MODULE 4

Different services provided by the transport layer

1. Process-to-Process Communication

The first duty of a transport-layer protocol is to provide **process-to-process communication**. A process is an application-layer entity (running program) that uses the services of the transport layer. The network layer is responsible for communication at the computer level (host-to-host communication). A network-layer protocol can deliver the message only to the destination computer. However, this is an incomplete delivery. The message still needs to be handed to the correct process. This is where a transport-layer protocol takes over. A transport-layer protocol is responsible for delivery of the message to the appropriate process. Figure shows the domains of a network layer and a transport layer.

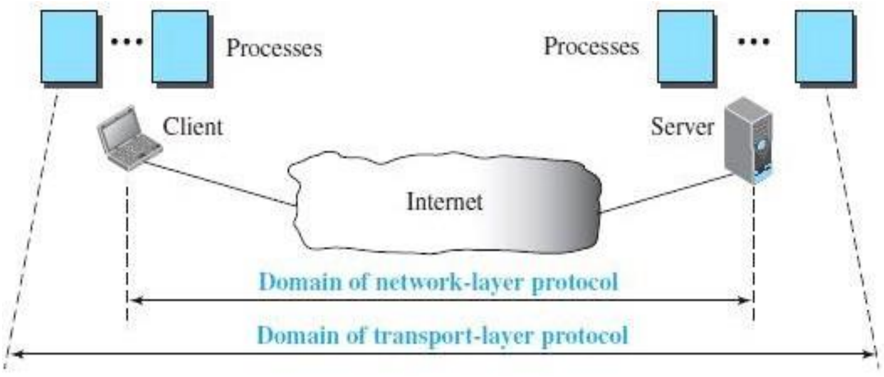

Processes
Processes
Client
Server
Internet
Domain of network-layer protocol
Domain of transport-layer protocol

Figure 1: Network layer versus transport layer

2. Encapsulation and Decapsulation

To send a message from one process to another, the transport-layer protocol encapsulates and decapsulates messages (Figure 2). Encapsulation happens at the sender site. When a process has a message to send, it passes the message to the transport layer along with a pair of socket addresses and some other pieces of information, which depend on the transport-layer protocol. The transport layer receives the data and adds the transport-layer header. The packets at the transport layer in the Internet are called *user datagrams*, *segments*, or *packets*, depending on what transport-layer protocol we use. In general,transport-layer payloads are called as *packets.*

Decapsulation happens at the receiver site. When the message arrives at the destination transport layer, the header is dropped and the transport layer delivers the message to the process running at the application layer. The sender socket address is passed to the process in case it needs to respond to the message received.

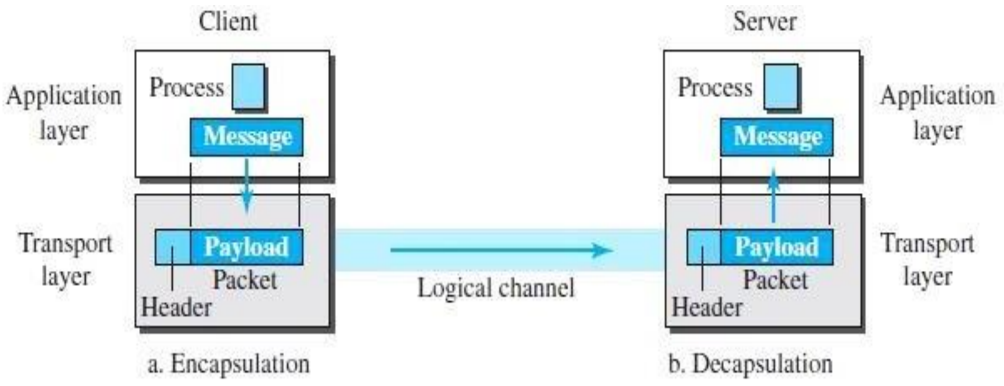


Figure 2: Encapsulation and decapsulation

3. Multiplexing and Demultiplexing

Whenever an entity accepts items from more than one source, this is referred to as multiplexing (many to one); whenever an entity delivers items to more than one source, this is referred to as demultiplexing (one to many). The transport layer at the source performs multiplexing; the transport layer at the destination performs demultiplexing

Figure 3 shows communication between a client and two servers. Three client processes are running at the client site, P1, P2, and P3. The processes P1 and P3 need to send requests to the corresponding server process running in a server. The client process P2 needs to send a request to the corresponding server process running at another server. The transport layer at the client site accepts three messages from the three processes and creates three packets. It acts as a multiplexer. The packets 1 and 3 use the same logical channel to reach the transport layer of the first server. When they arrive at the server, the transport layer does the job of a demultiplexer and distributes the messages to two different processes. The transport layer at the second server receives packet 2 and delivers it to the corresponding process. Note that we still have demultiplexing although there is only one message.

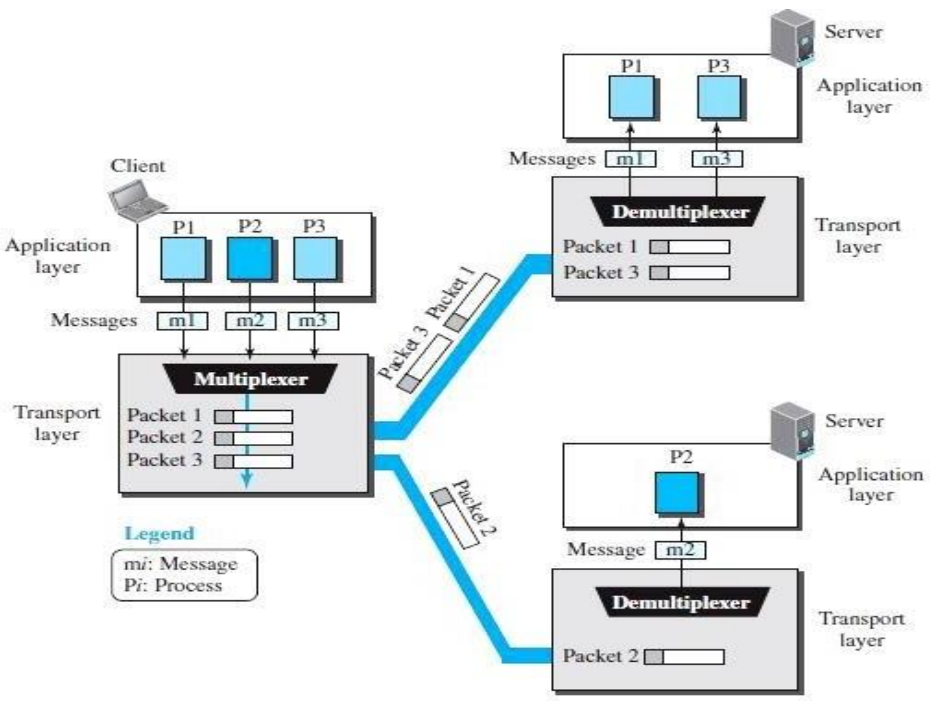


Figure 3: Multiplexing and demultiplexing

4. Flow Control

Whenever an entity produces items and another entity consumes them, there should be a balance between production and consumption rates. If the items are produced faster than they can be consumed, the consumer can be overwhelmed and may need to discard some items. If the items are produced more slowly than they can be consumed, the consumer must wait, and the system becomes less efficient. Flow control is related to the first issue. We need to prevent losing the data items at the consumer site.

Pushing or Pulling

Delivery of items from a producer to a consumer can occur in one of two ways: *pushing* or *pulling*. If the sender delivers items whenever they are produced without a prior request from the consumer the delivery is referred to as *pushing*. If the producer delivers the items after the consumer has requested them, the delivery is referred to as *pulling*. Figure 4 shows these two types of delivery.

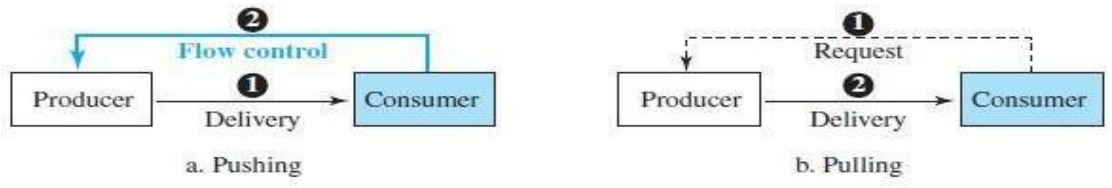


Figure 4: Pushing or pulling

Flow Control at Transport Layer

In communication at the transport layer, we are dealing with four entities: sender process, sender transport layer, receiver transport layer, and receiver process. The sending process at the application layer is only a producer. It produces message chunks and pushes them to the transport layer.

The sending transport layer has a double role: It is both a consumer and a producer. It consumes the messages pushed by the producer. It encapsulates the messages in packets and pushes them to the receiving transport layer. The receiving transport layer also has a double role: it is the consumer for the packets received from the sender and the producer that decapsulates the messages and delivers them to the application layer. The last delivery, however, is normally a pulling delivery; the transport layer waits until the application-layer process asks for messages.

Figure 5 shows that we need at least two cases of flow control: from the sending transport layer to the sending application layer and from the receiving transport layer to the sending transport layer.

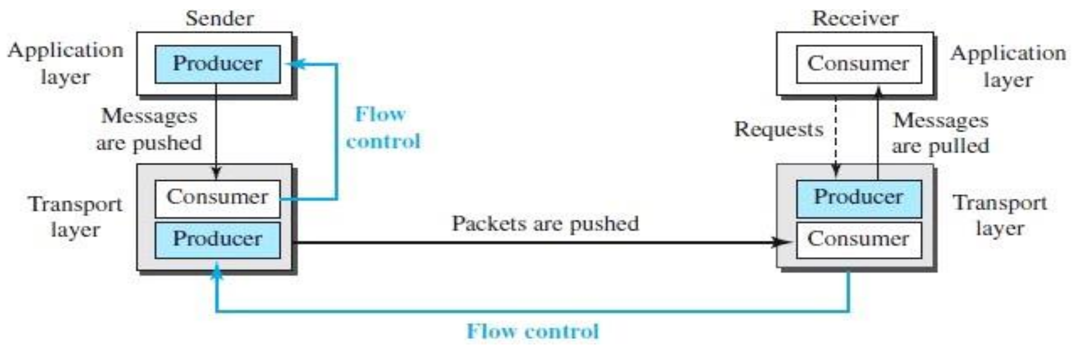


Figure 5: Flow control at the transport layer

5. Error Control

In the Internet, since the underlying network layer (IP) is unreliable, we need to make the transport layer reliable if the application requires reliability. Reliability can be achieved to add error control services to the transport layer. Error control at the transport layer is responsible for

1. Detecting and discarding corrupted packets.

2. Keeping track of lost and discarded packets and resending them.

3. Recognizing duplicate packets and discarding them.

4. Buffering out-of-order packets until the missing packets arrive.

Error control, unlike flow control, involves only the sending and receiving transport layers. Assume that the message chunks exchanged between the application and transport layers are error free. Figure 6 shows the error control between the sending and receiving transport layers. As with the case of flow control, the receiving transport layer manages error control, most of the time, by informing the sending transport layer about the problems.

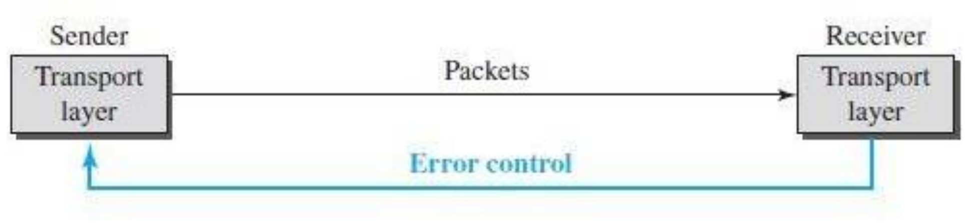


Figure 6: Error control at the transport layer

Sequence Numbers

Error control requires that the sending transport layer knows which packet is to be resent and the receiving transport layer knows which packet is a duplicate, or which packet has arrived out of order. This can be done if the packets are numbered. We can add a field to the transport-layer packet to hold the **sequence number** of the packet. When a packet is corrupted or lost, the receiving transport layer can somehow inform the sending transport layer to resend that packet using the sequence number. The receiving transport layer can also detect duplicate packets if two received packets have the same sequence number. The out-of-order packets can be recognized by observing gaps in the sequence numbers. Packets are numbered sequentially. However, because we need to include the sequence number of each packet in the header, we need to set a limit. If the header of the packet allows m bits for the sequence number, the sequence numbers range from 0 to 2^m -1.

For example, if m is 4, the only sequence numbers are 0 through 15, inclusive. However, we can wrap around the sequence. So the sequence numbers in this case are

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ...

In other words, the sequence numbers are modulo 2^m.

Acknowledgment

The receiver side can send an acknowledgment (ACK) for each of a collection of packets that have arrived safe and sound. The receiver can simply discard the corrupted packets. The sender can detect lost packets if it uses a timer. When a packet is sent, the sender starts a timer. If an ACK does not arrive before the timer expires, the sender resends the packet.

Duplicate packets can be silently discarded by the receiver. Out-of-order packets can be either discarded (to be treated as lost packets by the sender), or stored until the missing one arrives.

Combination of Flow and Error Control

- Flow control requires the use of two buffers, one at the sender site and the other at the receiver site.
- Error control requires the use of sequence and acknowledgment numbers by both sides.
- These two requirements can be combined if we use two numbered buffers, one at the sender, one at the receiver.

Sliding window with a neat diagram.

Sliding Window

Since the sequence numbers use modulo 2^m, a circle can represent the sequence numbers from 0 to $2^m - 1$ (Figure 7). The buffer is represented as a set of slices, called the ***sliding window,*** that occupies part of the circle at any time. At the sender site, when a packet is sent, the corresponding slice is marked. When all the slices are marked, it means that the buffer is full and no further messages can be accepted from the application layer.

When an acknowledgment arrives, the corresponding slice is unmarked. If some consecutive slices from the beginning of the window are unmarked, the window slides over the range of the corresponding sequence numbers to allow more free slices at the end of the window. Figure 7 shows the sliding window at the sender. The sequence numbers are in modulo 16 (m = 4) and the size of the window is 7. The sliding window is just an abstraction: the actual situation uses computer variables to hold the sequence numbers of the next packet to be sent and the last packet sent.

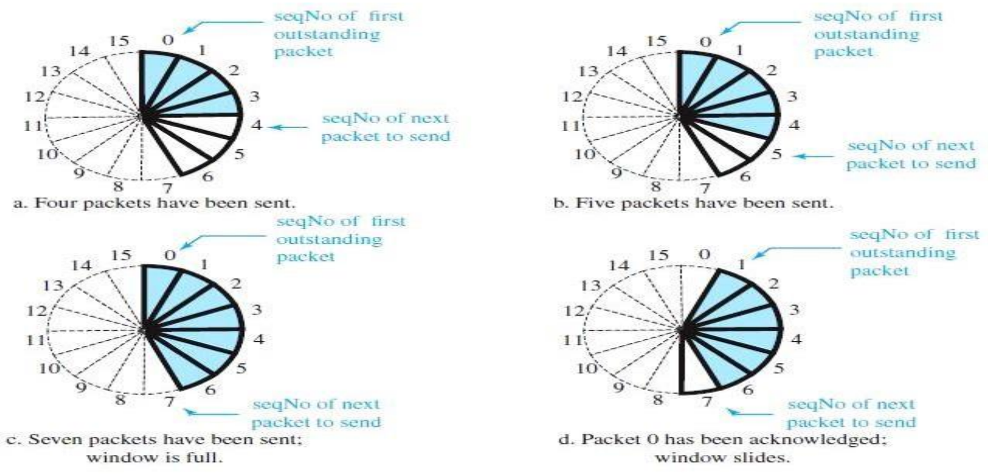

seqNo of first outstanding packet
seqNo of next packet to send
a. Four packets have been sent.
seqNo of first outstanding packet
seqNo of next packet to send
b. Five packets have been sent.
seqNo of first outstanding packet
seqNo of next packet to send
c. Seven packets have been sent; window is full.
seqNo of first outstanding packet
seqNo of next packet to send
d. Packet 0 has been acknowledged; window slides.

Figure 7: Sliding window in circular format

Most protocols show the sliding window using linear representation. The idea is the same, but it normally takes less space on paper. Figure 8 shows this representation.

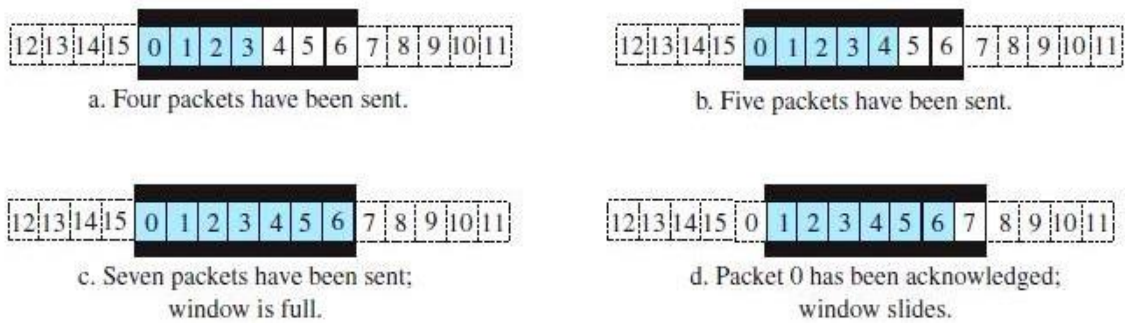

12 13 14 15 0 1 2 3 4 5 6 7 8 9 10 11
a. Four packets have been sent.
b. Five packets have been sent.
c. Seven packets have been sent; window is full.
d. Packet 0 has been acknowledged; window slides.

Figure 8: Sliding window in linear format

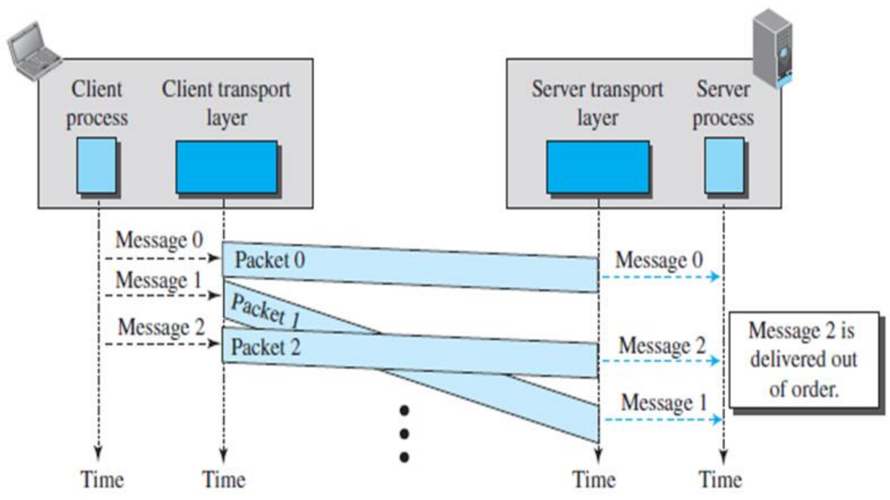
6. Congestion Control

Congestion in a network may occur if the load on the network—the number of packets sent to the network is greater than the capacity of the network, the number of packets a network can handle. Congestion control refers to the mechanisms and techniques that control the congestion and keep the load below the capacity.

Congestion happens in any system that involves waiting. For example, congestion happens on a freeway because any abnormality in the flow, such as an accident during rush hour, creates blockage. Congestion in a network or internetwork occurs because routers and switches have queues—buffers that hold the packets before and after processing. A router, for example, has an input queue and an output queue for each interface. If a router cannot process the packets at the same rate at which they arrive, the queues become overloaded and congestion occurs. Congestion at the transport layer is actually the result of congestion at the network layer, which manifests itself at the transport layer.

Connectionless oriented service

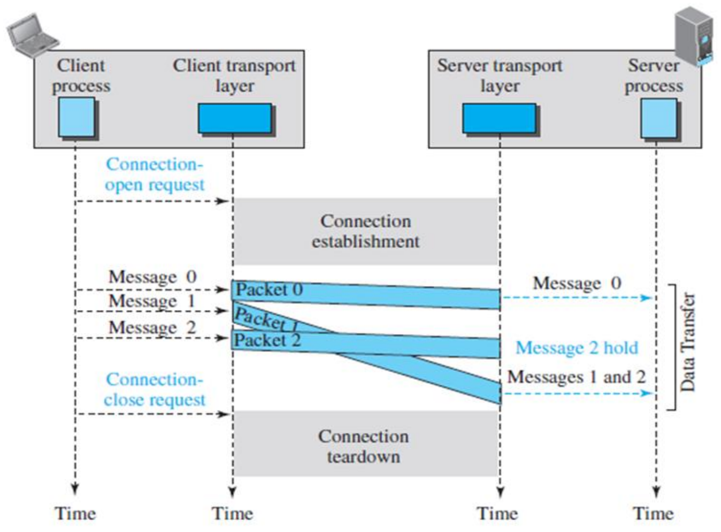
- The packets are sent from one party to another with no need for connection establishment or connection release.
- The packets are not numbered
- They may be delayed or lost or may arrive out of sequence.
- There is no acknowledgment either
- No facility for reordering of lost/corrupted packets.
- UDP, is connectionless.
- No flow control, error control, or congestion control can be effectively implemented in a connectionless service



Connection oriented service

- A connection establishment between the sender

  and the receiver, data transfer and connection release
- The packets are numbered
- There is an acknowledgment both way
- Facility for reordering of lost/corrupted packets
- TCP,SCTP



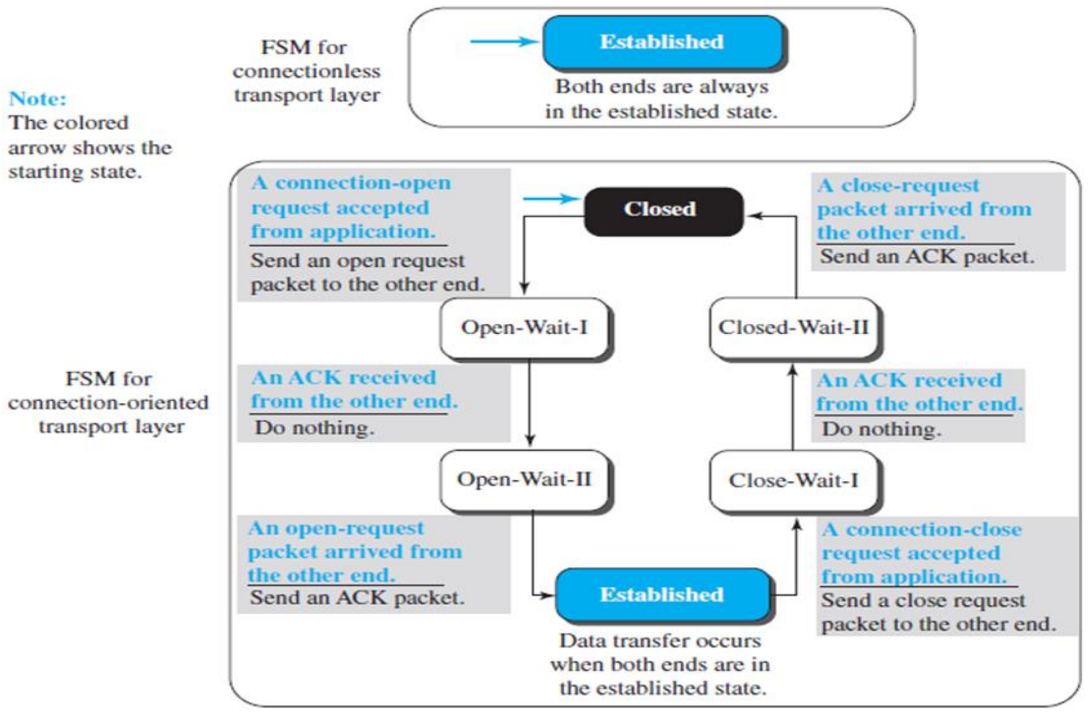
Finite State Machine

Connectionless transport layer as an FSM with only one state: the established state. The machine on each end (client and server) is always in the established state, ready to send and receive transport-layer packets.

Figure shows a representation of a transport layer using an FSM. Using this tool, each transport layer (sender or receiver) is taught as a machine with a finite number of states. The machine is always in one of the states until an *event* occurs. Each event is associated with two reactions: defining the list (possibly empty) of actions to be performed and determining the next state (which can be the same as the current state). One of the states must be defined as the initial state, the state in which the machine starts when it turns on.

An FSM in a connection-oriented transport layer, on the other hand, needs to go through three states before reaching the established state. The machine also needs to go through three states before closing the connection. The machine is in the closed state when there is no connection. It remains in this state until a request for opening the connection arrives from the local process; the machine sends an open request packet to the remote transport layer and moves to the open-wait-I state. When an acknowledgment is received from the other end, the local FSM moves to the open-wait-II state. When the machine is in this state, a unidirectional connection has been established, but if a bidirectional connection is needed, the machine needs to wait in this state until the other end also requests a connection. When the request is received, the machine sends an acknowledgment and moves to the established state.

Data and data acknowledgment can be exchanged between the two ends when they are both in the established state To tear down a connection, the application layer sends a close request message to its local transport layer. The transport layer sends a close-request packet to the other end and moves to close-wait-I state. When an acknowledgment is received from the other end, the machine moves to the close-wait-II state and waits for the close-request packet from the other end. When this packet arrives, the machine sends an acknowledgment and moves to the closed state.


Note:
The colored arrow shows the starting state.
FSM for connectionless transport layer
Established
Both ends are always in the established state.
FSM for connection-oriented transport layer
A connection-open request accepted from application.
Send an open request packet to the other end.
Closed
A close-request packet arrived from the other end.
Send an ACK packet.
Open-Wait-I
Closed-Wait-II
An ACK received from the other end.
Do nothing.
An ACK received from the other end.
Do nothing.
Open-Wait-II
Close-Wait-I
An open-request packet arrived from the other end.
Send an ACK packet.
Established
A connection-close request accepted from application.
Send a close request packet to the other end.
Data transfer occurs when both ends are in the established state.

TRANSPORT LAYER PROTOCOLS

Simple Protocol

- Simple connectionless protocol with neither flow nor error control
- Assume that the receiver can immediately handle any packet it receives

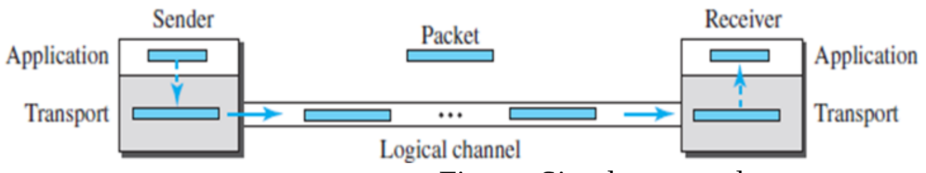


Figure: Simple protocol

The transport layer at the sender gets a message from its application layer, makes a packet out of it, and sends the packet. The transport layer at the receiver receives a packet from its network layer, extracts the message from the packet, and delivers the message to its application layer. The transport layers of the sender and receiver provide transmission services for their application layers.

FSMs

- The sender site should not send a packet until its application layer has a message to send.
- The receiver site cannot deliver a message to its application layer until a packet arrives.
- Each FSM has only one state, the ready state. The sending machine remains in the ready state until a request comes from the process in the application layer.
- When this event occurs, the sending machine encapsulates the message in a packet and sends it to the receiving machine.
- The receiving machine remains in the ready state until a packet arrives from the sending machine. When this event occurs, the receiving machine decapsulates the message out of the packet and delivers it to the process at the application layer.

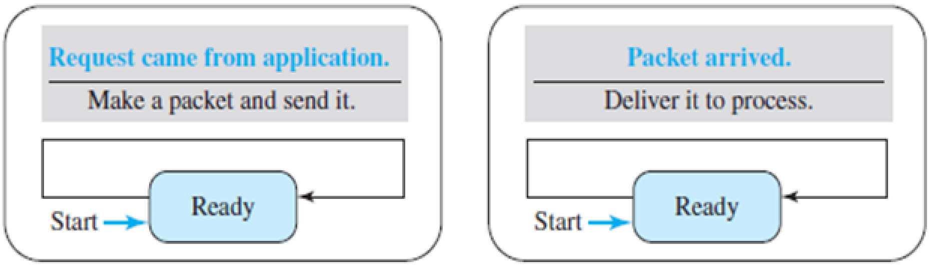


Figure: FSM of Simple protocol

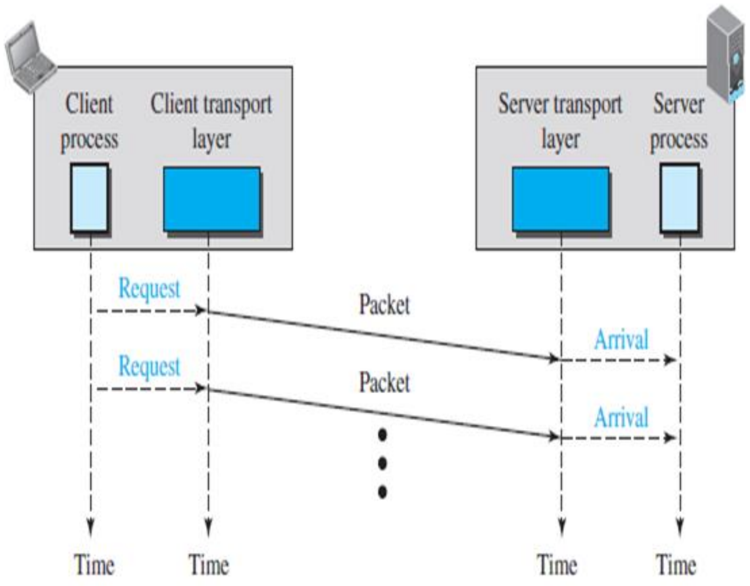
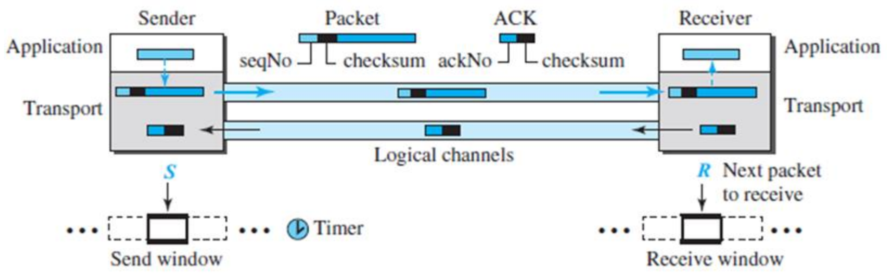


Figure: Flow diagram of Simple protocol

Stop-and-Wait Protocol

- Connection-oriented protocol
- Uses both flow and error control

Both the sender and the receiver use a sliding window of size 1



Sequence Numbers

Assume that the sender has sent the packet with sequence number x. Three things can happen.

1. The packet arrives safe and sound at the receiver site; the receiver sends an acknowledgment. The acknowledgment arrives at the sender site, causing the sender to send the next packet numbered x + 1

2. The packet is corrupted or never arrives at the receiver site; the sender resends the packet (numbered x) after the time-out. The receiver returns an acknowledgment.

3. The packet arrives safe and sound at the receiver site; the receiver sends an acknowledgment, but the acknowledgment is corrupted or lost

Acknowledgment Numbers

The acknowledgment numbers always announce the
sequence number of the next packet expected by the receiver.

For example, if packet 0 has arrived safe and sound, the receiver sends an ACK with acknowledgment 1. (meaning packet 1 is expected next)

If packet 1 has arrived safe and sound, the receiver sends an ACK with acknowledgment 0 (meaning packet 0 is expected).

In the Stop-and-Wait protocol, the acknowledgment number always announces, in modulo-2 arithmetic, the sequence number of the next packet expected.

FSMs

Sender

The sender is initially in the ready state, but it can move between the ready and blocking state. The variable S is initialized to 0.

❑ Ready state. When the sender is in this state, it is only waiting for one event to occur. If a request comes from the application layer, the sender creates a packet with the sequence number set to S. A copy of the packet is stored, and the packet is sent. The sender then starts the only timer. The sender then moves to the blocking state.

❑ Blocking state. When the sender is in this state, three events can occur:

a. If an error-free ACK arrives with the ackNo related to the next packet to be sent, which means ackNo = (S + 1) modulo 2, then the timer is stopped. The window slides, S = (S + 1) modulo 2. Finally, the sender moves to the ready state.

b. If a corrupted ACK or an error-free ACK with the ackNo ≠ (S + 1) modulo 2 arrives, the ACK is discarded.

c. If a time-out occurs, the sender resends the only outstanding packet and restarts the timer.

Receiver

The receiver is always in the ready state. Three events may occur:

a. If an error-free packet with seqNo = R arrives, the message in the packet is delivered to the application layer. The window then slides, R = (R + 1) modulo 2. Finally an ACK with ackNo = R is sent.

b. If an error-free packet with seqNo R arrives, the packet is discarded, but an ACK with ackNo = R is sent.

c. If a corrupted packet arrives, the packet is discarded.

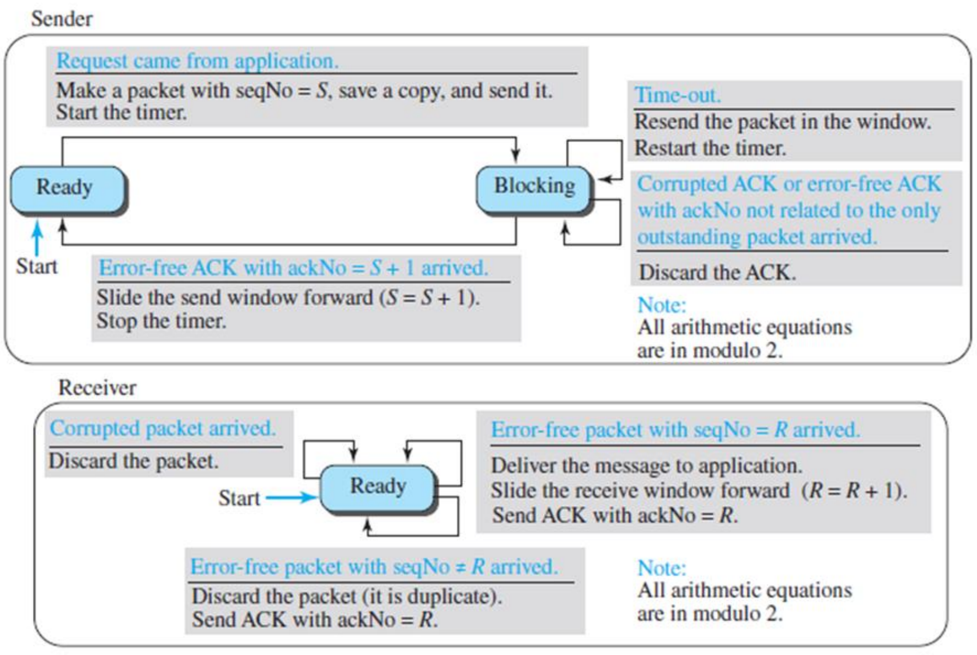

Sender
Request came from application.
Make a packet with seqNo = S, save a copy, and send it.
Start the timer.
Time-out.
Resend the packet in the window.
Restart the timer.
Ready
Blocking
Start
Corrupted ACK or error-free ACK with ackNo not related to the only outstanding packet arrived.
Discard the ACK.
Error-free ACK with ackNo = S + 1 arrived.
Slide the send window forward (S = S + 1).
Stop the timer.
Note: All arithmetic equations are in modulo 2.
Receiver
Corrupted packet arrived.
Discard the packet.
Ready
Start
Error-free packet with seqNo = R arrived.
Deliver the message to application.
Slide the receive window forward (R = R + 1).
Send ACK with ackNo = R.
Error-free packet with seqNo ≠ R arrived.
Discard the packet (it is duplicate).
Send ACK with ackNo = R.
Note: All arithmetic equations are in modulo 2.

Figure: FSM of Stop and Wait protocol

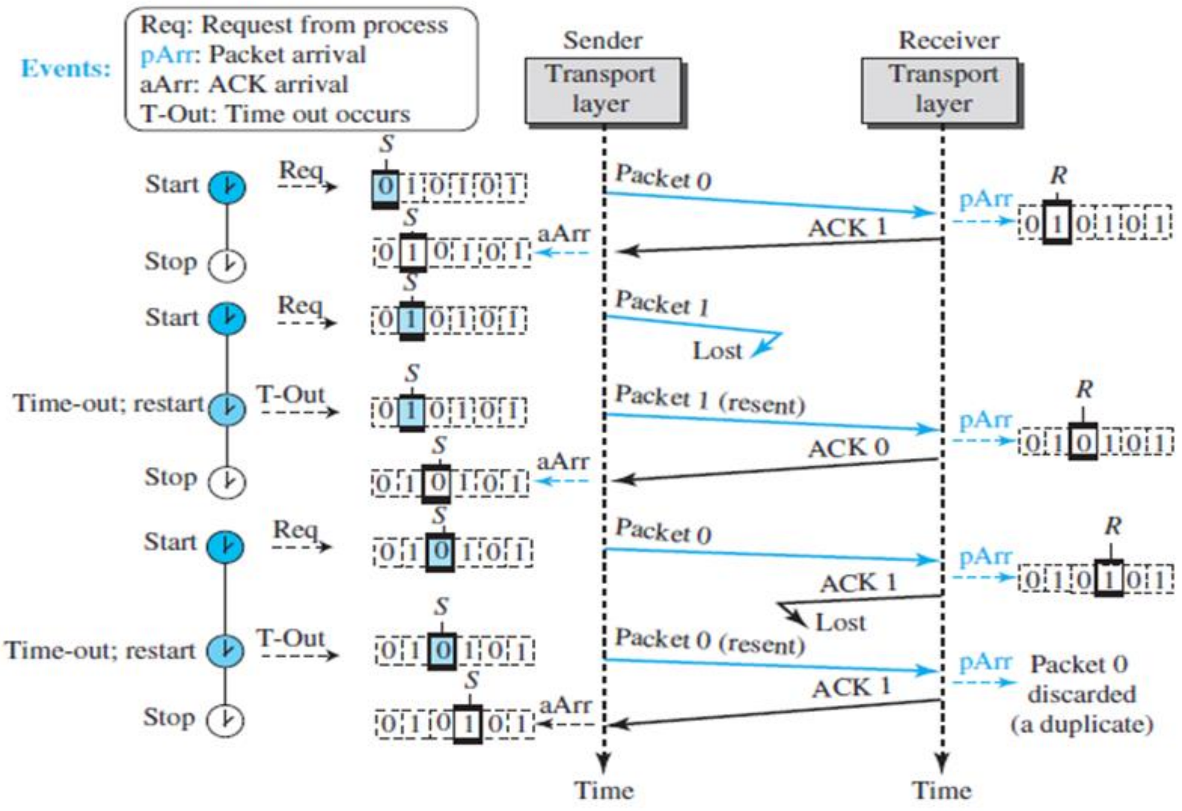


Figure: Flow diagram of Stop and Wait protocol

Go-Back N protocol

To improve the efficiency of transmission (to fill the pipe), multiple packets must be in transition while the sender is waiting for acknowledgment. In other words, we need to let more than one packet be outstanding to keep the channel busy while the sender is waiting for acknowledgment. One of the protocol is called ***Go-Back-N*** **(GBN)**. The key to Go-back-*N* is that we can send several packets before receiving acknowledgments, but the receiver can only buffer one packet. We keep a copy of the sent packets until the acknowledgments arrive. Figure 9 shows the outline of the protocol.

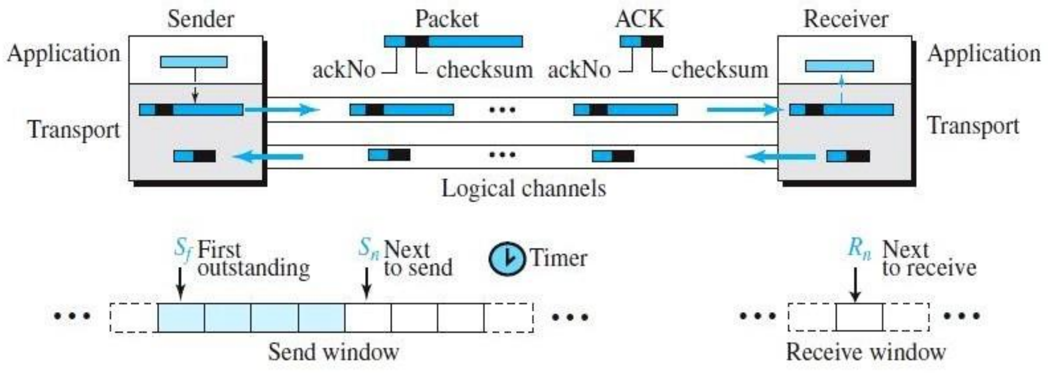


Figure 9: Go-Back-N protocol

Send Window

The send window is an imaginary box covering the sequence numbers of the data packets that can be in transit or can be sent. In each window position, some of the sequence numbers define the packets that have been sent; others define those that can be sent. The maximum size of the window is $2^m - 1$, we let the size be fixed and set to the maximum value, Figure 10 shows a sliding window of size 7 (m = 3) for the Go-Back-N protocol.

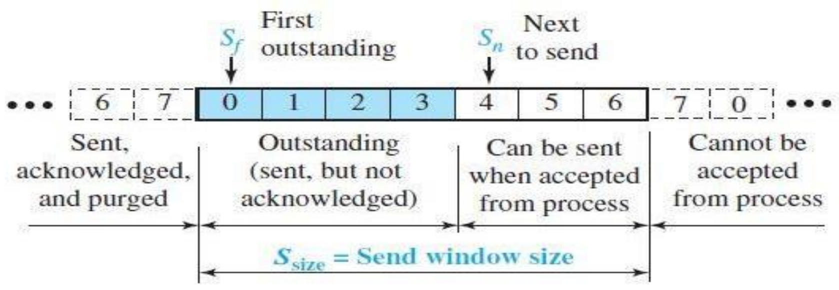

S_f First outstanding
S_n Next to send
6 7 0 1 2 3 4 5 6 7 0
Sent, acknowledged, and purged
Outstanding (sent, but not acknowledged)
Can be sent when accepted from process
Cannot be accepted from process
S_{size} = Send window size

Figure 10: Send window for Go-Back-N

The send window at any time divides the possible sequence numbers into four regions. The first region, left of the window, defines the sequence numbers belonging to packets that are already acknowledged. The sender does not worry about these packets and keeps no copies of them. The second region, colored, defines the range of sequence numbers belonging to the packets that have been sent, but have an unknown status. The sender needs to wait to find out if these packets have been received or were lost. These are called as *outstanding* packets. The third range, white in the figure, defines the range of sequence numbers for packets that can be sent; however, the corresponding data have not yet been received from the application layer. Finally, the fourth region, right of the window, defines sequence numbers that cannot be used until the window slides.

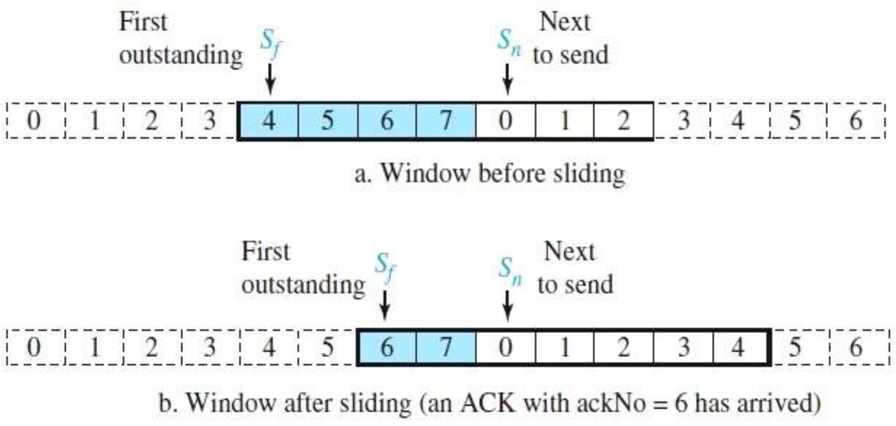


Figure 11: Sliding the send window

Figure 11 shows how a send window can slide one or more slots to the right when an acknowledgment arrives from the other end. In the figure, an acknowledgment with ack No = 6 has arrived. This means that the receiver is waiting for packets with sequence no 6.

Receive Window

The receive window makes sure that the correct data packets are received and that the correct acknowledgments are sent. In Go-Back-N, the size of the receive window is always 1. The receiver is always looking for the arrival of a specific packet. Any packet arriving out of order is discarded and needs to be resent. Figure 12 shows the receive window. It needs only one variable, Rn (receive window, next packet expected), to define this abstraction. The sequence numbers to the left of the window belong to the packets already received and acknowledged; the sequence numbers to the right of this window define the packets that cannot be received. Any received packet with a sequence number in these two regions is discarded. Only a packet with a sequence number matching the value of Rn is accepted and acknowledged.

The receive window also slides, but only one slot at a time. When a correct packet is received, the window slides, Rn = (Rn + 1) modulo 2^m.

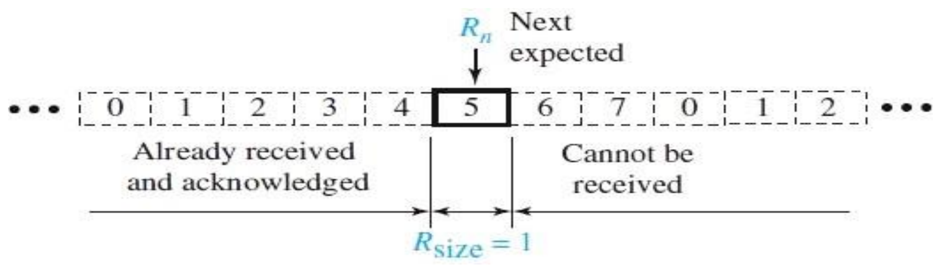


Figure 12: Receive window for Go-Back-N

FSMs

Figure 13 shows the FSMs for the GBN protocol.

Sender

The sender starts in the ready state, but thereafter it can be in one of the two states: ready or blocking. The two variables are normally initialized to 0 (Sf = Sn = 0).

- **Ready state.** Four events may occur when the sender is in ready state.

a. If a request comes from the application layer, the sender creates a packet with the sequence number set to Sn. A copy of the packet is stored, and the packet is sent. The sender also starts the only timer if it is not running. The value of Sn is now incremented, (Sn = Sn + 1) modulo 2^m. If the window is full, Sn = (Sf + Ssize) modulo 2^m, the sender goes to the blocking state.

b. If an error-free ACK arrives with ackNo related to one of the outstanding packets,the sender slides the window (set Sf = ackNo), and if all outstanding packets are acknowledged (ackNo = Sn), then the timer is stopped. If all outstanding packets are not acknowledged, the timer is restarted.

c. If a corrupted ACK or an error-free ACK with ack number not related to the outstanding packet arrives, it is discarded.

d. If a time-out occurs, the sender resends all outstanding packets and restarts the timer.

- **Blocking state.** Three events may occur in this case:

a. If an error-free ACK arrives with ackNo related to one of the outstanding packets, the sender slides the window (set Sf = ackNo) and if all outstanding packets are acknowledged (ackNo

= Sn), then the timer is stopped. If all outstanding packets are not acknowledged, the timer is restarted. The sender then moves to the ready state.

b. If a corrupted ACK or an error-free ACK with the ackNo not related to the outstanding packets arrives, the ACK is discarded.

c. If a time-out occurs, the sender sends all outstanding packets and restarts the timer.

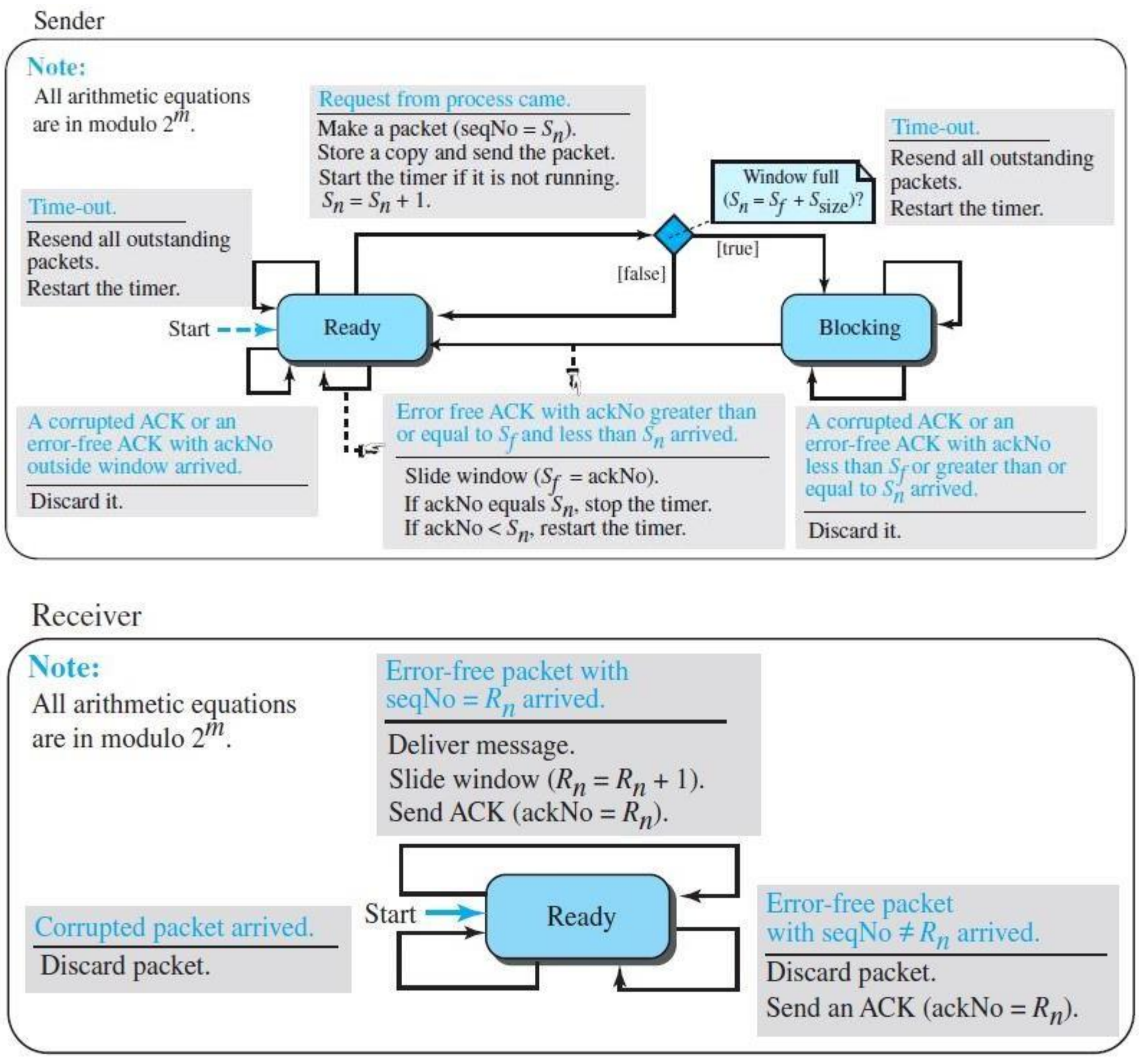

Sender
Note:
All arithmetic equations are in modulo 2^m.
Request from process came.
Make a packet (seqNo = S_n).
Store a copy and send the packet.
Start the timer if it is not running.
$S_n = S_n + 1$.
Time-out.
Resend all outstanding packets.
Restart the timer.
Window full ($S_n = S_f + S_{size}$)?
[false]
[true]
Time-out.
Resend all outstanding packets.
Restart the timer.
Start
Ready
Blocking
A corrupted ACK or an error-free ACK with ackNo outside window arrived.
Discard it.
Error free ACK with ackNo greater than or equal to S_f and less than S_n arrived.
Slide window (S_f = ackNo).
If ackNo equals S_n, stop the timer.
If ackNo < S_n, restart the timer.
A corrupted ACK or an error-free ACK with ackNo less than S_f or greater than or equal to S_n arrived.
Discard it.
Receiver
Note:
All arithmetic equations are in modulo 2^m.
Error-free packet with seqNo = R_n arrived.
Deliver message.
Slide window ($R_n = R_n + 1$).
Send ACK (ackNo = R_n).
Corrupted packet arrived.
Discard packet.
Start
Ready
Error-free packet with seqNo ≠ R_n arrived.
Discard packet.
Send an ACK (ackNo = R_n).

Figure 13 FSMs for the Go-Back-N protocol

Receiver

The receiver is always in the ready state. The only variable, Rn, is initialized to 0. Three events may occur:

a. If an error-free packet with seq No = Rn arrives, the message in the packet is delivered to the application layer. The window then slides, Rn = (Rn + 1) modulo 2m. Finally an ACK is sent with ack No = Rn.

b. If an error-free packet with seqNo outside the window arrives, the packet is discarded, but an ACK with ackNo = Rn is sent.

c. If a corrupted packet arrives, it is discarded.

Send Window Size

The size of the send window must be less than 2^m is because for example, choose m = 2, which means the size of the window can be 2^m - 1, or 3. Figure 14 compares a window size of 3 against a window size of 4. If the size of the window is 3 (less than 2^m) and all three acknowledgments are lost, the only timer expires and all three packets are resent. Thereceiver is now expecting packet 3, not packet 0, so the duplicate packet is correctly discarded. On the other hand, if the size of the window is 4 (equal to 22) and all acknowledgments are lost, the sender will send a duplicate of packet 0. However, this time the window of the receiver expects to receive packet 0 (in the next cycle), so it accepts packet 0, not as a duplicate, but as the first packet in the next cycle. This is an error. This shows that the size of the send window must be less than 2^m.

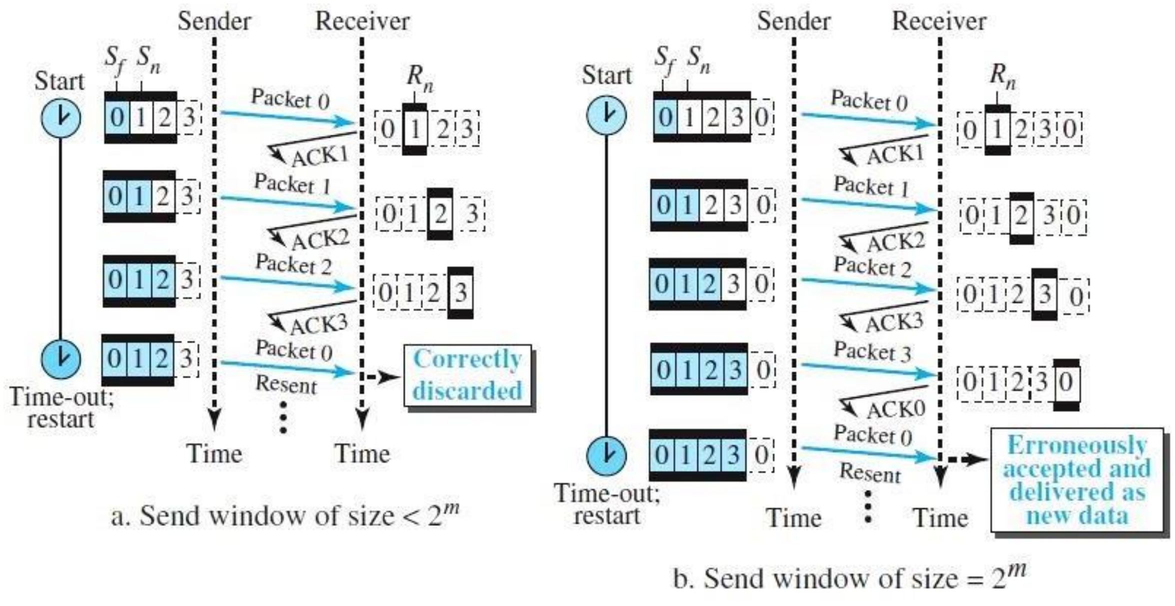


Figure 14 Send window size for Go-Back-N

Example 1

Figure 15 shows an example of Go-Back-*N*. This is an example of a case where the forward channel is reliable, but the reverse is not. No data packets are lost, but some ACKs are delayed and one is lost. The example also shows how cumulative acknowledgments can help if acknowledgments are delayed or lost.

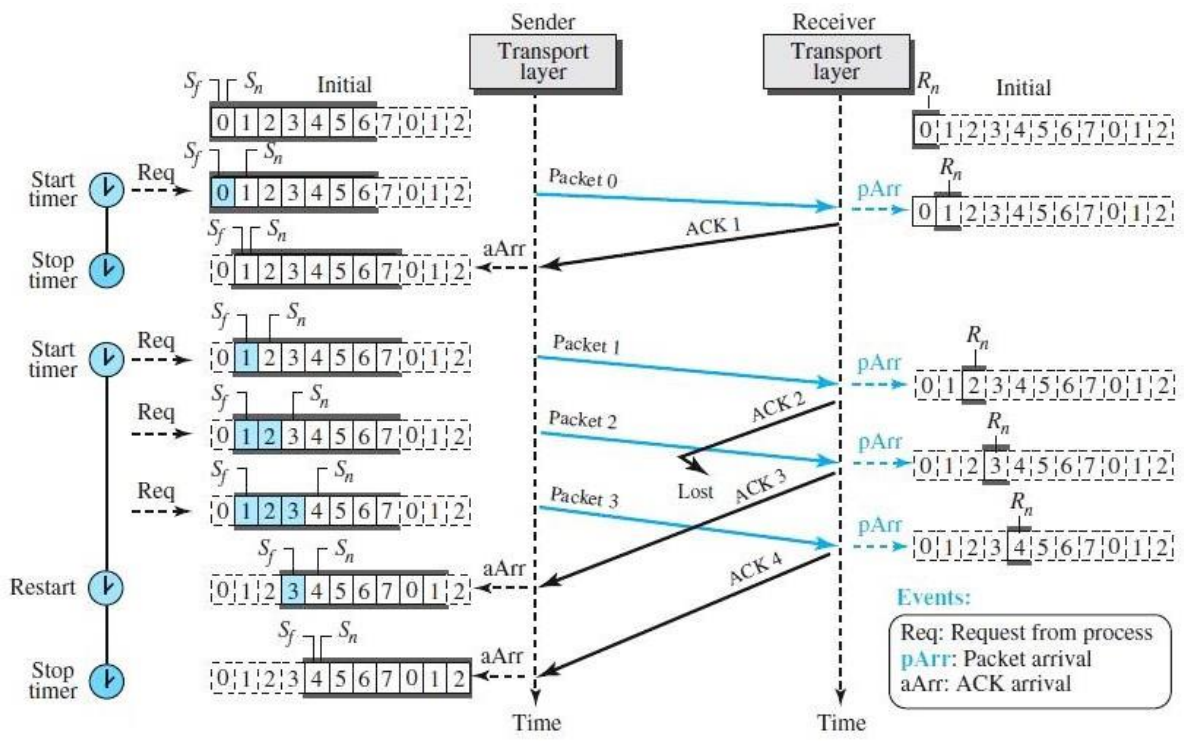


Figure15: Flow diagram for Example 1

After initialization, there are some sender events. Request events are triggered by message chunks from the application layer; arrival events are triggered by ACKs received from the network layer. There is no time-out event here because all outstanding packets are acknowledged before the timer expires. Although ACK 2 is lost, ACK 3 is cumulative and serves as both ACK 2 and ACK 3. There are four events at the receiver site.

Example 2

Figure 16 shows what happens when a packet is lost. Packets 0, 1, 2, and 3 are sent. However, packet 1 is lost. The receiver receives packets 2 and 3, but they are discarded because they are received out of order (packet 1 is expected). When the receiver receives packets 2 and 3, it sends ACK1 to show that it expects to receive packet 1. However, these ACKs are not useful for the sender because the ackNo is equal to *Sf* , not greater than *Sf*. So the sender discards them. When the time-out occurs, the sender resends packets 1, 2, and 3, which are acknowledged.

Go-Back-N versus Stop-and-Wait

The Stop-and-Wait protocol is actually a Go-Back-N protocol in which there are only two sequence numbers and the send window size is 1. In other words, m = 1 and $2^m - 1 = 1$. In Go-Back-N, we said that the arithmetic is modulo 2^m; in Stop-and-Wait it is modulo 2, which is the same as 2^m when m = 1.

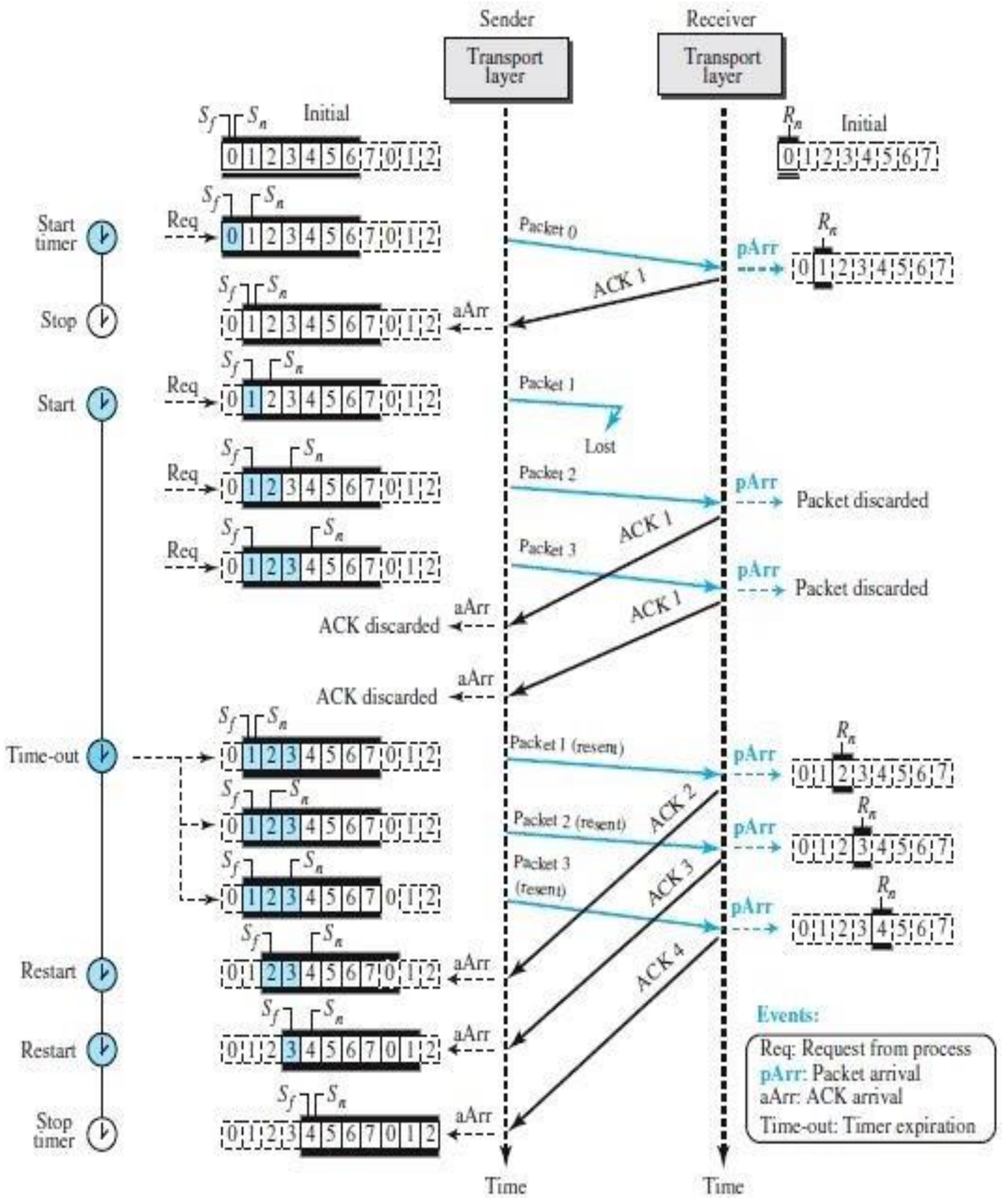


Figure 16: Flow diagram for Example 2

Selective-Repeat Protocol

The Go-Back-*N* protocol simplifies the process at the receiver. The receiver keeps track of only one variable, and there is no need to buffer out-of-order packets; they are simply discarded. However, this protocol is inefficient if the underlying network protocol loses a lot of packets. Each time a single packet is lost or corrupted, the sender resends all outstanding packets, even though some of these packets may have been received safe and sound but out of order. If the network layer is losing many packets because of congestion in the network, the resending of all of these outstanding packets makes the congestion worse, and eventually more packets are lost. This has an avalanche effect that may result in the total collapse of the network.

Another protocol, called the **Selective-Repeat (SR) protocol,** has been devised, which, as the name implies, resends only selective packets, those that are actually lost.
The outline of this protocol is shown in Figure 17.

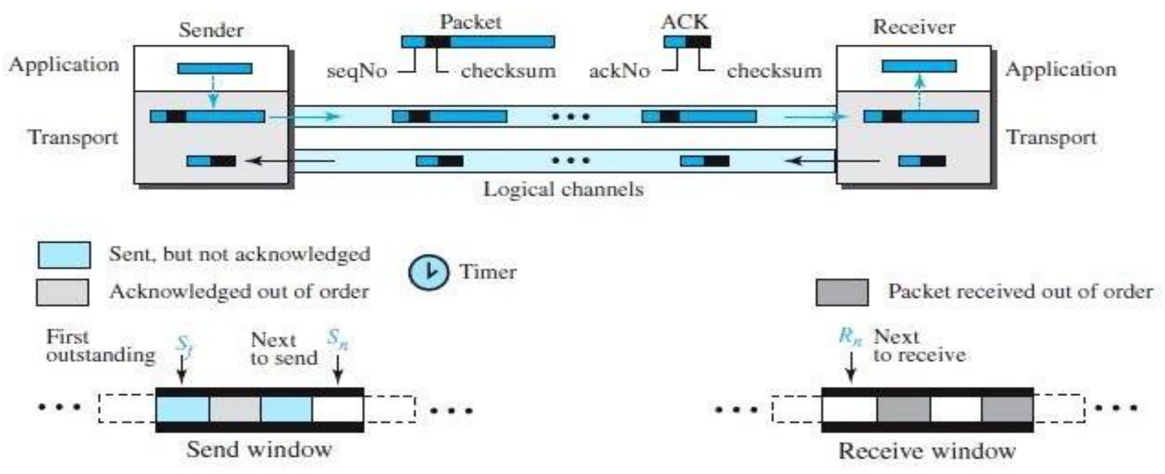


Figure 17: Outline of Selective-Repeat

Windows

The Selective-Repeat protocol also uses two windows: a send window and a receive window. However, there are differences between the windows in this protocol and the ones in Go-Back-N. First, the maximum size of the send window is much smaller; it is 2^m-1. The reason for this will be discussed later. Second, the receive window is the same size as the send window.

The send window maximum size can be 2^m-1. For example, if m = 4, the sequence numbers go from 0 to 15, but the maximum size of the window is just 8 (it is 15 in the Go-Back-N Protocol). The Selective-Repeat send window in Figure 18.1 to emphasize the size.

The receive window in Selective-Repeat is totally different from the one in Go-Back-N. The size of the receive window is the same as the size of the send window (max 2^m-1). The Selective-Repeat protocol allows as many packets as the size of the receive window to arrive out of order and be kept until there is a set of consecutive packets to be delivered to the application layer. Because the sizes of the send window and receive window are the same, all the packets in the send packet can arrive out of order and be stored until they can be delivered. To emphasize that in a reliable protocol the receiver never delivers packets out of order to the application layer.

Figure 18.2 shows the receive window in Selective-Repeat. Those slots inside the window that are shaded define packets that have arrived out of order and are waiting for the earlier transmitted packet to arrive before delivery to the application layer.

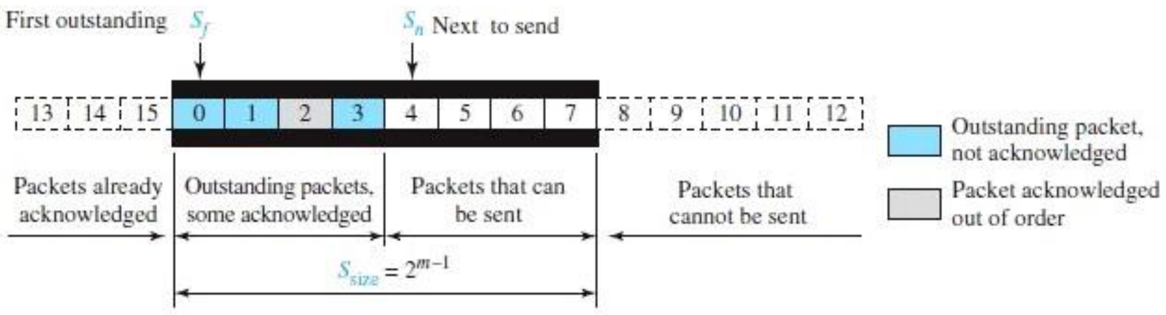

First outstanding S_f
S_n Next to send
13 14 15 0 1 2 3 4 5 6 7 8 9 10 11 12
Packets already acknowledged
Outstanding packets, some acknowledged
Packets that can be sent
Packets that cannot be sent
$S_{size} = 2^{m-1}$
Outstanding packet, not acknowledged
Packet acknowledged out of order

Figure 18.1: Send window for Selective-Repeat protocol

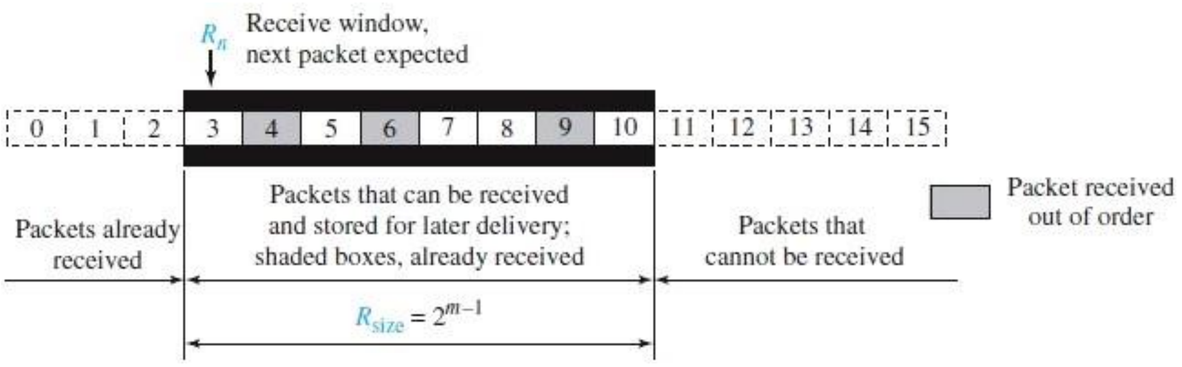

R_n Receive window, next packet expected
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Packets already received
Packets that can be received and stored for later delivery; shaded boxes, already received
Packets that cannot be received
$R_{size} = 2^{m-1}$
Packet received out of order

Figure 18.2: Receive window for Selective-Repeat protocol

Example 3

Assume a sender sends 6 packets: packets 0, 1, 2, 3, 4, and 5. The sender receives an ACK with ackNo = 3. What is the interpretation if the system is using GBN or SR?

Solution

If the system is using GBN, it means that packets 0, 1, and 2 have been received uncorrupted and the receiver is expecting packet 3. If the system is using SR, it means that packet 3 has been received uncorrupted; the ACK does not say anything about other packets.

FSMs

Figure 19 shows the FSMs for the Selective-Repeat protocol. It is similar to the ones for the GBN, but there are some differences.

Sender

The sender starts in the ready state, but later it can be in one of the two states: ready or blocking. The following shows the events and the corresponding actions in each state.

➢ **Ready state.** Four events may occur in this case:

a. If a request comes from the application layer, the sender creates a packet with the sequence number set to *Sn*. A copy of the packet is stored, and the packet is sent. If the timer is not running, the sender starts the timer. The value of *Sn* is now incremented, *Sn* = (*Sn* + 1) modulo 2*m*. If the window is full, *Sn* = (*Sf* + *S*size) modulo 2*m*, the sender goes to the blocking state.

b. If an error-free ACK arrives with ackNo related to one of the outstanding packets, that packet is marked as acknowledged. If the ackNo = *Sf*, the window slides to the right until the *Sf* points to the first unacknowledged packet (all consecutive acknowledged packets are now outside the window). If there are outstanding packets, the timer is restarted; otherwise, the timer is stopped.

c. If a corrupted ACK or an error-free ACK with ackNo not related to an outstanding packet arrives, it is discarded.

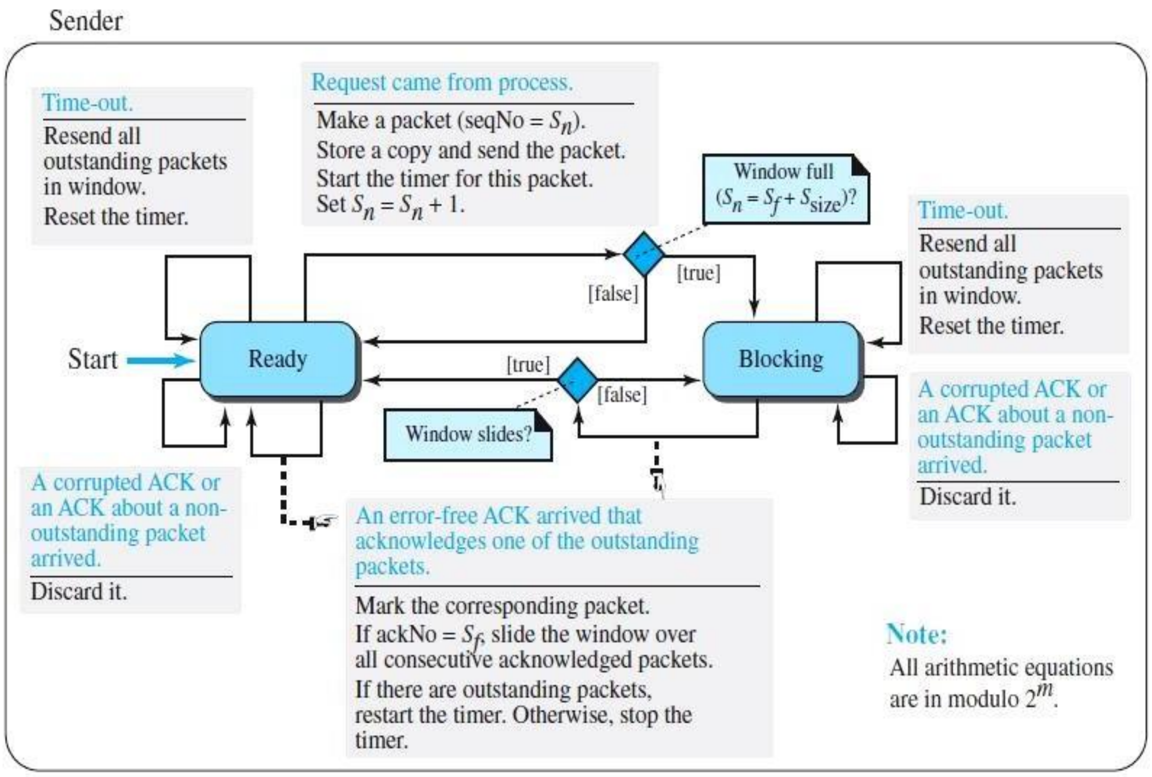
d. If a time-out occurs, the sender resends all unacknowledged packets in the window and restarts the timer.

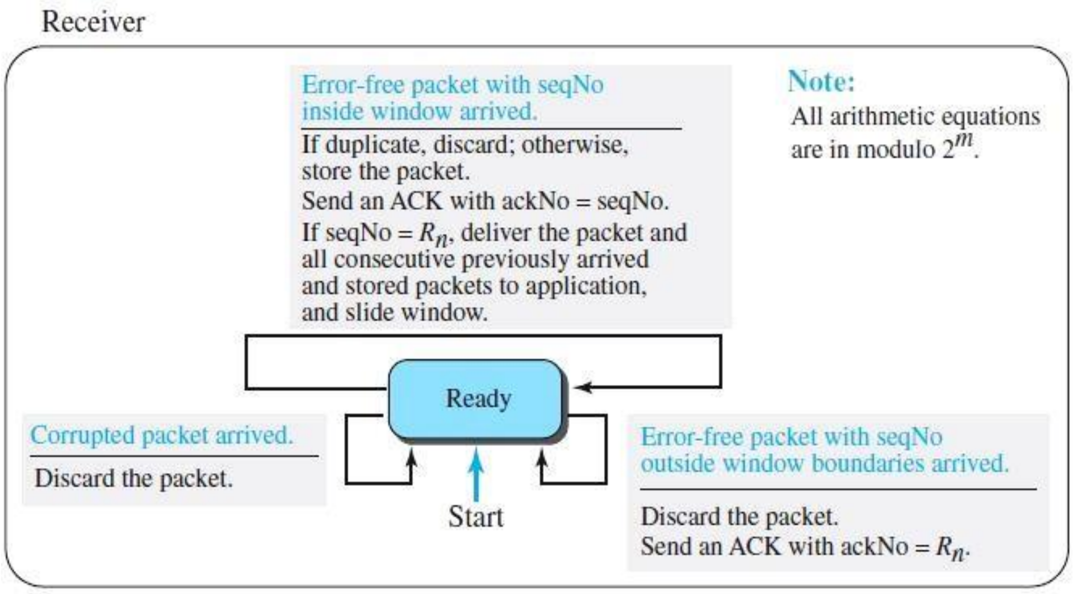
➢ **Blocking state.** Three events may occur in this case:

a. If an error-free ACK arrives with ackNo related to one of the outstanding packets, that packet is marked as acknowledged. In addition, if the ackNo = *Sf*, the window is slid to the right until the *Sf* points to the first unacknowledged packet (all consecutive acknowledged packets are now outside the window). If the window
has slid, the sender moves to the ready state.

b. If a corrupted ACK or an error-free ACK with the ackNo not related to outstanding packets arrives, the ACK is discarded.

c. If a time-out occurs, the sender resends all unacknowledged packets in the window and restarts the timer.


Sender
Time-out.
Resend all outstanding packets in window.
Reset the timer.
Request came from process.
Make a packet (seqNo = S_n).
Store a copy and send the packet.
Start the timer for this packet.
Set $S_n = S_n + 1$.
Window full ($S_n = S_f + S_{size}$)?
[false]
[true]
Time-out.
Resend all outstanding packets in window.
Reset the timer.
Start
Ready
Blocking
[true]
[false]
Window slides?
A corrupted ACK or an ACK about a non-outstanding packet arrived.
Discard it.
An error-free ACK arrived that acknowledges one of the outstanding packets.
Mark the corresponding packet.
If ackNo = S_f, slide the window over all consecutive acknowledged packets.
If there are outstanding packets, restart the timer. Otherwise, stop the timer.
A corrupted ACK or an ACK about a non-outstanding packet arrived.
Discard it.
Note:
All arithmetic equations are in modulo 2^m.


Receiver
Error-free packet with seqNo inside window arrived.
If duplicate, discard; otherwise, store the packet.
Send an ACK with ackNo = seqNo.
If seqNo = R_n, deliver the packet and all consecutive previously arrived and stored packets to application, and slide window.
Note:
All arithmetic equations are in modulo 2^m.
Ready
Corrupted packet arrived.
Discard the packet.
Start
Error-free packet with seqNo outside window boundaries arrived.
Discard the packet.
Send an ACK with ackNo = R_n.

Receiver

The receiver is always in the ready state. Three events may occur:

a. If an error-free packet with seqNo in the window arrives, the packet is stored and an ACK with ackNo = seqNo is sent. In addition, if the seqNo = *Rn*, then the packet and all previously arrived consecutive packets are delivered to the application layer and the window slides so that the *Rn* points to the first empty slot.

b. If an error-free packet with seqNo outside the window arrives, the packet is discarded, but an ACK with ackNo = *Rn* is returned to the sender. This is needed to let the sender slide its window if some ACKs related to packets with seqNo < *Rn* were lost.

c. If a corrupted packet arrives, the packet is discarded.

Example 4

This example is similar to Example 2 (Figure 16) in which packet 1 is lost. Selective-Repeat behaviour is shown in this case. Figure 19 shows the situation.

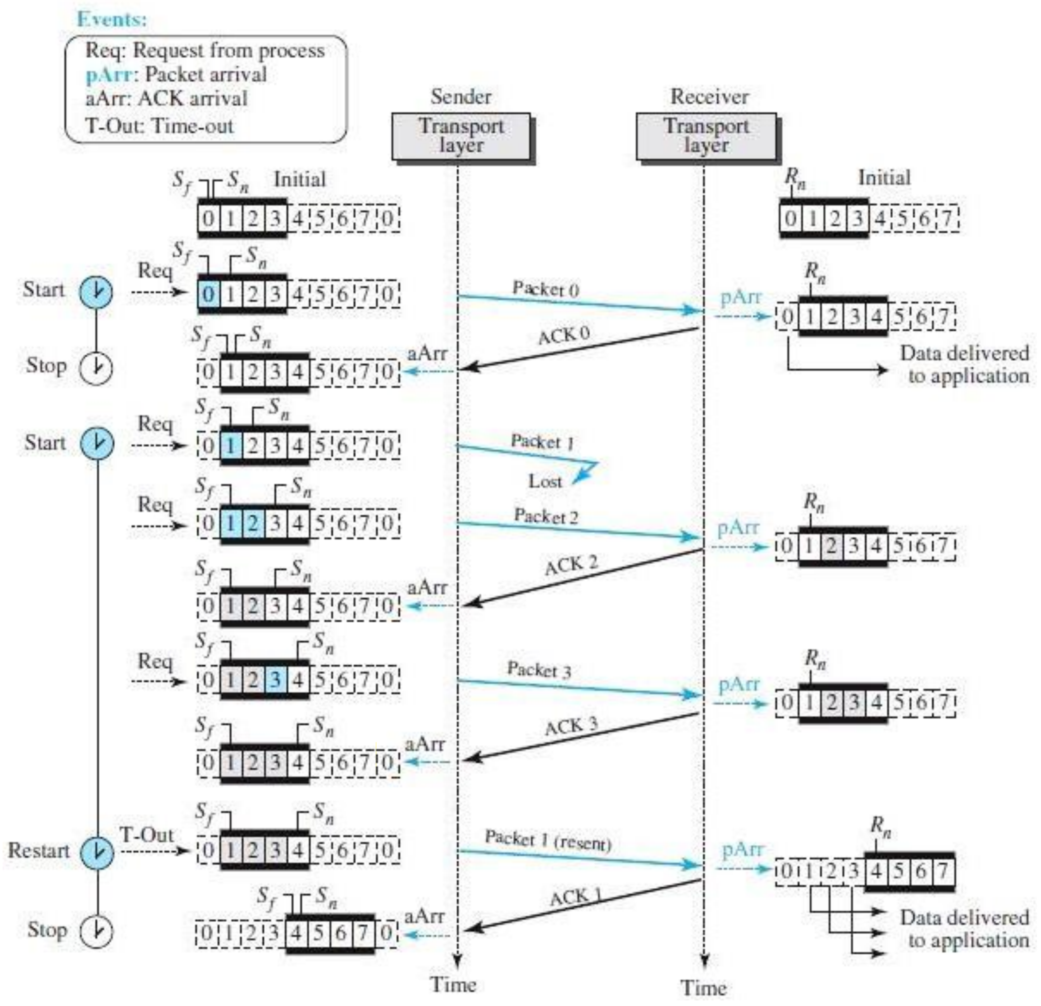


Figure 19: Flow diagram for Example 4

At the sender, packet 0 is transmitted and acknowledged. Packet 1 is lost. Packets 2 and 3 arrive out of order and are acknowledged. When the timer times out, packet 1 (the only unacknowledged packet) is resent and is acknowledged. The send window then slides.

At the receiver site we need to distinguish between the acceptance of a packet and its delivery to the application layer. At the second arrival, packet 2 arrives and is stored and marked (shaded slot), but it cannot be delivered because packet 1 is missing. At the next arrival, packet 3 arrives and is marked and stored, but still none of the packets can be delivered. Only at the last arrival, when finally a copy of packet 1 arrives, can packets 1, 2, and 3 be delivered to the application layer. There are two conditions for the delivery of packets to the application layer: First, a set of consecutive packets must have arrived. Second, the set starts from the beginning of the window. After the first arrival, there was only one packet and it started from the beginning of the window. After the last arrival, there are three packets and the first one starts from the beginning of the window. The key is that a reliable transport layer promises to deliver packets in order.

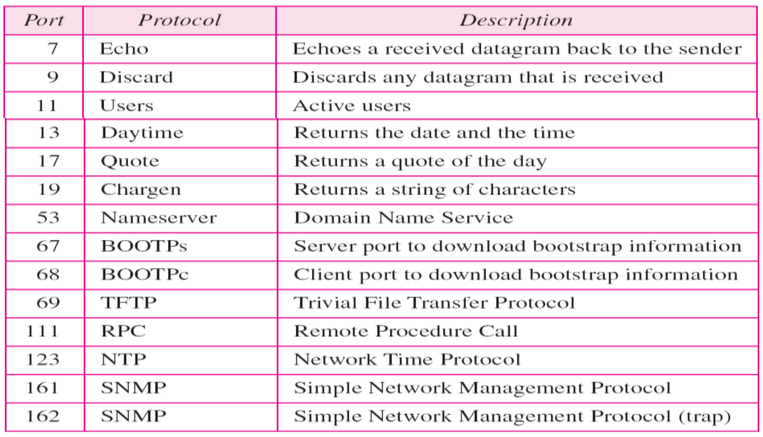
Window Sizes

We can now show why the size of the sender and receiver windows can be at most one-half of 2^m. For an example, we choose $m = 2$, which means the size of the window is $2^m/2$ or $2^{(m-1)} = 2$. Figure 23.36 compares a window size of 2 with a window size of 3. If the size of the window is 2 and all acknowledgments are lost, the timer for packet 0 expires and packet 0 is resent. However, the window of the receiver is now expecting packet 2, not packet 0, so this duplicate packet is correctly discarded (the sequence number 0 is not in the window). When the size of the window is 3 and all acknowledgments are lost, the sender sends a duplicate of packet 0. However, this time, the window of the receiver expects to receive packet 0 (0 is part of the window), so it accepts packet 0, not as a duplicate, but as a packet in the next cycle. This is clearly an error.

USER DATAGRAM PROTOCOL (UDP)

- The User Datagram Protocol (UDP) is called a connectionless, unreliable transport protocol. It does not add anything to the services of IP except to provide process-to-process communication instead of host-to-host communication.
- If a process wants to send a small message and does not care much about reliability, it can use UDP.

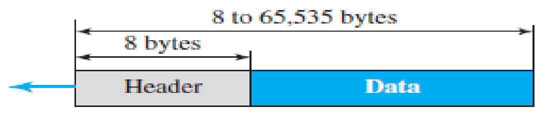
Table : *Well-known ports used with UDP*

- 

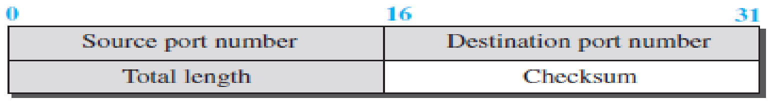
Port	Protocol	Description
7	Echo	Echoes a received datagram back to the sender
9	Discard	Discards any datagram that is received
11	Users	Active users
13	Daytime	Returns the date and the time
17	Quote	Returns a quote of the day
19	Chargen	Returns a string of characters
53	Nameserver	Domain Name Service
67	BOOTPs	Server port to download bootstrap information
68	BOOTPc	Client port to download bootstrap information
69	TFTP	Trivial File Transfer Protocol
111	RPC	Remote Procedure Call
123	NTP	Network Time Protocol
161	SNMP	Simple Network Management Protocol
162	SNMP	Simple Network Management Protocol (trap)

UDP packets, called user datagrams, have a fixed-size header of 8 bytes made of four fields, each of 2 bytes (16 bits). The first two fields define the source and destination port numbers. The third field defines the total length of the user datagram, header plus data. The 16 bits can define a total length of 0 to 65,535 bytes. However, the total length needs to be less because a UDP user datagram is stored in an IP datagram with the total length of 65,535 bytes. The last field can carry the optional checksum

User datagram packet format



a. UDP user datagram



b. Header format

Example: The following is the content of a UDP header in hexadecimal format. CB84000D001C001C

a. What is the source port number?

b. What is the destination port number?

c. What is the total length of the user datagram?

d. What is the length of the data?

e. Is the packet directed from a client to a server or vice versa?

f. What is the client process?

Solution:a. The source port number is the first four hexadecimal digits (CB84)16, which means that the source port number is 52100.

b. The destination port number is the second four hexadecimal digits (000D)16, which means that the destination port number is 13.

c. The third four hexadecimal digits (001C)16 define the length of the whole UDP packet as 28 bytes.

d. The length of the data is the length of the whole packet minus the length of the header, or 28 − 8 = 20 bytes.

e. Since the destination port number is 13 (well-known port), the packet is from the client to the server.

f. The client process is the Daytime

UDP Services

Process-to-Process Communication

UDP provides process-to-process communication using **socket addresses,** a combination of IP addresses and port numbers.

Connectionless Services

UDP provides a connectionless service. This means that each user datagram sent by UDP is an independent datagram. There is no relationship between the different user datagrams even if they are coming from the same source process and going to the same destination program. The user datagrams are not numbered. Also, unlike TCP, there is no connection establishment and no connection termination. This means that each user datagram can travel on a different path. One of the ramifications of being connectionless is that the process that uses UDP cannot send a stream of data to UDP and expect UDP to chop them into different, related user datagrams. Instead each request must be small enough to fit into one user datagram. Only those processes sending short messages, messages less than 65,507 bytes (65,535 minus 8 bytes for the UDP header and minus 20 bytes for the IP header), can use UDP.

Flow Control

UDP is a very simple protocol. There is no flow control, and hence no window mechanism. The receiver may overflow with incoming messages. The lack of flow control means that the process using UDP should provide for this service, if needed.

Error Control

There is no error control mechanism in UDP except for the checksum. This means that the sender does not know if a message has been lost or duplicated. When the receiver detects an error through the checksum, the user datagram is silently discarded. The lack of error control means that the process using UDP should provide for this service, if needed.

Checksum

UDP checksum calculation includes three sections: a pseudoheader, the UDP header, and the data coming from the application layer. The pseudoheader is the part of the header of the IP packet in which the user datagram is to be encapsulated with some fields filled with 0s (see Figure 20).

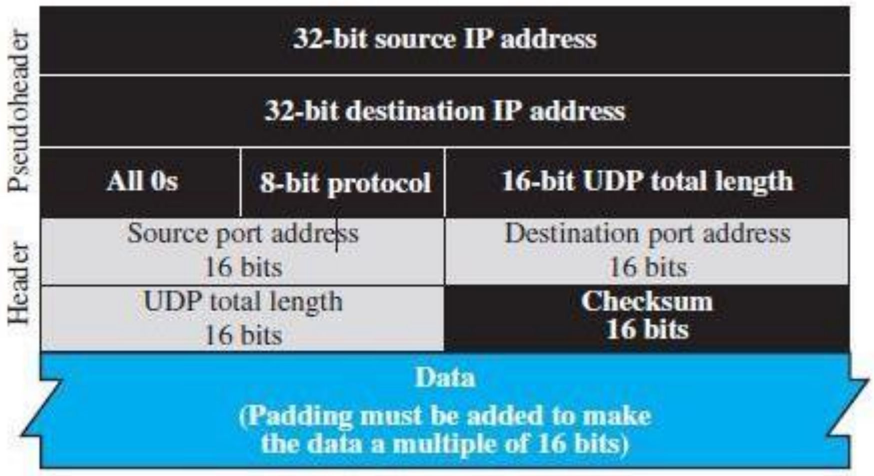


Figure 20: Pseudoheader for checksum calculation

If the checksum does not include the pseudoheader, a user datagram may arrive safe and sound. However, if the IP header is corrupted, it may be delivered to the wrong host. The protocol field is added to ensure that the packet belongs to UDP, and not to TCP. We will see later that if a process can use either UDP or TCP, the destination port number can be the same. The value of the protocol field for UDP is 17. If this value is changed during transmission, the checksum calculation at the receiver will detect it and UDP drops the packet. It is not delivered to the wrong protocol.

Example : What value is sent for the checksum in each one of the following hypothetical situations? **a.** The sender decides not to include the checksum. **b.** The sender decides to include the checksum, but the value of the sum is all 1s. **c.** The sender decides to include the checksum, but the value of the sum is all 0s.

Solution: a. The value sent for the checksum field is all 0s to show that the checksum is not calculated.

b. When the sender complements the sum, the result is all 0s; the sender complements the result again before sending. The value sent for the checksum is all 1s.

c. This situation never happens because it implies that the value of every term included in the calculation of the sum is all 0s, which is impossible; some fields in the pseudoheader have nonzero values.

Congestion Control

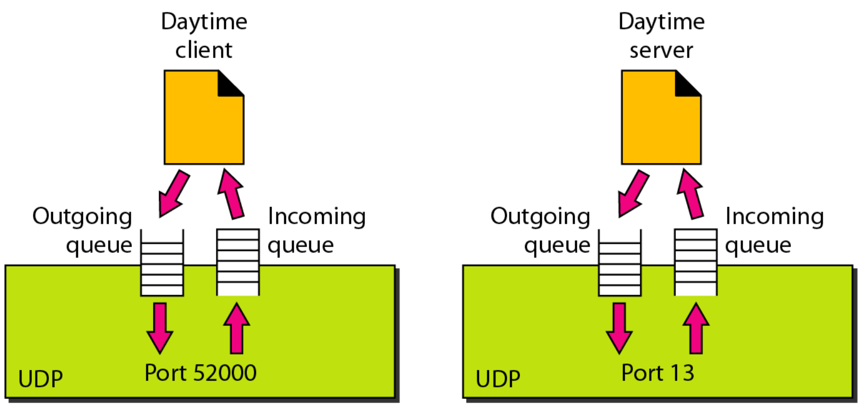
Since UDP is a connectionless protocol, it does not provide congestion control. UDP assumes that the packets sent are small and sporadic and cannot create congestion in the network. This assumption may or may not be true today, when UDP is used for interactive real-time transfer of audio and video.

Encapsulation and Decapsulation

To send a message from one process to another, the UDP protocol encapsulates and decapsulates messages.

Queuing

In UDP, queues are associated with ports. At the client site, when a process starts, it requests a port number from the operating system. Some implementations create both an incoming and an outgoing queue. associated with each process. Other implementations create only an incoming queue associated with each process.



Multiplexing and Demultiplexing

In a host running a TCP/IP protocol suite, there is only one UDP but possibly several processes that may want to use the services of UDP. To handle this situation, UDP multiplexes and demultiplexes.

Typical UDP Applications

The following shows some typical applications that can benefit more from the services of UDP than from those of TCP.

- ❑ UDP is suitable for a process that requires simple request-response communication with little concern for flow and error control. It is not usually used for a process such as FTP that needs to send bulk data.
- ❑ UDP is suitable for a process with internal flow- and error-control mechanisms. For example, the Trivial File Transfer Protocol (TFTP) process includes flow and error control. It can easily use UDP.

- ❑ UDP is a suitable transport protocol for multicasting. Multicasting capability is embedded in the UDP software but not in the TCP software.
- ❑ UDP is used for management processes such as SNMP.
- ❑ UDP is used for some route updating protocols such as Routing Information Protocol (RIP).
- ❑ UDP is normally used for interactive real-time applications that cannot tolerate uneven delay between sections of a received message.

TRANSMISSION CONTROL PROTOCOL(TCP)

TCP is a connection-oriented protocol; it creates a virtual connection between two TCPs to send data. In addition, TCP uses flow and error control mechanisms at the transport level. i,e connection-oriented-reliable TCP.

- **TCP** is a connection-oriented, reliable protocol.
- TCP explicitly defines connection establishment, data transfer, and connection teardown phases to provide a connection-oriented service.
- TCP uses a combination of GBN and SR protocols to provide reliability.
- To achieve reliability, TCP uses checksum (for error detection), retransmission of lost or corrupted packets, cumulative and selective acknowledgments, and timers.

Table : *Well-known ports used by TCP*

Port	*Protocol*	*Description*
7	Echo	Echoes a received datagram back to the sender
9	Discard	Discards any datagram that is received
11	Users	Active users
13	Daytime	Returns the date and the time
17	Quote	Returns a quote of the day
19	Chargen	Returns a string of characters
20	FTP, Data	File Transfer Protocol (data connection)
21	FTP, Control	File Transfer Protocol (control connection)
23	TELNET	Terminal Network
25	SMTP	Simple Mail Transfer Protocol
53	DNS	Domain Name Server
67	BOOTP	Bootstrap Protocol
79	Finger	Finger
80	HTTP	Hypertext Transfer Protocol
111	RPC	Remote Procedure Call

TCP services and features

- Process-to-Process Communication
- Stream Delivery Service
- Full-Duplex Communication
- Multiplexing and Demultiplexing
- Connection-Oriented Service
- Reliable Service

- **Process-to-Process Communication**

TCP provides process-to-process communication using port numbers.

- **Stream Delivery Service**

TCP, unlike UDP, is a stream-oriented protocol. In UDP, a process sends messages with predefined boundaries to UDP for delivery. UDP adds its own header to each of these messages and delivers it to IP for transmission. Each message from the process is called a user datagram, and becomes, eventually, one IP datagram. Neither IP nor UDP recognizes any relationship between the datagrams.

TCP, on the other hand, allows the sending process to deliver data as a stream of bytes and allows the receiving process to obtain data as a stream of bytes. TCP creates an environment in which the two processes seem to be connected by an imaginary "tube" that carries their bytes across the Internet. This imaginary environment is depicted in Figure 21. The sending process produces (writes to) the stream and the receiving process consumes (reads from) it.

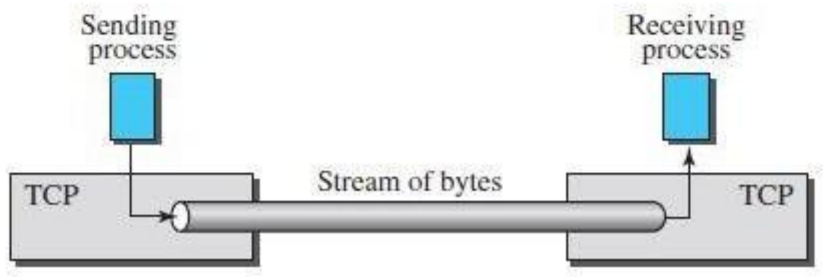


Figure 21: Stream delivery

- **Sending and Receiving Buffers**

One way to implement a buffer is to use a circular array of 1-byte locations as shown in Figure 22. For simplicity, it is shown as two buffers of 20 bytes each; normally the buffers are hundreds or thousands of bytes, depending on the implementation. We also show the buffers as the same size, which is not always the case. The figure shows the movement of the data in one direction. At the sender, the buffer has three types of chambers. The white section contains empty chambers that can be filled by the sending process (producer). The colored area holds bytes that have been sent but not yet acknowledged. The TCP sender keeps these bytes in the buffer until it receives an acknowledgment. The shaded area contains bytes to be sent by the sending TCP. However, as we will see later in this chapter, TCP may be able to send only part of this shaded section. This could be due to the slowness of the receiving process or to congestion in the network. Also note that, after the bytes in the colored chambers are acknowledged, the chambers are recycled and available for use by the sending process.

The operation of the buffer at the receiver is simpler. The circular buffer is divided into two areas (shown as white and colored). The white area contains empty chambers to be filled by bytes received from the network. The colored sections contain received bytes that can be read by the receiving process. When a byte is read by the receiving process, the chamber is recycled and added to the pool of empty chambers.

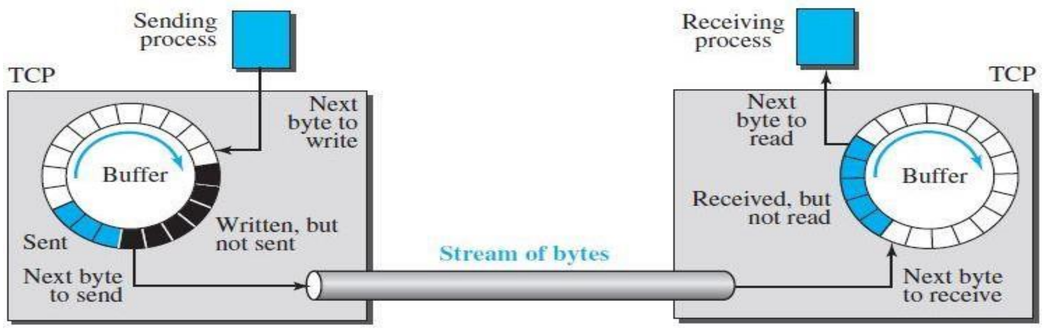


Figure 22: Sending and receiving buffers

➢ **Segments**

Although buffering handles the disparity between the speed of the producing and consuming processes, we need one more step before we can send data. The network layer, as a service provider for TCP, needs to send data in packets, not as a stream of bytes. At the transport layer, TCP groups a number of bytes together into a packet called a segment.

TCP adds a header to each segment (for control purposes) and delivers the segment to the network layer for transmission. The segments are encapsulated in an IP datagram and transmitted. This entire operation is transparent to the receiving process. Segments may be received out of order, lost or corrupted, and resent. All of these are handled by the TCP receiver with the receiving application process unaware of TCP’s activities. Figure 23 shows how segments are created from the bytes in the buffers.

Segments are not necessarily all the same size. In the figure, for simplicity, it is shown one segment carrying 3 bytes and the other carrying 5 bytes. In reality, segments carry hundreds, if not thousands, of bytes.

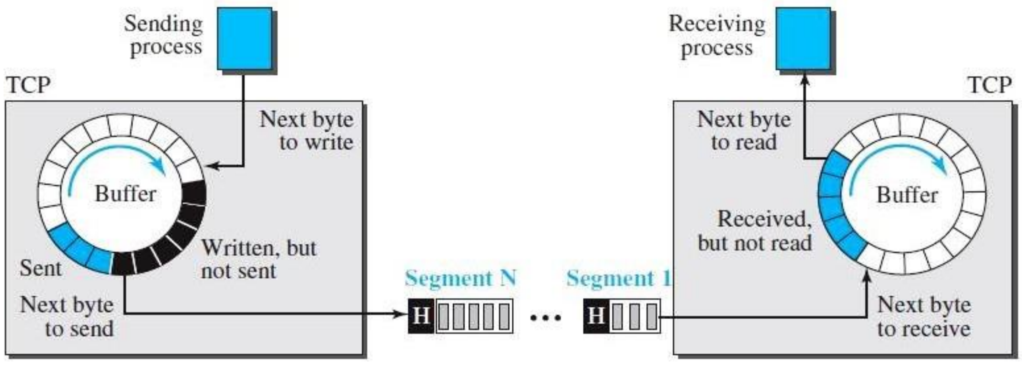


Figure 23: TCP segments

- **Full-Duplex Communication**

TCP offers full-duplex service, where data can flow in both directions at the same time. Each TCP endpoint then has its own sending and receiving buffer, and segments move in both directions.

- **Multiplexing and Demultiplexing**

Like UDP, TCP performs multiplexing at the sender and demultiplexing at the receiver. However, since TCP is a connection-oriented protocol, a connection needs to be established for each pair of processes.

- **Connection-Oriented Service**

TCP, unlike UDP, is a connection-oriented protocol. When a process at site A wants to send to and receive data from another process at site B, the following three phases occur:

1. The two TCP's establish a logical connection between them.

2. Data are exchanged in both directions.

3. The connection is terminated.

This is a logical connection, not a physical connection. The TCP segment is encapsulated in an IP datagram and can be sent out of order, or lost or corrupted, and then resent. Each may be routed over a different path to reach the destination. There is no physical connection. TCP creates a stream-oriented environment in which it accepts the responsibility of delivering the bytes in order to the other site.

- **Reliable Service**

TCP is a reliable transport protocol. It uses an acknowledgment mechanism to check the safe and sound arrival of data.

TCP Features

Numbering System

Although the TCP software keeps track of the segments being transmitted or received, there is no field for a segment number value in the segment header. Instead, there are two fields, called the sequence number and the acknowledgment number. These two fields refer to a byte number and not a segment number.

Byte Number

TCP numbers all data bytes (octets) that are transmitted in a connection. Numbering is independent in each direction. When TCP receives bytes of data from a process, TCP stores them in the sending buffer and numbers them. The numbering does not necessarily start from 0. Instead, TCP chooses an arbitrary number between 0 and 2^{32} - 1 for the number of the first byte. For example, if the number happens to be 1057 and the total data to be sent is 6000 bytes, the bytes are numbered from 1057 to 7056. We will see that byte numbering is used for flow and error control.

Sequence Number

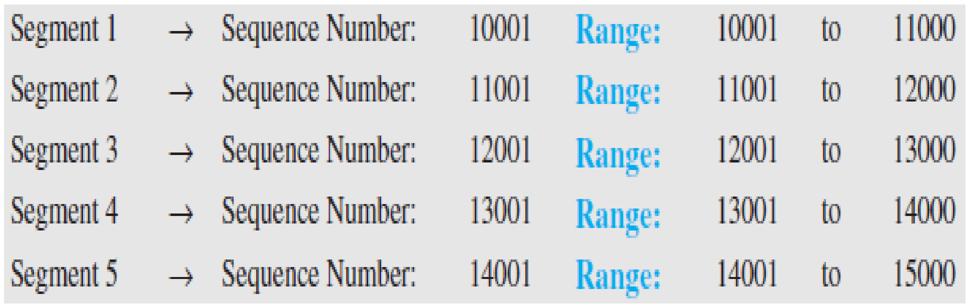
After the bytes have been numbered, TCP assigns a sequence number to each segment that is being sent. The sequence number, in each direction, is defined as follows:

1. The sequence number of the first segment is the ISN (initial sequence number), which is a random number.
2. The sequence number of any other segment is the sequence number of the previous segment plus the number of bytes (real or imaginary) carried by the previous segment.

Acknowledgment Number

Communication in TCP is full duplex; when a connection is established, both parties can send and receive data at the same time. Each party numbers the bytes, usually with a different starting byte number. The sequence number in each direction shows the number of the first byte carried by the segment. Each party also uses an acknowledgment number to confirm the bytes it has received. However, the acknowledgment number defines the number of the next byte that the party expects to receive. In addition, the acknowledgment number is cumulative, which means that the party takes the number of the last byte that it has received, safe and sound, adds 1 to it, and announces this sum as the acknowledgment number. The term *cumulative* here means that if a party uses 5643 as an acknowledgment number, it has received all bytes from the beginning up to 5642. Note that this does not mean that the party has received 5642 bytes, because the first byte number does not have to be 0.

EXAMPLE: Suppose a TCP connection is transferring a file of 5000 bytes. The first byte is numbered 10,001. What are the sequence numbers for each segment if data are sent in five segments, each carrying 1000 bytes?



Segment 1	→	Sequence Number:	10001	Range:	10001	to	11000
Segment 2	→	Sequence Number:	11001	Range:	11001	to	12000
Segment 3	→	Sequence Number:	12001	Range:	12001	to	13000
Segment 4	→	Sequence Number:	13001	Range:	13001	to	14000
Segment 5	→	Sequence Number:	14001	Range:	14001	to	15000

TCP segment format Segment

A packet in TCP is called a segment.

Format

The format of a segment is shown in Figure 23.1. 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.

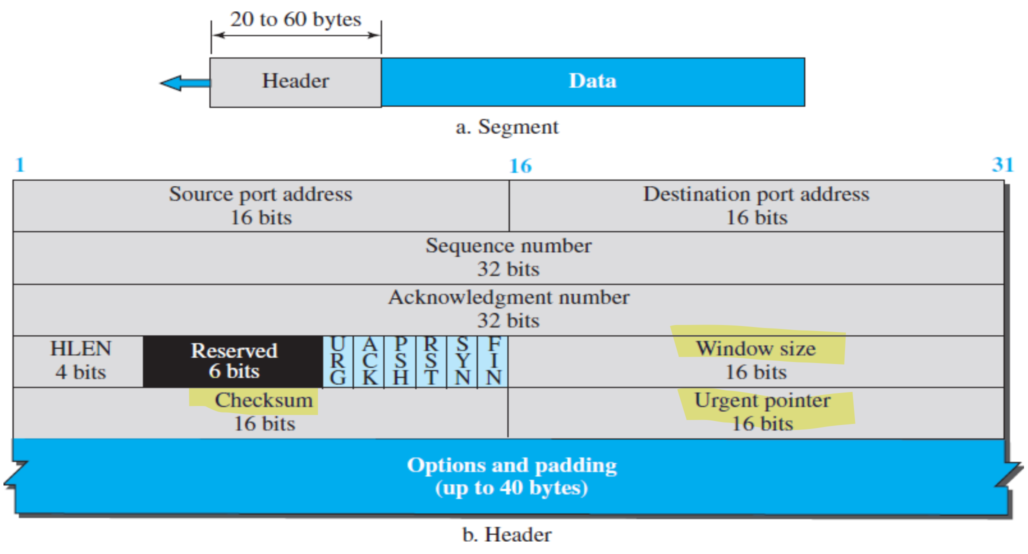


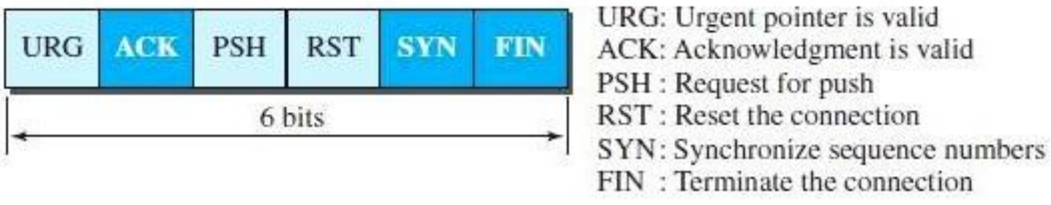
Figure 23.1: TCP segment format

❑ 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.

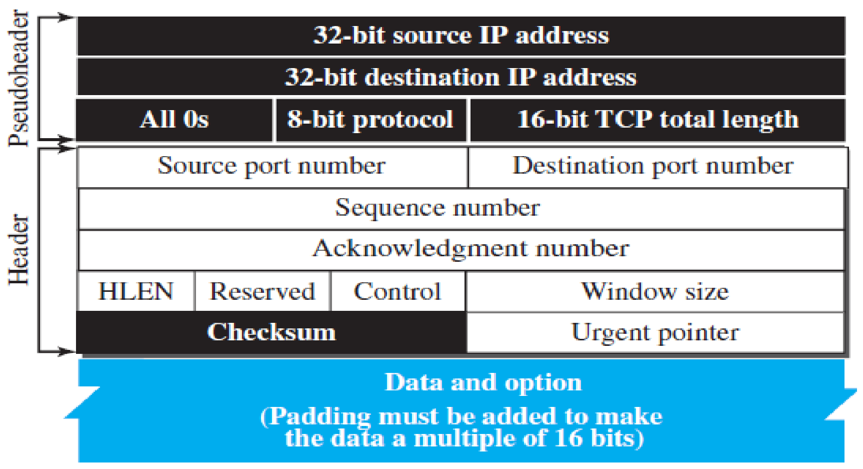
❑ Sequence number. This 32-bit field defines the number assigned to the first byte of data contained in this segment. As we said before, TCP is a stream transport protocol. To ensure connectivity, each byte to be transmitted is numbered. The sequence number tells the destination which byte in this sequence is the first byte in the segment. During connection establishment, each party uses a random number generator to create an initial sequence number (ISN), which is usually different in each direction.

- 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).
- Control. This field defines 6 different control bits or flags, as shown in Figure 24.8. 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. A brief description of each bit is shown in the figure below.


URG
ACK
PSH
RST
SYN
FIN
6 bits
URG: Urgent pointer is valid
ACK: Acknowledgment is valid
PSH : Request for push
RST : Reset the connection
SYN: Synchronize sequence numbers
FIN : Terminate the connection

- **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. The sender must obey the dictation of the receiver in this case.
- **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. However, the use of the checksum in the UDP datagram is optional, whereas the use of the checksum for TCP is mandatory.
- **Urgent pointer.** This 16-bit field, which is valid only if the urgent flag is set, is used when the segment contains urgent data. It defines a value that must be added to the sequence number to obtain the number of the last urgent byte in the data section of the segment.
- **Options.** There can be up to 40 bytes of optional information in the TCP header.

Pseudoheader added to the TCP datagram



Encapsulation

A TCP segment encapsulates the data received from the application layer. The TCP segment is encapsulated in an IP datagram, which in turn is encapsulated in a frame at the data-link layer.

A TCP Connection

In TCP, connection-oriented transmission requires three phases: connection establishment, data transfer, connection termination. a connection-oriented transport protocol establishes a logical path between the source and destination. All of the segments belonging to a message are then sent over this logical path.

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. The process starts with the server. The server program tells its TCP that it is ready to accept a connection. This request is called a passive open. Although the server TCP is ready to accept a connection from any machine in the world, it cannot make the connection itself.

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. TCP can now start the three-way handshaking process, as shown in Figure

The three steps in this phase are as follows.

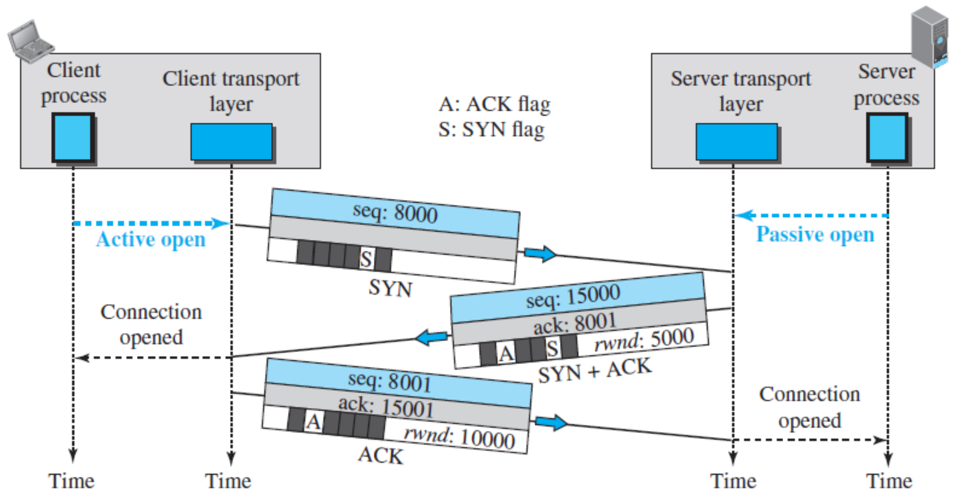
1. The client sends the first segment, a SYN segment, in which only the SYN flag is set. This segment is for synchronization of sequence numbers. The client in the example chooses a random number as the first sequence number and sends this number to the server. This sequence number is called the initial sequence number (ISN). Note that this segment does not contain an acknowledgment number. It does not define the window size either; a window size definition makes sense only when a segment

includes an acknowledgment. The segment can also include some options. The SYN segment is a control segment and carries no data, it consumes one sequence number because it needs to be acknowledged.

2. The server sends the second segment, a SYN + ACK segment with two flag bits set as: 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 the segment contains an acknowledgment, it also needs to define the receive window size, rwnd (to be used by the client), Since this segment is playing the role of a SYN segment, it needs to be acknowledged and therefore, consumes one sequence number.

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. The ACK segment does not consume any sequence numbers if it does not carry data, but some implementations allow this third segment in the connection phase to carry the first chunk of data from the client. In this case, the segment consumes as many sequence numbers as the number of data bytes.

Figure *Connection establishment using three-way handshaking*



SYN Flooding Attack

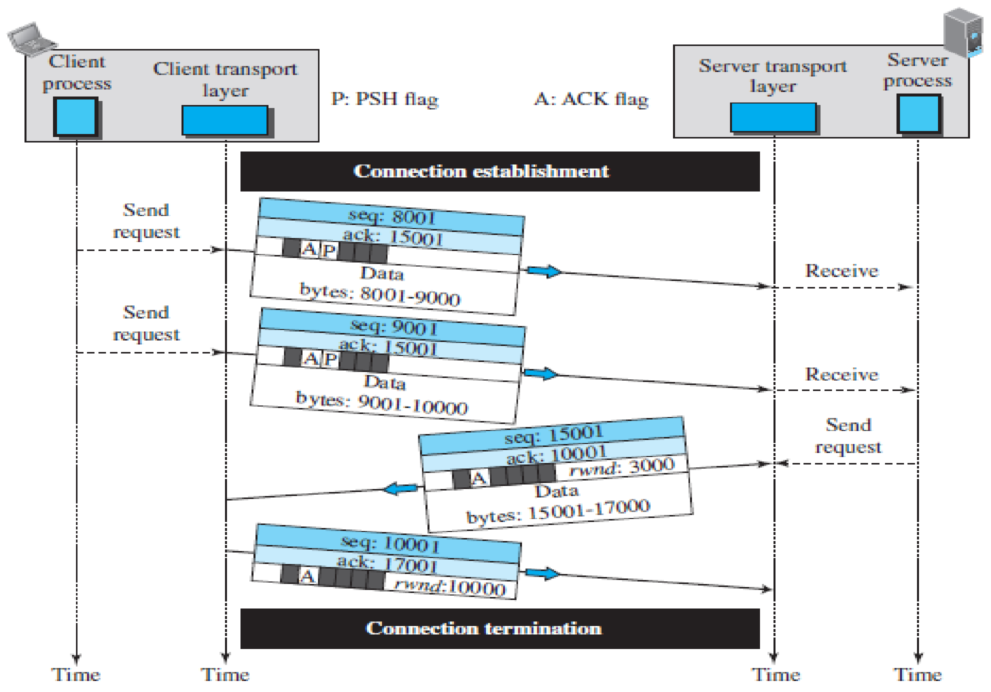
SYN flooding attack belongs to a group of security attacks known as a ***denial of service attack,*** in which an attacker monopolizes a system with so many service requests that the system overloads and denies service to valid requests.

Data Transfer

After connection is established, bidirectional data transfer can take place. The client and server can send data and acknowledgments in both directions.

In this example, after a connection is established, the client sends 2,000 bytes of data in two segments. The server then sends 2,000 bytes in one segment. The client sends one more segment. The first three segments carry both data and acknowledgment, but the last segment carries only an acknowledgment because there is no more data to be sent. Note the values of the sequence and acknowledgment numbers. The data segments sent by the client have the PSH (push) flag set so that the server TCP knows to deliver data to the server process as soon as they are received. The segment from the server, on the other hand, does not set the push flag. Most TCP implementations have the option to set or not to set this flag.

Figure *Data transfer*



Urgent Data

TCP is a stream-oriented protocol. This means that the data is presented from the application program to TCP as a stream of bytes. Each byte of data has a position in the stream. However, there are occasions in which an application program needs to send urgent bytes, some bytes that need to be treated in a special way by the application at the other end. The solution is to send a segment with the URG bit set. The sending application program tells the sending TCP that the piece of data is urgent. The sending TCP creates a segment and inserts the urgent data at the beginning of the segment. The rest of the segment can contain normal data from the buffer. The urgent pointer field in the header defines the end of the urgent data (the last byte of urgent data).

For example, if the segment sequence number is 15000 and the value of the urgent pointer is 200, the first byte of urgent data is the byte 15000 and the last byte is the byte 15200. The rest of the bytes in the segment (if present) are nonurgent.

Connection Termination

Either of the two parties involved in exchanging data (client or server) can close the connection, although it is usually initiated by the client. Most implementations allow two options for connection termination: three-way handshaking and four-way handshaking with a half-close option.

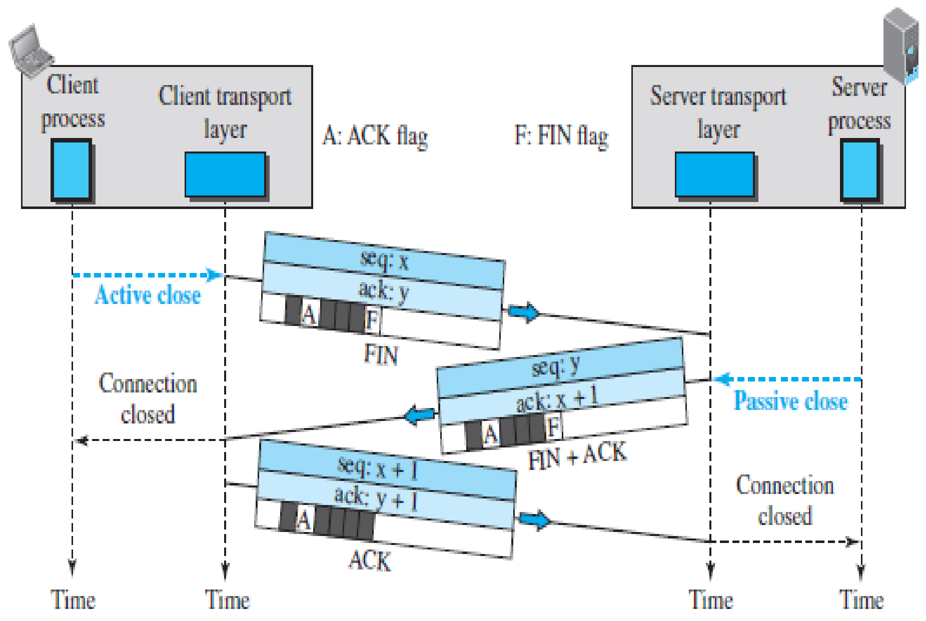
Three-Way Handshaking

1.The client TCP, after receiving a close command from the client process, sends the first segment, a FIN segment in which the FIN flag is set. A FIN segment can include the last chunk of data sent by the client or it can be just a control segment as shown in the figure. If it is only a control segment, it consumes only one sequence number because it needs to be acknowledged.

2.The server TCP, after receiving the FIN segment, informs its process of the situation and sends the second segment, a FIN + ACK segment, to confirm the receipt of the FIN segment from the client and at the same time to announce the closing of the connection in the other direction. This segment can also contain the last chunk of data from the server. If it does not carry data, it consumes only one sequence number because it needs to be acknowledged.

3. The client TCP sends the last segment, an ACK segment, to confirm the receipt of the FIN segment from the TCP server. This segment contains the acknowledgment number, which is one plus the sequence number received in the FIN segment from the server. This segment cannot carry data and consumes no sequence numbers.

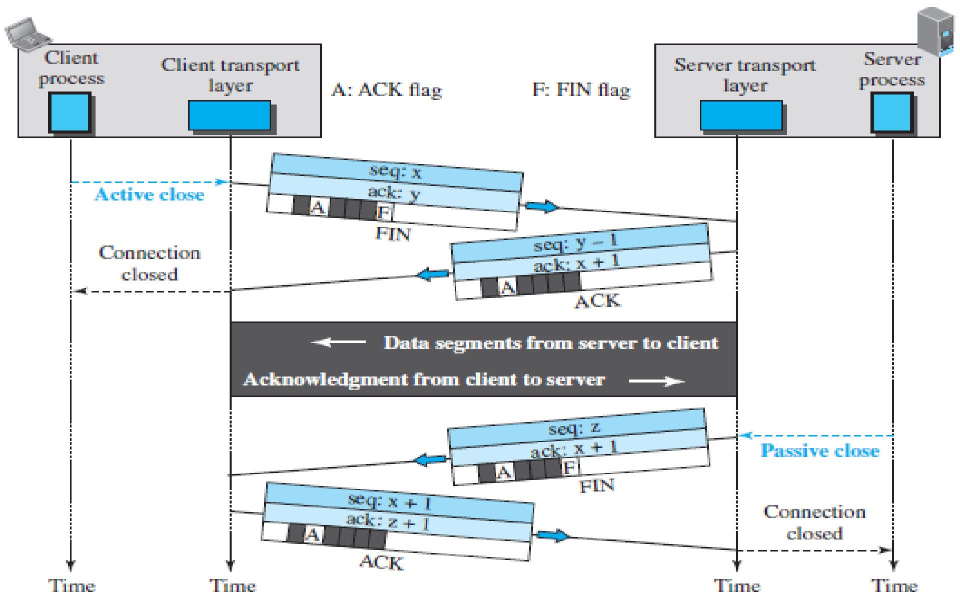
Figure Connection termination using three-way handshaking



Half-Close

In TCP, one end can stop sending data while still receiving data. This is called a ***halfclose.*** Either the server or the client can issue a half-close request. It can occur when the server needs all the data before processing can begin.

The data transfer from the client to the server stops. The client half-closes the connection by sending a FIN segment. The server accepts the half-close by sending the ACK segment. The server, however, can still send data. When the server has sent all of the processed data, it sends a FIN segment, which is acknowledged by an ACK from the client. After half-closing the connection, data can travel from the server to the client and acknowledgments can travel from the client to the server. The client cannot send any more data to the server.

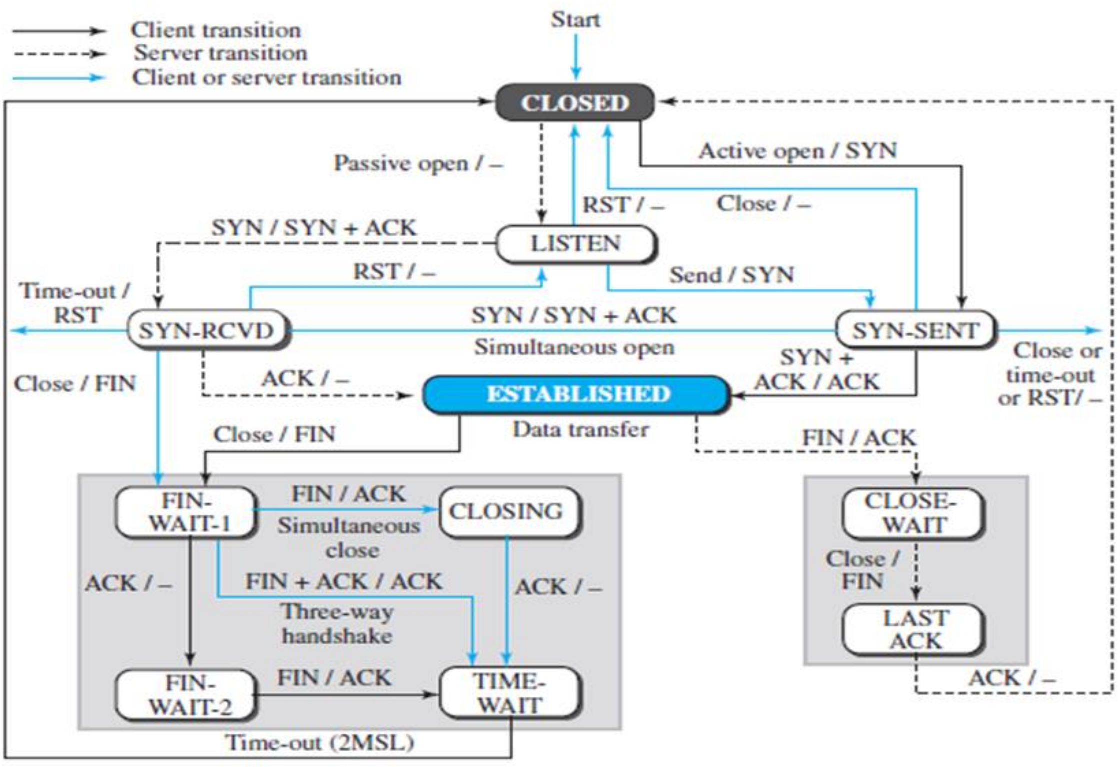


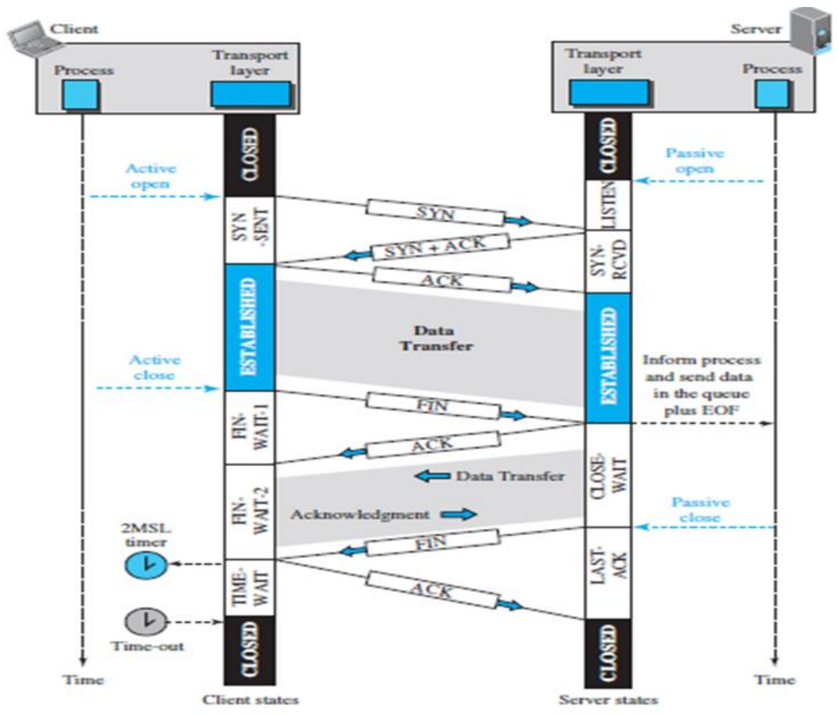
Connection Reset

TCP at one end may deny a connection request, may abort an existing connection, or may terminate an idle connection. All of these are done with the RST (reset) flag.

State Transition Diagram

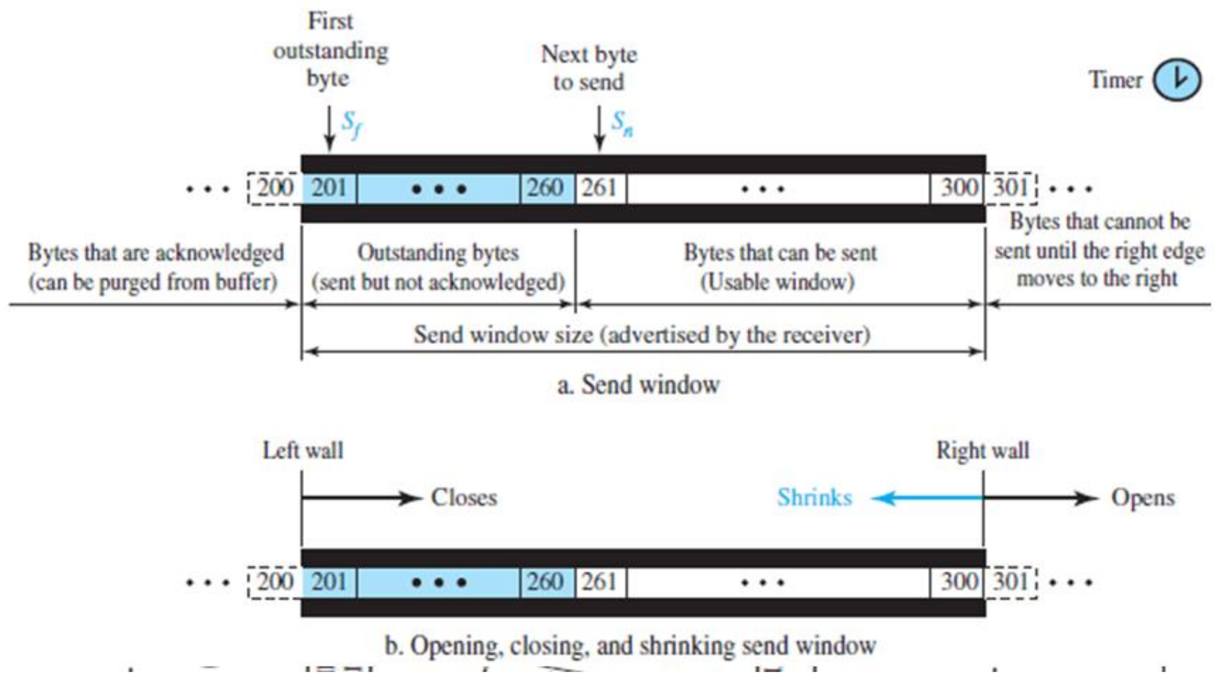
The figure shows the two FSMs used by the TCP client and server combined in one diagram. The rounded-corner rectangles represent the states. The transition from one state to another is shown using directed lines. Each line has two strings separated by a slash. The first string is the input, what TCP receives. The second is the output, what TCP sends. The dotted black lines in the figure represent the transition that a server normally goes through; the solid black lines show the transitions that a client normally goes through. However, in some situations, a server transitions through a solid line or a client transitions through a dotted line. The colored lines show special situations. Note that the rounded-corner rectangle marked ESTABLISHED is in fact two sets of states, a set for the client and another for the server, that are used for flow and error control,





Windows in TCP

TCP uses two windows (send window and receive window) for each direction of data transfer, which means four windows for a bidirectional communication.

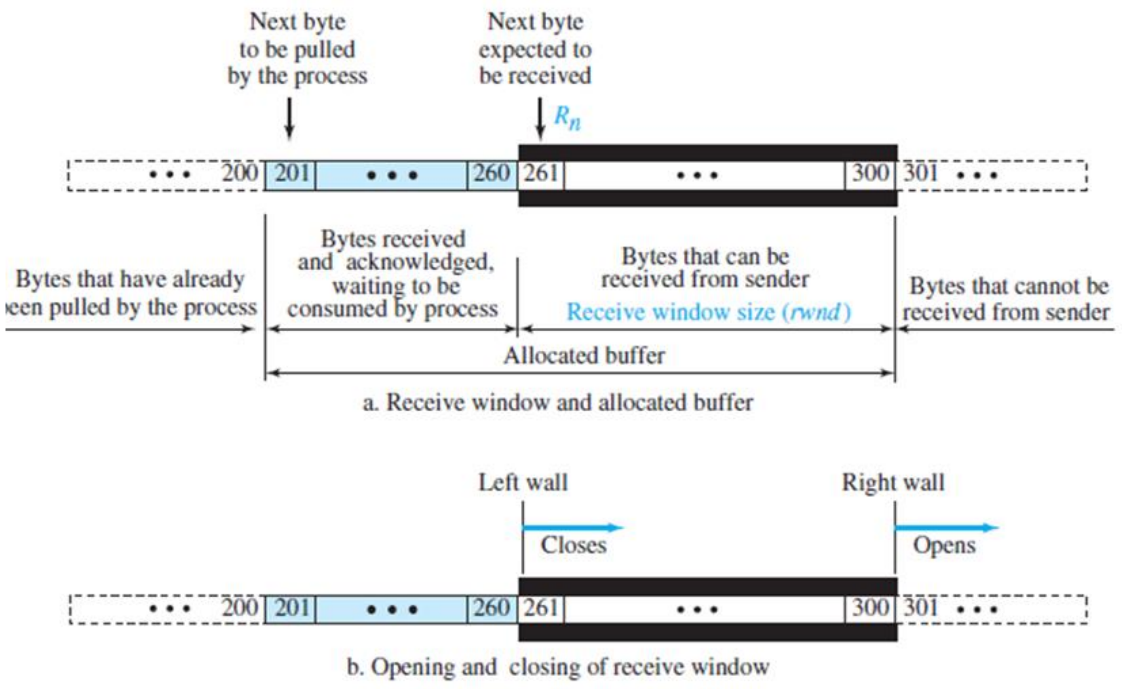

First outstanding byte
S_f
Next byte to send
S_n
Timer
200
201
260
261
300
301
Bytes that are acknowledged (can be purged from buffer)
Outstanding bytes (sent but not acknowledged)
Bytes that can be sent (Usable window)
Bytes that cannot be sent until the right edge moves to the right
Send window size (advertised by the receiver)
a. Send window
Left wall
Closes
Right wall
Shrinks
Opens
b. Opening, closing, and shrinking send window

The send window in TCP is similar to the one used with the **Selective-Repeat protocol**,but with some differences:

1. The window size in SR is the number of packets, but the window size in TCP is the number of bytes.

2. TCP can store data received from the process and send them later, but we assume that the sending TCP is capable of sending segments of data as soon as it receives them from its process.

3. Another difference is the number of timers. The theoretical Selective-Repeat protocol may use several timers for each packet sent, but as mentioned before, theTCP protocol uses only one timer.


Next byte to be pulled by the process
Next byte expected to be received
R_n
200
201
260
261
300
301
Bytes that have already been pulled by the process
Bytes received and acknowledged, waiting to be consumed by process
Bytes that can be received from sender
Receive window size (rwnd)
Bytes that cannot be received from sender
Allocated buffer
a. Receive window and allocated buffer
Left wall
Closes
Right wall
Opens
b. Opening and closing of receive window

There are two differences between the receive window in TCP and the one we used for SR.

1. TCP allows the receiving process to pull data at its own pace. The receive window size is then always smaller than or equal to the buffer size. The receive window size determines the number of bytes that the receive window can accept from the sender before being overwhelmed (flow control). In other words, the receive window size, normally called *rwnd*, can be determined as:

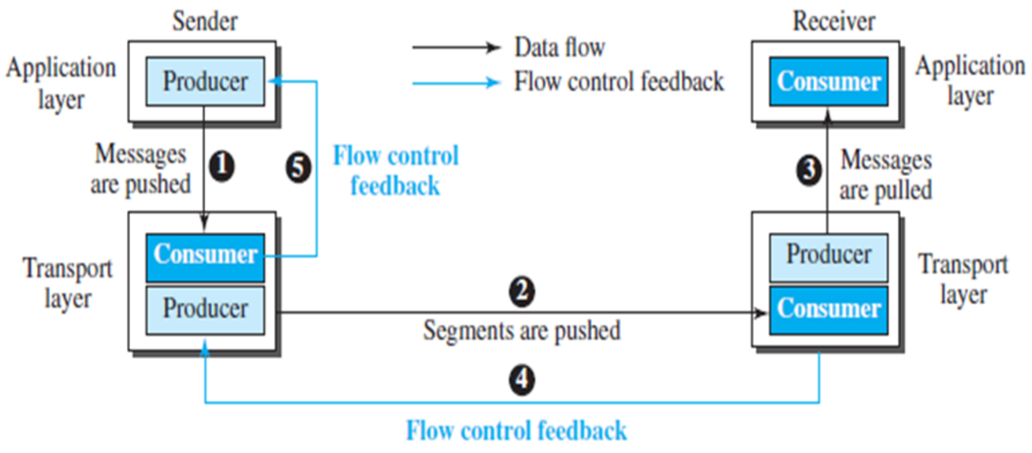
rwnd = **buffer size - number of waiting bytes to be pulled**

2. The way acknowledgments are used in the TCP protocol.

An acknowledgement in SR is selective, defining the uncorrupted packets that have been received. The major acknowledgment mechanism in TCP is a cumulative acknowledgment announcing the next expected byte to receive (TCP looks like GBN).

The new version of TCP, however, uses both cumulative and selective acknowledgments;

Flow Control



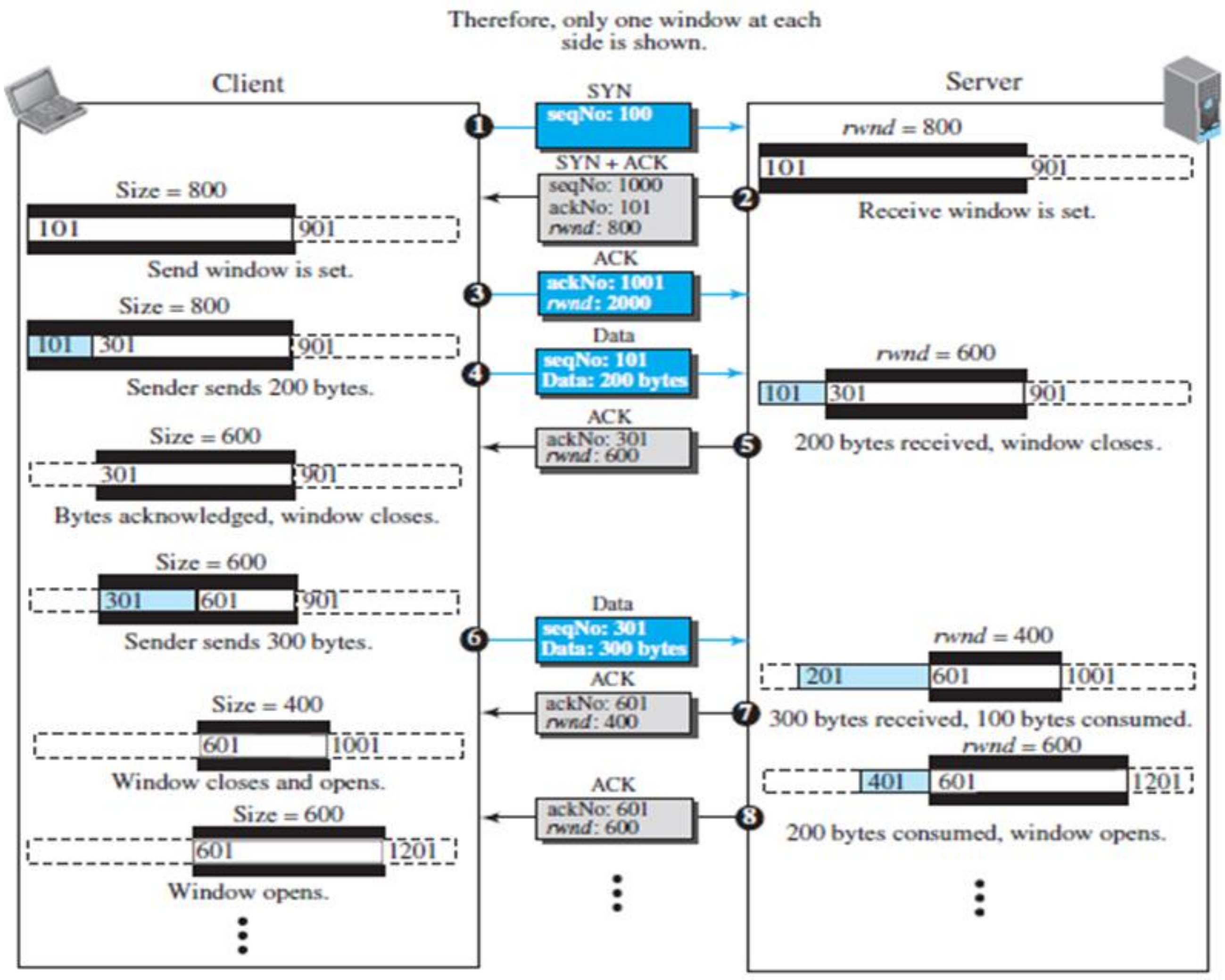
- *Flow control balances the rate a producer creates data with the rate* a consumer can use the data.
- TCP separates flow control from error control.
- Figure shows unidirectional data transfer between a sender and a receiver; Bidirectional data transfer can be deduced from the unidirectional process.
- Most implementations of TCP do not provide flow control feedback from the receiving process to the receiving TCP.
- The receiving TCP controls the sending TCP; the sending TCP controls the sending process.
- Flow control feedback from the sending TCP to the sending process (path 5) is achieved through simple rejection of data by the sending TCP when its window is full.

Opening and Closing Windows

- To achieve flow control, TCP forces the sender and the receiver to adjust their window sizes,
- The receive window closes when more bytes arrive from the sender; it opens) when more bytes are pulled by the process.
- We assume that it does not shrink.
- The opening, closing, and shrinking of the send window is controlled by the receiver.

A Scenario

- Figure shows a simple example of unidirectional data transfer (from client to server).
- We assume that no segment is corrupted, lost, duplicated, or has arrived out of order.
- Note that only two windows for unidirectional data transfer is shown.



-

Eight segments are exchanged between the client and server:

1. The first segment is from the client to the server (a SYN segment) to request connection.
The client announces its initial seqNo = 100. When this segment arrives at the server, it allocates a buffer size of 800 (an assumption) and sets its window to cover the whole buffer (*rwnd* = 800). Note that the number of the next byte to arrive is 101.

2. The second segment is from the server to the client. This is an ACK + SYN segment.
The segment uses ackNo = 101 to show that it expects to receive bytes starting from 101. It also announces that the client can set a buffer size of 800 bytes.

3. The third segment is the ACK segment from the client to the server. The client has defined a rwnd of size 2000.

4. After the client has set its window with the size (800) dictated by the server, the process pushes 200 bytes of data. The TCP client numbers these bytes 101 to 300. It then creates a segment and sends it to the server. The segment shows the starting byte number as 101 and the segment carries 200 bytes. The window of the client is then adjusted to show that 200 bytes of data are sent but waiting for acknowledgment. When this segment is received at the server, the bytes are stored, and the receive window closes to show that the next byte expected is byte 301; the stored bytes occupy 200 bytes of buffer.

5. The fifth segment is the feedback from the server to the client. The server acknowledges bytes up to and including 300 (expecting to receive byte 301). The segment also carries the size of the receive window after decrease (600). The client, after receiving this segment, purges the acknowledged bytes from its window and closes its window to show that the next byte to send is byte 301. The window size, however, decreases to 600 bytes. Although the allocated buffer can store 800 bytes, the window cannot open (moving its right wall to the right) because the receiver does not let it.

6. Segment 6 is sent by the client after its process pushes 300 more bytes. The segment defines seqNo as 301 and contains 300 bytes. When this segment arrives at the server, the server stores them, but it has to reduce its window size. After its process has pulled 100 bytes of data, the window closes from the left for the amount of 300 bytes, but opens from the right for the amount of 100 bytes. The result is that the size is only reduced 200 bytes. The receiver window size is now 400 bytes.

7. In segment 7, the server acknowledges the receipt of data, and announces that its window size is 400. When this segment arrives at the client, the client has no choice but to reduce its window again and set the window size to the value of rwnd = 400 advertised by the server. The send window closes from the left by 300 bytes, and opens from the right by 100 bytes.

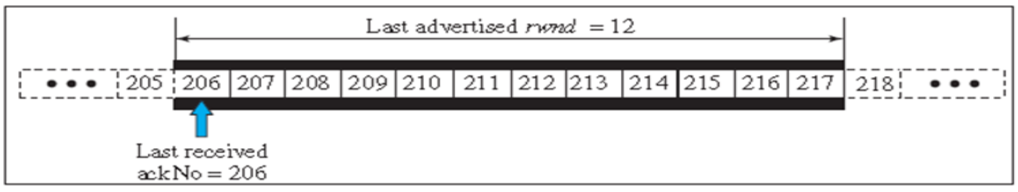
8. Segment 8 is also from the server after its process has pulled another 200 bytes. Its window size increases. The new rwnd value is now 600. The segment informs the client that the server still expects byte 601, but the server window size has expanded to 600. After this segment arrives at the client, the client opens its window by 200 bytes without closing it. The result is that its window size increases to 600 bytes.

Shrinking of Windows

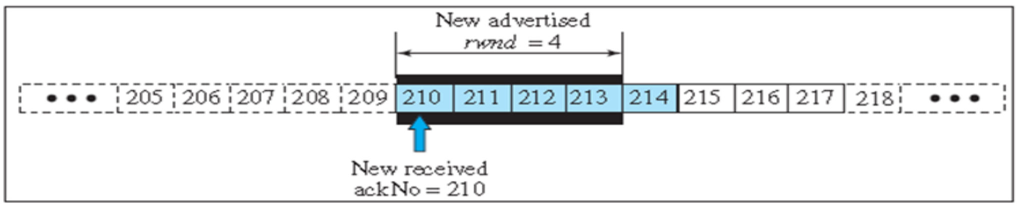
- The receive window cannot shrink. The send window, can shrink if the receiver defines a value for ***rwnd** that results in shrinking the* window.
- However, some implementations do not allow shrinking of the send window.
- *To prevent shrinking of the send* window the receiver needs to keep the following relationship between

new ackNo + new *rwnd* ≥ *last ackNo* + *last rwnd*

- This inequality is a mandate for the receiver to check its advertisement.



a. The window after the last advertisement



b. The window after the new advertisement; window has shrunk

- Part a of the figure shows the values of the last acknowledgment and *rwnd.*
- *Part b shows the* situation in which the sender has sent bytes 206 to 214.
- Bytes 206 to 209 are acknowledged and purged.
- The new advertisement, however, defines the new value of *rwnd as 4, in which 210 + 4 <* 206 + 12.
- When the send window shrinks, it creates a problem: byte 214, which has already been sent, is outside the window.
- the receiver does not know which of the bytes 210 to 217 has already been sent.
- One way to prevent this situation is to let the receiver postpone its feedback until enough buffer locations are available in its window.

- The receiver should wait until more bytes are consumed by its process

Window Shutdown

- Shrinking the send window by moving its right wall to the left is strongly discouraged.
- However, there is one exception: the receiver can temporarily shut down the window by sending a *rwnd of 0.*
- *This can happen if for some reason the receiver* does not want to receive any data from the sender for a while.
- In this case, the sender does not actually shrink the size of the window, but stops sending data until a new advertisement has arrived.

Silly Window Syndrome

- A serious problem can arise in the sliding window operation when either the sending application program creates data slowly or the receiving application program consumes data slowly, or both.
- This results in the sending of data in very small segments, which reduces the efficiency of the operation.
- For example:- if TCP sends segments containing only 1 byte of data, it means that a 41-byte datagram (20 bytes of TCP header and 20 bytes of IP header) transfers only 1 byte of user data.
- The inefficiency is even worse after accounting for the data-link layer and physical-layer overhead. This problem is called the ***silly window syndrome.***

Syndrome Created by the Sender

- The sending TCP may create a silly window syndrome if it is serving an application program that creates data slowly, for example, 1 byte at a time.
- The application program writes 1 byte at a time into the buffer of the sending TCP.
- The result is a lot of 41-byte segments that are traveling through an internet.
- The solution is to prevent the sending TCP from sending the data byte by byte. The sending TCP must be forced to wait and collect data to send in a larger block.
- How long should the sending TCP wait? If it waits too long, it may delay the process.
- If it does not wait long enough, it may end up sending small segments.

Nagle's algorithm

1. **The sending TCP sends the first piece of data it receives from the sending application** program even if it is only 1 byte.

2. After sending the first segment, the sending TCP accumulates data in the output buffer and waits until either the receiving TCP sends an acknowledgment or until enough data have accumulated to fill a maximum-size segment.

3. Step 2 is repeated for the rest of the transmission. Segment 3 is sent immediately if an acknowledgment is received for segment 2, or if enough data have accumulated to fill a maximum-size segment.

Syndrome Created by the Receiver

- The receiving TCP may create a silly window syndrome if it is serving an application program that consumes data slowly, for example, 1 byte at a time.
  - Suppose that the sending application program creates data in blocks of 1 kB, but the receiving application program consumes data 1 byte at a time.
  - Also suppose that the input buffer of the receiving TCP is 4 kB.
  - The sender sends the first 4 kB of data. The receiver stores it in its buffer. Now its buffer is full. It advertises a window size of zero, which means the sender should stop sending data.

1. **Clark's solution is to send an acknowledgment as soon as the data arrive, but to announce a** window size of zero until either there is enough space to accommodate a segment of maximum size or until at least half of the receive buffer is empty.
2. The second solution is to delay sending the acknowledgment. This means that when a segment arrives, it is not acknowledged immediately. The receiver waits until there is a decent amount of space in its incoming buffer before acknowledging the arrived segments.

   Delayed acknowledgment also has another advantage: it reduces traffic. The receiver does not have to acknowledge each segment.

   However, there also is a disadvantage in that the delayed acknowledgment may result in the sender unnecessarily retransmitting the unacknowledged segments. The TCP protocol balances the advantages and disadvantages. It now defines that the acknowledgment should not be delayed by more than 500 ms.

Error Control

- TCP provides reliability using error control.
- Error control includes
  - mechanisms for detecting and resending corrupted segments,
  - resending lost segments,
  - storing out-of order segments until missing segments arrive,
  - Detecting and discarding duplicated segments.
- Error control in TCP is achieved through the use of three simple tools:
  - checksum, acknowledgment, and time-out.

1. Checksum

   Each segment includes a checksum field, which is used to check for a corrupted segment.

   If a segment is corrupted, as detected by an invalid checksum, the segment is discarded by the destination TCP and is considered as lost.

   TCP uses a 16-bit checksum that is mandatory in every segment.

2. Acknowledgment

TCP uses acknowledgments to confirm the receipt of data segments.

Control segments that carry no data, but consume a sequence number, are also acknowledged.

ACK segments are never acknowledged.

ACK segments do not consume sequence numbers and are not acknowledged.

- TCP used only one type of acknowledgment: **cumulative acknowledgment.**
- Today, some TCP implementations also use **selective acknowledgment.**

3. Cumulative Acknowledgment (ACK)

TCP was originally designed to acknowledge receipt of segments cumulatively.

The receiver sends the next byte it expects to receive, ignoring all segments received and stored out of order. This is sometimes referred to as positive cumulative acknowledgment, or ACK.

The word positive means that no feedback is provided for discarded, lost, or duplicate segments.

The 32-bit ACK field in the TCP header is used for cumulative acknowledgments, and its value is valid only when the ACK flag bit is set to 1.

4. Selective Acknowledgment (SACK)

More and more implementations are adding another type of acknowledgment called selective acknowledgment, or SACK.

A SACK does not replace an ACK, but reports additional information to the sender.

A SACK reports a block of bytes that is out of order, and also a block of bytes that is duplicated, i.e., received more than once. However, since there is no provision in the TCP header for adding this type of information, SACK is implemented as an option at the end of the TCP header.

Generating Acknowledgments

- When end A sends a data segment to end B, it must include (piggyback) an acknowledgment that gives the next sequence number it expects to receive.
- When the receiver has no data to send and it receives an in-order segment (with expected sequence number) and the previous segment has already been acknowledged, the receiver delays sending an ACK segment until another segment arrives or until a period of time (normally 500 ms) has passed.
- When a segment arrives with a sequence number that is expected by the receiver, and the previous in-order segment has not been acknowledged, the receiver immediately sends an ACK segment.
- When a segment arrives with an out-of-order sequence number that is higher than expected, the receiver immediately sends an ACK segment announcing the sequence number of the next expected segment.
- When a missing segment arrives, the receiver sends an ACK segment to announce the next sequence number expected.
- If a duplicate segment arrives, the receiver discards the segment, but immediately sends an acknowledgment indicating the next in-order segment expected

5. Retransmission

When a segment is sent, it is stored in a queue until it is acknowledged. When the retransmission timer expires or when the sender receives three duplicate ACKs for the first segment in the queue, that segment is retransmitted.

Retransmission after RTO

- The sending TCP maintains one **retransmission time-out (RTO)** for each connection.
- When the timer matures, i.e. times out, TCP resends the segment in the front of the queue and restarts the timer.

- The value of RTO is dynamic in TCP and is updated based on the **round-trip time (RTT)** of segments.
- RTT is the time needed for a segment to reach a destination and for an acknowledgment to be received

  Retransmission after Three Duplicate ACK Segments

  The previous rule about retransmission of a segment is sufficient if the value of RTO is not large. To expedite service throughout the Internet by allowing senders to retransmit without waiting for a time out, most implementations today follow the three duplicate ACKs rule and retransmit the missing segment immediately. This feature is called fast retransmission.

  In this version, if three duplicate acknowledgments (i.e., an original ACK plus three exactly identical copies) arrive for a segment, the next segment is retransmitted without waiting for the time-out.

6. Out-of-Order Segments

Data may arrive out of order and be temporarily stored by the receiving TCP, but TCP guarantees that no out-of-order data are delivered to the process.

TCP Congestion Control

Congestion Window

- The size of the send window is controlled by the receiver using the value of *rwnd*, which is advertised in each segment traveling in the opposite direction.
- The use of this strategy guarantees that the receive window is never overflowed with the received bytes.
- There is no congestion at the other end, but there may be congestion in the middle.
- IP is a simple protocol with no congestion control. TCP, itself, needs to be responsible for this problem.
- To control the number of segments to transmit, TCP uses another variable called a congestion window, cwnd, whose size is controlled by the congestion situation in the network.
- **Actual window size** = **minimum (*rwnd, cwnd*)**

Congestion Detection

- The TCP sender uses the occurrence of two events as signs of congestion in the network: time-out and receiving three duplicate ACKs.
- An earlier version of TCP, called Taho TCP, treated both events (time-out and three duplicate ACKs) similarly,
- Later version of TCP, called Reno TCP, treats these two signs differently.

Congestion Policies

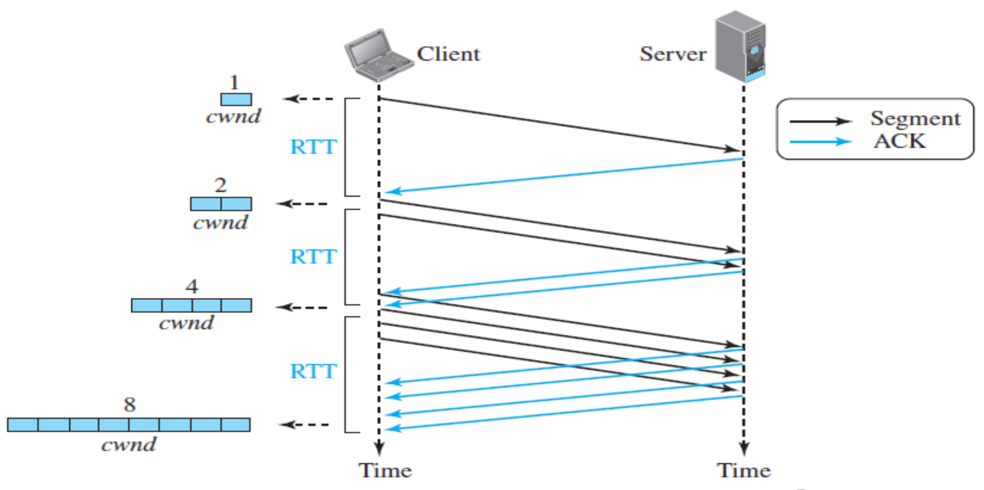
TCP's general policy for handling congestion is based on three algorithms:

- slow start
- congestion avoidance, and
- Fast recovery

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), but it increases one MSS each time an acknowledgment arrives.
- The MSS is a value negotiated during the connection establishment, using an option of the same name.
- The algorithm **starts slowly, but grows exponentially**.
- The sender starts with cwnd = 1. This means that the sender can send only one segment. After the first ACK arrives, the acknowledged segment is purged from the window, which means there is now one empty segment slot in the window.
- The size of the congestion window is also increased by 1 because no congestion in the network. The size of the window is now 2. After sending two segments and receiving two individual acknowledgments for them, the size of the congestion window now becomes 4, and so on.

re *Slow start, exponential increase*

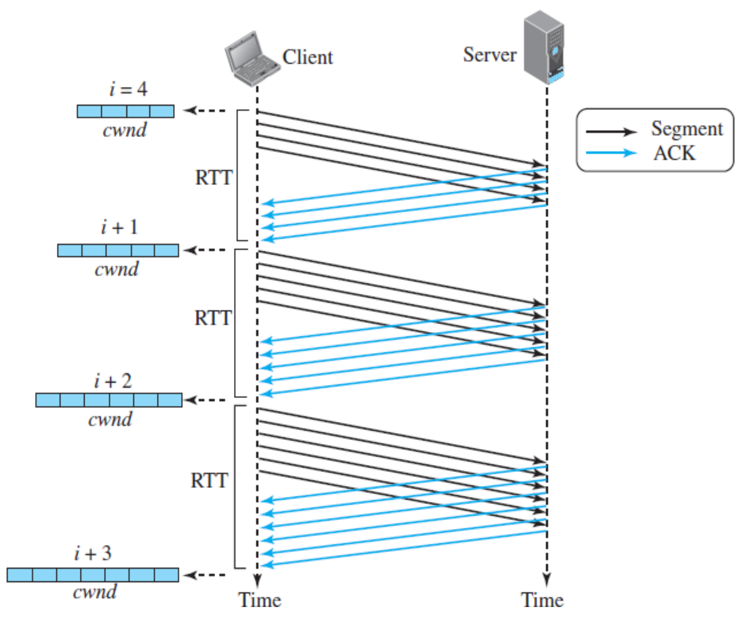


Start	$\rightarrow$	$cwnd = 1 \longrightarrow 2^0$
After 1 RTT	$\rightarrow$	$cwnd = cwnd + 1 \quad = 1 + 1 = 2 \longrightarrow 2^1$
After 2 RTT	$\rightarrow$	$cwnd = cwnd + 2 \quad = 2 + 2 = 4 \longrightarrow 2^2$
After 3 RTT	$\rightarrow$	$cwnd = cwnd + 4 \quad = 4 + 4 = 8 \longrightarrow 2^3$

Congestion Avoidance: Additive Increase

- 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. A window is the number of segments transmitted during RTT.

Congestion avoidance, additive increase



- The sender starts with *cwnd* = 4. This means that the sender can send only 4 segments. After 4 ACKs arrive, the acknowledged segments are purged from the window, which means there is now one extra empty segment slot in the window.
- The size of the congestion window is also increased by 1. The size of window is now 5. After sending five segments and receiving five acknowledgments for them, the size of the congestion window now becomes 6, and so on.
- In other words, the size of the congestion window in this algorithm is also a function of the number of ACKs that have arrived and can be determined as follows:

If an ACK arrives , *cwnd* = *cwnd* + (1/*cwnd*).

Fast Recovery

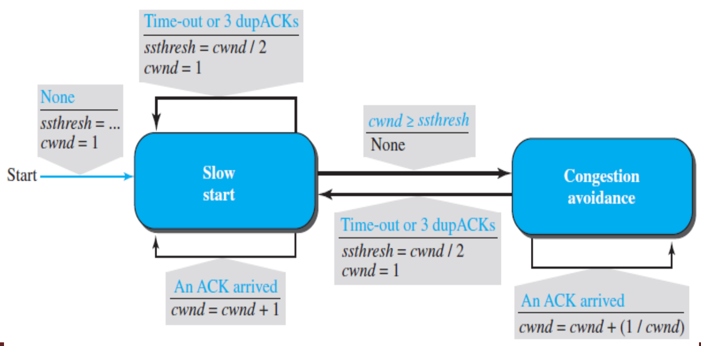
- The **fast-recovery** algorithm is optional in TCP.
- The old version of TCP did not use it, but the new versions try to use it. It starts when three duplicate ACKs arrive, which is interpreted as light congestion in the network.
- Like congestion avoidance, this algorithm is also an additive increase, but it increases the size of the congestion window when a duplicate ACK arrives.

- Three versions of congestion policies in TCP:
    - Taho TCP
    - Reno TCP and
    - New Reno TCP

Taho TCP

- The early TCP, known as *Taho TCP*, used only two different algorithms in their congestion policy: *slow start* and *congestion avoidance.*
- Figure show the FSM for this version of TCP.
- However, some small trivial actions have deleted, such as incrementing and resetting the number of duplicate ACKs, to make the FSM less crowded and simpler.

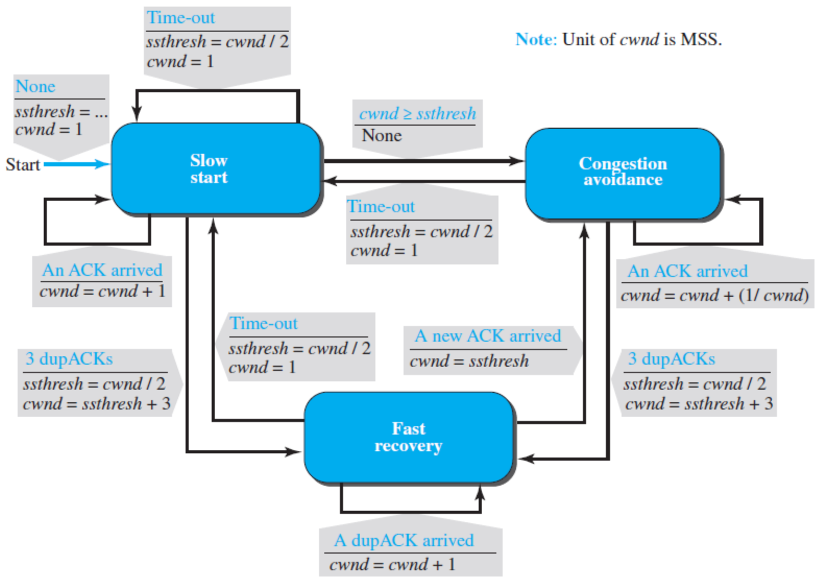
Figure FSM for Taho TCP



-

Reno TCP

- A newer version of TCP, called *Reno TCP*, added a new state to the congestion-control FSM, called the fast-recovery state. This version treated the two signals of congestion. time-out and the arrival of three duplicate ACKs, differently.
- In this version, if a time-out occurs, TCP moves to the slow-start state on the other hand, if three duplicate ACKs arrive, TCP moves to the fast-recovery state and remains there as long as more duplicate ACKs arrive.
- The fast-recovery state is a state somewhere between the slow-start and the congestion-avoidance states.



NewReno TCP

- A later version of TCP, called *NewReno TCP,* made an extra optimization on the Reno TCP.
- In this version, TCP checks to see if more than one segment is lost in the current window when three duplicate ACKs arrive.
- When TCP receives three duplicate ACKs, it retransmits the lost segment until a new ACK (not duplicate) arrives.
- If the new ACK defines the end of the window when the congestion was detected, TCP is certain that only one segment was lost.
- NewReno TCP retransmits segment to avoid receiving more and more duplicate ACKs for it.