the possibility of the sender overflowing the receiver-buffer.

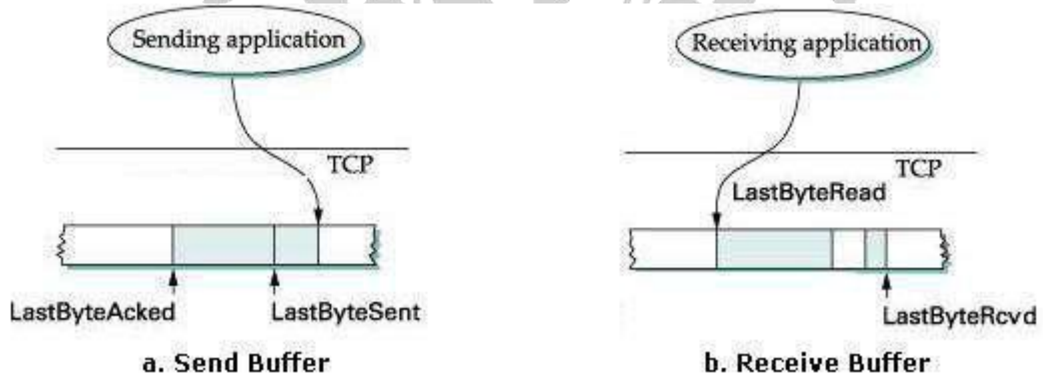


Figure 2.32: a) Send buffer and b) Receive Buffer

- As shown in Figure 2.32, we define the following variables:
    1) MaxSendBuffer: A send-buffer allocated to the sender.
    2) MaxRcvBuffer: A receive-buffer allocated to the receiver.
    3) LastByteSent: The no. of the last bytes sent to the send-buffer at the sender.
    4) LastByteAcked: The no. of the last bytes acknowledged in the send-buffer at the sender.
    5) LastByteRead: The no. of the last bytes read from the receive-buffer at the receiver.
    6) LastByteRcvd: The no. of the last bytes arrived & placed in receive-buffer at the receiver.

Send Buffer

- Sender maintains a send buffer, divided into 3 segments namely
    1) Acknowledged data
    2) Unacknowledged data and
    3) Data to be transmitted
- Send buffer maintains 2 pointers: LastByteAcked and LastByteSent. The relation b/w these two is:

LastByteAcked ≤ LastByteSent

Receive Buffer

- Receiver maintains receive buffer to hold data even if it arrives out-of-order.
- Receive buffer maintains 2 pointers: LastByteRead and LastByteRcvd. The relation b/w these two is:

LastByteRead ≤ LastByteRcvd + 1

Flow Control Operation

- Sender prevents overflowing of send buffer by maintaining

LastByteWritten − LastByteAcked ≤ MaxSendBuffer

- Receiver avoids overflowing receive buffer by maintaining

LastByteRcvd − LastByteRead ≤ MaxRcvBuffer

- Receiver throttles the sender by advertising a window that is smaller than the amount of free space that it can buffer as:

AdvertisedWindow = MaxRcvBuffer − (LastByteRcvd − LastByteRead)

2.5.6 TCP Connection Management

2.5.6.1 Connection Setup & Data Transfer

- To setup the connection, three segments are sent between the two hosts. Therefore, this process is referred to as a three-way handshake.
- Suppose a client-process wants to initiate a connection with a server-process.
- Figure 2.33 illustrates the steps involved:

  - ➢ The client first sends a connection-request segment to the server.
  - ➢ The connection-request segment contains:
    1) SYN bit is set to 1.
    2) Initial sequence-number (client_isn).
  - ➢ The SYN segment is encapsulated within an IP datagram and sent to the server.

  Step 2: Server sends a connection-granted segment to the Client

  - ➢ Then, the server
    → extracts the SYN segment from the datagram
    → allocates the buffers and variables to the connection and
    → sends a connection-granted segment to the client.
  - ➢ The connection-granted segment contains:
    1) SYN bit is set to 1.
    2) Acknowledgment field is set to client_isn+1.
    3) Initial sequence-number (server_isn).

  - ➢ Finally, the client
    → allocates buffers and variables to the connection and
    → sends an ACK segment to the server
  - ➢ The ACK segment acknowledges the server.
  - ➢ SYN bit is set to zero, since the connection is established.

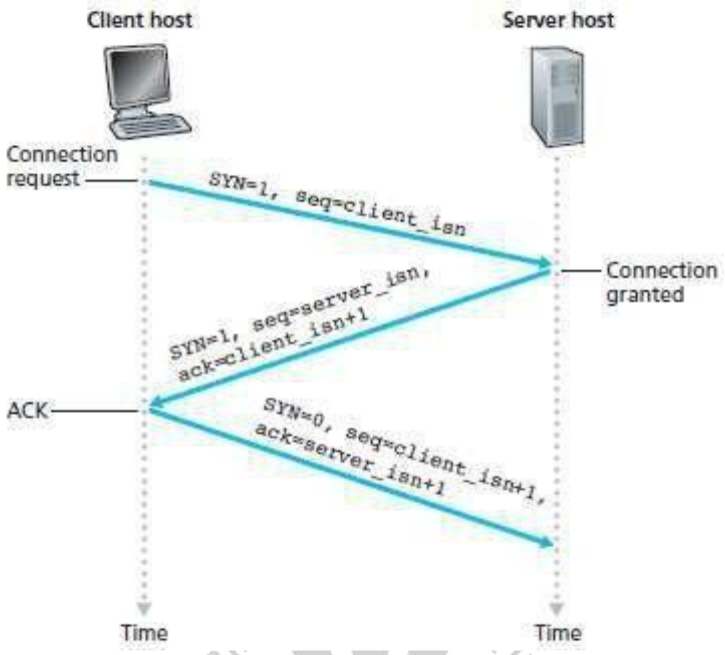


Figure 2.33: TCP three-way handshake: segment exchange

2.5.6.2 Connection Release

- Either of the two processes in a connection can end the connection.
- When a connection ends, the ―resources‖ in the hosts are de-allocated.
- Suppose the client decides to close the connection.
- Figure 2.34 illustrates the steps involved:
    1) The client-process issues a close command.
        - ¤ Then, the client sends a shutdown-segment to the server.
        - ¤ This segment has a FIN bit set to 1.
    2) The server responds with an acknowledgment to the client.
    3) The server then sends its own shutdown-segment.
        - ¤ This segment has a FIN bit set to 1.
    4) Finally, the client acknowledges the server‘s shutdown-segment.

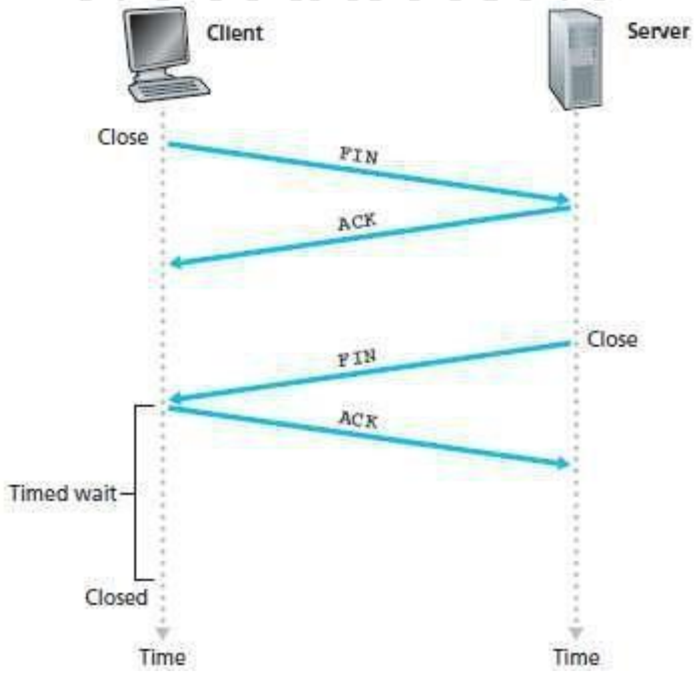


Figure 2.34: Closing a TCP connection

2.6 Principles of Congestion Control

2.6.1 The Causes and the Costs of Congestion

2.6.1.1 Scenario 1: Two Senders, a Router with Infinite Buffers

- Two hosts (A & B) have a connection that shares a single-hop b/w source & destination.
- This is illustrated in Figure 2.35.

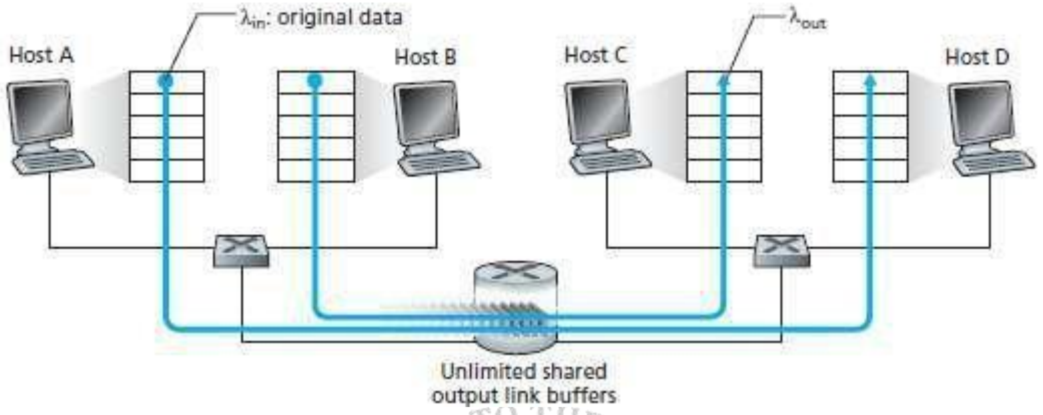

λ_{in}: original data
λ_{out}
Host A
Host B
Host C
Host D
Unlimited shared output link buffers

Figure 2.35: Congestion scenario 1: Two connections sharing a single hop with infinite buffers

- Let
  Sending-rate of Host-A = λin bytes/sec
  Outgoing Link's capacity = R

- Packets from Hosts A and B pass through a router and over a shared outgoing link.
- The router has buffers.
- The buffers stores incoming packets when packet-arrival rate exceeds the outgoing link's capacity.

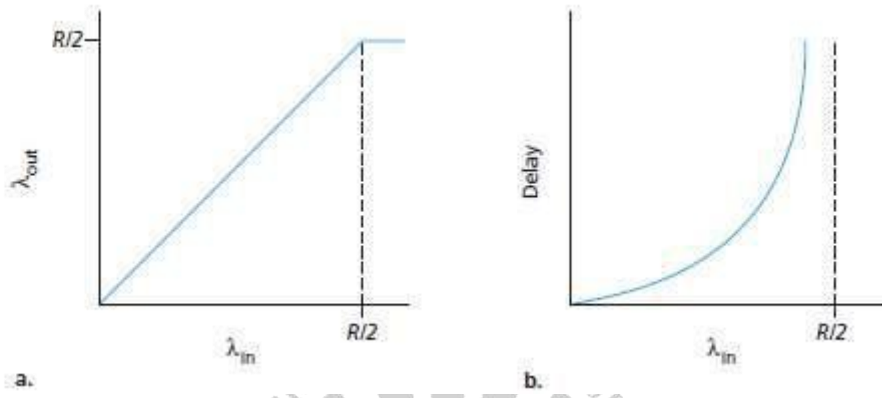


Figure 2.36: Congestion scenario 1: Throughput and delay as a function of host sending-rate

- Figure 2.36 plots the performance of Host-A's connection.

Left Hand Graph
- The left graph plots the per-connection throughput as a function of the connection-sending-rate.
- For a sending-rate b/w 0 and R/2, the throughput at the receiver equals the sender's sending-rate.
  However, for a sending-rate above R/2, the throughput at the receiver is only R/2. (Figure 2.36a)
- Conclusion: he link cannot deliver packets to a receiver at a steady-state rate that exceeds R/2.

Right Hand Graph
- The right graph plots the average delay as a function of the connection-sending-rate (Figure 2.36b).
- As the sending-rate approaches R/2, the average delay becomes larger and larger.

- Conclusion: Large queuing delays are experienced as the packet arrival rate nears the link capacity.

2.6.1.2 Scenario 2: Two Senders and a Router with Finite Buffers
- Here, we have 2 assumptions (Figure 2.37):
  1) The amount of router buffering is finite.
     ➢ Packets will be dropped when arriving to an already full buffer.
  2) Each connection is reliable.
     ➢ If a packet is dropped at the router, the sender will eventually retransmit it.
- Let
  Application's sending-rate of Host-A = λin bytes/sec
  Transport -layer's sending-rate of Host-A = λin bytes/sec (also called offered-load to network) Outgoing Link's capacity = R

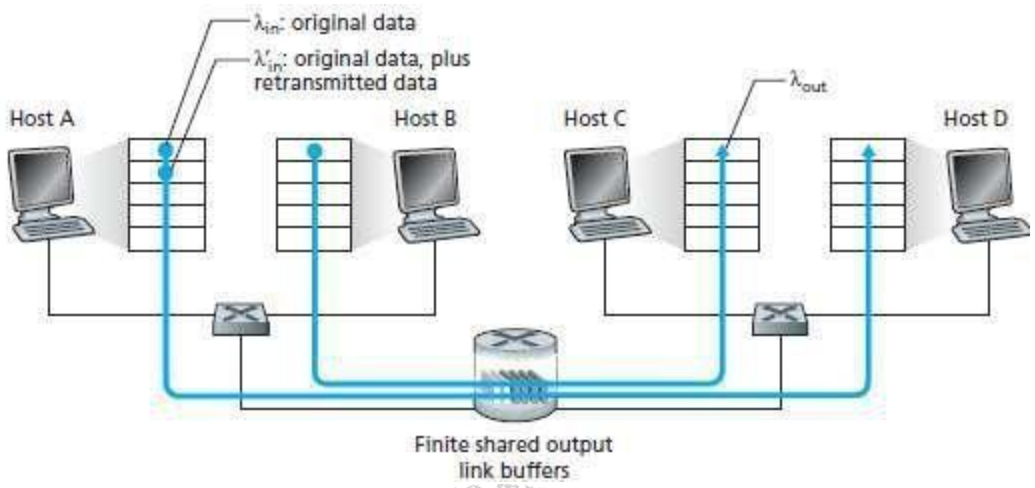

λ_{in}: original data
λ'_{in}: original data, plus retransmitted data
λ_{out}
Host A
Host B
Host C
Host D
Finite shared output link buffers

Figure 2.37: Scenario 2: Two hosts (with retransmissions) and a router with finite buffers

Case 1 (Figure 2.38(a)):
- Host-A sends a packet only when a buffer is free.
- In this case,
  - → no loss occurs
  - → λin will be equal to λin', and
  - → throughput of the connection will be equal to λin.
- The sender retransmits only when a packet is lost.
- Consider the offered-load λin' = R/2.
- The rate at which data are delivered to the receiver application is R/3.
- The sender must perform retransmissions to compensate for lost packets due to buffer overflow.

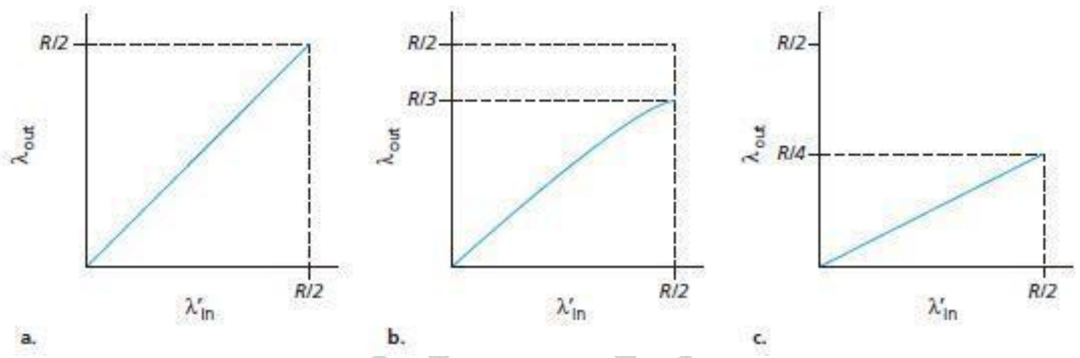


Figure 2.38: Scenario 2 performance with finite buffers

Case 3 (Figure 2.38(c)):
- The sender may time out & retransmit a packet that has been delayed in the queue but not yet lost.
- Both the original data packet and the retransmission may reach the receiver.
- The receiver needs one copy of this packet and will discard the retransmission.
- The work done by the router in forwarding the retransmitted copy of the original packet was wasted.

2.6.1.3 Scenario 3: Four Senders, Routers with Finite Buffers, and Multihop Paths
- Four hosts transmit packets, each over overlapping two-hop paths.
- This is illustrated in Figure 2.39.

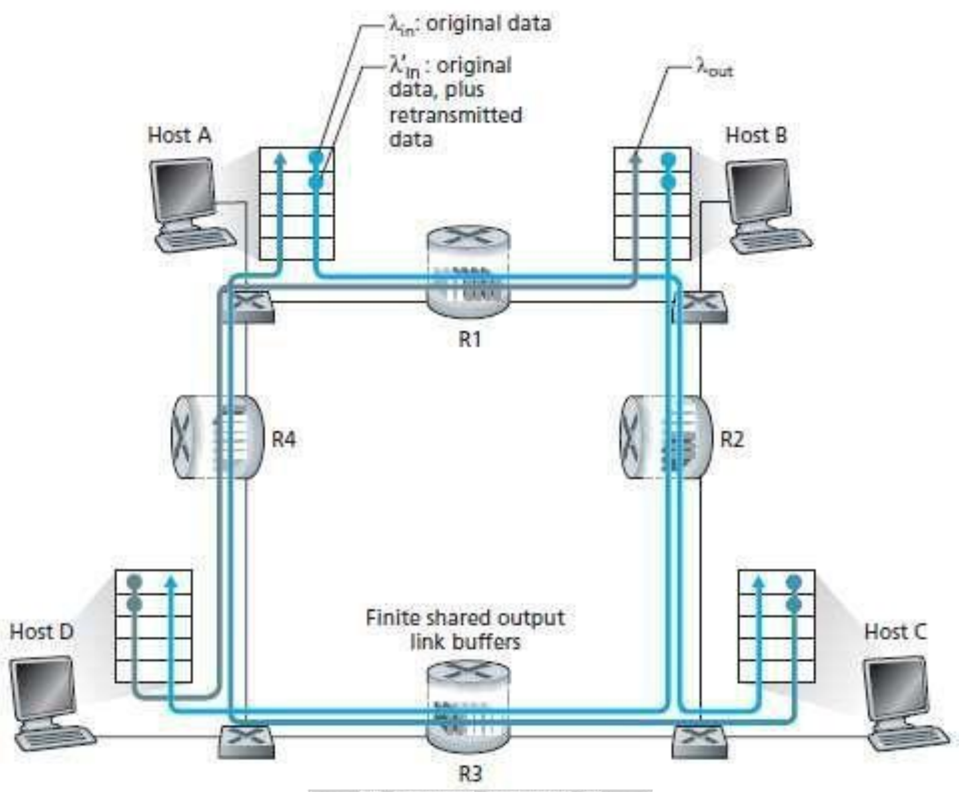

λ_{in}: original data
λ'_{in}: original data, plus retransmitted data
λ_{out}
Host A
Host B
R1
R4
R2
Host D
Host C
Finite shared output link buffers
R3

Figure 2.39: Four senders, routers with finite buffers, and multihop paths

- 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.
- Case-1: For extremely small values of λin,
  → buffer overflows are rare (as in congestion scenarios 1 and 2) and
  → the throughput approximately equals the offered-load.
- Case-2: For slightly larger values of λin, the corresponding throughput is also larger. This is because
  → more original data is transmitted into the network
  → data is delivered to the destination and
  → overflows are still rare.
- Case-3: For extremely larger values of λin.
  - ➢ Consider router R2.
  - ➢ The A–C traffic arriving to router R2 can have an arrival rate of at most R regardless of the value of λin.
    where R = the capacity of the link from R1 to R2,.
  - ➢ If λin is extremely large for all connections, then the arrival rate of B–D traffic at R2 can be much larger than that of the A–C traffic.
  - ➢ The A–C and B–D traffic must compete at router R2 for the limited amount of buffer-space.
  - ➢ Thus, the amount of A–C traffic that successfully gets through R2 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.
  - ➢ When a packet is dropped along a path, the transmission capacity ends up having been wasted.

2.6.2 Approaches to Congestion Control

- Congestion-control approaches can be classified based on whether the network-layer provides any explicit assistance to the transport-layer:

  - ➢ The network-layer provides no explicit support to the transport-layer for congestion-control.

- ➢ Even the presence of congestion must be inferred by the end-systems based only on observed network-behavior.
- ➢ Segment loss is taken as an indication of network-congestion and the window-size is decreased accordingly.

2) Network Assisted congestion Control

- ➢ Network-layer components provide explicit feedback to the sender regarding congestion.
- ➢ This feedback may be a single bit indicating congestion at a link.
- ➢ Congestion information is fed back from the network to the sender in one of two ways:
  - i) Direct feedback may be sent from a network-router to the sender (Figure 2.40).
  - ¤ This form of notification typically takes the form of a choke packet.
  - ii) A router marks 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.
  - ¤ This form of notification takes at least a full round-trip time.

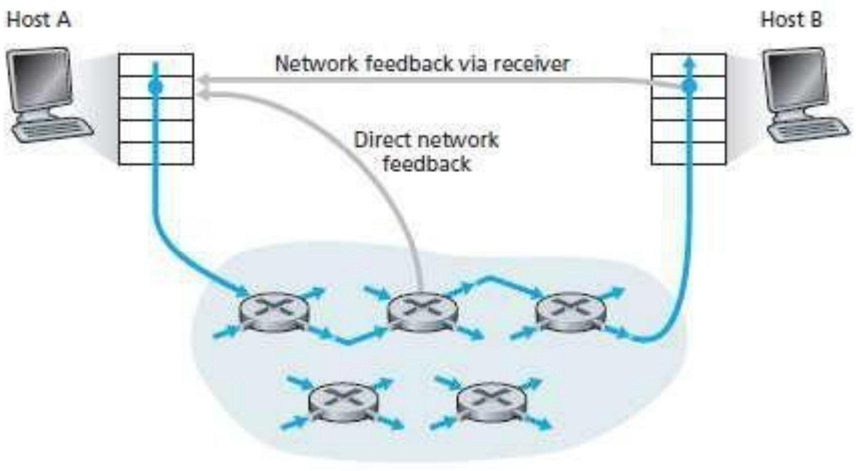


Figure 2.40: Two feedback pathways for network-indicated congestion information

2.6.3 Network Assisted Congestion Control Example: ATM ABR Congestion Control

- ATM (Asynchronous Transfer Mode) protocol uses network-assisted approach for congestion-control.
- ABR (Available Bit Rate) has been designed as an elastic data-transfer-service.
  - i) When the network is underloaded, ABR has to take advantage of the spare available bandwidth.
  - ii) When the network is congested, ABR should reduce its transmission-rate.

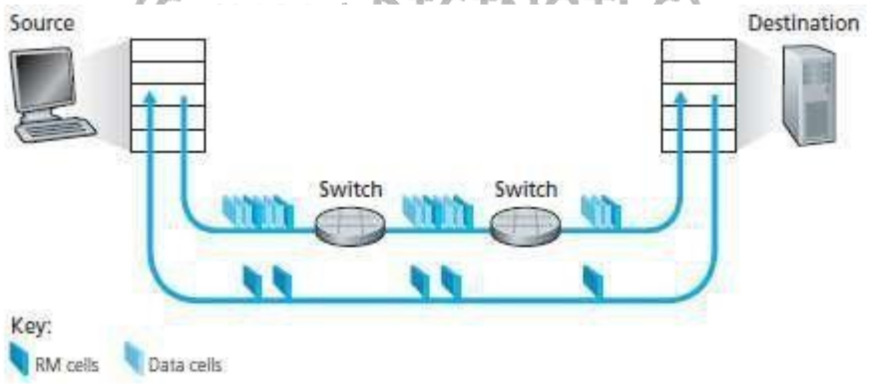


Figure 2.41: Congestion-control framework for ATM ABR service

- Figure 2.41 shows the framework for ATM ABR congestion-control.
- Data-cells are transmitted from a source to a destination through a series of intermediate switches.
- RM-cells are placed between the data-cells. (RM Resource Management).
- The RM-cells are used to send congestion-related information to the hosts & switches.
- When an RM-cell arrives at a destination, the cell will be sent back to the sender

- Thus, RM-cells can be used to provide both
  → direct network feedback and
  → network feedback via the receiver.

2.6.3.1 Three Methods to indicate Congestion

- ATM ABR congestion-control is a rate-based approach.
- ABR provides 3 mechanisms for indicating congestion-related information:

  - Each data-cell contains an EFCI bit. (EFCI □ Explicit forward congestion indication)
  - A congested-switch sets the EFCI bit to 1 to signal congestion to the destination.
  - The destination must check the EFCI bit in all received data-cells.
  - If the most recently received data-cell has the EFCI bit set to 1, then the destination
    → sets the CI bit to 1 in the RM-cell (CI □ congestion indication)
    → sends the RM-cell back to the sender.
  - Thus, a sender can be notified about congestion at a network switch.

  - The rate of RM-cell interspersion is a tunable parameter.
  - The default value is one RM-cell every 32 data-cells. (NI □ No Increase)
  - The RM-cells have a CI bit and a NI bit that can be set by a congested-switch.
  - A switch
    → sets the NI bit to 1 in a RM-cell under mild congestion and
    → sets the CI bit to 1 under severe congestion conditions.

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

2.7 TCP Congestion Control

2.7.1 TCP Congestion Control

- TCP has congestion-control mechanism.
- TCP uses end-to-end congestion-control rather than network-assisted congestion-control
- Here is how it works:
  - Each sender limits the rate at which it sends traffic into its connection as a function of perceived congestion.
    i) If sender perceives that there is little congestion, then sender increases its data-rate.
    ii) If sender perceives that there is congestion, then sender reduces its data-rate.
- This approach raises three questions:
  1) How does a sender limit the rate at which it sends traffic into its connection?
  2) How does a sender perceive that there is congestion on the path?
  3) What algorithm should the sender use to change its data-rate?
- The sender keeps track of an additional variable called the congestion-window (cwnd).
- The congestion-window imposes a constraint on the data-rate of a sender.
- The amount of unacknowledged-data at a sender will not exceed minimum of (cwnd & rwnd), that is:

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

- The sender‗s data-rate is roughly cwnd/RTT bytes/sec.
- Explanation of Loss event:
  - A ―loss event‖ at a sender is defined as the occurrence of either
    → timeout or
    → receipt of 3 duplicate ACKs from the receiver.
  - Due to excessive congestion, the router-buffer along the path overflows. This causes a datagram to be dropped.
  - The dropped datagram, in turn, results in a loss event at the sender.
  - The sender considers the loss event as an indication of congestion on the path.
- How congestion is detected?
  - Consider the network is congestion-free.
  - Acknowledgments for previously unacknowledged segments will be received at the sender.
  - TCP
    → will take the arrival of these acknowledgments as an indication that all is well and
    → will use acknowledgments to increase the window-size (& hence data-rate).

- ➢ TCP is said to be self-clocking because
  → acknowledgments are used to trigger the increase in window-size
- ➢ Congestion-control algorithm has 3 major components:
  1) Slow start
  2) Congestion avoidance and
  3) Fast recovery.

2.7.1.1 Slow Start

- When a TCP connection begins, the value of cwnd is initialized to 1 MSS.
- TCP doubles the number of packets sent every RTT on successful transmission.
- Here is how it works:
  - ➢ As shown in Figure 2.42, the TCP
    → sends the first-segment into the network and
    → waits for an acknowledgment.
  - ➢ When an acknowledgment arrives, the sender
    → increases the congestion-window by one MSS and
    → sends out 2 segments.
  - ➢ When two acknowledgments arrive, the sender
    → increases the congestion-window by one MSS and
    → sends out 4 segments.
  - ➢ This process results in a doubling of the sending-rate every RTT.
- Thus, the TCP data-rate starts slow but grows exponentially during the slow start phase.

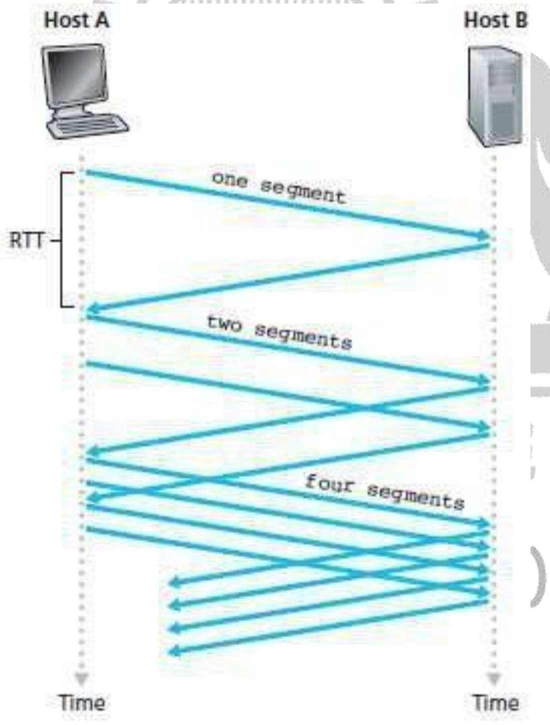


Figure 2.42: TCP slow start

- When should the exponential growth end?
  - ➢ Slow start provides several answers to this question.
    1) If there is a loss event, the sender
    → sets the value of cwnd to 1 and
    → begins the slow start process again. (ssthresh –slow start threshold‖)
    → sets the value of ssthresh to cwnd/2.
    2) When the value of cwnd equals ssthresh, TCP enters the congestion avoidance state.
    3) When three duplicate ACKs are detected, TCP
    → performs a fast retransmit and
    → enters the fast recovery state.
- TCP‘s behavior in slow start is summarized in FSM description in Figure 2.43.

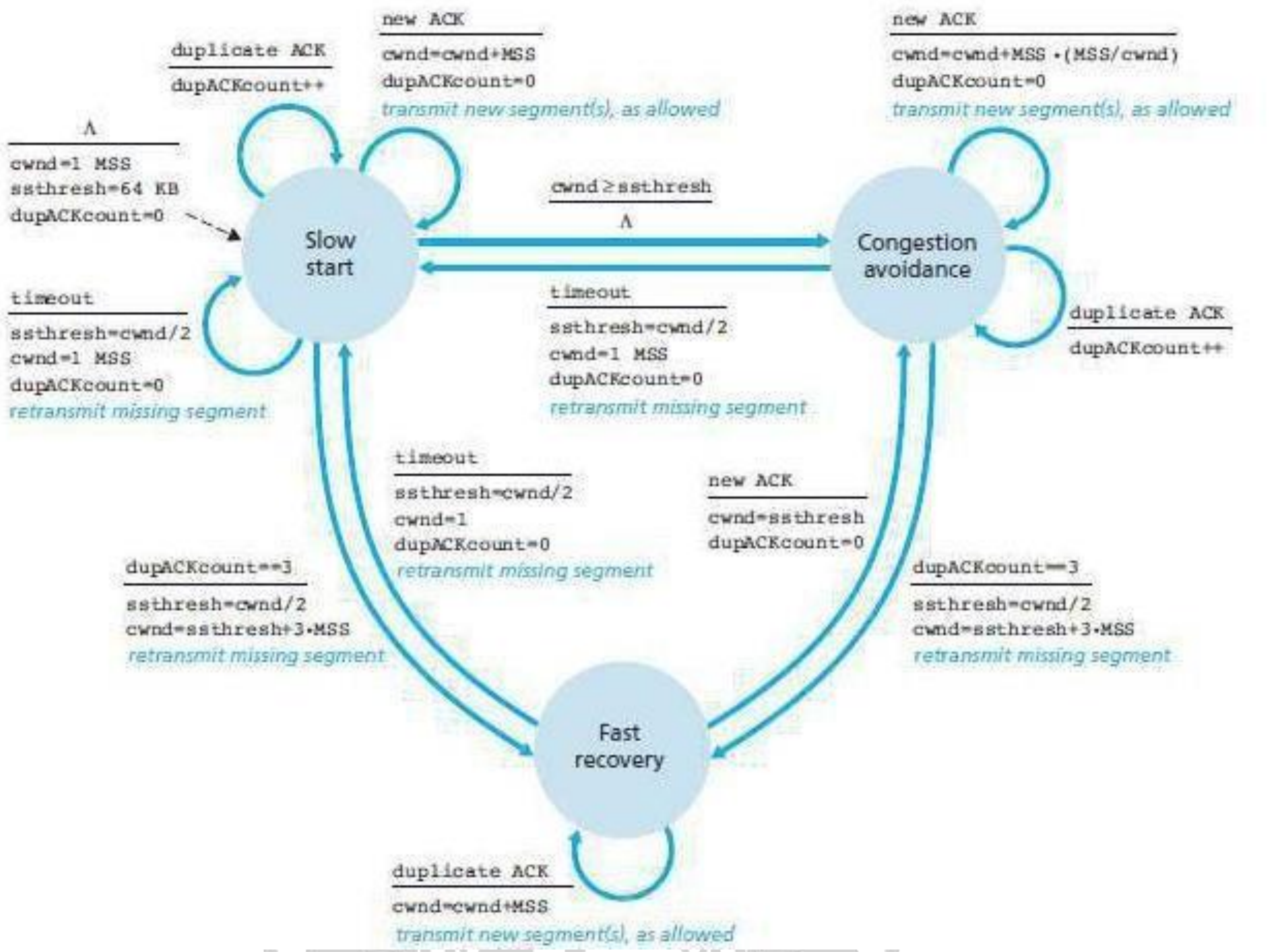


Figure 2.43: FSM description of TCP congestion-control

2.7.1.2 Congestion Avoidance
- On entry to congestion-avoidance state, the value of cwnd is approximately half its previous value.
- Thus, the value of cwnd is increased by a single MSS every RTT.
- The sender must increases cwnd by MSS bytes (MSS/cwnd) whenever a new acknowledgment arrives
- When should linear increase (of 1 MSS per RTT) end?
  1) When a timeout occurs.
  ➢ When the loss event occurred,
    → value of cwnd is set to 1 MSS and
    → value of ssthresh is set to half the value of cwnd.
  2) When triple duplicate ACK occurs.
  ➢ When the triple duplicate ACKs were received,
    → value of cwnd is halved.
    → value of ssthresh is set to half the value of cwnd.

2.7.1.3 Fast Recovery
- The value of cwnd is increased by 1 MSS for every duplicate ACK received.
- When an ACK arrives for the missing segment, the congestion-avoidance state is entered.
- If a timeout event occurs, fast recovery transitions to the slow-start state.
- When the loss event occurred
  → value of cwnd is set to 1 MSS, and
  → value of ssthresh is set to half the value of cwnd.
- There are 2 versions of TCP:

  ➢ An early version of TCP was known as TCP Tahoe.
  ➢ TCP Tahoe
    → cut the congestion-window to 1 MSS and
    → entered the slow-start phase after either
      i) timeout-indicated or

ii) triple-duplicate-ACK-indicated loss event.

2) TCP Reno

- The newer version of TCP is known as TCP Reno.
- TCP Reno incorporated fast recovery.
- Figure 2.44 illustrates the evolution of CP‘s congestion-window for both Reno and Tahoe.

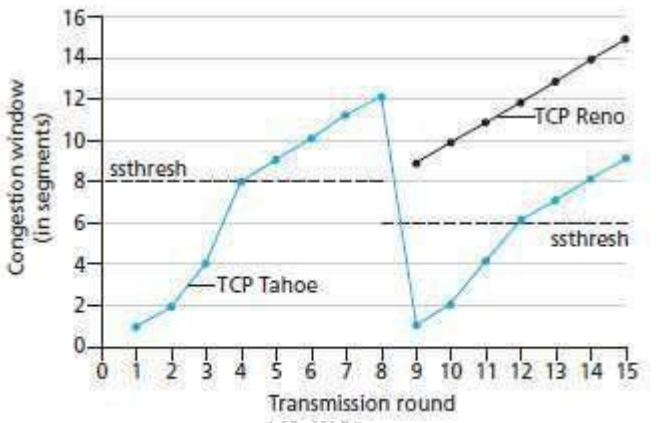


Figure 2.44: Evolution of TCP‘s congestion-window (Tahoe and Reno)

2.7.1.4 TCP Congestion Control: Retrospective

- TCP‘s congestion-control consists of (AIMD additive increase, multiplicative decrease)
  - → Increasing linearly (additive) value of cwnd by 1 MSS per RTT and
  - → Halving (multiplicative decrease) value of cwnd on a triple duplicate-ACK event.
- For this reason, TCP congestion-control is often referred to as an AIMD.
- AIMD congestion-control gives rise to the ―saw tooth‖ behavior shown in Figure 2.45.
- TCP
  - → increases linearly the congestion-window-size until a triple duplicate-ACK event occurs and
  - → decreases then the congestion-window-size by a factor of 2

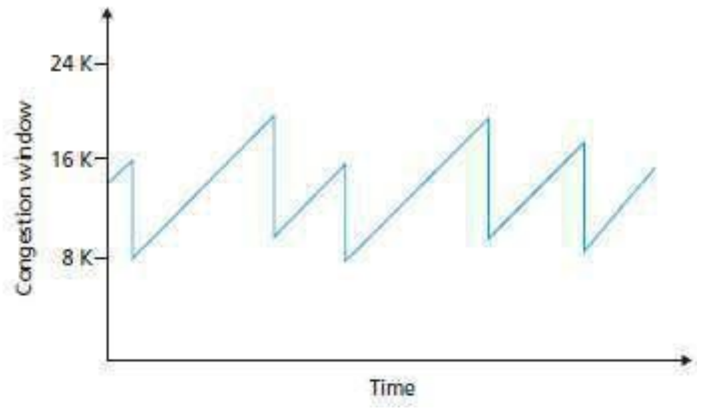


Figure 2.45: Additive-increase, multiplicative-decrease congestion-control

2.7.2 Fairness

- Congestion-control mechanism is fair if each connection gets equal share of the link-bandwidth.
- As shown in Figure 2.46, consider 2 TCP connections sharing a single link with transmission-rate R.
- Assume the two connections have the same MSS and RTT.

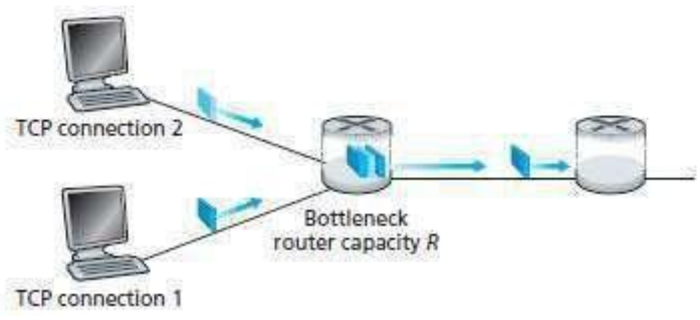


Figure 2.46: Two TCP connections sharing a single bottleneck link

- Figure 2.47 plots the throughput realized by the two TCP connections.
    - If TCP shares the link-bandwidth equally b/w the 2 connections,
      then the throughput falls along the 45-degree arrow starting from the origin.

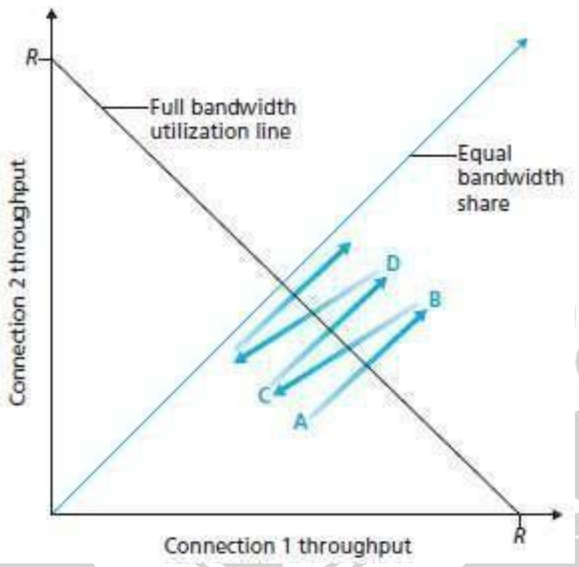


Figure 2.47: Throughput realized by TCP connections 1 and 2

2.7.2.1 Fairness and UDP

- Many multimedia-applications (such as Internet phone) often do not run over TCP.
- Instead, these applications prefer to run over UDP. This is because
  → applications can pump their audio into the network at a constant rate and
  → occasionally lose packets.

2.7.2.2 Fairness and Parallel TCP Connections

- Web browsers use multiple parallel-connections to transfer the multiple objects within a Web page.
- Thus, the application gets a larger fraction of the bandwidth in a congested link.
- ∴ Web-traffic is so pervasive in the Internet; multiple parallel-connections are common nowadays.