# Chapter 3 : Transport layer

Residing between the application and network layers, the transport layer is a central piece of the layered network architecture.

It has the critical role of providing communication services directly to the application processes running on different hosts.

## 3.1 Introduction and Transport-Layer Services

A transport-layer protocol provides **logical communication** between application processes running on different hosts.

By *logical communication*, we mean that from an application's perspective, it is as if the hosts running the processes were directly connected; in reality, the hosts may be on opposite sides of the planet, connected via numerous routers and a wide range of link types.

Transport-layer protocols are implemented in the end systems but not in network routers/switches.

### Source application to destination application delivery flow:

- Transport layer the application messages into smaller chunks known as **segments** and adds a transport-layer header to each chunk.
- 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 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.
- There are mainly two transport-layer protocols—TCP and UDP. Each of these protocols provides a different set of transport-layer services to the invoking application.

### Transport layer Vs. Network Layer

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

**Example:**

Consider two houses, one on the East Coast and the other on the West Coast, with each house being home to a dozen kids. The kids in the East Coast household are cousins of the kids in the West Coast household. The kids in the two households love to write to each other—each kid writes each cousin every week.

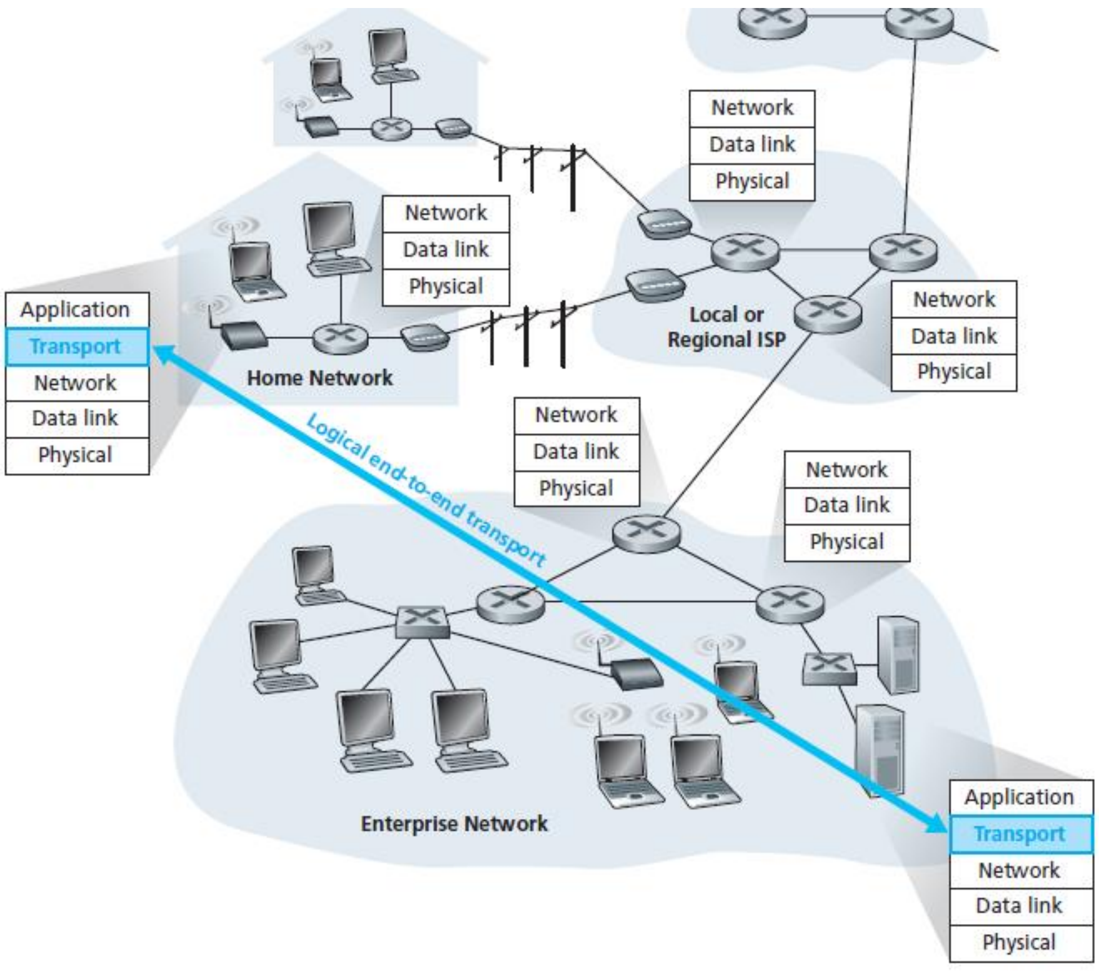
Letter from home to home will be delivered by the traditional postal service in a separate envelope. In each of the households there is one kid—Ann in the West Coast house and Bill in the East Coast house—responsible for mail collection and mail distribution to other siblings.

This household example serves as a nice analogy for explaining how the transport layer relates to the network layer:

- application messages = letters in envelopes
- processes = cousins
- hosts (also called end systems) = houses
- transport-layer protocol = Ann and Bill
- network-layer protocol = postal service (including mail carriers)

Note that Ann and Bill do all their work within their respective homes; they are not involved in sorting mail in any intermediate mail center or in moving mail from one mail center to another.

Similarly, transport-layer protocols live in the end systems. Within an end system, a transport protocol moves messages from application processes to the network layerand vice versa, but it doesn't have any knowledge about how the messages are moved within the network core.



## 3.2 Multiplexing and Demultiplexing:

Transport-layer multiplexing and demultiplexing extends host-to-host delivery service provided by the network layer to a process-to-process delivery service for applications running on the hosts.

A process (as part of a network application) can have one or more **sockets** – door between application layer and transport layer, through which data passes from underlying network to the process and from the process to the network.

The transport layer in the hosts does not actually sends or receives data directly to and from a process, but instead to an intermediary socket.

**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**.

**Demultiplexing:**

The job of delivering the data in a transport-layer segment received from network layer to the correct socket is called **demultiplexing**.

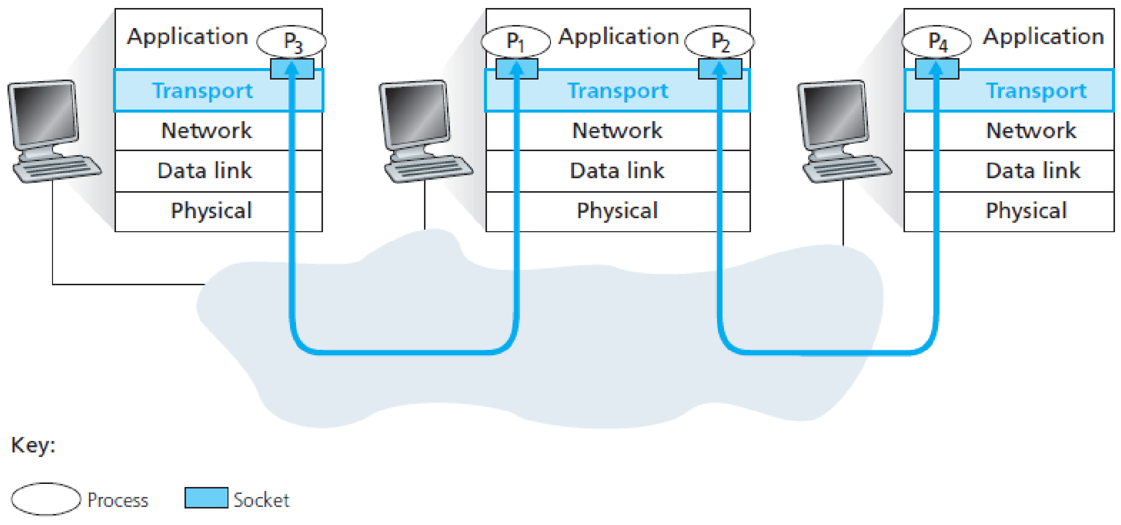


**Figure 3.2** ♦ Transport-layer multiplexing and demultiplexing

**Transport-layer multiplexing requires:**

(1) sockets having unique identifiers, and

(2) each segment having 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.

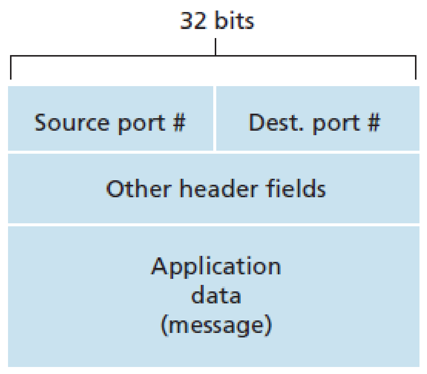


**Figure 3.3** ♦ Source and destination port-number fields in a transport-layer segment

### 3.2.1 Connectionless Multiplexing and Demultiplexing (UDP) :

Python program running in a host can create a UDP socket with the line:

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

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

Alternatively, we can add a line into our Python program after we create the socket to associate a specific port number (say, 19157) to this UDP socket via the socket bind() method:

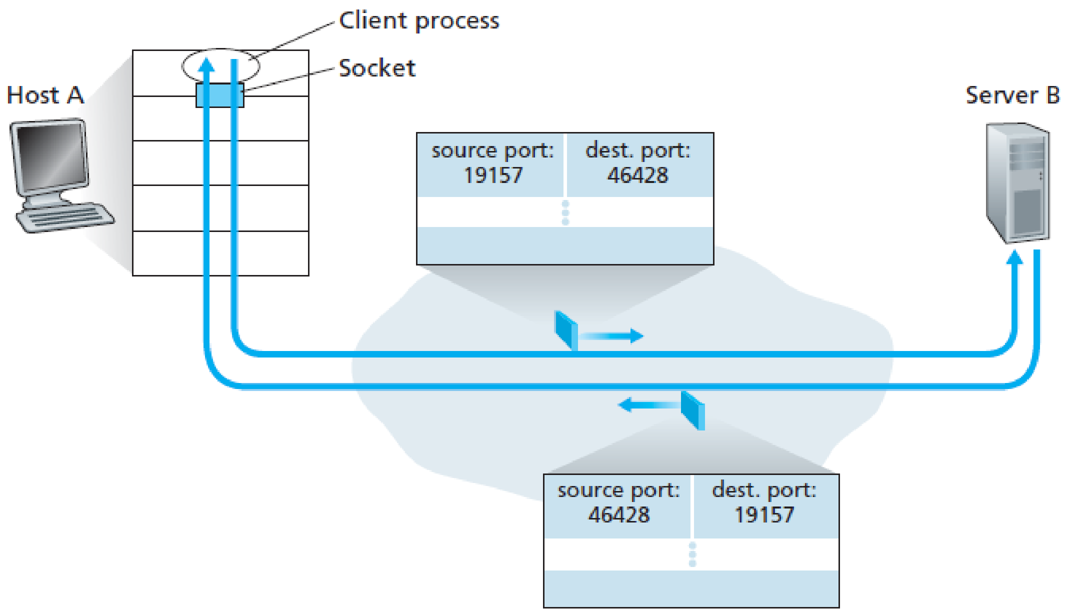
**clientSocket.bind((‘’, 19157))**

### Example flow:

- 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 two 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.

- As UDP segments arrive from the network, Host B directs (demultiplexes) each segment to the appropriate socket by examining the segment's destination port number.
- 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.**

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



### 3.3.2 Connection-Oriented Multiplexing and Demultiplexing (TCP):

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).

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

### Example flow:

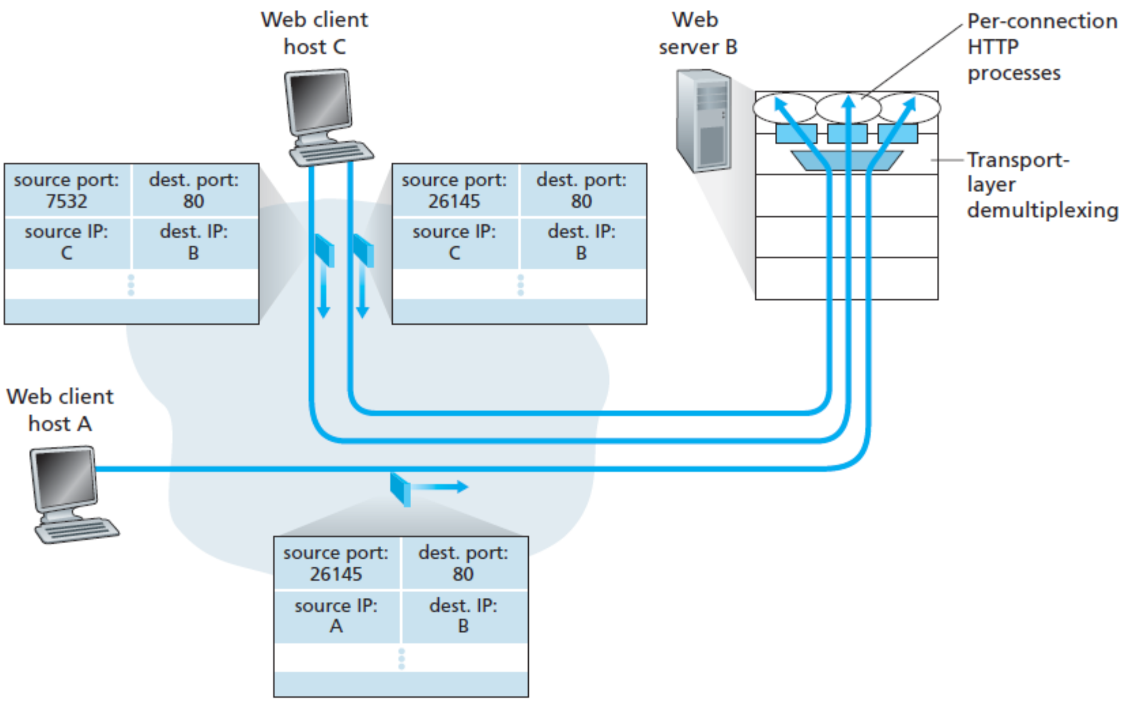
- The TCP server application has a "welcoming socket," that waits for connection establishment requests from TCP clients on port number 12000.
- The TCP client creates a socket and sends a connection establishment request segment with the lines:

**clientSocket = socket(AF_INET, SOCK_STREAM)**
**clientSocket.connect((serverName,12000))**

- A connection-establishment request contains TCP segment with destination port number 12000 and a source port number chosen by the client.
- When the server process receives the incoming connection-request segment with destination port 12000, it locates the server process that is waiting to accept a connection on port number
- 12000. The server process then creates a new socket:

  **connectionSocket, addr = serverSocket.accept()**
- The transport layer at the server notes the following four values in the connection-request segment: (1) the source port number in the segment, (2) the IP address of the source host, (3) the destination port number in the segment, and (4) its own IP address

**If destination IP and port addresses are same for multiple client requests, then TCP creates separate process for each connection request.**



## 3.3 Connectionless Transport: UDP

**UDP (User datagram protocol) is** a no-frills, bare-bones transport protocol (A protocol with minimal set of basic services).

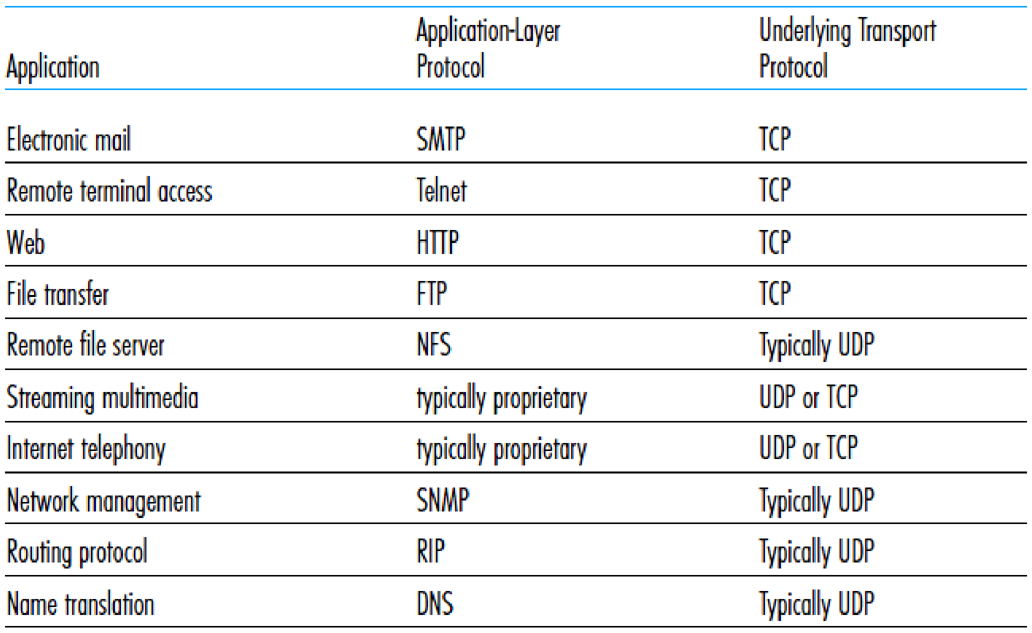
UDP, defined in [RFC 768], does just about as little as a transport protocol can do. Aside from the multiplexing/demultiplexing function and some light error checking, it adds nothing to IP.

UDP takes messages from the application process, attaches source and destination port number fields for the multiplexing/demultiplexing service, adds two other small fields, and passes the resulting segment to the network layer.

In UDP there is no handshaking between sending and receiving transport-layer entities before sending a segment. For this reason, UDP is said to be ***connectionless***.

Some example applications using UDP:

- DNS(Domain Name System)
- SNMP(Simple Network Management Protocol)
- RIP ( Routing Information Protocol)



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

**Figure 3.6 ♦** Popular Internet applications and their underlying transport protocols

## Why to use UDP?

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

1. ***Finer application-level control over what data is sent, and when:***
   Under UDP, as soon as an application process passes data to UDP, UDP will package the data inside a UDP segment and immediately pass the segment to the network layer. No congestion control mechanisms is implemented so no extra overhead of retransmission of lost segments over sender and receiver.
2. ***No connection establishment.***
   Thus UDP does not introduce any delay to establish a connection. This is probably the principal reason why DNS runs over UDP rather than TCP—DNS would be much slower if it ran over TCP.
3. ***No connection state.***
   TCP maintains connection state in the end systems. This connection state includes receive and send buffers, congestion-control parameters, and sequence and acknowledgment number parameters. UDP, on the other hand, does not maintain connection state and does not track any of these parameters. For this reason, a server

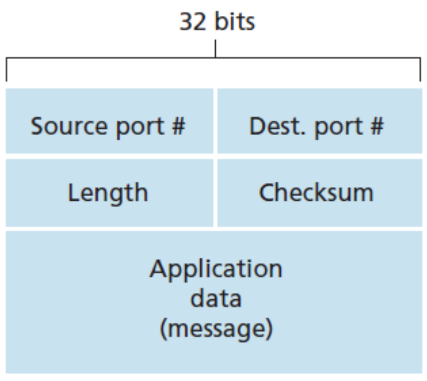
devoted to a particular application can typically support many more active clients when the application runs over UDP rather than TCP.

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

### 4.3.1 UDP Segment Structure:

The UDP header has only four fields each consisting of two bytes (16 bits):

- **The port numbers** (Source and destination) allow the 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 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 1's 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.

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

$$\begin{array}{l} 1011101110110101 \\ \underline{1000111100001100} \\ 0100101011000010 \end{array}$$

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.**

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.

## 3.4 Principles of Reliable Data Transfer

If one had to identify a "top-ten" list of fundamentally important problems in all of networking, reliable data transfer would be a candidate to lead the list.

It is the responsibility of a **reliable data transfer protocol** to implement reliability service (no packet loss, no bit errors).

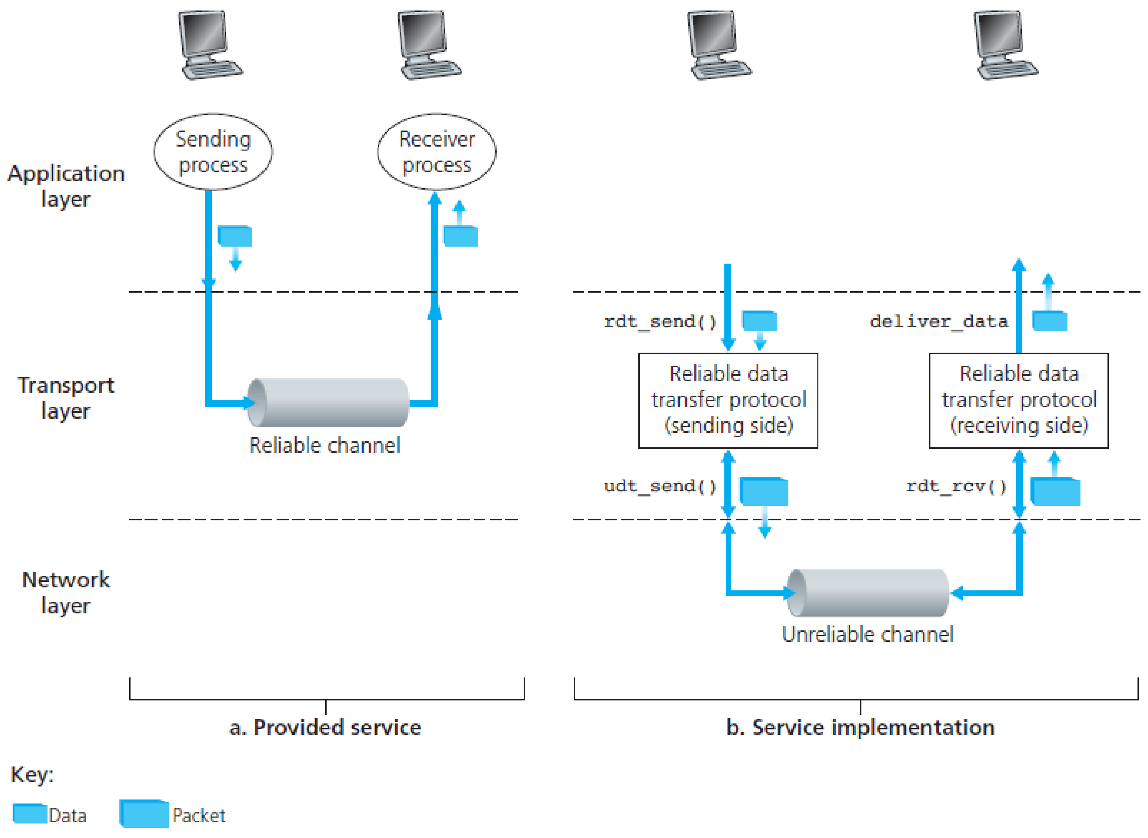
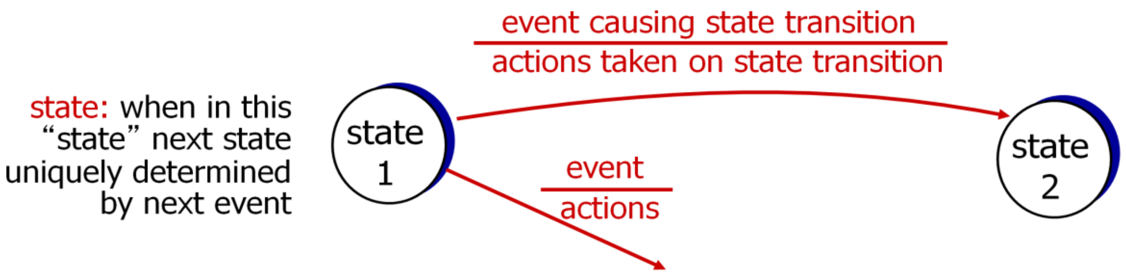


**Figure 3.8** ♦ Reliable data transfer: Service model and service implementation

- 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. (Here rdt stands for *reliable data transfer)*
- 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().**
- **udt_send():** called by rdt, to transfer packet over unreliable channel to receiver

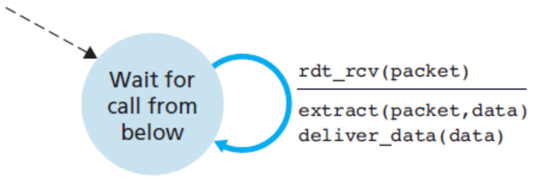
### 3.4.1 Building a Reliable Data Transfer Protocol

We will use finite state machines (FSM) to specify event and actions of sender and 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 labelling the transition, and the actions taken when the event occurs are shown below the horizontal line. When no action is taken on an event, or no event occurs and an action is taken, we'll use the symbol below or above the horizontal

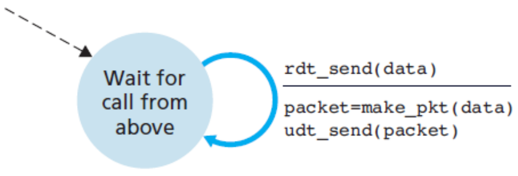

event causing state transition
actions taken on state transition
state: when in this "state" next state uniquely determined by next event
state 1
event
actions
state 2

### rdt 1.0 : Reliable Data Transfer over a Perfectly Reliable Channel

we will first consider the simplest case, in which the underlying channel is completely reliable.



b. rdt1.0: receiving side



a. rdt1.0: sending side

The sending side of rdt simply accepts data from the upper layer via the **rdt_send(data)** event, creates a packet containing the data via the action **make_pkt(data**) and sends the packet into the channel.

On the receiving side, rdt receives a packet from the underlying channel via the **rdt_rcv(packet)** event, removes the data from the packet via the action **extract (packet, data)** and passes the data up to the upper layer via the action **deliver_data(data)**.

Here all packet flow is from the sender to receiver; with a perfectly reliable channel there is no need for the receiver side to provide any feedback to the sender since nothing can go wrong!

## rdt2.0 : Transfer over a Channel with Bit Errors

A more realistic model of the underlying channel is one in which bits in a packet may be corrupted. Such bit errors typically occur in the physical components of a network as a packet is transmitted, propagates, or is buffered.

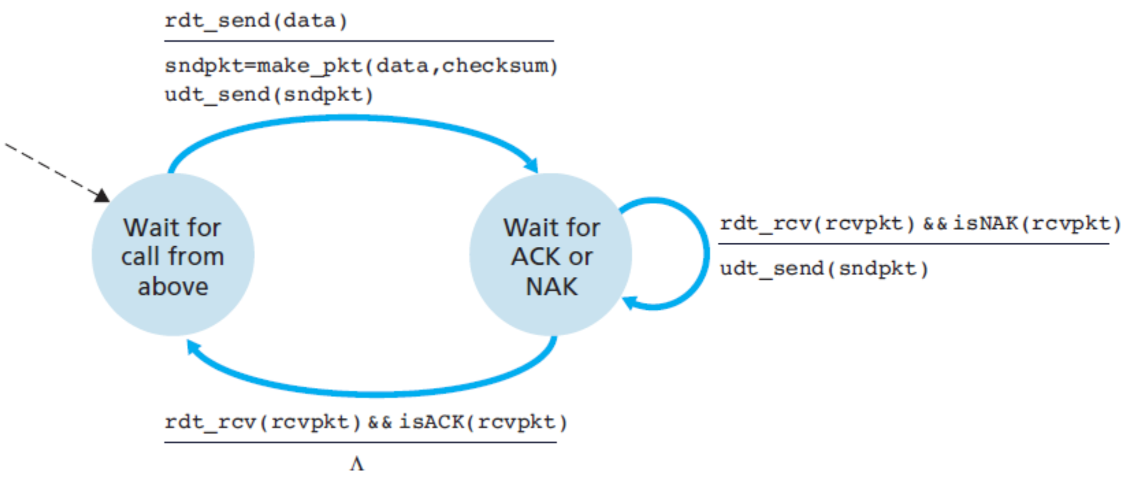
To recover from bit errors new mechanism is required which will inform sender regarding its message is correctly received or not : ARQ ( Automatic Repeat Request) protocol.

It includes two types of messages :

- **Acknowledgment** : for correctly received packet
- **Negative acknowledgement** : for corrupted packet

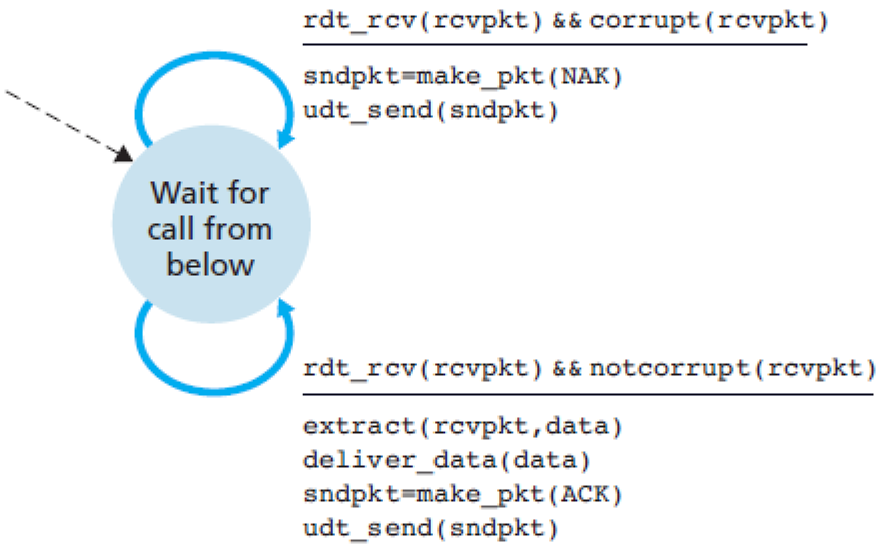
Fundamentally, three additional protocol capabilities are required in ARQ protocols to handle the presence of bit errors:

- ***Error detection :***First, a mechanism is needed to allow the receiver to detect when bit errors have occurred. The Internet checksum field is for exactly this purpose.
- ***Receiver feedback : T***he only way for the sender to learn whether or not a packet was received correctly is to provide explicit feedback to the sender. The positive (ACK) and negative (NAK) acknowledgment replies in the message-dictation scenario are examples of such feedback. Our rdt2.0 protocol. Feedback is 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.


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

a. rdt2.0: sending side

- When the **rdt_send(data)** event occurs, the sender will create a packet containing the data to be sent, along with a packet checksum and then send the packet via the **udt_send(sndpkt)** operation.
- In the rightmost state, the sender protocol is waiting for an ACK or a NAK packet from the receiver. If an ACK packet is received **(the notation rdt_rcv(rcvpkt) && isACK (rcvpkt)**), then packet received correctly by receiver 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.
- When the sender is in the wait-for-ACK-or-NAK state, it *cannot* get more data from the upper layer; that is, the rdt_send() event can not occur; that will happen only after the sender receives an ACK and leaves this state.
- Because of this behavior, protocols such as rdt2.0 are known as **stop-and-wait** protocols.



b. rdt2.0: receiving side

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

## Problem in rdt2.0:

it has a fatal error. No possibility of corruption of the ACK or NAK has been accounted. Minimally, we will need to add checksum bits to ACK/NAK packets in order to detect such errors.

Consider three possibilities for handling corrupted ACKs or NAKs:

1. Add third type of message indicating ACK/NAK is corrupted.
2. A second alternative is to add enough checksum bits to allow the sender not only to detect, but also to recover from, bit errors.
3. 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.

## Solution to duplicate packets:

- A simple solution to this new problem is to add a new field to the data packet and have the sender number its data packets by putting a **sequence number** into this field.
- The receiver then need only check this sequence number to determine whether or not the received packet is a retransmission.
- For this simple case of a stop-and wait protocol, a 1-bit sequence number will be sufficient.

## Updated rdt2.1 with sequence number added to packet:

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.

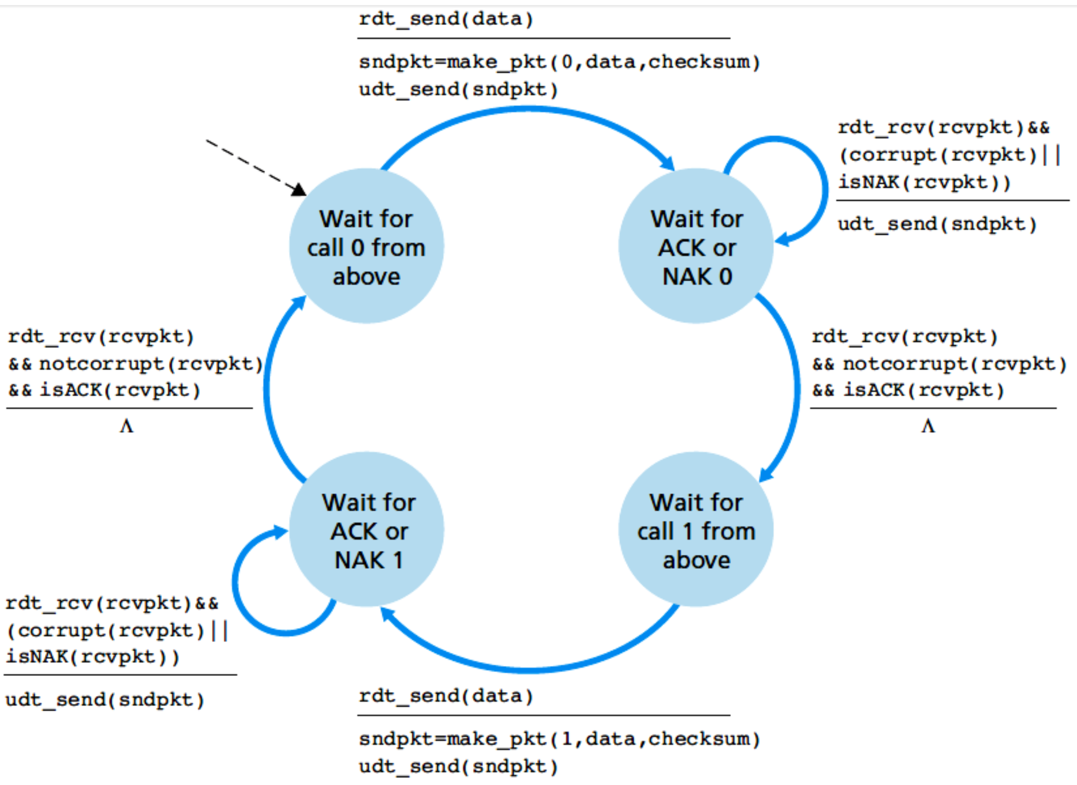


**Figure 3.11** ♦ rdt2.1 sender

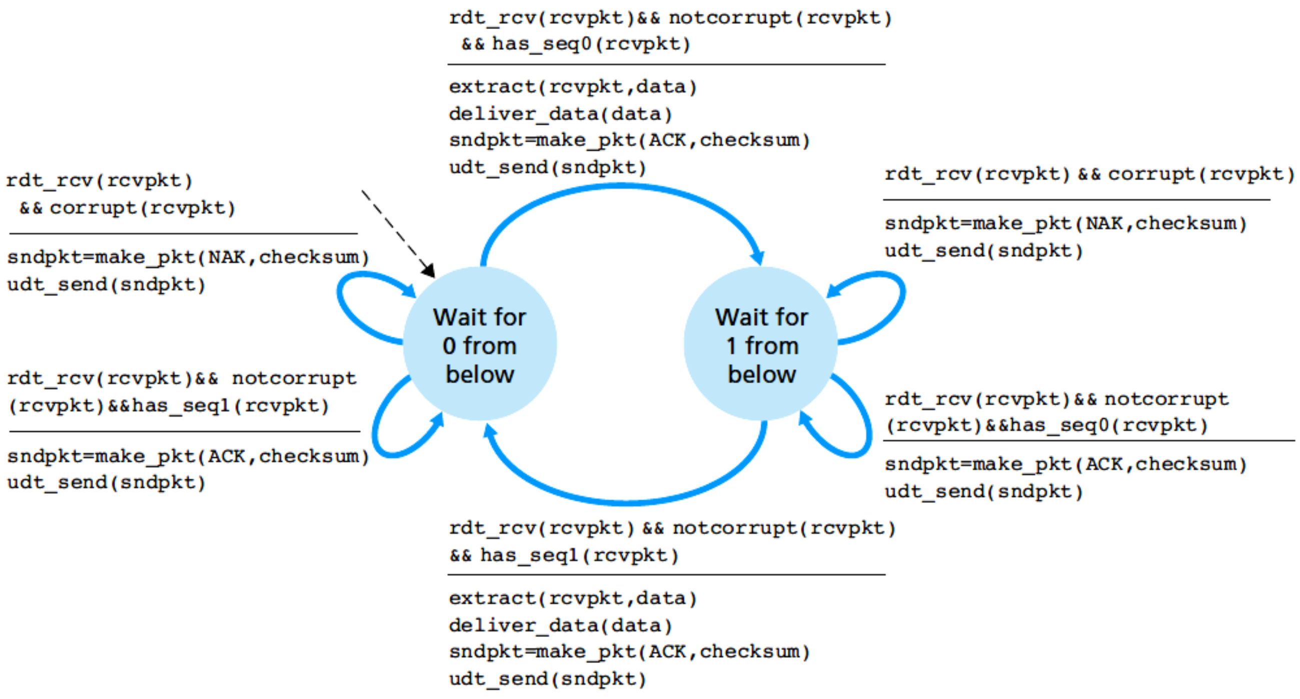

rdt_rcv(rcvpkt)&& notcorrupt(rcvpkt)
&& has_seq0(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
sndpkt=make_pkt(ACK,checksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt)
&& corrupt(rcvpkt)
sndpkt=make_pkt(NAK,checksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && corrupt(rcvpkt)
sndpkt=make_pkt(NAK,checksum)
udt_send(sndpkt)
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 3.12** ♦ rdt2.1 receiver

## Negative acknowledgement free protocol : rdt2.2

The same effect as a NAK can be achieved if, instead of sending a NAK, an ACK for the last correctly received packet is sent.

- The 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 : include the ACK,0 or ACK,1 argument in make_pkt() in the receiver FSM
- The sender must now check the sequence number of the packet being acknowledged by a received ACK message : include the 0 or 1 argument in isACK() in the sender FSM.
- 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 and that is why it is being ACKed twie.

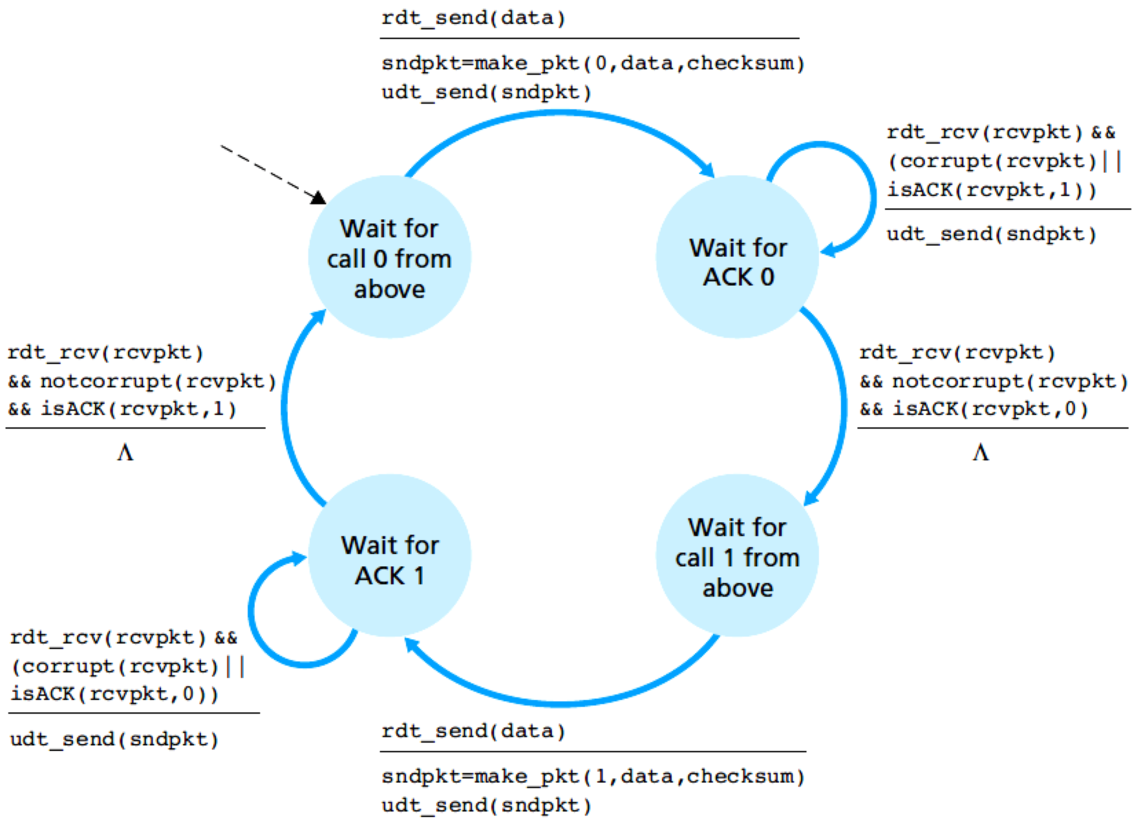

rdt_send(data)
sndpkt=make_pkt(0,data,checksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && (corrupt(rcvpkt)|| isACK(rcvpkt,1))
udt_send(sndpkt)
Wait for call 0 from above
Wait for ACK 0
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,1)
Λ
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,0)
Λ
Wait 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 3.13** ♦ rdt2.2 sender

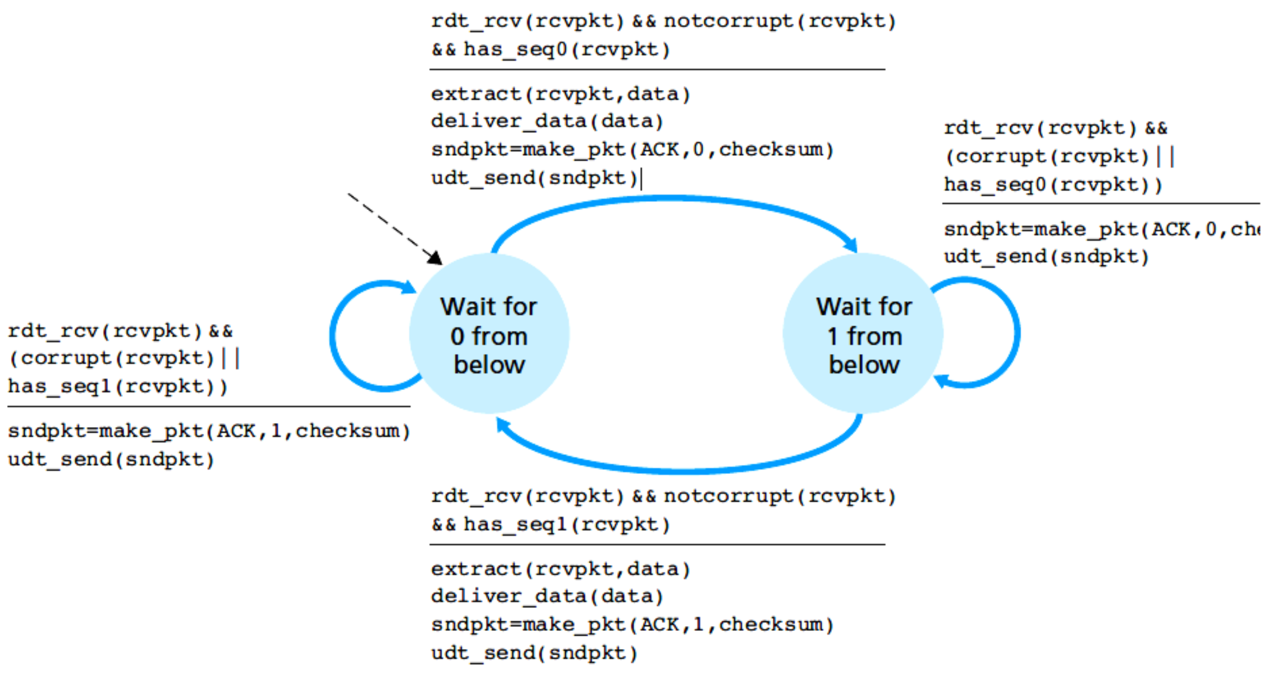

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq0(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
sndpkt=make_pkt(ACK,0,checksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && (corrupt(rcvpkt)|| has_seq0(rcvpkt))
sndpkt=make_pkt(ACK,0,ch
udt_send(sndpkt)
Wait for 0 from below
Wait for 1 from below
rdt_rcv(rcvpkt)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

**Figure 3.14** ♦ rdt2.2 receiver

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

Suppose now that in addition to corrupting bits, the underlying channel can *lose* packets as well.

Two additional concerns must now be addressed by the protocol:

- How to detect packet loss and what to do when packet loss occurs.
- The use of checksumming, sequence numbers, ACK packets, and retransmission techniques.

The sender transmits a data packet and either that packet, or the receiver's ACK gets lost, no reply is forthcoming at the sender from the receiver. After waiting for certain amount of time the sender can simply retransmit the data packet.

**How long to wait for ACK?**

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

Note that if a packet experiences a particularly large delay, the sender may retransmit the packet even though neither the data packet nor its ACK have been lost. This introduces the possibility of **duplicate data packets** in the sender-to-receiver channel. Happily, protocol rdt2.2 already has enough functionality (that is, sequence numbers) to handle the case of duplicate packets.

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 is sent, (2) respond to a timer interrupt and (3) stop the timer.

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

**Note : Receiver rdt 3.0 is same as receiver rdt 2.2**

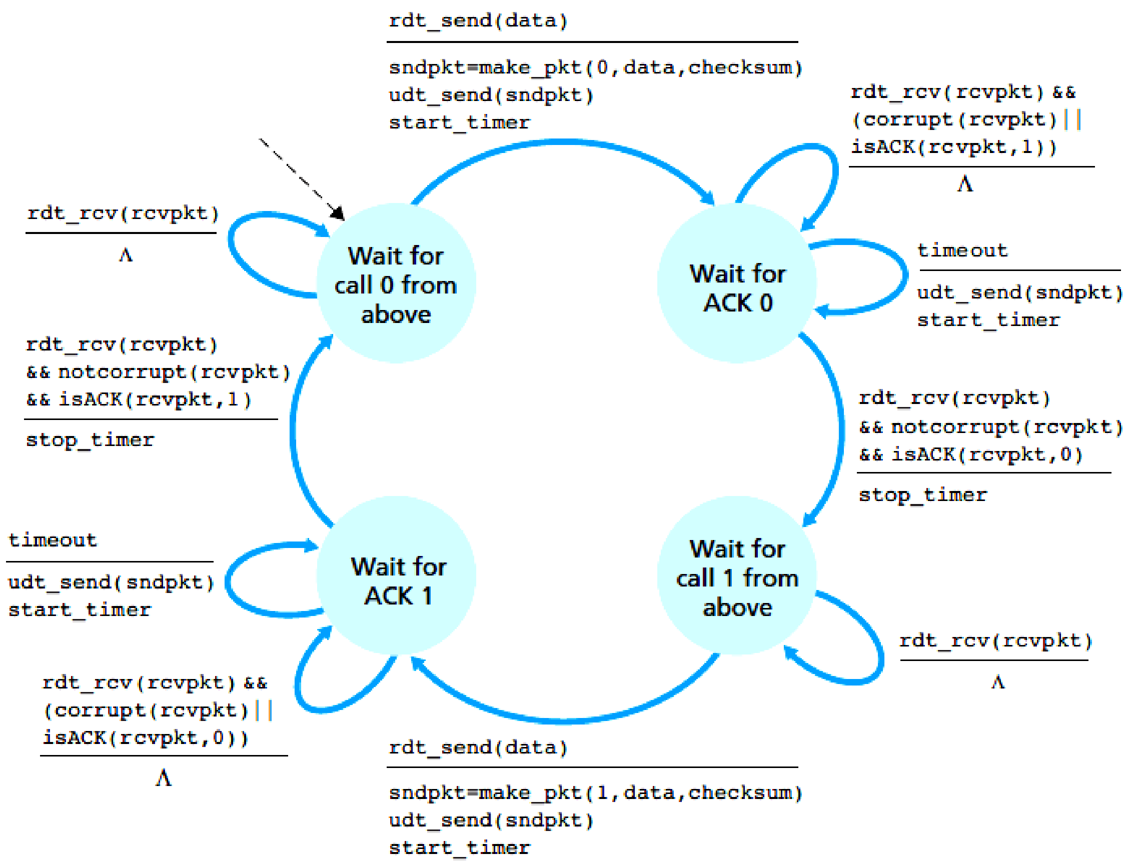
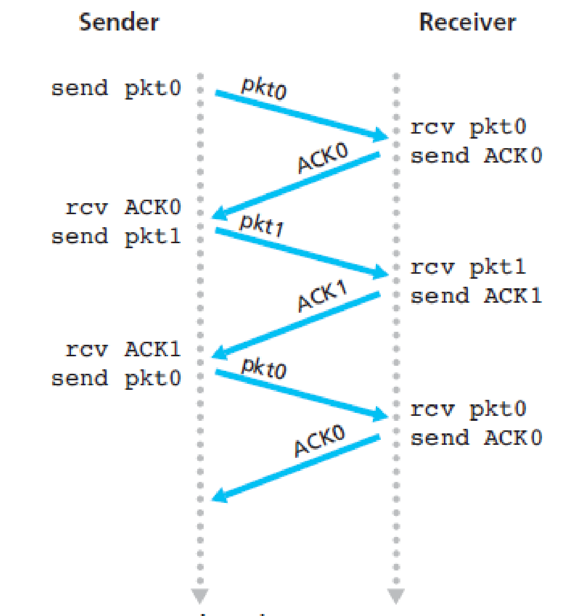
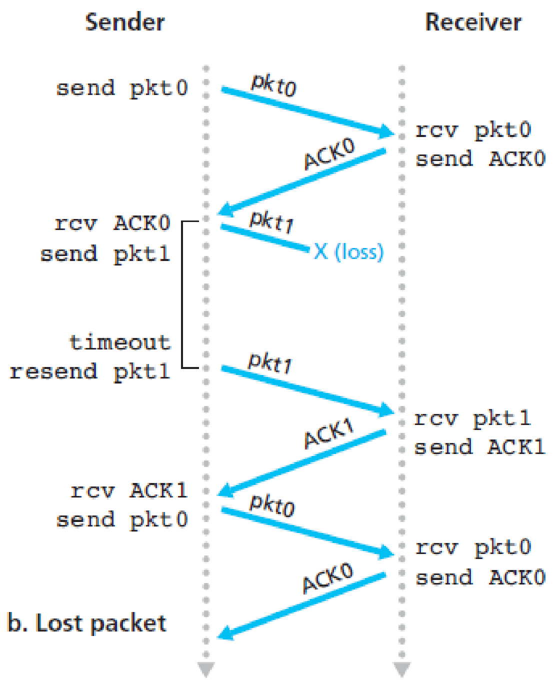

rdt_send(data)
sndpkt=make_pkt(0,data,checksum)
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt) && (corrupt(rcvpkt)|| isACK(rcvpkt,1))
Λ
rdt_rcv(rcvpkt)
Λ
Wait for call 0 from above
Wait for ACK 0
timeout
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,1)
stop_timer
rdt_rcv(rcvpkt)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

Figure 3.15 ♦ rdt3.0 sender

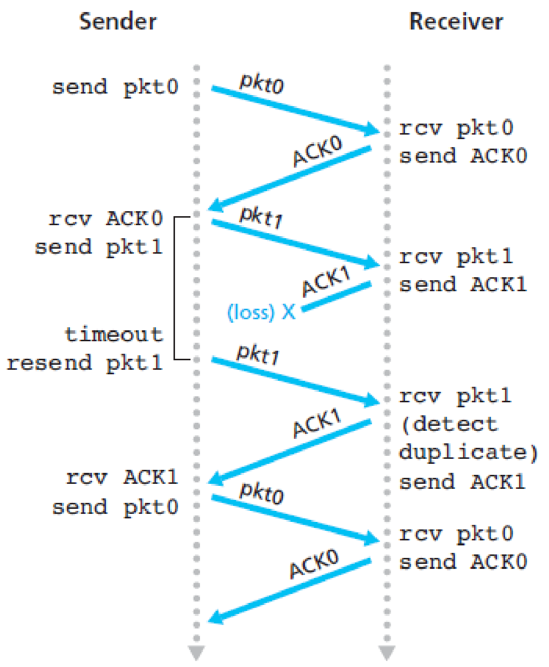
**Operations of rdt3.0, the alternating-bit protocol :**



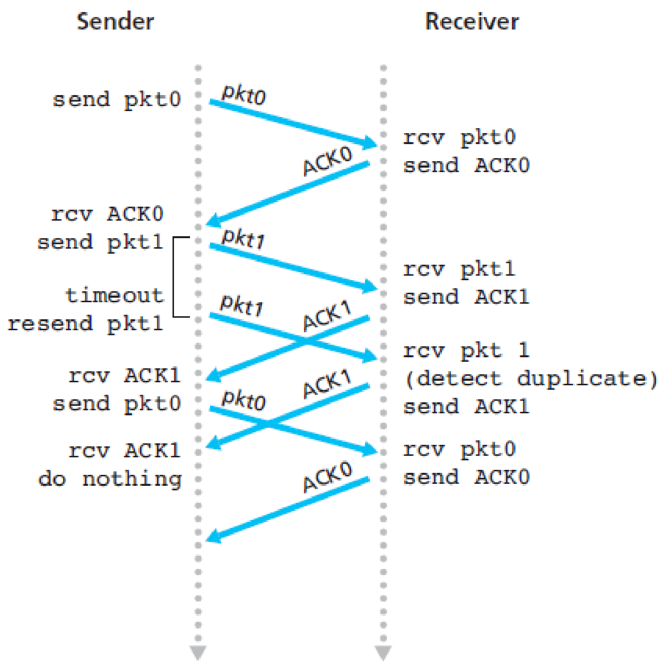
a. Operation with no loss



b. Lost packet



c. Lost ACK



d. Premature timeout

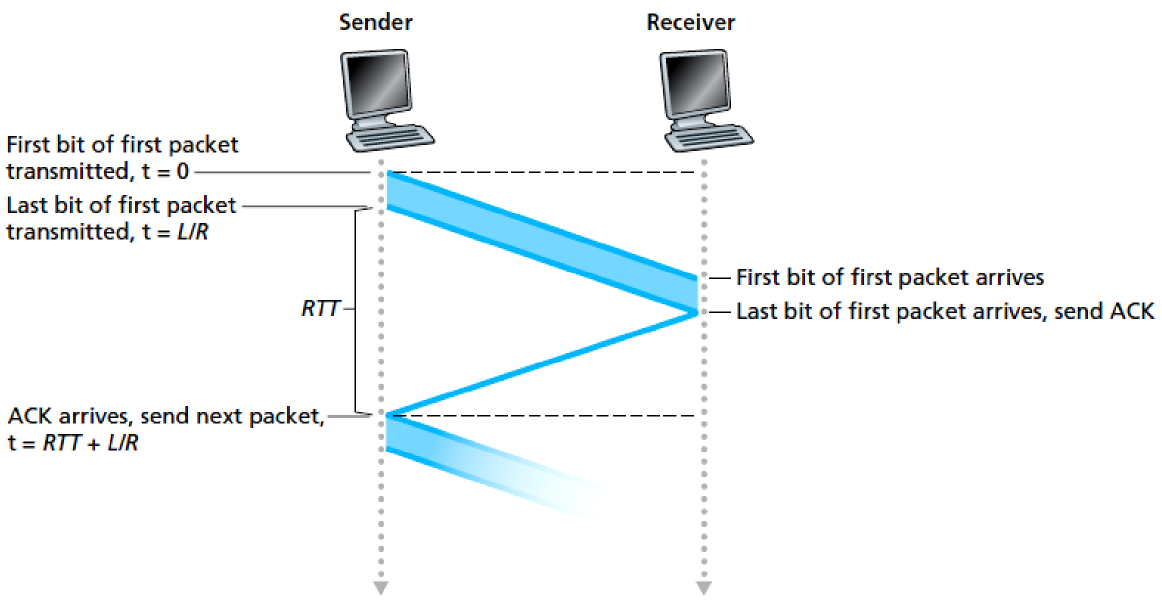
## 3.4.2 **Pipelined Reliable Data Transfer Protocols**

In stop and wait protocol sender cannot send next packet until it has successfully received acknowledgment of the previous packet.

- Consider an idealized case of two hosts, one located on the West Coast of the United States and the other located on the East Coast.
- RTT, is approximately 30 milliseconds. Suppose that they are connected by a channel with a transmission rate, *R*, of 1 Gbps . With a packet size *L* of 1,000 bytes.
- Including both header fields and data, the time needed to actually transmit the packet into the 1 Gbps link is:

$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$

- Below figure shows that with our stop-and-wait protocol, if the sender begins sending the packet at $t = 0$, then at $t = L/R = 8$ microseconds, the last bit enters the channel at the sender side.
- The packet then makes its 15-msec cross-country journey, with the last bit of the packet emerging at the receiver at $t = \text{RTT}/2 + L/R = 15.008$ msec
- The ACK emerges back at the sender at $t = \text{RTT} + L/R = 30.008$ msec. At this point, the sender can now transmit the next message (30.008 msec).


Sender
Receiver
First bit of first packet transmitted, t = 0
Last bit of first packet transmitted, t = L/R
RTT
First bit of first packet arrives
Last bit of first packet arrives, send ACK
ACK arrives, send next packet, t = RTT + L/R

a. Stop-and-wait operation

If we define the **utilization** of the sender (or the channel) as the fraction of time the sender is actually busy sending bits into the channel

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

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

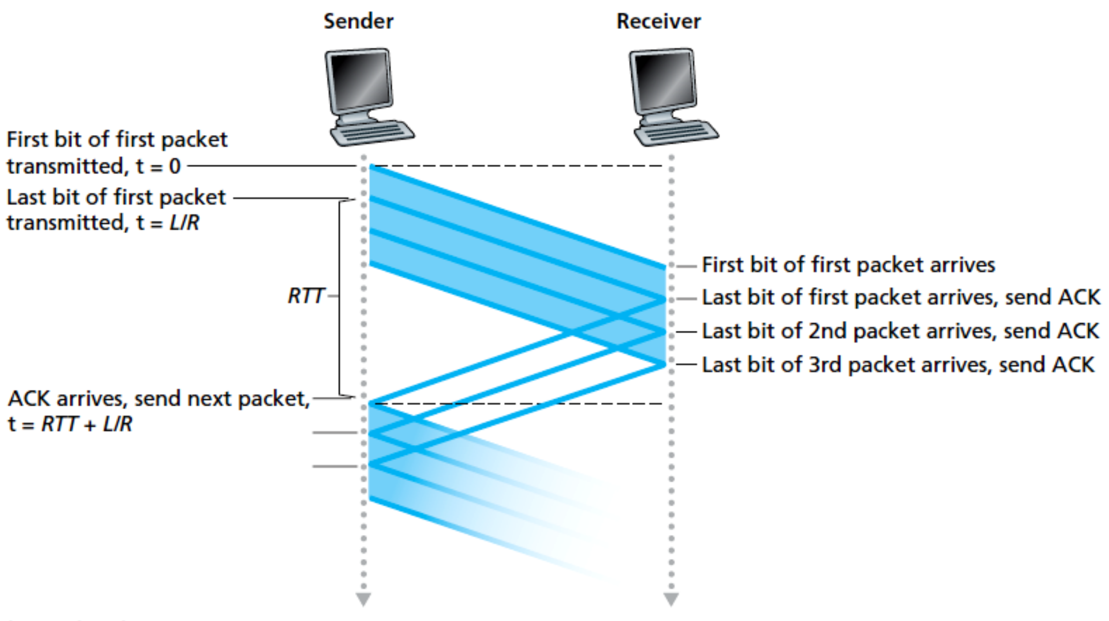
The solution to this particular performance problem is simple: Rather than operate in a stop-and-wait manner, the sender is allowed to send multiple packets without waiting for acknowledgments. This technique is known as **pipelining**.

Pipelining has the following consequences for reliable data transfer protocols:

- The range of sequence numbers must be increased to allocate each packet a unique sequence number.
- 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.

Consider using same access link sender can send 3 packets at a time without waiting for ACK. Utilization of link will be increased by 3 times :

$$U_{sender} = \frac{3L/R}{RTT + L/R} = \frac{.0024}{30.008} = 0.00081$$


Sender
Receiver
First bit of first packet transmitted, t = 0
Last bit of first packet transmitted, t = L/R
RTT
ACK arrives, send next packet, t = RTT + 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

b. Pipelined operation

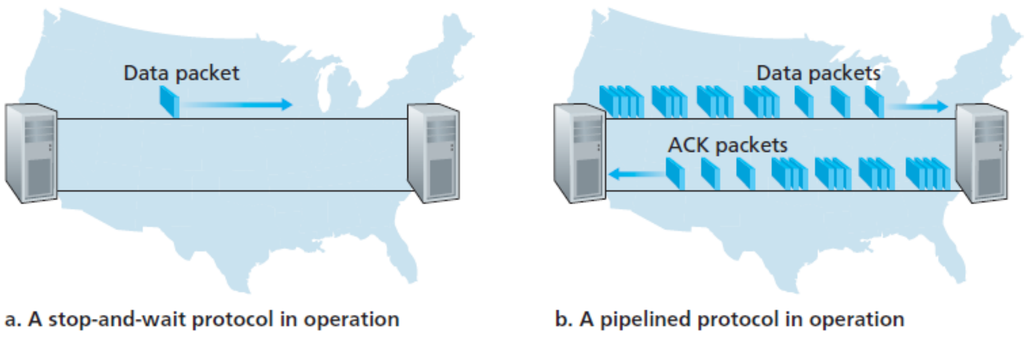

Data packet
a. A stop-and-wait protocol in operation
Data packets
ACK packets
b. A pipelined protocol in operation

**Figure 3.17** ♦ Stop-and-wait versus pipelined protocol

## Go-Back-N (GBN) :

In a **Go-Back-N (GBN) protocol**, the sender is allowed to transmit multiple packets upto N without waiting for an acknowledgment. Figure 3.19 shows the sender's view of the range of sequence numbers in a GBN protocol.

- **base** : points to the sequence number of the oldest unacknowledged packet.
- **nextseqnum** : points to be the smallest unused sequence number

Four intervals in the range of sequence numbers can be identified:

- **Sequence numbers in the interval [0,base-1]** : packets already transmitted and acknowledged.
- **The interval [base,nextseqnum-1]** : packets sent but not yet acknowledged.
- **The interval [nextseqnum,base+N-1]** : packets that can be sent immediately when data is available from application layer.
- **sequence numbers >= base+N** : cannot be used.

As the protocol operates, this window slides forward over the sequence number space. For this reason, *N* is often referred to as the **window size** and the GBN protocol itself as a **sliding-window protocol**.

**Sequence No. range :** If *k* is the number of bits in the packet sequence number field, the range of sequence numbers is thus $[0,2^k - 1]$.

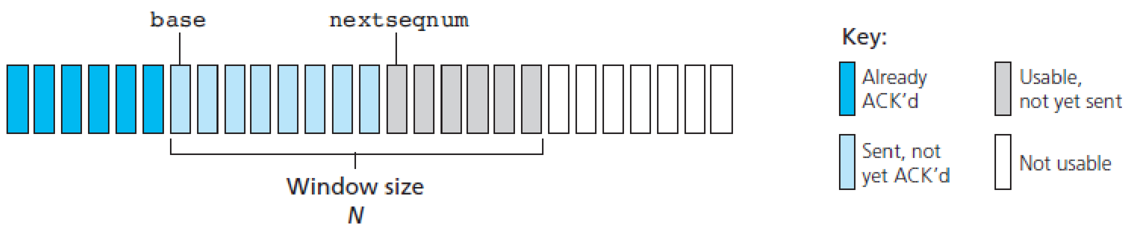


**Figure 3.19** ♦ Sender's view of sequence numbers in Go-Back-N

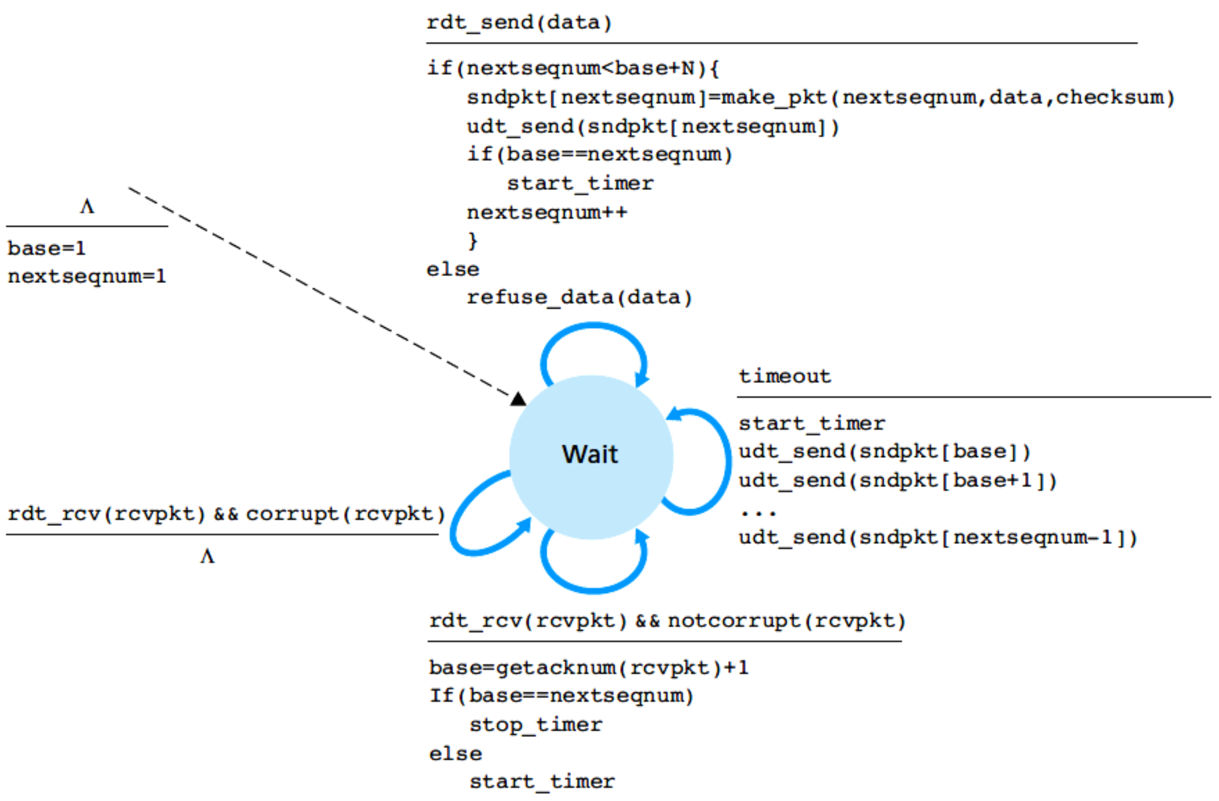
**Events/Actions at the sender side GBN :**

1. ***Invocation from above.:*** When rdt_send() is called from above, the sender first checks to see if the window is full:
    - If the window is not full, a packet is created and sent, and variables are appropriately updated.
    - If the window is full, the sender simply returns the data back to the upper layer,
2. ***Receipt of an ACK:*** In our GBN protocol, an acknowledgment for a packet with sequence number *n* will be taken to be a **cumulative acknowledgment**, indicating that all packets with a sequence number up to and including *n* have been correctly received at the receiver.
3. ***A timeout event.:*** If a timeout occurs, the sender resends *all* packets that have been previously sent but that have not yet been acknowledged.
   GBN sender uses only a single timer set 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.

**Events/Actions at receiver:**

If a packet with sequence number *n* is received correctly and is in order then 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.
- Of course, 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.



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

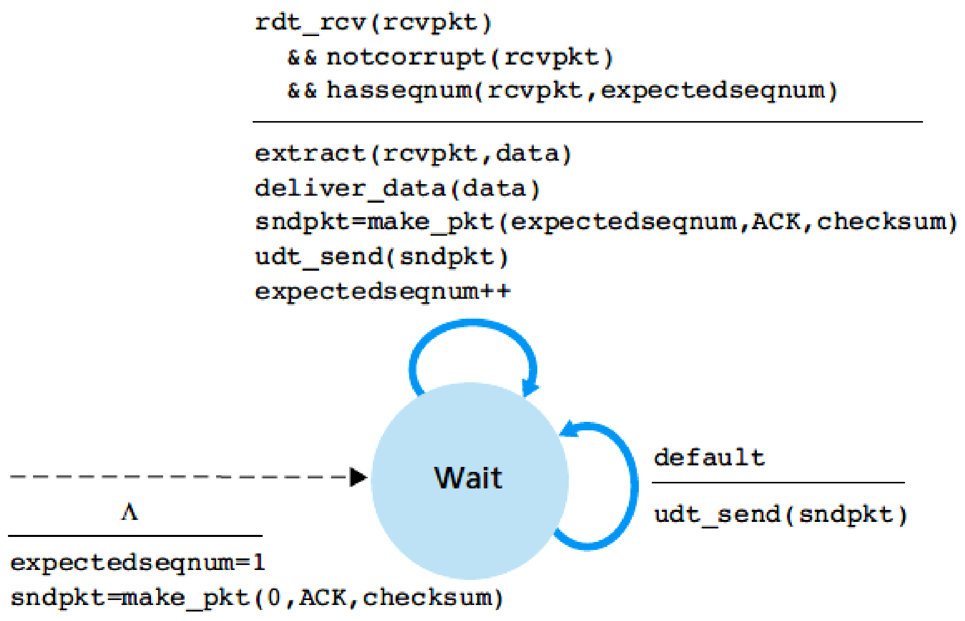
```
Λ
________________
base=1
nextseqnum=1
```

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

```
rdt_rcv(rcvpkt) && corrupt(rcvpkt)
________________________________
Λ
```

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

**Figure 3.20** ♦ Extended FSM description of GBN sender



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

```
default
________________
udt_send(sndpkt)
```

```
Λ
________________
expectedseqnum=1
sndpkt=make_pkt(0,ACK,checksum)
```

**Figure 3.21** ♦ Extended FSM description of GBN receiver

**Operational flow of GBN:**

Figure 3.22 shows the operation of the GBN protocol for the case of a window size of four packets (seq. No. 0 to 3)

As each successive ACK (for example, ACK0 and ACK1) is received, the window slides forward and the sender can transmit one new packet (pkt4 and pkt5, respectively).

On the receiver side, packet 2 is lost and thus packets 3, 4, and 5 are found to be out of order and are discarded.

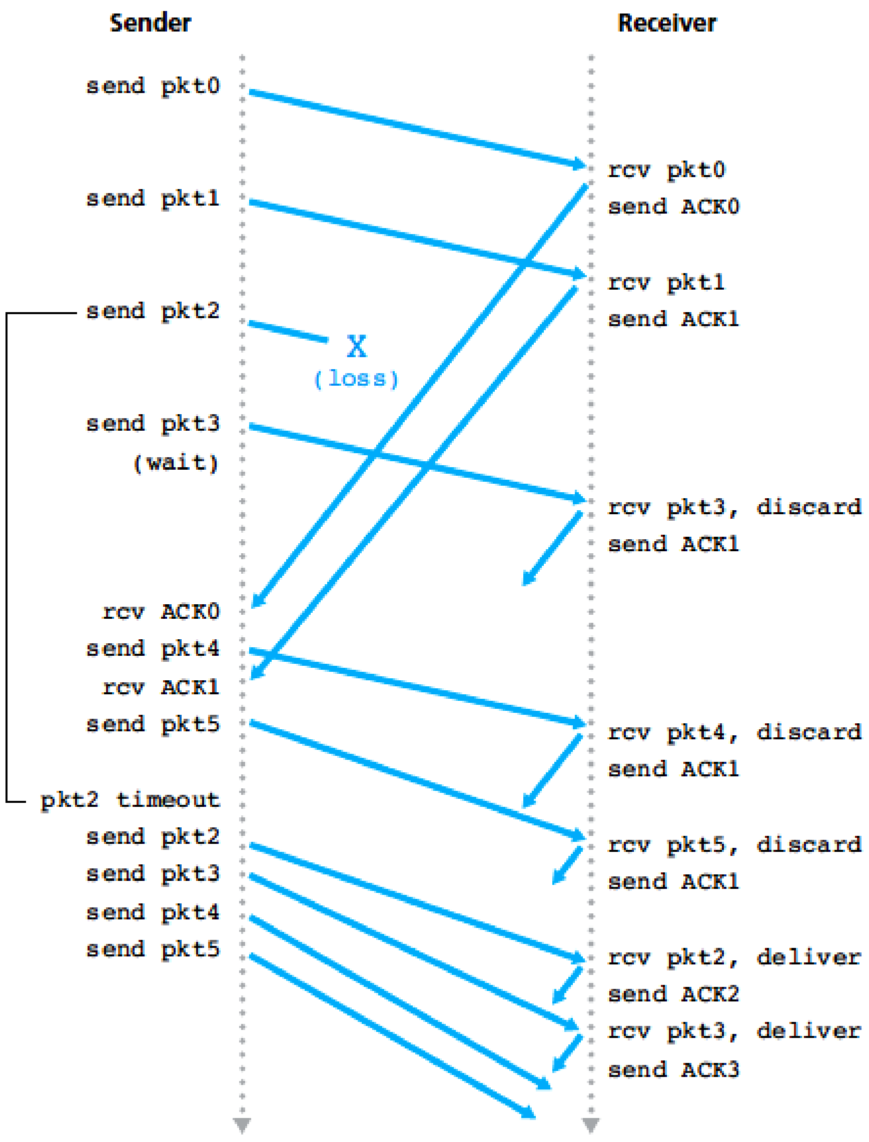


**Figure 3.22** ♦ Go-Back-N in operation

## Selective Repeat (SR):

Unlike GBN, selective-repeat protocols avoid unnecessary retransmissions by having the sender retransmit only those packets those were lost or corrupted at the receiver.

- The receiver *individually* acknowledges correctly received packets.
- A window size of *N* will again be used to limit the number of outstanding, unacknowledged packets in the pipeline.
- The SR receiver will acknowledge a correctly received packet whether or not it is in order. Out-of-order packets are buffered until any missing packets are received.

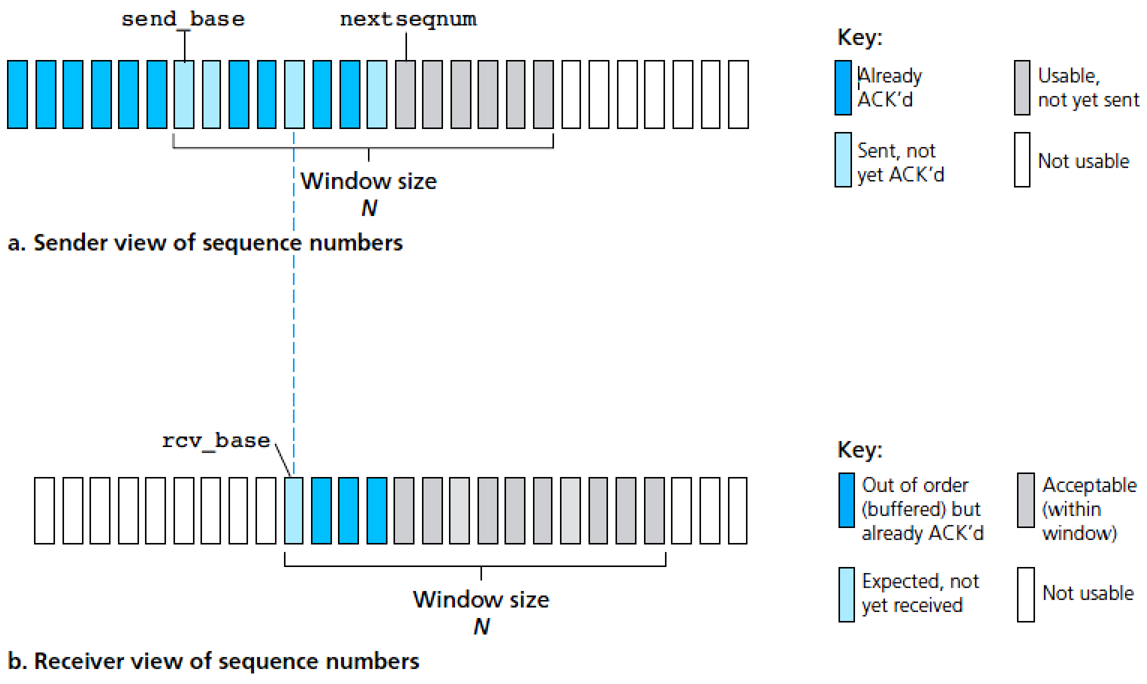

send_base
nextseqnum
Window size
N
a. Sender view of sequence numbers
Key:
Already ACK'd
Usable, not yet sent
Sent, not yet ACK'd
Not usable
rcv_base
Window size
N
b. Receiver view of sequence numbers
Key:
Out of order (buffered) but already ACK'd
Acceptable (within window)
Expected, not yet received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

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

## SR sender event and actions:

- **Data received from above :**the SR sender checks the next available sequence number for the packet. If the sequence number is within the sender's window, the data is packetized and sent; otherwise it is either buffered or returned to the upper layer.
- ***Timeout***.: Each packet must have its own logical timer. When timeout event occurs resend packet *n* and restart the timer.

- ***ACK received.*** If an ACK is received, the SR sender marks that packet as having been received. If the packet's sequence number is equal to send_base, the window will slide and can receive new packet from upper layer.

## SR receiver event and actions:

- ***Packet with sequence number in* [rcv_base, rcv_base+N-1] *is correctly received :*** accept the packet and send selective ACK to sender. If the packet has a sequence number equal to the base of the receive window, 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 number of packets delivered to the upper layer
- ***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.
- ***Otherwise.*** Ignore the packet.

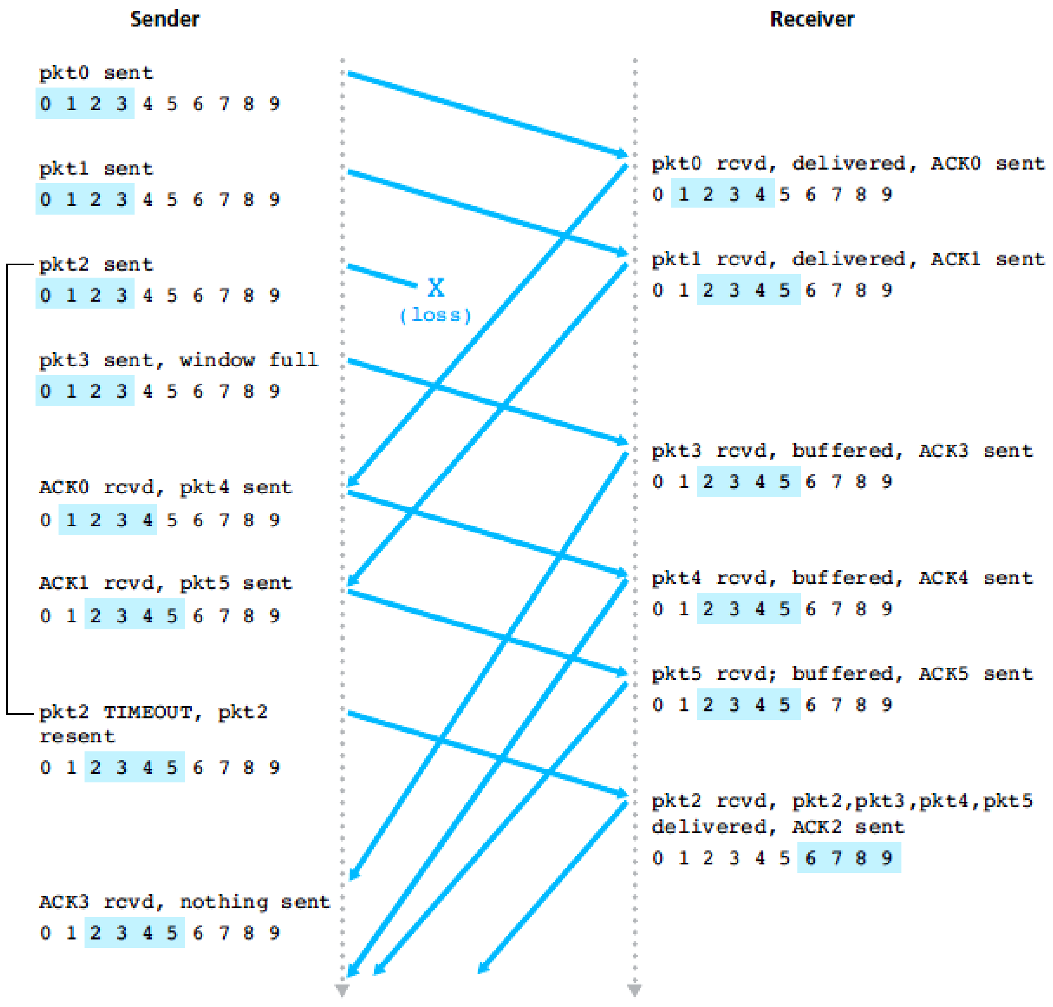


**Figure 3.26** ♦ SR operation

## Selective repeat problem:

Consider sequence number range is 0, 1, 2, 3, and a window size N=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.

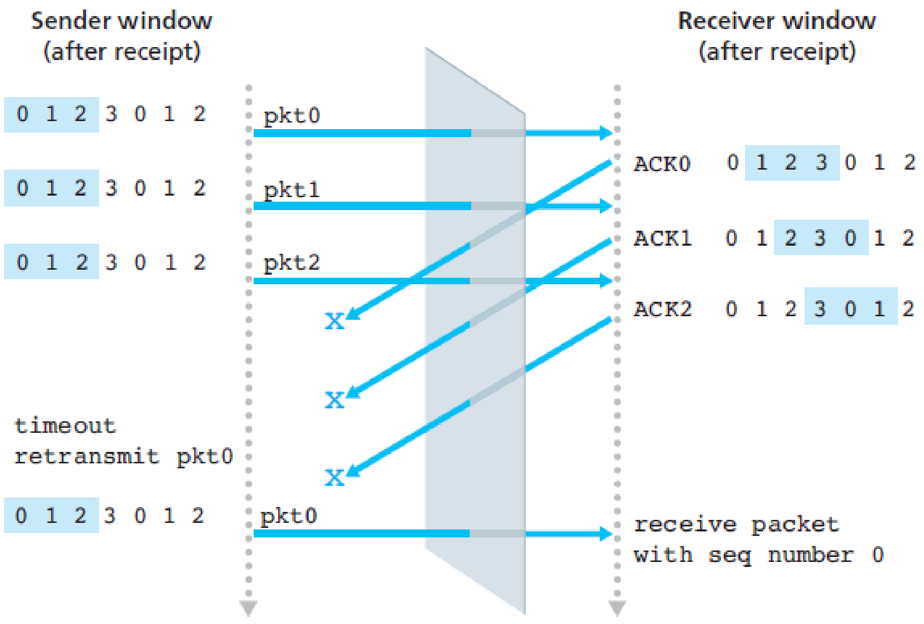
Consider the ACKs for the first three packets are lost and the sender retransmits these packets. The receiver thus next receives a packet with sequence number 0—a copy of the first packet sent but receiver was waiting for new packet with seq no. 0.

Here **due to small window size**, receiver considers retransmitted old packet as a new packet and accepts it – Wrong delivery of data !!

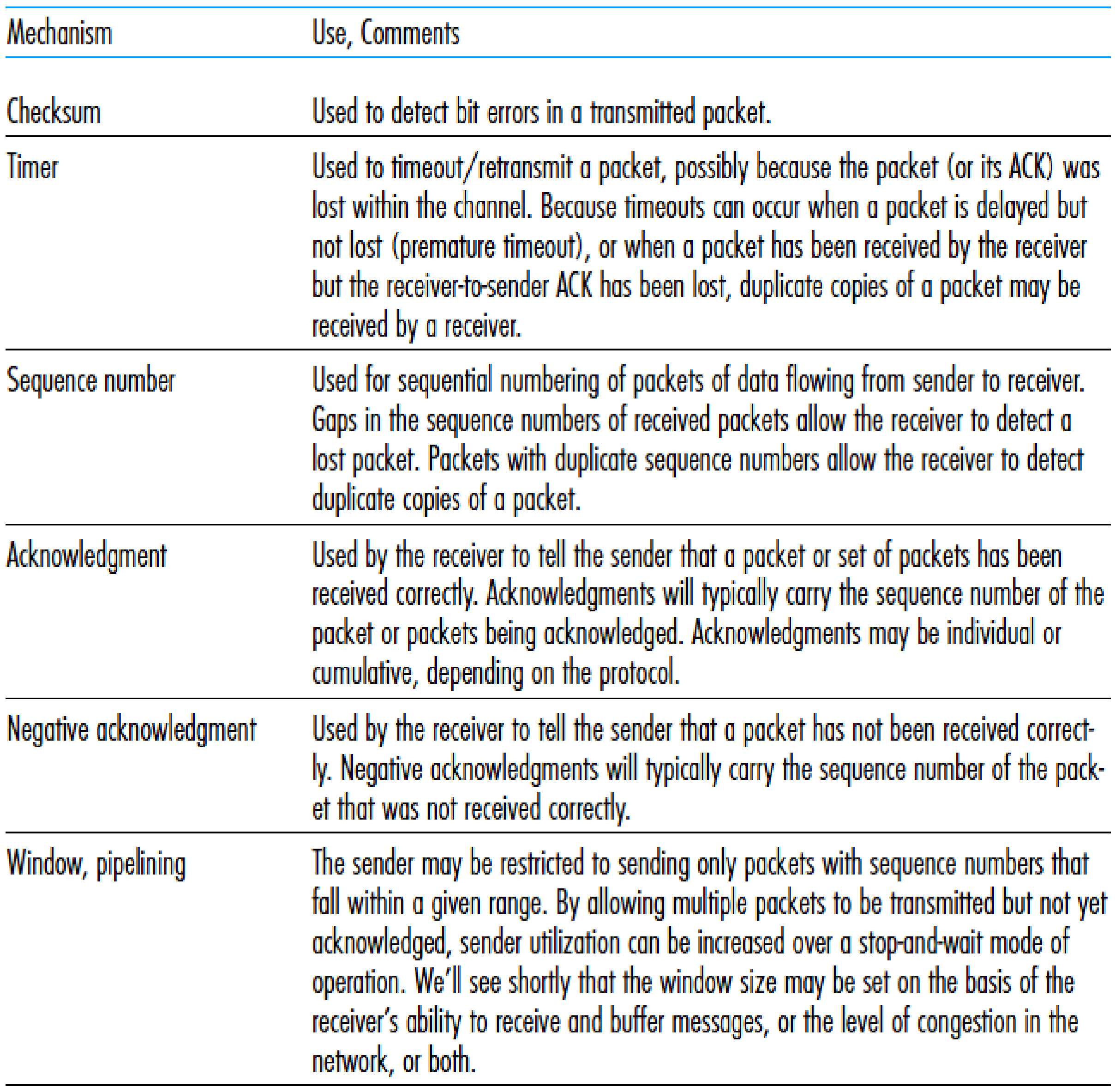
**What should be the window size then to avoid such problem?**

**Window size must be less than or equal to half the size of the sequence number space for SR protocols.**

**Window size <= seq# space / 2**



## Summary of reliable data transfer and their use.



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

## 3.4 Connection-Oriented Transport: TCP

### TCP is:

- TCP—the Internet's transport-layer, connection-oriented, reliable transport protocol.
- TCP is said to be **connection-oriented** because before one application process can begin to send data to another, the two processes must first "handshake" with each other—that is, they must send some preliminary segments to each other to establish the parameters of the ensuing data transfer.
- A TCP connection provides a **full-duplex service**: If there is a TCP connection between Process A on one host and Process B on another host, then application layer data can flow from Process A to Process B at the same time as application layer data flows from Process B to Process A.
- A TCP connection is also always **point-to-point**, that is, between a single sender and a single receiver.
- The maximum amount of data that can be grabbed and placed in a segment is limited by the **maximum segment size (MSS).**
- TCP pairs each chunk of client data with a TCP header, thereby forming **TCP segments**. The segments are passed down to the network layer, where they are separately encapsulated within network-layer IP datagrams.

### 3.5.2 TCP Segment Structure:

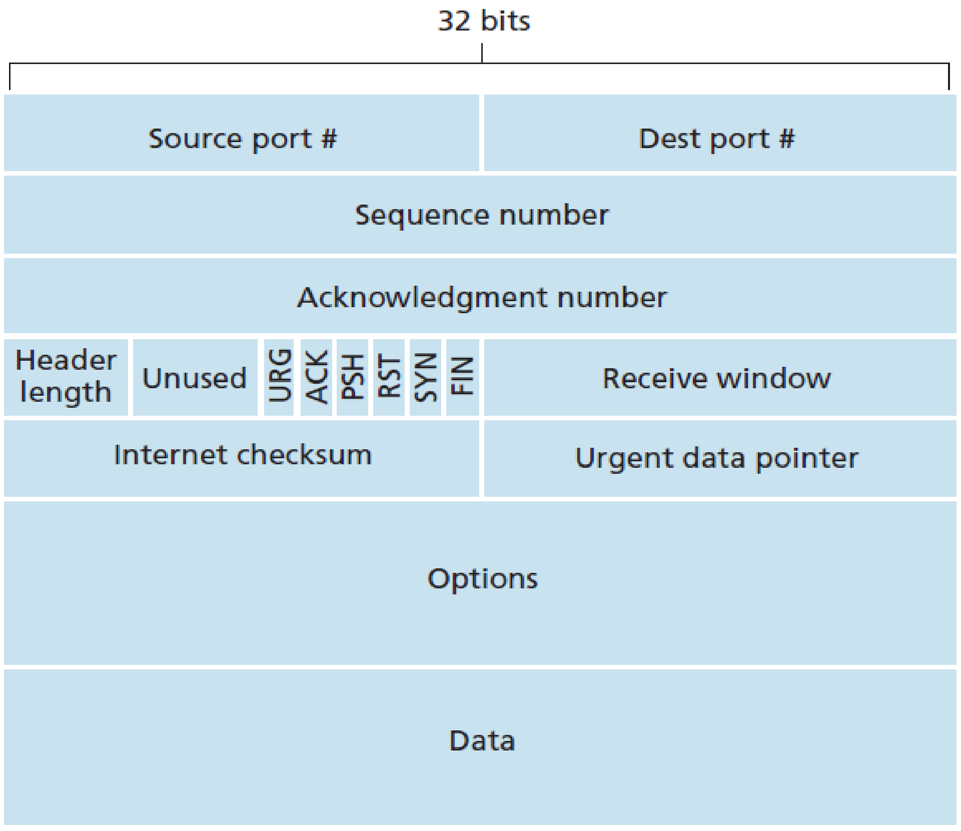
The segment consists of a header of 20 to 60 bytes, followed by data from the application program. The header is 20 bytes if there are no options and up to 60 bytes if it contains options.

A TCP segment header contains the following fields:

- **Source port address.** This is a 16-bit field that defines the port number of the application program in the host that is sending the segment.
- **Destination port address.** This is a 16-bit field that defines the port number of the application program in the host that is receiving the segment.
- **Sequence number.** This 32-bit field defines the number assigned to the first byte of data contained in this segment. As we said before, TCP is a stream transport protocol.To ensure connectivity, each byte to be transmitted is numbered. The

sequence number tells the destination which byte in this sequence is the first byte in the segment.

- **Acknowledgment number.** This 32-bit field defines the byte number that the receiver of the segment is expecting to receive from the other party. If the receiver of the segment has successfully received byte number *x* from the other party, it returns *x+1* as the acknowledgment number. Acknowledgment and data can be piggybacked together.
- **Header length.** This 4-bit field indicates the number of 4-byte words in the TCP header. The length of the header can be between 20 and 60 bytes. Therefore, the value of this field is always between 5 (5 *4 =20) and 15 (15 *4 =60).
- **Reserved.** This is a 6-bit field reserved for future use.



- **Control Flags.** This field defines 6 different control bits or flags as shown in Figure. One or more of these bits can be set at a time. These bits enable flow control, connection establishment and termination, connection abortion, and the mode of data transfer in TCP.

  → The **ACK bit** is used to indicate that the value carried in the acknowledgment field is valid; that is, the segment contains an acknowledgment for a segment that has been successfully received.

- → The **RST**, **SYN**, and **FIN** bits are used for connection setup and teardown.
- → Setting the **PSH** bit indicates that the receiver should pass the data to the upper layer immediately. The **URG** bit is used to indicate that there is data in this segment that the sending-side upper-layer
- → entity has marked as "urgent." The location of the last byte of this urgent data is indicated by the 16-bit **urgent data pointer field**

- **Window size.** This field defines the window size of the sending TCP in bytes. Note that the length of this field is 16 bits, which means that the maximum size of the window is 65,535 bytes. This value is normally referred to as the receiving window (*rwnd*) and is determined by the receiver.
- **Checksum.** This 16-bit field contains the checksum. The calculation of the checksum for TCP follows the same procedure as the one described for UDP.
- **Urgent pointer.** This 16-bit field, which is valid only if the urgent flag is set, is used when the segment contains urgent data.
- **Options.** There can be up to 40 bytes of optional information in the TCP header for example **end of operation(EOP), Timestamp** etc.

## 3.5.3 Round-Trip Time Estimation and Timeout.

TCP 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.

**Too long** Timeout will delay the transmissions of other packets and **too short** timeout will increase unnecessary retransmissions.

### How to measure Timeout ??

- Measure the sample RTT, denoted **SampleRTT**, for a segment that is amount of time between when the segment is sent and when an acknowledgment for the segment is received.
- The **SampleRTT** values will fluctuate from segment to segment due to congestion in the routers and to the varying load on the end systems, therefore it is natural to take

some sort of average of the SampleRTT values denoted as **EstimatedRTT** according to the following formula:

**First Time : EstimatedRTT = SampleRTT**

**Next all measurements:**

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

**typical value: α= 0.125**

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

**First Time: DevRTT = SampleRTT/2**

**Next all measurements:**

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

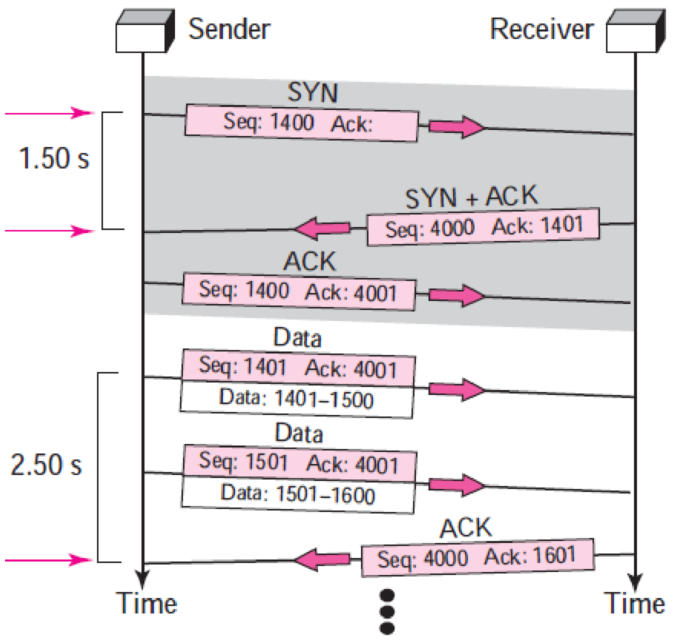
**typically, β = 0.25**

- Finally measurement of Timeout interval based on **EstimatedRTT** and **DevRTT**

**TimeoutInterval = EstimatedRTT + 4*DevRTT**

**Example:**

| **For the first packet:** | **For the next packet :** |
|---|---|
| **SampledRTT = 1.5** | **SRTT = 2.5** |
| **EstimatedRTT = 1.5** | **ERTT = 7/8 ×1.5 + (1/8) × 2.5 = 1.625** |
| **DevRTT = (1.5) / 2 = 0.75** | **DRTT = 3/4 (0.75) + (1/4) × \|1.625 − 2.5\| = 0.78** |
| **TimeOut = 1.5 + 4 × 0.75 = 4.5** | **TimeOut = 1.625 + 4 × 0.78 = 4.74** |



## 3.4.4 TCP Reliable Data Transfer

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

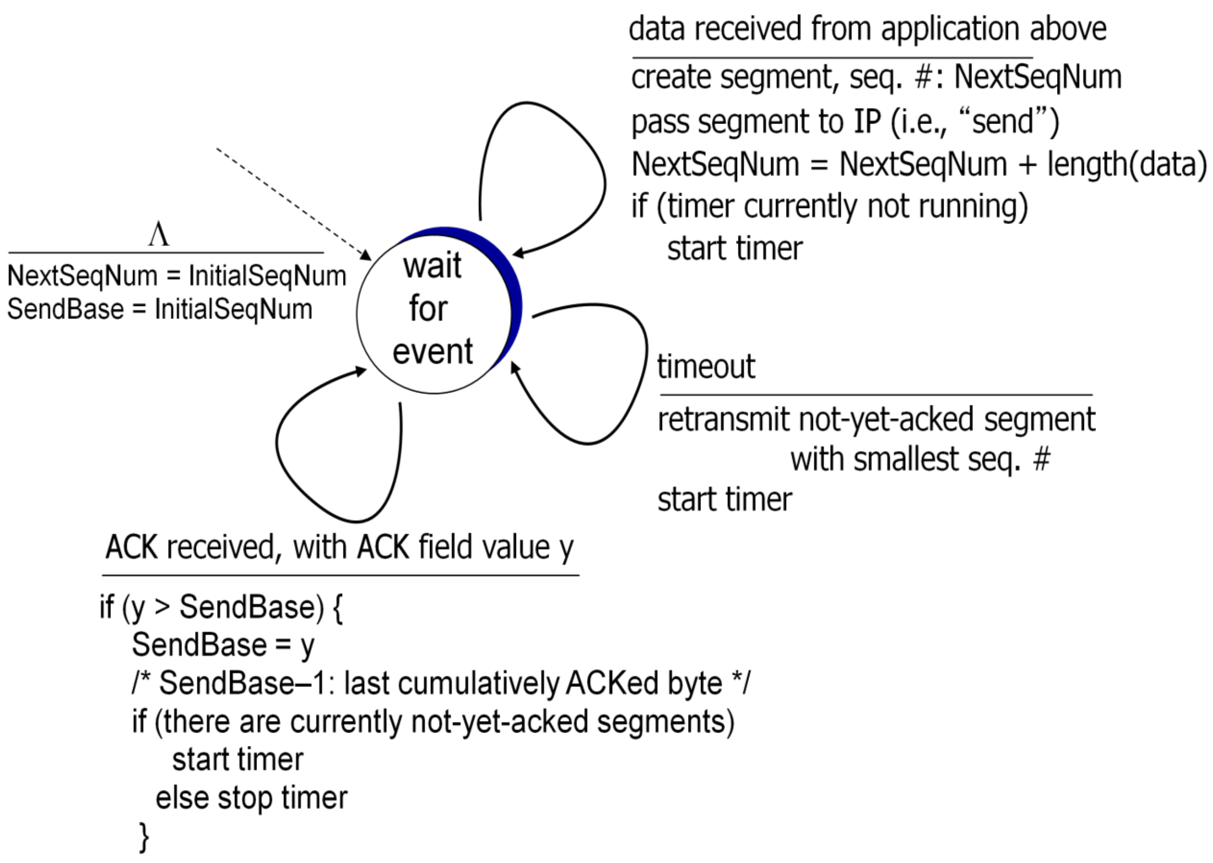
- TCP creates a **reliable data transfer service** on top of IP's unreliable best effort service.
- TCP's reliable data transfer service ensures that the data stream that a process reads out of its TCP receive buffer is uncorrupted, without gaps, without duplication, and in sequence.
- TCP timer management procedures use only a *single* retransmission timer, even if there are multiple transmitted but not yet acknowledged segments.
- It uses pipelined approach and cumulative acknowledgement.

Retransmission can be triggred by two events :

- Timer timeout
- 3 duplicate acknowledgements

### A simplified TCP sender process flow:

1. ***Upon receiving data from application layer***
   a. create segment with sequence number (sequence number  is byte-stream number of first data byte in segment)
   b. start timer if not already running for oldest unacknowledged segment
   *c.* set expiration *interval :* ***TimeOutInterval***
2. ***Upon timeout :***
   *a. retransmit segment that caused timeout*
   *b. restart timer for the retransmitted packet*
3. ***Upon receiving ACK :***
   a. if ack acknowledges previously unacked segments  then update the sendbase value.
   b. start timer if there are  still unacked segments.


wait
for
event
Λ
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum
data received from application above
timeout
ACK received, with ACK field value y

```
create segment, seq. #: NextSeqNum
pass segment to IP (i.e., "send")
NextSeqNum = NextSeqNum + length(data)
if (timer currently not running)
    start timer
```

```
retransmit not-yet-acked segment
        with smallest seq. #
start timer
```

```
if (y > SendBase) {
   SendBase = y
   /* SendBase–1: last cumulatively ACKed byte */
   if (there are currently not-yet-acked segments)
       start timer
     else stop timer
  }
```

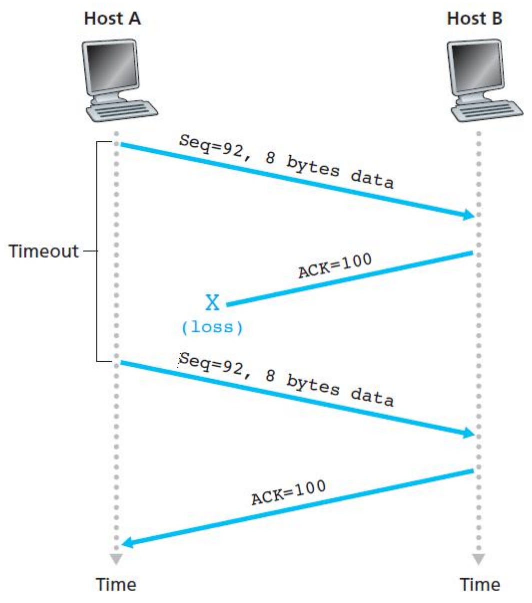
**TCP Receiver Events and actions :**

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

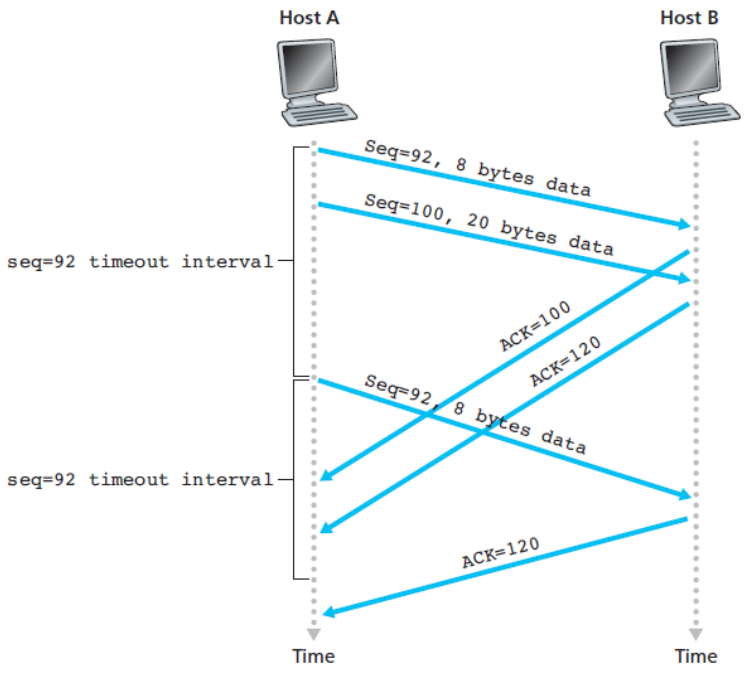
## A Few Interesting Scenarios:

### 1. Retransmission due to a lost acknowledgment.

- consider host A has send packet with seq.no : 92 to host B
- Host B received packet and sent an ACK :100 (next expected byte).
- ACK get lost in the network and did not reach to Host A.
- After timeout at sender retransmits the packet with seq.no : 92



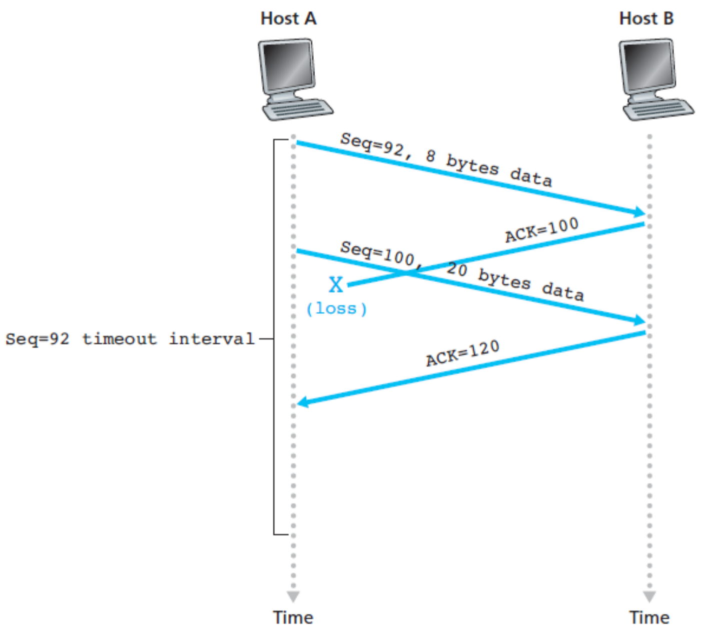
### 2. Premature timeout



- Host A sends two Packets with seq.no :92 and 100
- Before receiving any ACK of sent packets timer for first packet expires.
- Sender retransmits the packet with seq.no :92 and restarts the timer.
- Old ACK for both packets arrived during retransmitted packet’s timer interval.
- Second packet will not be retransmitted because ACK is received before timer expiration for the retransmitted packet.

## 3. Cumulative ACK

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



## 4. Doubling the Timeout Interval

Whenever the timeout event occurs, TCP retransmits the not-yet acknowledged segment with the smallest sequence number. But each time TCP retransmits, it sets the next timeout interval to twice the previous value, rather than deriving it from the last EstimatedRTT and DevRTT.

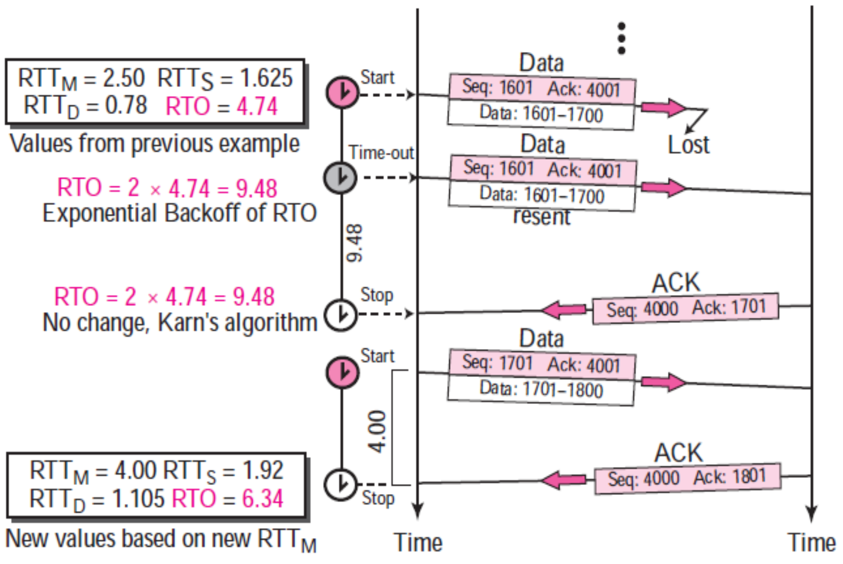
**For retransmitted segment double the next timer timeout time : Exponential backoff**

**Here, $RTT_M$ = SampledRTT**

**$RTT_S$ = EstimatedRTT**

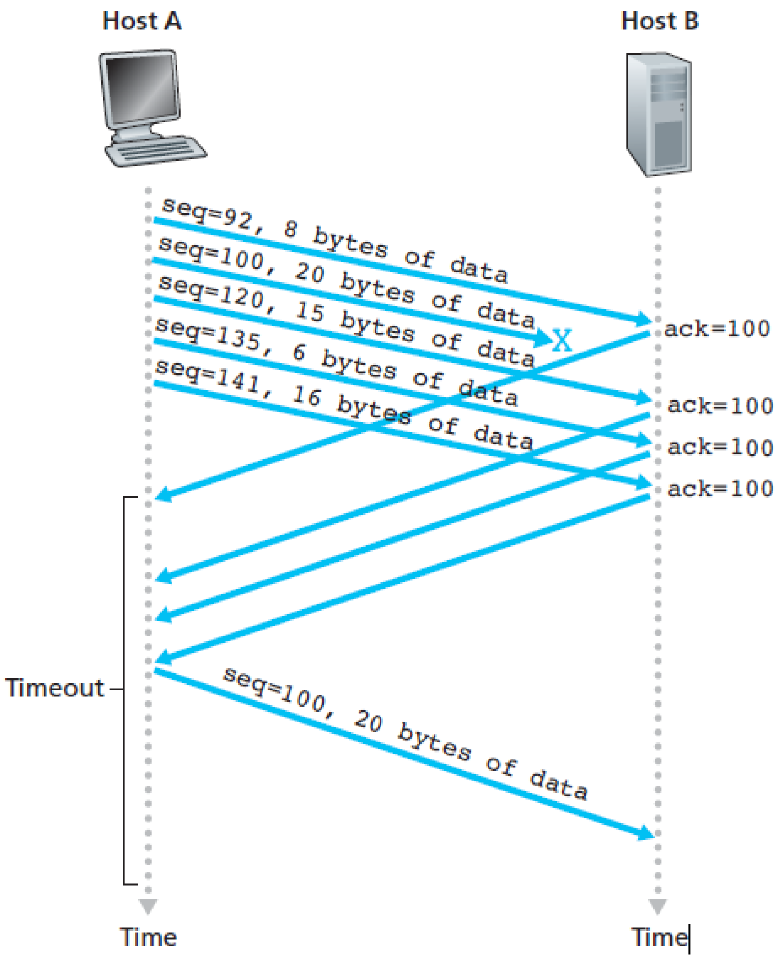
**$RTT_D$ = DevRTT**

**RTO=Retransmission Timer**



## 5. TCP Fast Retransmit

- When a segment is lost, this long timeout period forces the sender to delay resending the lost packet, thereby increasing the end-to end delay.
- The sender can often detect packet loss well before the timeout event occurs by so-called duplicate ACKs.
- **duplicate ACK** is an ACK that reacknowledges a segment for which the sender has already received an earlier acknowledgment.
- 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** that is retransmitting the missing segment *before* that segment's timer expires.



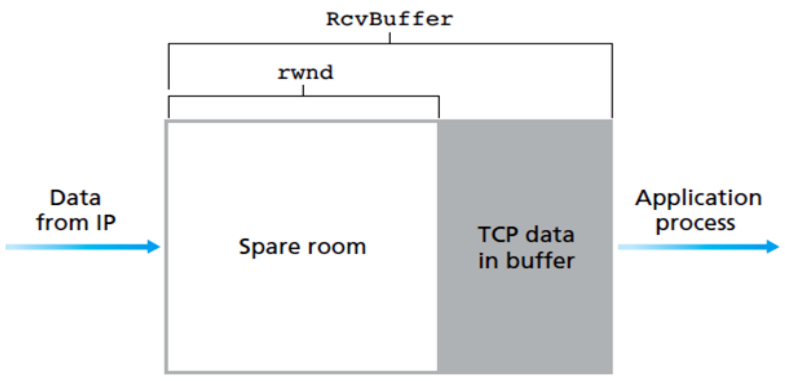
## Is TCP Go-Back-N or Selective Repeat ??

- Cumulative ACK and One Timer at sender side: properties of Go-Back-N
- Buffering out of order packets at receiver and Selective retransmission : properties of Selective Repeat

**So, TCP's error-recovery mechanism is probably best categorized as a hybrid of GBN and SR protocols**

## 3.4.5 TCP Flow Control:

- TCP provides a **flow-control service** to its applications to eliminate the possibility of the sender overflowing the receiver's buffer.
- Flow control is thus a speed-matching service—matching the rate at which the sender is sending against the rate at which the receiving application is reading.
- TCP provides flow control by having the sender maintain a variable called the receive window (rwnd).
- It gives the sender an idea of how much free buffer space is available at the receiver.



### Receiver side 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
- To prevent overflow :

**LastByteRcvd – LastByteRead <= RcvBuffer**

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

  **rwnd = RcvBuffer – [LastByteRcvd – LastByteRead]**

### Sender side variables:

- **LastByteSent**
- **LastByteAcked**
- difference between these two variables, **LastByteSent – LastByteAcked**, is the amount of unacknowledged data that A has sent into the connection.
- To prevent the overflow at receiver:

  **LastByteSent – LastByteAcked <= rwnd**
- If rwnd is set to 0 by receiver then sender will stop sending data but , keep sending 1 byte packet to receiver to know whether rwnd is set to new value or not to prevent deadlock.

## 3.4.6 TCP Connection Management

In TCP, connection-oriented transmission requires three phases: connection establishment, data transfer, and connection termination.

### Connection Establishment:

TCP transmits data in full-duplex mode. When two TCPs in two machines are connected, they are able to send segments to each other simultaneously.

### Three-Way Handshaking:

The connection establishment in TCP is called **three-way handshaking.**

The server program tells its TCP that it is ready to accept a connection. This request is called a *passive open.*

The client program issues a request for an *active open*. A client that wishes to connect to an open server tells its TCP to connect to a particular server.

1. The client sends the first segment, a SYN segment, in which only the SYN flag is set. This segment is for synchronization of sequence numbers. The client chooses a random number as the first sequence number and sends this number to the server.

This sequence number is called the **initial sequence number** (ISN). Note that this segment does not contain an acknowledgment number. It does not define the window size either.

**A SYN segment cannot carry data, but it consumes one sequence number.**

2. The server sends the second segment, a SYN + ACK segment with two flag bits set: SYN and ACK. This segment has a dual purpose. First, it is a SYN segment for communication in the other direction. The server uses this segment to initialize a sequence number for numbering the bytes sent from the server to the client. The server also acknowledges the receipt of the SYN segment from the client by setting the ACK flag and displaying the next sequence number it expects to receive from the client. Because it contains an acknowledgment, it also needs to define the receive window size, *rwnd* (to be used by the client).
3. The client sends the third segment. This is just an ACK segment. It acknowledges the receipt of the second segment with the ACK flag and acknowledgment number field.

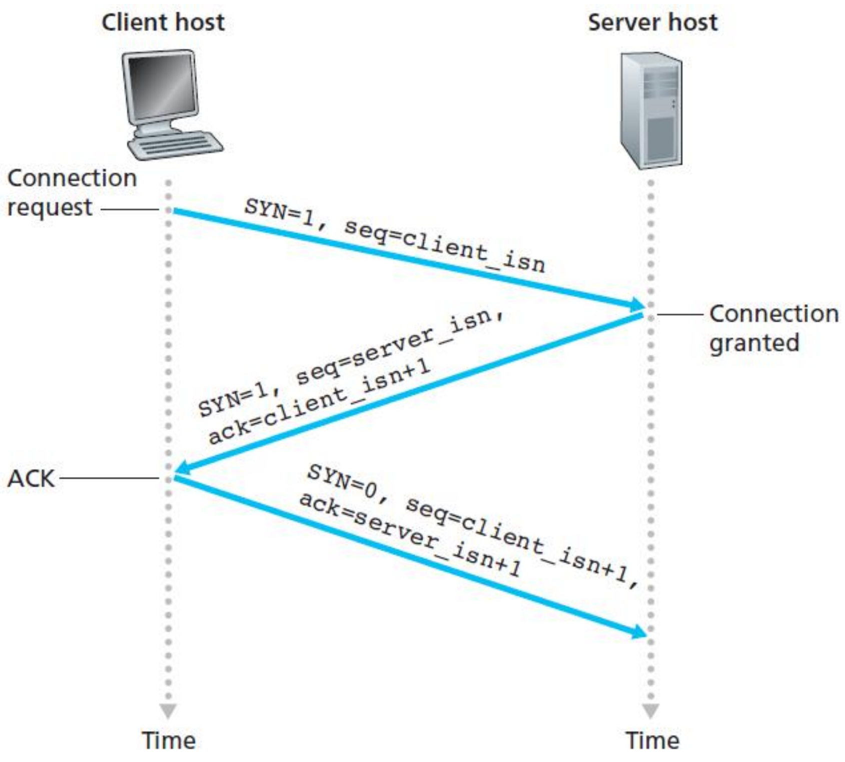


**Figure 3.39** ♦ TCP three-way handshake: segment exchange

## Connection Termination

Any of the two parties involved in exchanging data (client or server) can close the connection, although it is usually initiated by the client.

### *Three-Way Handshaking*

- The client application process issues a close command. This causes the client TCP to send a special TCP segment to the server process. This special segment has a flag bit in the segment's header, the FIN bit set to 1.
- When the server receives this segment, it sends the client an acknowledgment segment in return.
- The server then sends its own shutdown segment, which has the FIN bit set to 1.
- Finally, the client acknowledges the
- server's shutdown segment. At this point, all the resources in the two hosts are now deallocated.

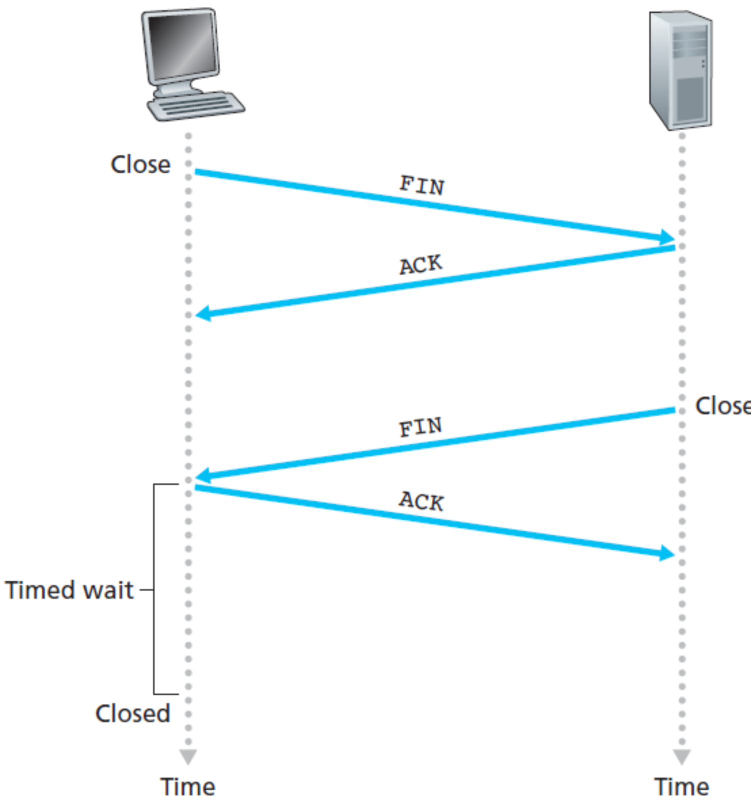


**Figure 3.40** ♦ Closing a TCP connection

## TCP States : Client side

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

- The client TCP begins in the CLOSED state. The application on the client side initiates a new TCP connection. This causes TCP in the client to send a SYN segment to TCP in the server.
- After having sent the SYN segment, the client TCP enters the SYN_SENT state. While in the SYN_SENT state, the client TCP waits for a segment from the server TCP that includes an acknowledgment for the client's previous segment and has the SYN bit set to 1.
- Having received such a segment, the client TCP enters the ESTABLISHED state. While in the ESTABLISHED state, the TCP client can send and receive TCP segments containing payload data.
- Suppose that the client application decides it wants to close the connection. This causes the client TCP to send a TCP segment with the FIN bit set to 1 and to enter the FIN_WAIT_1 state.
- While in the FIN_WAIT_1 state, the client TCP waits for a TCP segment from the server with an acknowledgment.
- When it receives this segment, the client TCP enters the FIN_WAIT_2 state. While in the FIN_WAIT_2 state, the client waits for another segment from the server with the FIN bit set to 1; after receiving this segment, the client TCP acknowledges the server's segment and enters the TIME_WAIT state.
- The TIME_WAIT state lets the TCP client resend the final acknowledgment in case the ACK is lost. The time spent in the TIME_WAIT state is implementation-dependent, but typical values are 30 seconds, 1 minute, and 2 minutes.
- After the wait, the connection formally closes and all resources on the client side (including port numbers) are released.

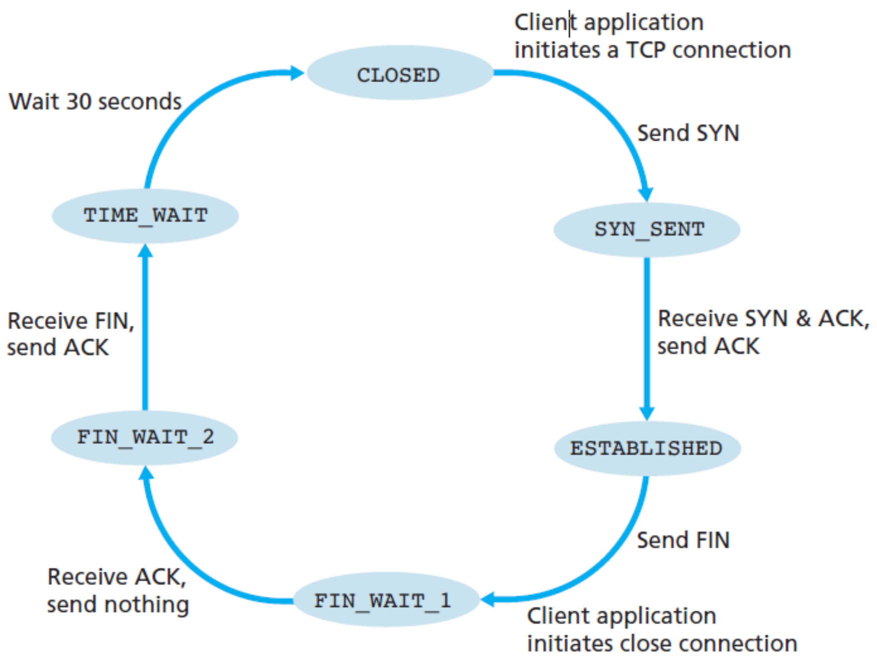

Client application initiates a TCP connection
CLOSED
Wait 30 seconds
Send SYN
TIME_WAIT
SYN_SENT
Receive FIN, send ACK
Receive SYN & ACK, send ACK
FIN_WAIT_2
ESTABLISHED
Send FIN
Receive ACK, send nothing
FIN_WAIT_1
Client application initiates close connection

**Figure 3.41** ♦ A typical sequence of TCP states visited by a client TCP

## TCP States : Server side

- Server application creates a Listen socket and enters into LISTEN state from CLOSED state.
- Whenever it receives SYN message from client , it sends SYN+ACK message and accept the connection request and enters into SYN_RCVD state.
- Upon receiving ACK from client in enters into ESTABLISHED state. . While in the ESTABLISHED state, the TCP server can send and receive TCP segments containing payload data.
- Suppose that the client application decides to close the connection and sends the FIN packet. Server will send ACK for Fin and enters into CLOSE_WAIT sate.
- Finally It will send FIN to client and enters into LAST_ACK state.
- Upon receiving final ACK from client server closes the connection and entred into CLOSED state for that particular connection.

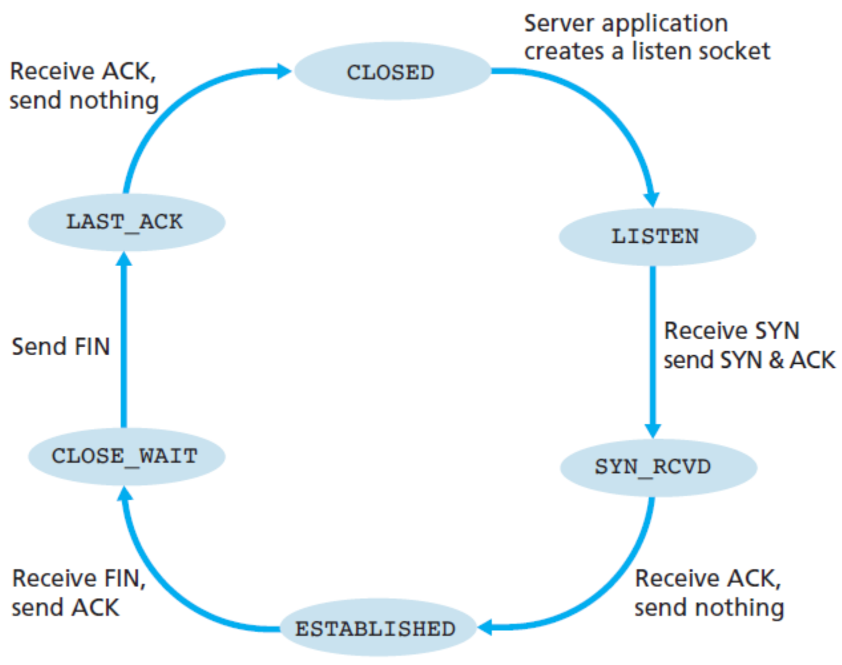


**Figure 3.42** ♦ A typical sequence of TCP states visited by a server-side TCP

## 3.6 Principles of Congestion Control:

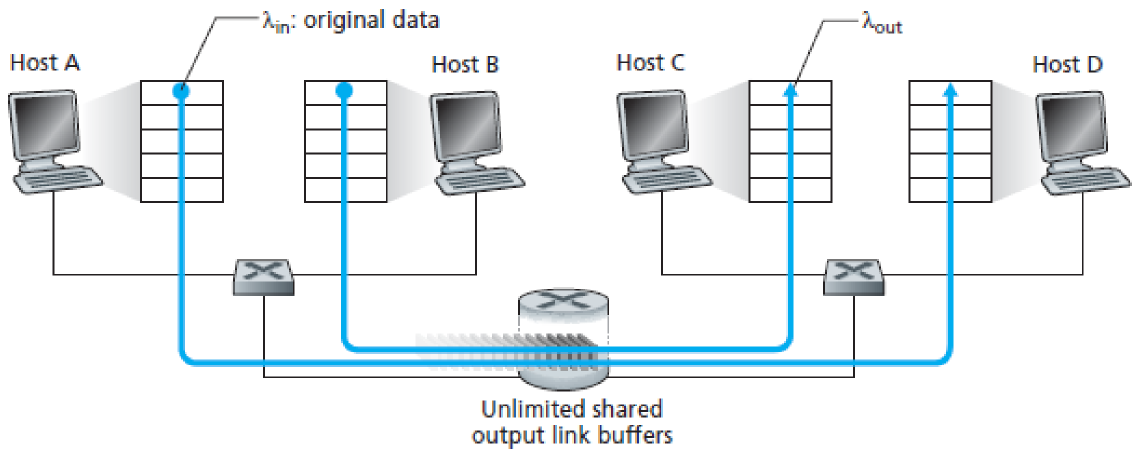
Network congestion—too many sources attempting to send data at too high a rate. To treat the cause of network congestion, mechanisms are needed to throttle senders in the face of network congestion.

### 3.6.1 The Causes and the Costs of Congestion

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

Consider two hosts (A and B), each having a connection that shares a single hop between source and Destination.

- Let's assume that the application in Host A and B is sending data into the connection at an average rate of $\lambda_{in}$ bytes/sec. The underlying transport-level protocol is very simple, Data is encapsulated and sent; no error recovery (for example, retransmission), flow control, or congestion control is performed.
- Figure 3.44 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.



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

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

- 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.

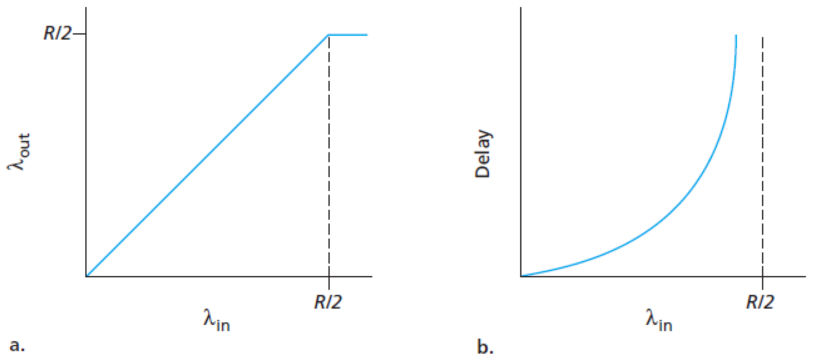


**Figure 3.44** ♦ Congestion scenario 1: Throughput and delay as a function of host sending rate

The right-hand graph in Figure 3.44, 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

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

## Scenario 2: Two Senders and a Router with Finite Buffers

Let us now slightly modify scenario 1 in the following two ways:

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

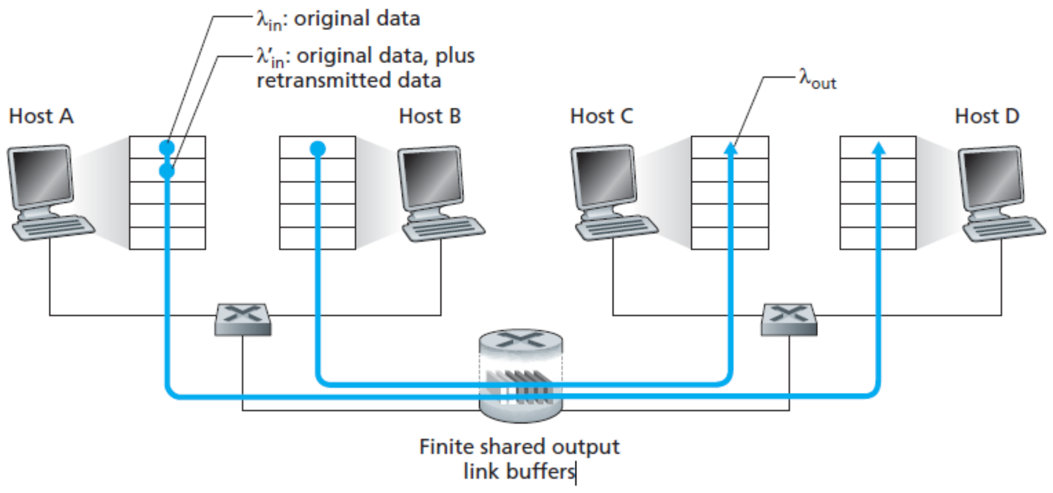

$\lambda_{in}$: original data
$\lambda'_{in}$: original data, plus retransmitted data
$\lambda_{out}$
Host A
Host B
Host C
Host D
Finite shared output link buffers

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

- 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, λin would be equal to λ`in and the throughput of the connection would be equal to λin. This case is shown in Figure 3.46(a).
- Consider next the slightly more realistic case that the sender retransmits when a packet is known for certain to be lost. Now connection contains original + some retransmitted data. Consider offered rate to each connection is R/2, But due to it contains some retransmitted data throughput will be less than R/2.
- According to Figure 3.46(b), at this value of the offered load, the rate at which data are delivered to the receiver application is *R*/3. Thus, out of the 0.5*R* units of data transmitted, 0.333*R* bytes/sec are original data and 0.166*R* bytes/ sec are retransmitted data.

***Here another cost of a congested network— the sender must perform retransmissions in order to compensate for dropped (lost) packets due to buffer overflow.***

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

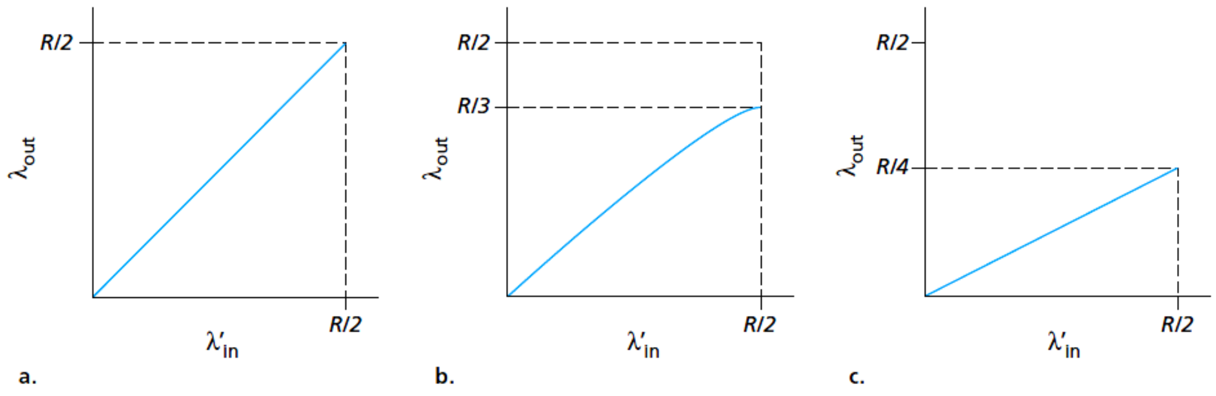


**Figure 3.46** ♦ Scenario 2 performance with finite buffers

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

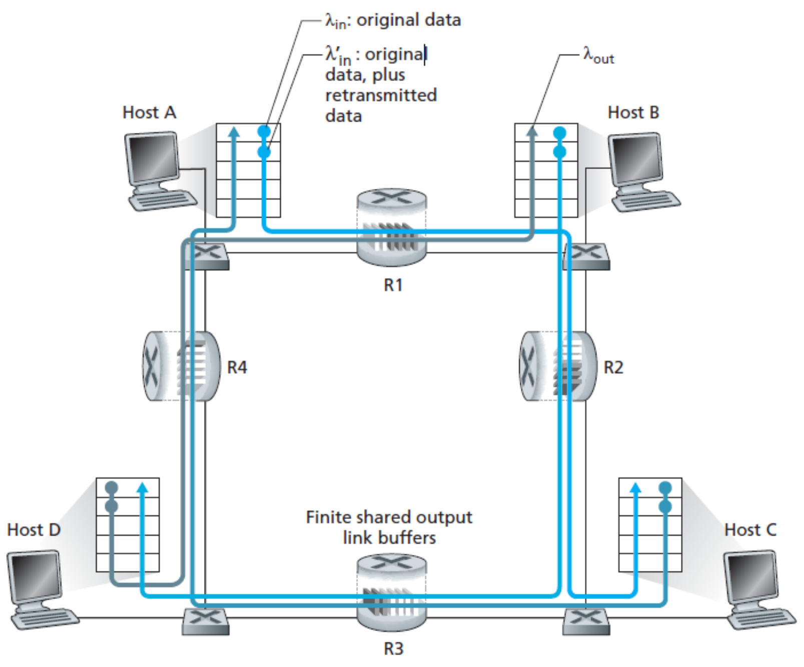

λin: original data
λ'in : original data, plus retransmitted data
λout
Host A
Host B
Host C
Host D
R1
R2
R3
R4
Finite shared output link buffers

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

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.

Consider the A–C traffic arriving to router R2 from R1 can have an arrival rate at R2 that is at most *R. The rate of A-C from R1 to 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.

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."

***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.***

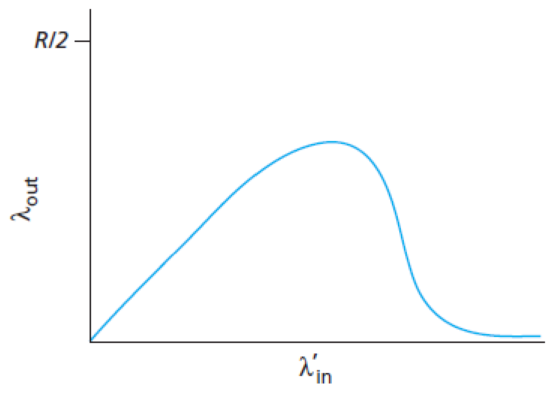


**Figure 3.48** ♦ Scenario 3 performance with finite buffers and multihop paths

## 3.6.2 Approaches to Congestion Control

### Two main approaches of congestion control

### 1. *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 behaviour (for example, packet loss and delay).

### 2. *Network-assisted congestion control:*

With network-assisted congestion control, network-layer components (that is, routers) provide explicit feedback to the sender regarding the congestion state in the network. This feedback may be as simple as a single bit indicating congestion at a link. This approach is used in ATM available bit-rate (ABR) congestion control.

For network-assisted congestion control, congestion information is typically fed back from the network to the sender in one of two ways, as shown in Figure 3.49. Direct feedback may be sent from a network router to the sender. This form of notification typically takes the form of a **choke packet.**

Second, the receiver then notifies the sender of the congestion indication. Note that this latter form of notification takes at least a full round-trip time.

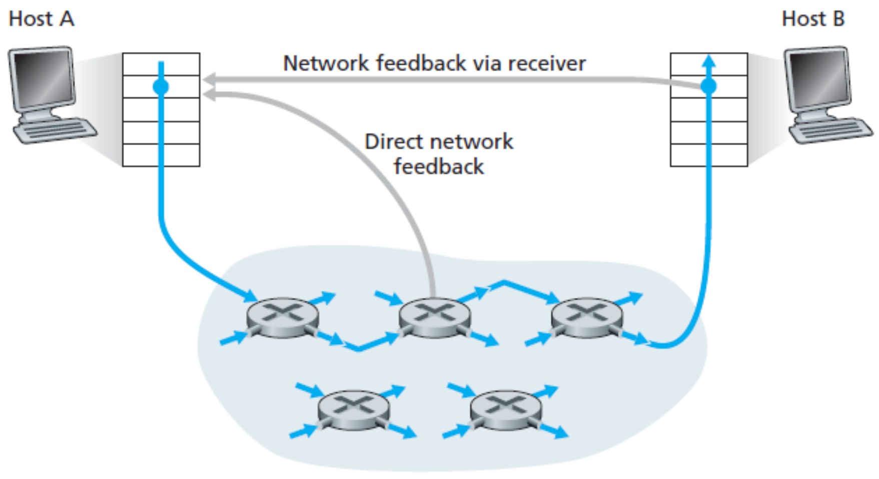


**Figure 3.49** ♦ Two feedback pathways for network-indicated congestion information

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

- 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.
- ABR has been designed as an elastic data transfer service.
- 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 limit its transmission rate to some predetermined minimum transmission rate.
- With ATM ABR service, data cells are transmitted from a source to a destination through a series of intermediate switches. Interspersed with the data are **resource-management cells (RM cells)**; these RM cells can be used to convey congestion-related information among the hosts and switches.

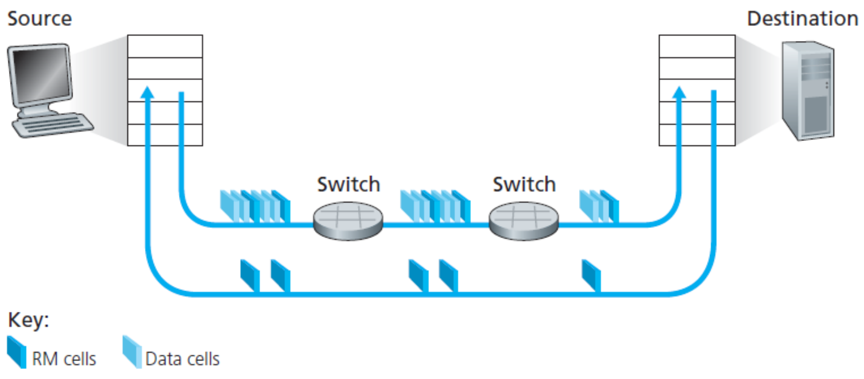


**Figure 3.50** ♦ Congestion-control framework for ATM ABR service

ABR provides three mechanisms for signaling congestion-related information from the switches to the receiver:

1. ***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**.**

2. ***CI and NI bits.***

RM cell is sent after 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 as it is.

3. ***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.

## 3.7 TCP Congestion Control

### Congestion Window:

Previously, we talked about flow control and tried to discuss solutions when the receiver is overwhelmed with data. We said that the sender window size is determined by the available buffer space in the receiver (rwnd). If the network cannot deliver the data as fast as it is created by the sender, it must tell the sender to slow down. In other words, in addition to the receiver, the network is a second entity that determines the size of the sender's window.

The sender has two pieces of information: the receiver-advertised window size and the congestion window size (cwnd).

**Actual window size** = **minimum (rwnd, cwnd)**

### Some guiding principles:

- A lost segment implies congestion →decrease the sending rate when a segment is lost.
- On arrival of ACK →The sender's rate can be increased.
- Bandwidth probing → Increase transmission rate in response to arriving ACKs until a loss event occurs.

### Congestion Policy

TCP's general policy for handling congestion is based on three phases: **slow start, congestion avoidance, and congestion detection.**

### *Slow Start: Exponential Increase*

The **slow start** algorithm is based on the idea that the size of the congestion window (cwnd) starts with one maximum segment size (MSS). The size of the window increases one MSS each time one acknowledgement arrives. As the name implies, the algorithm starts slowly, but grows exponentially.

- The sender starts with cwnd = 1 MSS. This means that the sender can send only one segment. After the first ACK arrives, the size of the congestion window is increased by 1, which means that cwnd is now 2. Now two more segments can be sent.
- When two more ACKs arrive, the size of the window is increased by 1 MSS for each ACK, which
- means cwnd is now 4. Now four more segments can be sent.

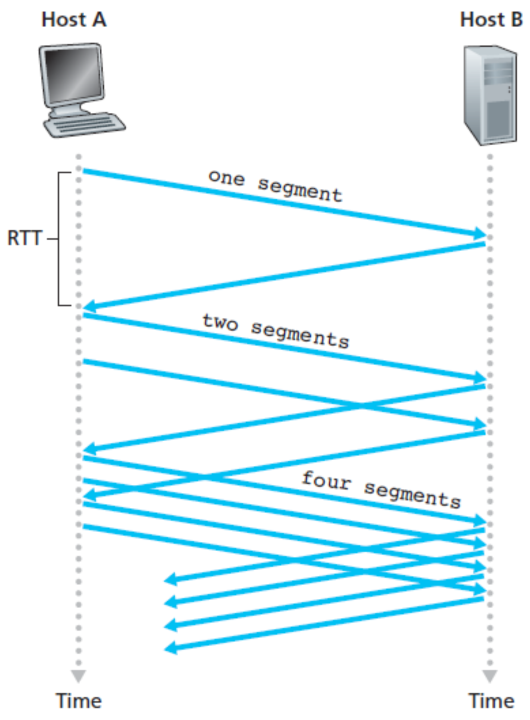


**Figure 3.51** ♦ TCP slow start

Slow start cannot continue indefinitely. There must be a threshold to stop this phase. The sender keeps track of a variable named *ssthresh* (slow start threshold). When the size of window in bytes reaches this threshold, slow start stops and the  ext phase starts.

## *Congestion Avoidance: Additive Increase*

To avoid congestion before it happens, one must slow down this exponential growth. TCP defines another algorithm called **congestion avoidance,** which increases the cwnd additively instead of exponentially.

When the size of the congestion window reaches the slow start threshold in the case where cwnd = *i*, the slow start phase stops and the additive phase begins. In this algorithm, each time the whole "window" of segments is acknowledged, the size of the congestion window is increased by one.

**Start --> cwnd** = ***i***

**After 1 RTT --> cwnd** = ***i*** + **1**

**After 2 RTT --> cwnd** = ***i*** + **2**

**After 3 RTT --> cwnd** = ***i*** + **3**

## *Congestion Detection: Multiplicative Decrease*

If congestion occurs, the congestion window size must be decreased. The only way a sender can guess that congestion has occurred is the need to retransmit a segment.

However, retransmission can occur in one of two cases: when the RTO timer times out or when three duplicate ACKs are received. In both cases, the size of the threshold is dropped to half (**multiplicative decrease**).

Most TCP implementations have two reactions:

**1. If a time-out occurs (TCP Tahoe),** there is a stronger possibility of congestion; a segment has probably been dropped in the network and there is no news about the following sent segments. In this case TCP reacts strongly:

**a.** It sets the value of the threshold to half of the current window size.

**b.** It reduces cwnd back to one segment.

**c.** It starts the slow start phase again.

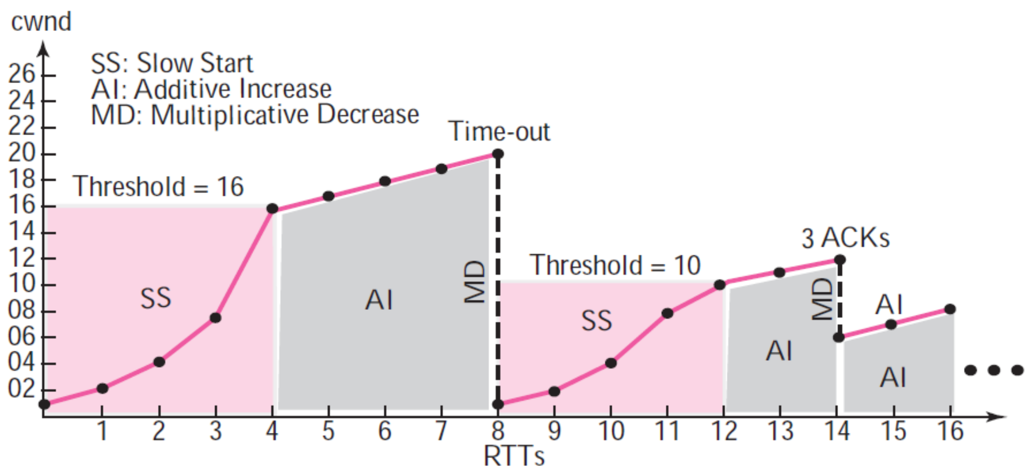
**2. If three duplicate ACKs are received (TCP Reno),** there is a weaker possibility of congestion; a segment may have been dropped but some segments after that have arrived safely since three duplicate ACKs are received. **This is called fast transmission and fast recovery**.

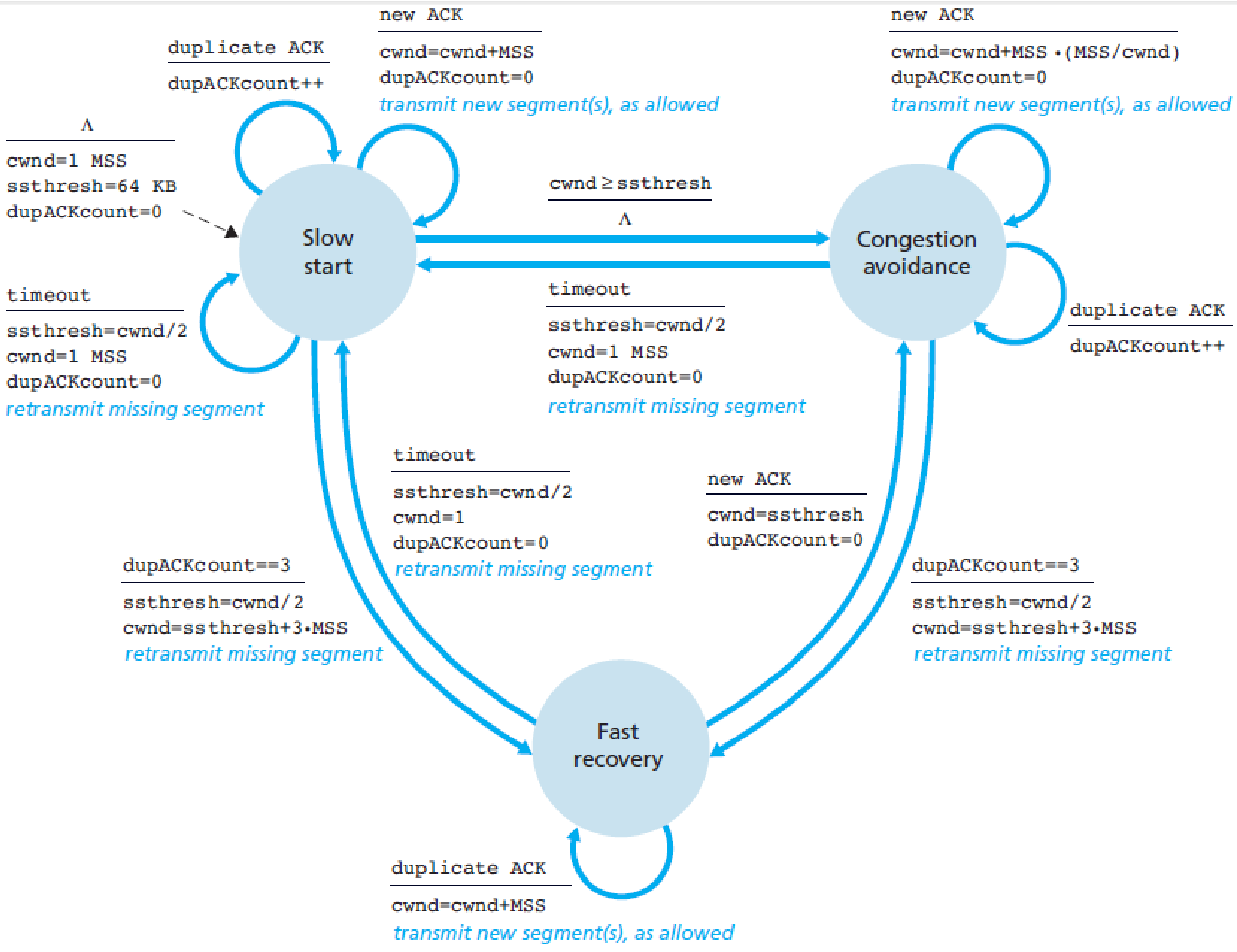
In this case, TCP has a weaker reaction as shown below:

**a.** It sets the value of the threshold to half of the current window size.

**b.** It sets cwnd to the value of the threshold (some implementations add three segment sizes to the threshold).

**c.** It starts the congestion avoidance phase




duplicate ACK
dupACKcount++
new ACK
cwnd=cwnd+MSS
dupACKcount=0
transmit new segment(s), as allowed
new ACK
cwnd=cwnd+MSS •(MSS/cwnd)
dupACKcount=0
transmit new segment(s), as allowed
Λ
cwnd=1 MSS
ssthresh=64 KB
dupACKcount=0
cwnd ≥ ssthresh
Λ
Slow start
Congestion avoidance
timeout
ssthresh=cwnd/2
cwnd=1 MSS
dupACKcount=0
retransmit missing segment
timeout
ssthresh=cwnd/2
cwnd=1 MSS
dupACKcount=0
retransmit missing segment
duplicate ACK
dupACKcount++
timeout
ssthresh=cwnd/2
cwnd=1
dupACKcount=0
retransmit missing segment
new ACK
cwnd=ssthresh
dupACKcount=0
dupACKcount==3
ssthresh=cwnd/2
cwnd=ssthresh+3•MSS
retransmit missing segment
dupACKcount==3
ssthresh=cwnd/2
cwnd=ssthresh+3•MSS
retransmit missing segment
Fast recovery
duplicate ACK
cwnd=cwnd+MSS
transmit new segment(s), as allowed