MODULE 5

1) What are the 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.

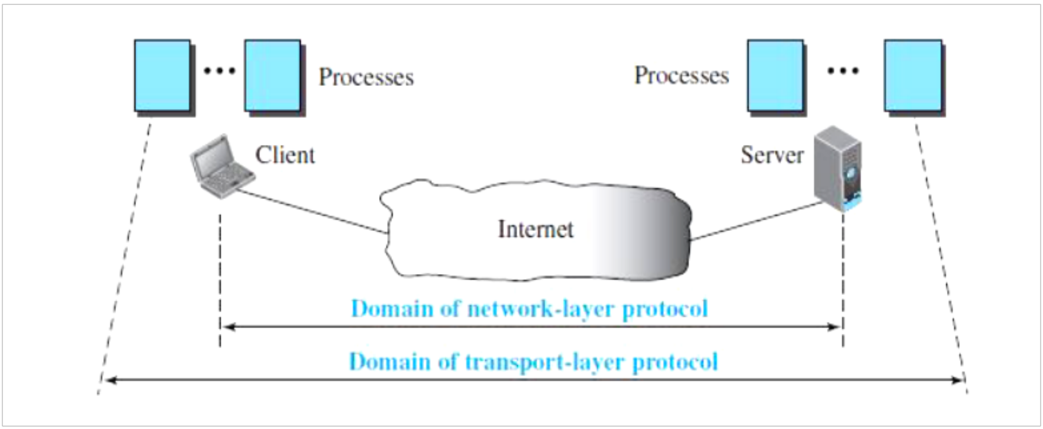

Processes
Processes
Client
Server
Internet
Domain of network-layer protocol
Domain of transport-layer protocol

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.

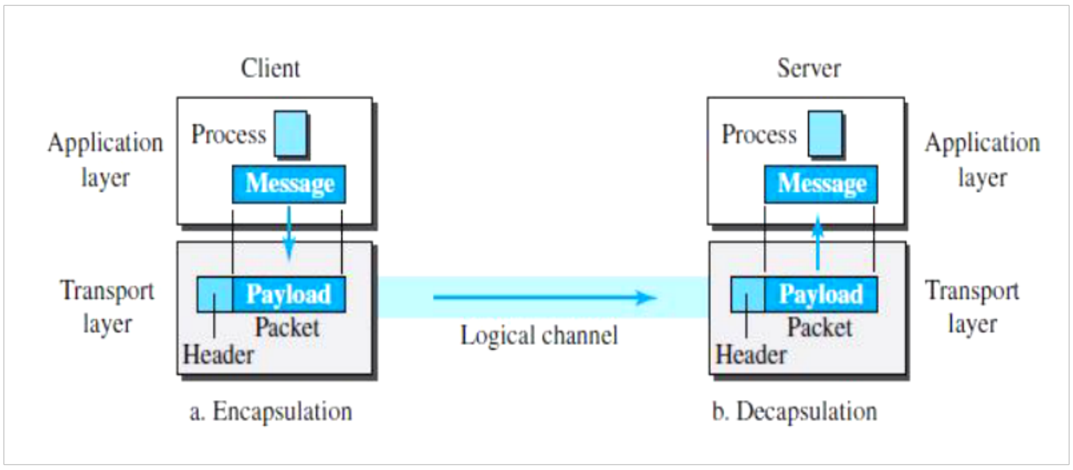


Figure 2: Encapsulation and decapsulation

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.

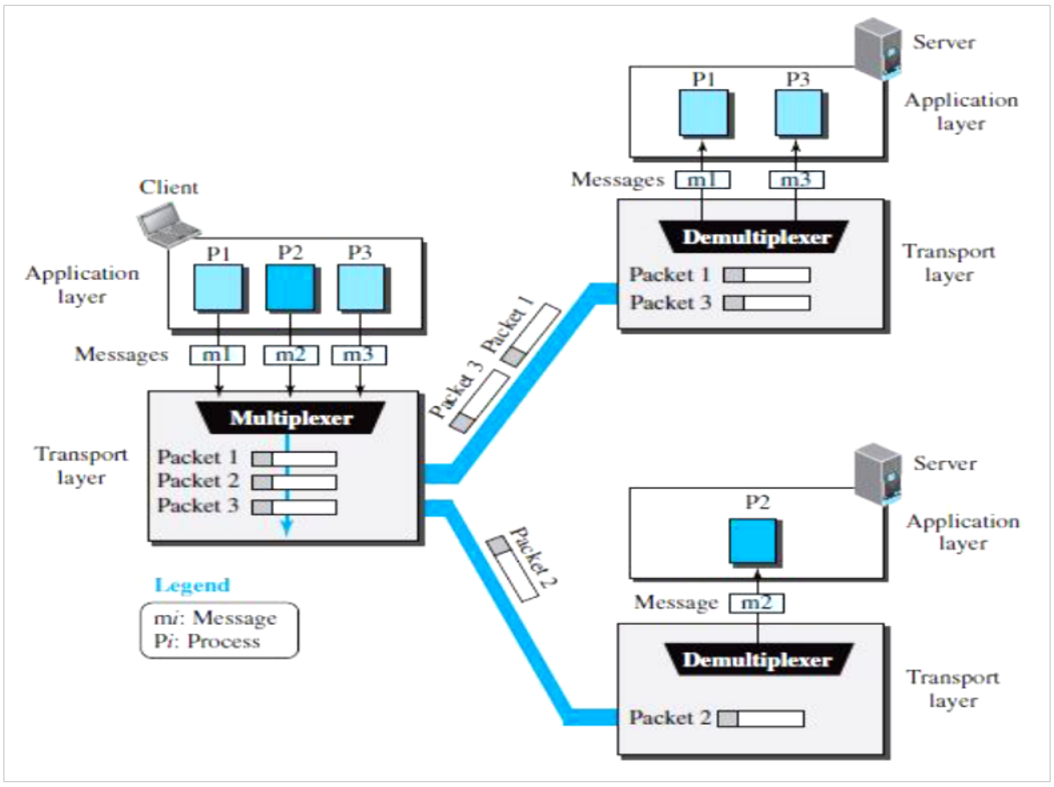


Figure 3: Multiplexing and demultiplexing

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.

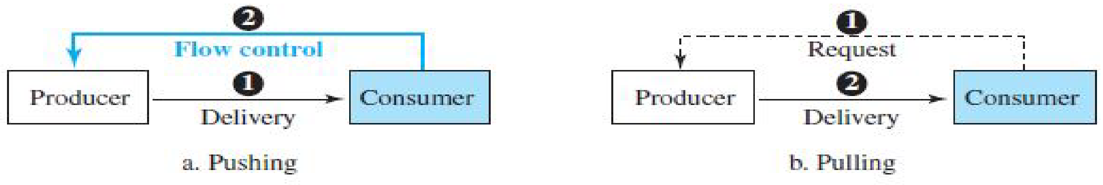


Figure 4: Pushing or pulling

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.

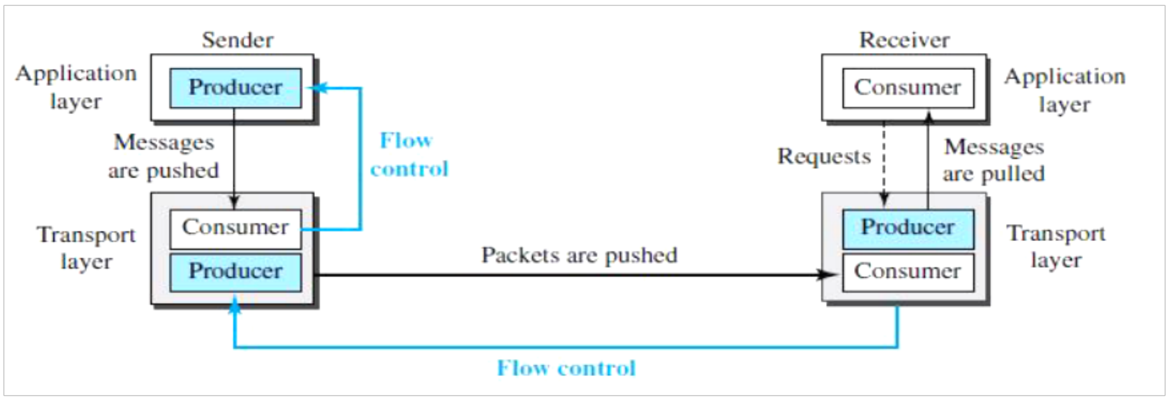


Figure 5: Flow control at the transport layer

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.

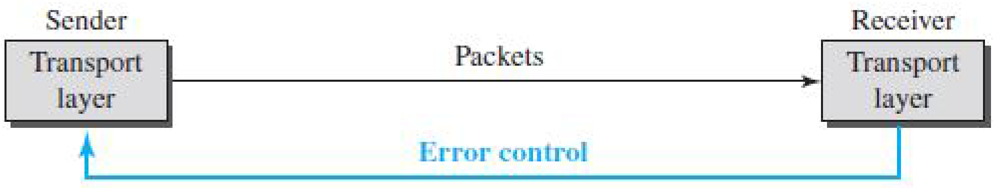


Figure 6: Error control at the transport layer

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.

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.

2) Explain the concept of 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.

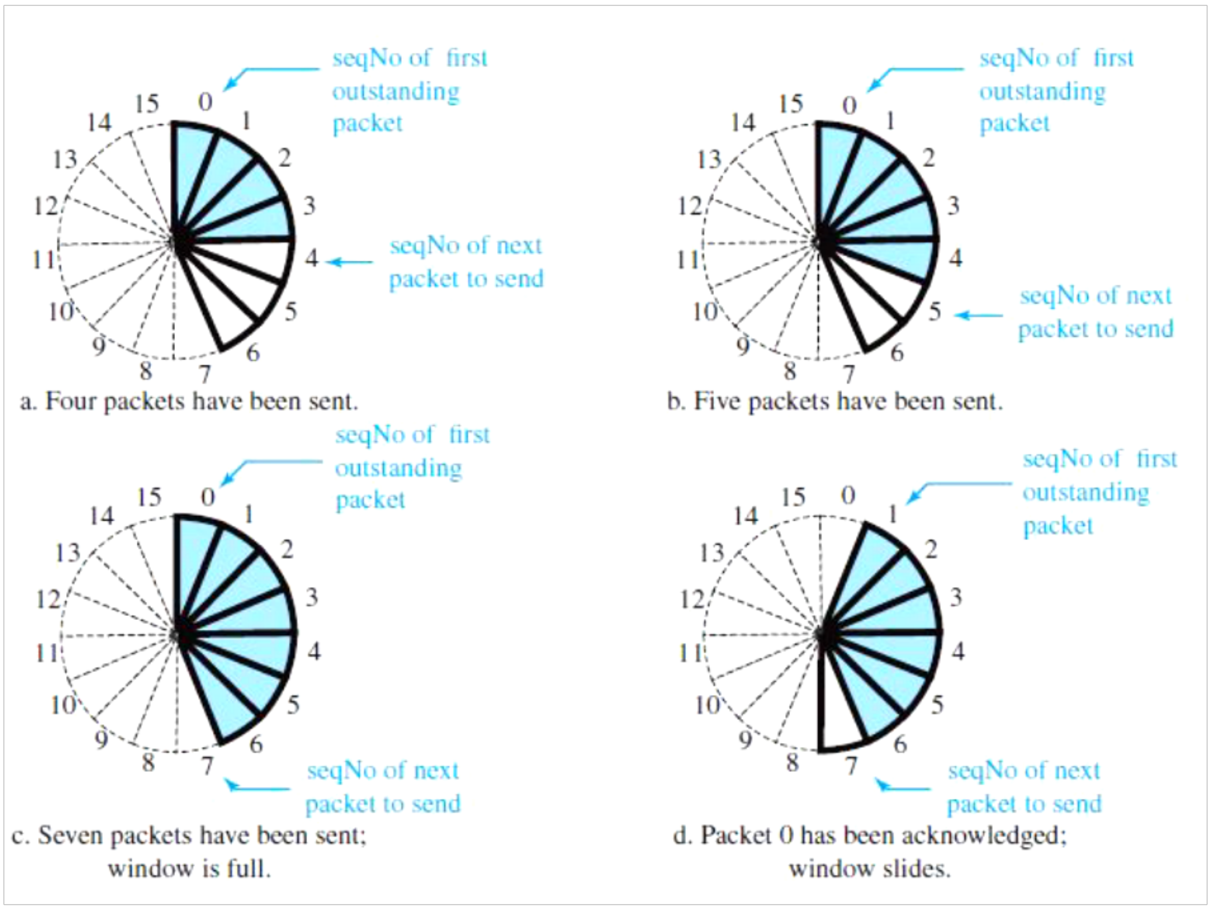

seqNo of first outstanding packet
seqNo of next packet to send
a. Four packets have been sent.
seqNo of first outstanding packet
seqNo of next packet to send
b. Five packets have been sent.
seqNo of first outstanding packet
seqNo of next packet to send
c. Seven packets have been sent; window is full.
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.

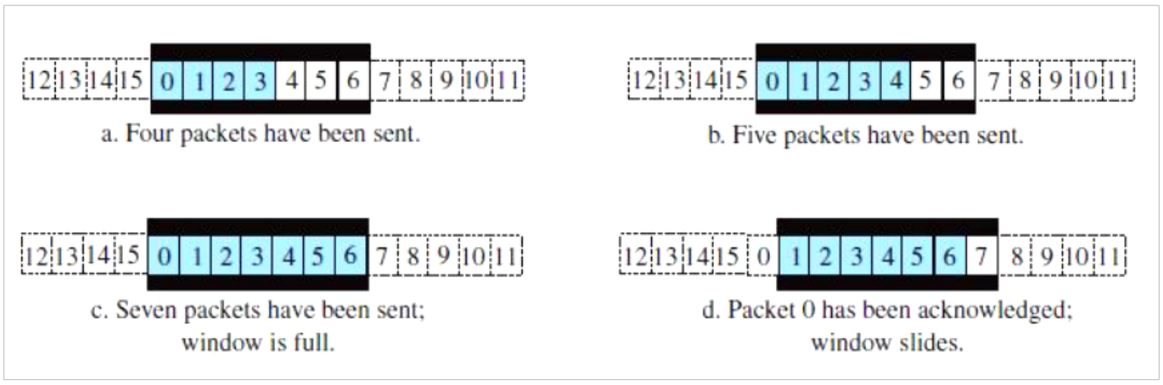

12 13 14 15 0 1 2 3 4 5 6 7 8 9 10 11
a. Four packets have been sent.
b. Five packets have been sent.
c. Seven packets have been sent; window is full.
d. Packet 0 has been acknowledged; window slides.

Figure 8: Sliding window in linear format

3. **Write outline and explain send window and receive window for Go back N protocol/ selective repeat 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.

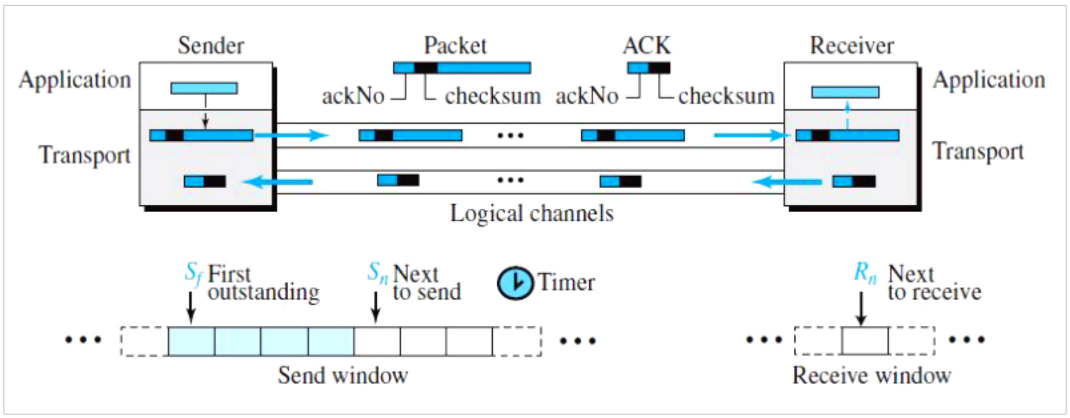
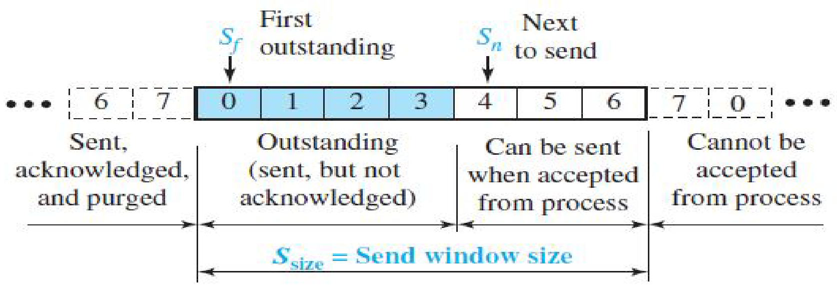


Figure 9: Go-Back-N protocol

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.

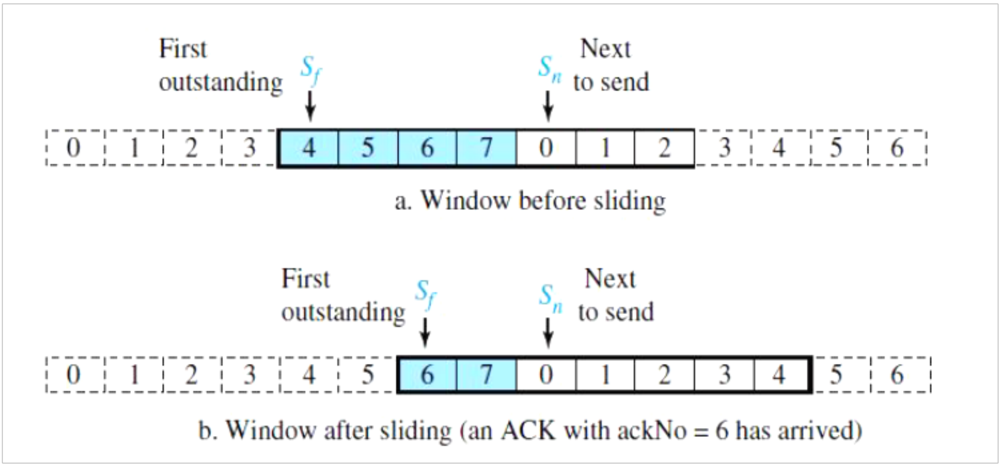


Figure 11: Sliding the send window

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.

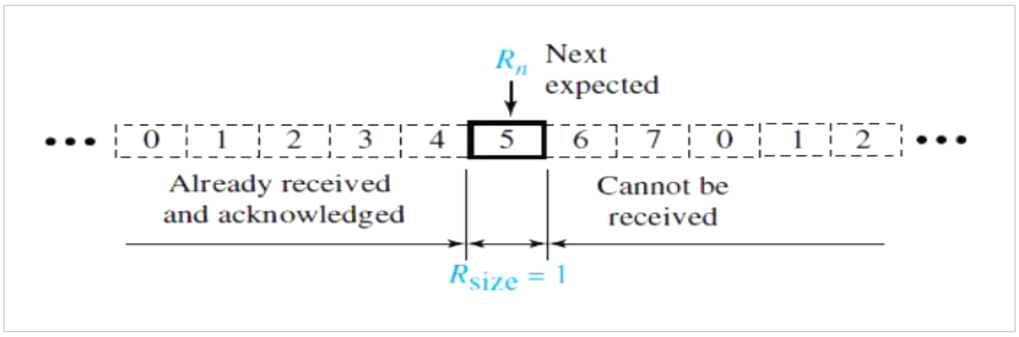


Figure 12: Receive window for Go-Back-N

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.

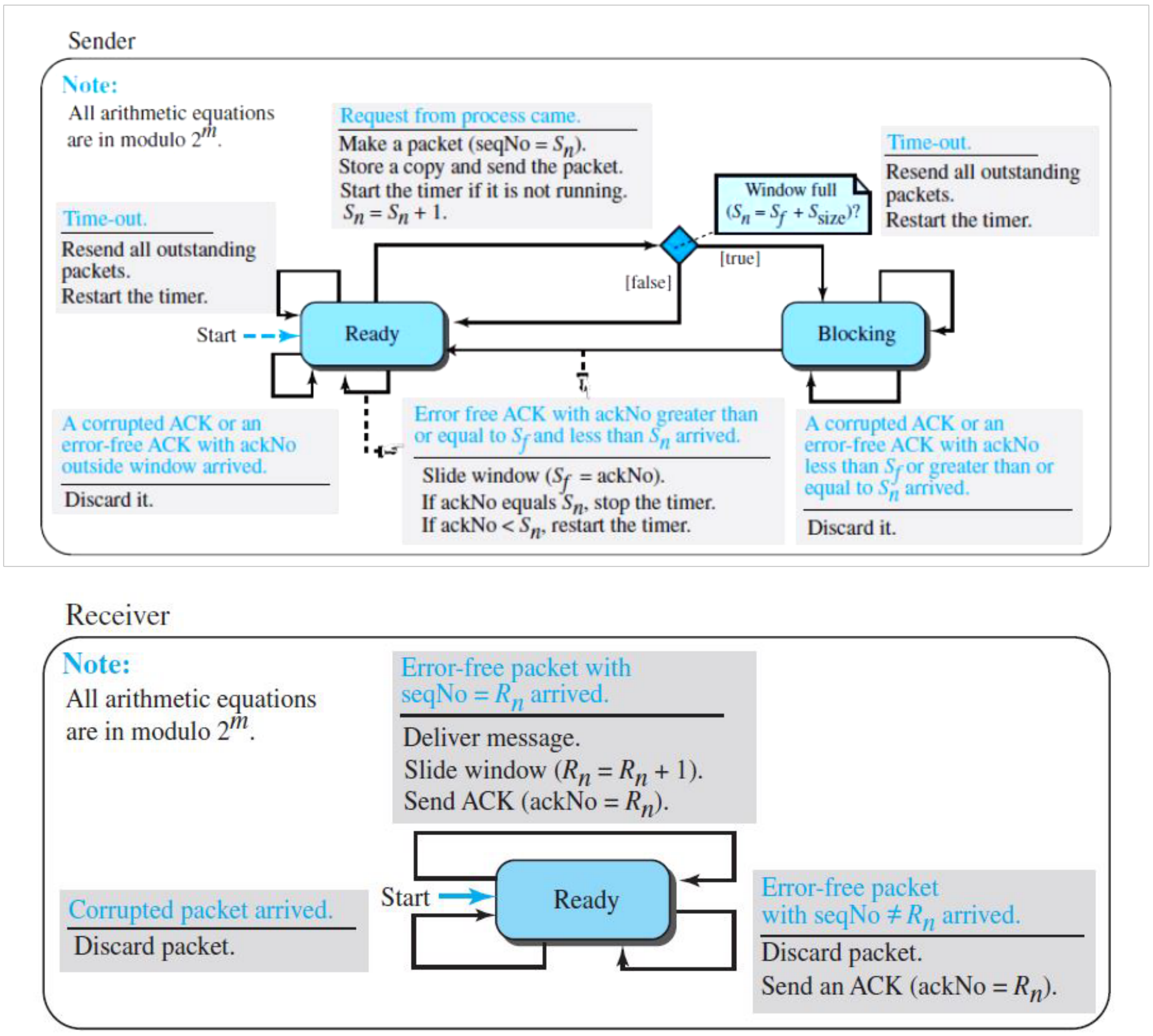

Sender
Note:
All arithmetic equations are in modulo 2^m.
Request from process came.
Make a packet (seqNo = S_n).
Store a copy and send the packet.
Start the timer if it is not running.
$S_n = S_n + 1$.
Window full ($S_n = S_f + S_{size}$)?
[true]
[false]
Time-out.
Resend all outstanding packets.
Restart the timer.
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. The receiver 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.

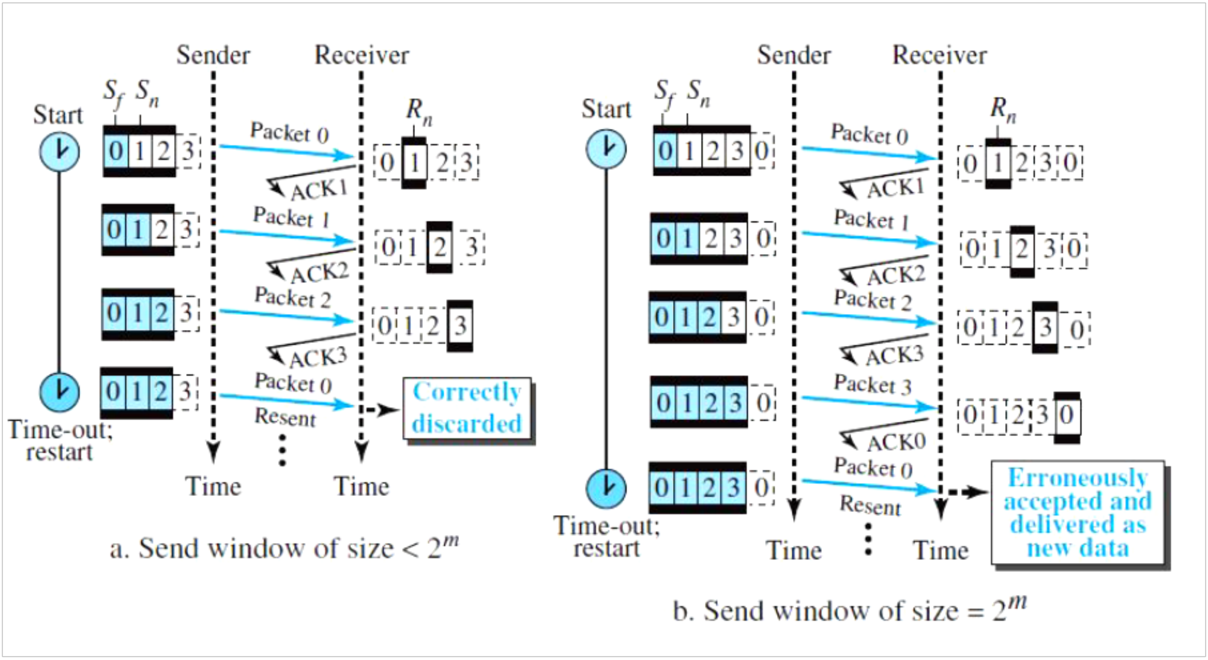


Figure 14 Send window size for Go-Back-N

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.

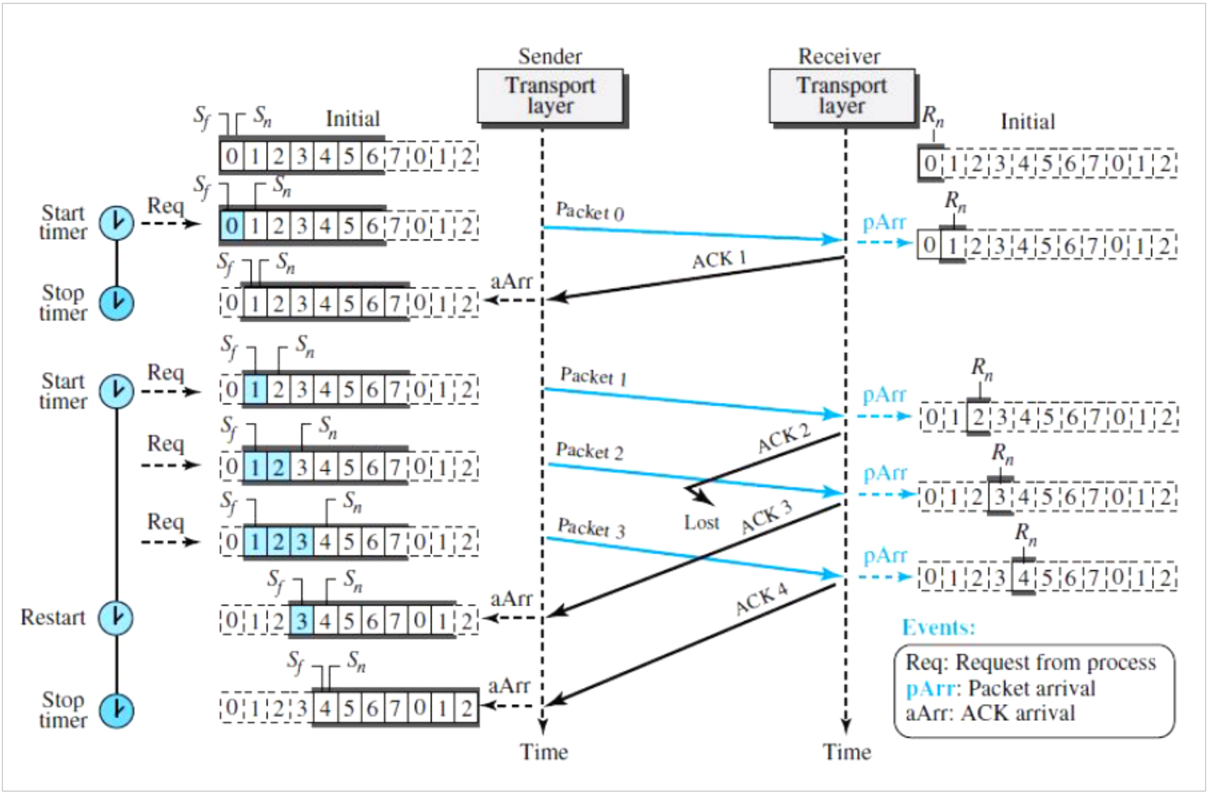


Figure15: Flow diagram for Example 1

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 S_f, not greater than S_f. 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.

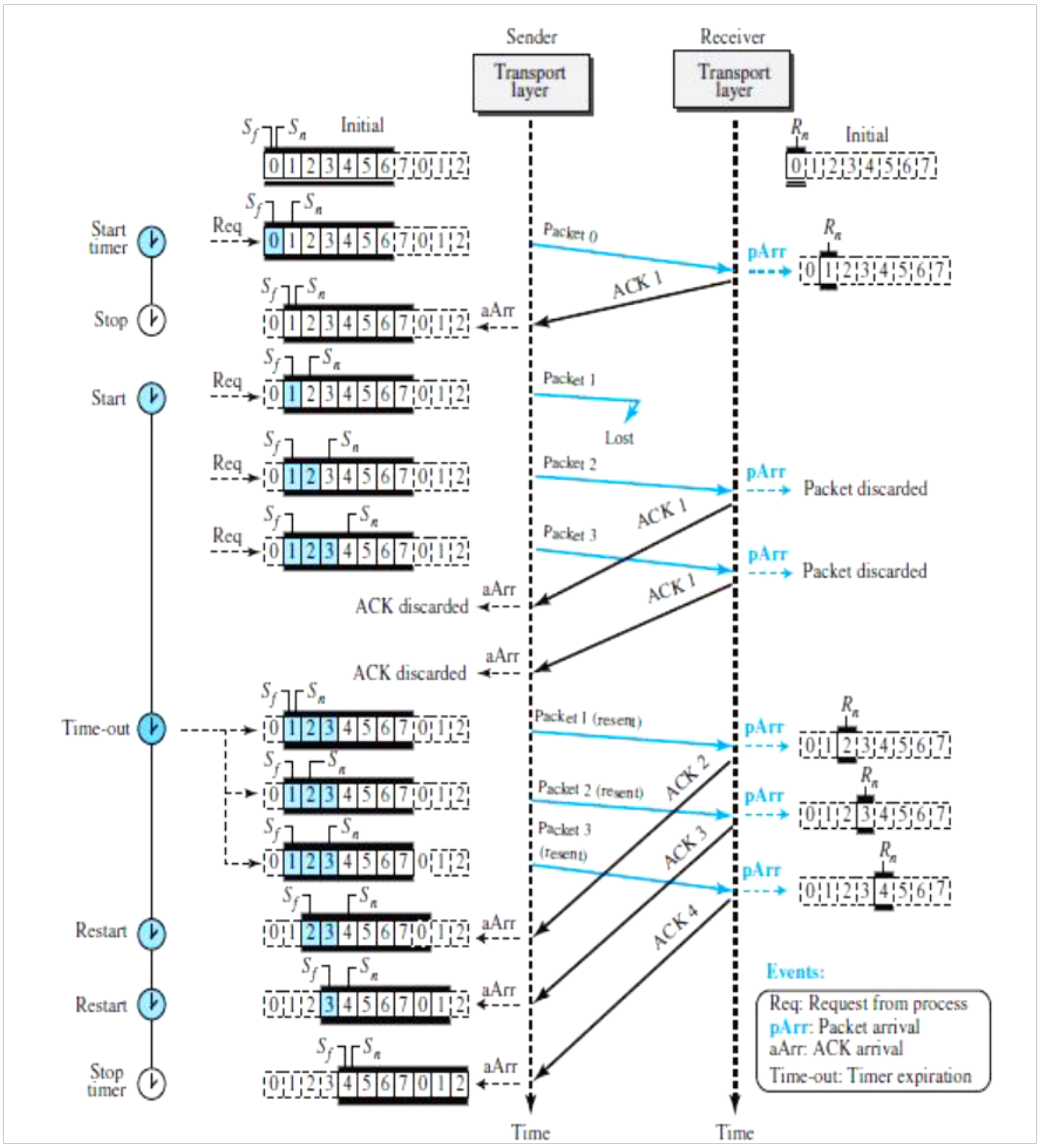


Figure 16: Flow diagram for Example 2

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.

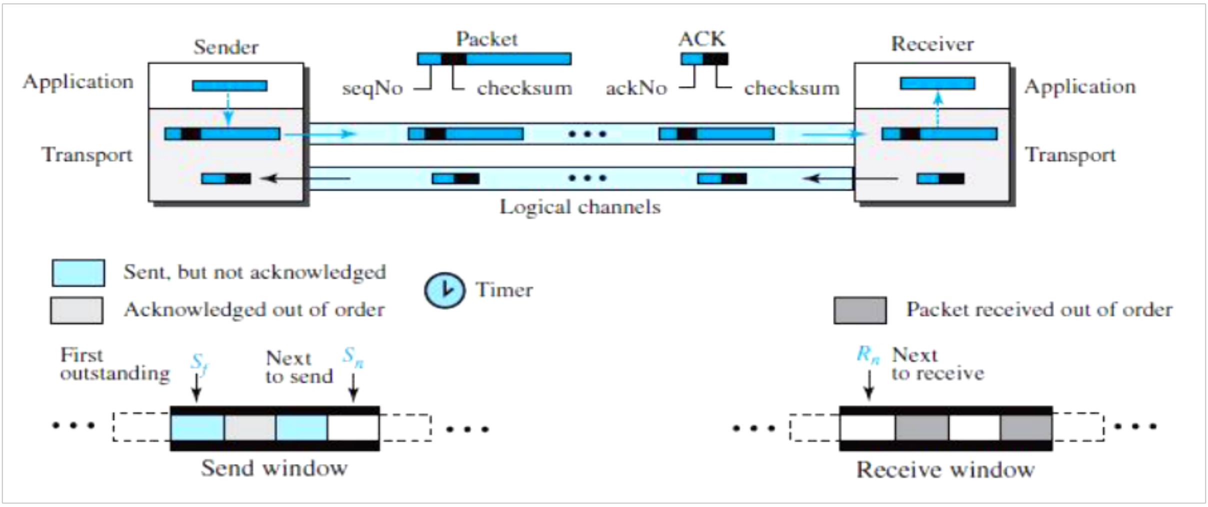


Figure 17: Outline of Selective-Repeat

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.

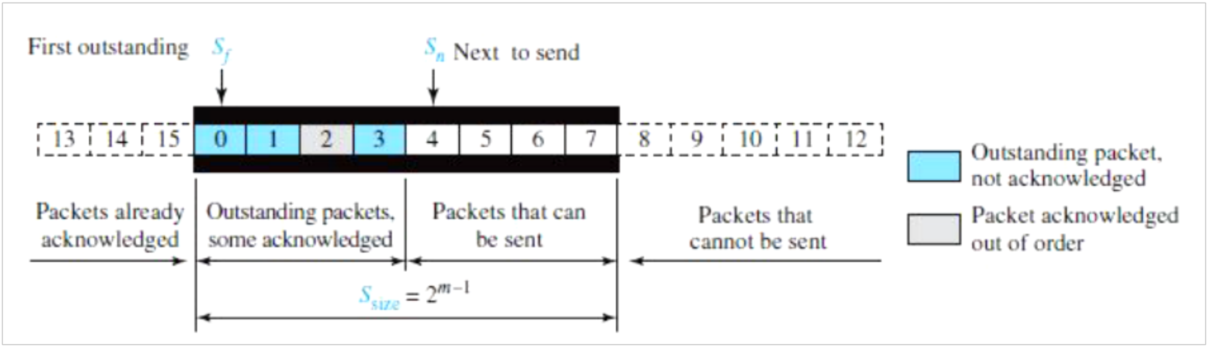

First outstanding
S_f
S_n Next to send
13
14
15
0
1
2
3
4
5
6
7
8
9
10
11
12
Packets already acknowledged
Outstanding packets, some acknowledged
Packets that can be sent
Packets that cannot be sent
$S_{size} = 2^{m-1}$
Outstanding packet, not acknowledged
Packet acknowledged out of order

Figure 18.1: Send window for Selective-Repeat protocol

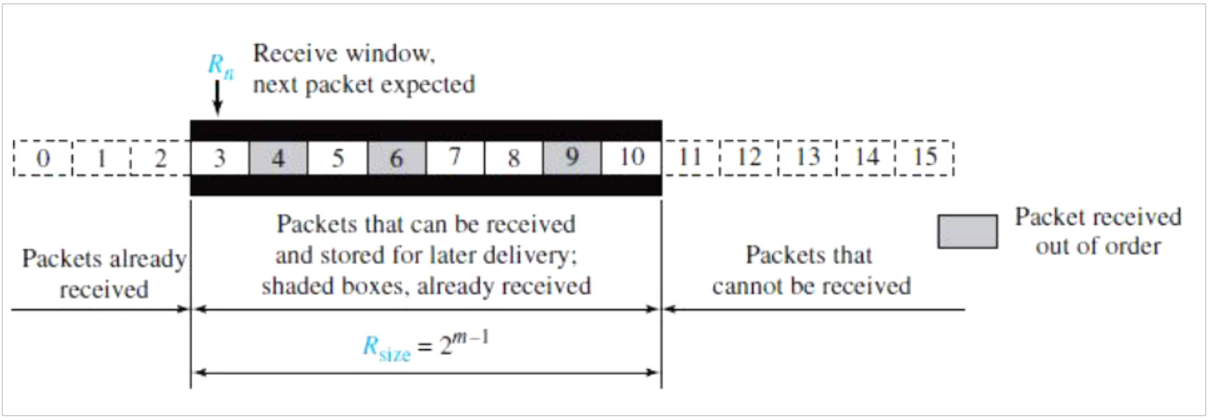

R_n
Receive window, next packet expected
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Packets already received
Packets that can be received and stored for later delivery; shaded boxes, already received
Packets that cannot be received
Packet received out of order
$R_{size} = 2^{m-1}$

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.

- **<u>Ready state</u>*.*** 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.

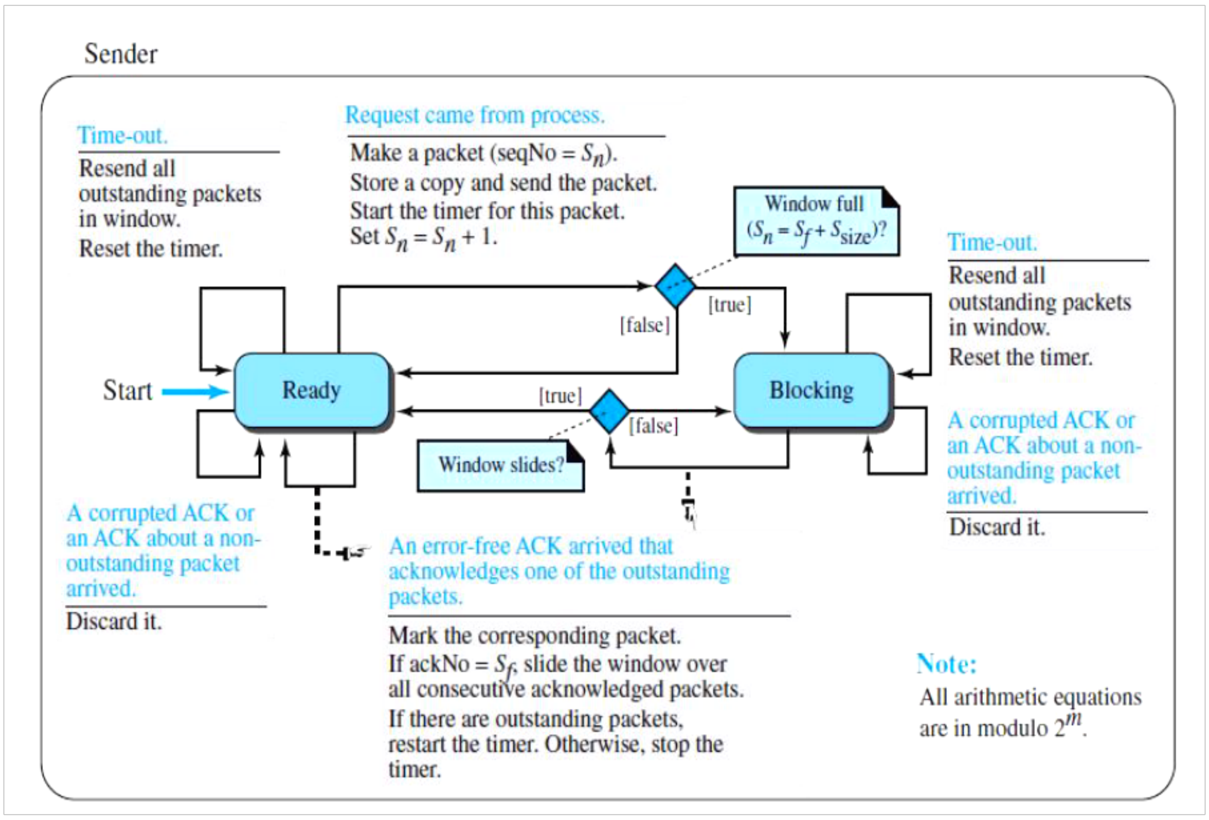
d. If a time-out occurs, the sender resends all unacknowledged packets in the window and restarts the timer.

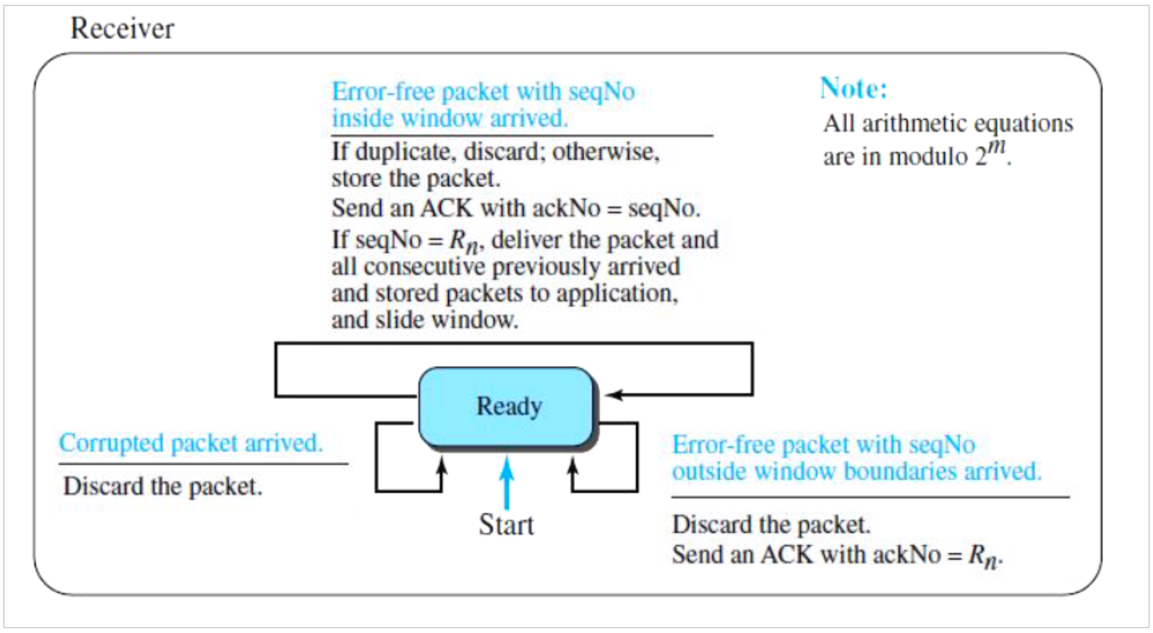
- **Blocking state*.*** Three events may occur in this case:

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 ($S_n = S_f + S_{size}$)?
[true]
[false]
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.
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.
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.

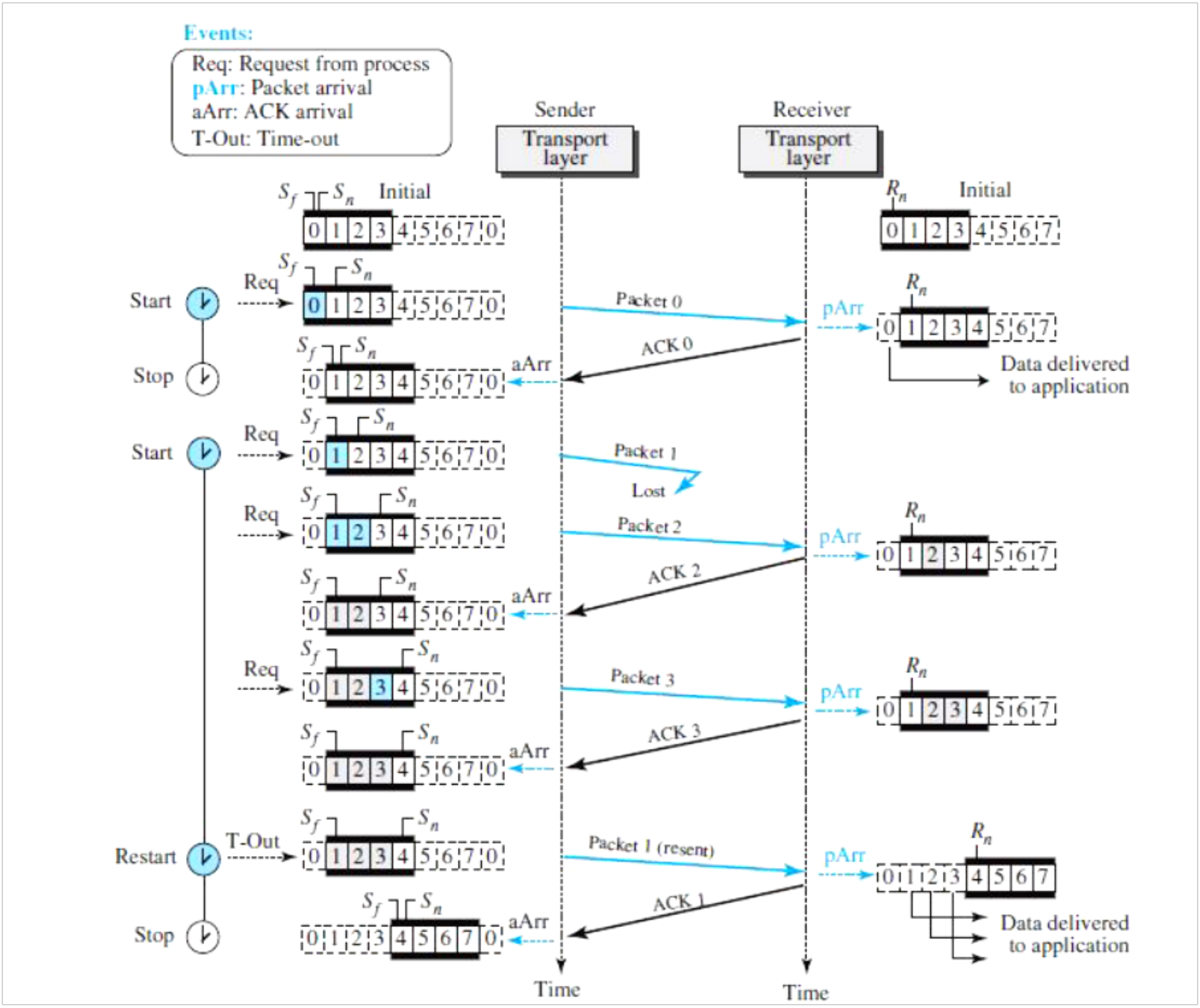


Figure 19: Flow diagram for Example 4

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.

4. **What are the services provided by UDP? Mention any four typical applications of UDP.**

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).

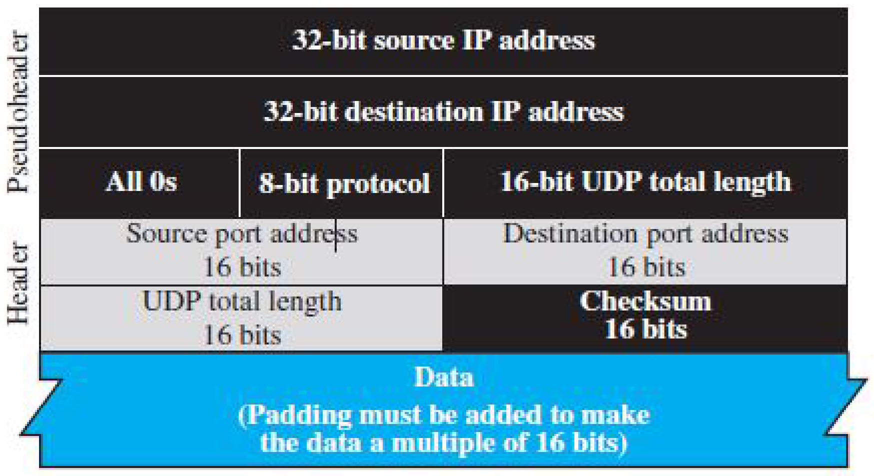

Pseudoheader
Header
32-bit source IP address
32-bit destination IP address
All 0s
8-bit protocol
16-bit UDP total length
Source port address 16 bits
Destination port address 16 bits
UDP total length 16 bits
Checksum 16 bits
Data
(Padding must be added to make the data a multiple of 16 bits)

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.

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 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.

5. What are the different TCP services and features? Explain them

- **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.

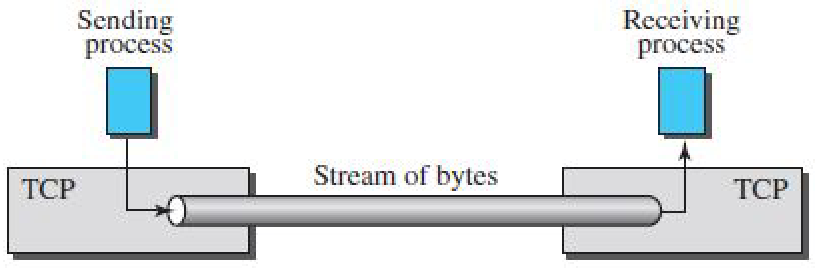


Figure 21: Stream delivery

- **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.

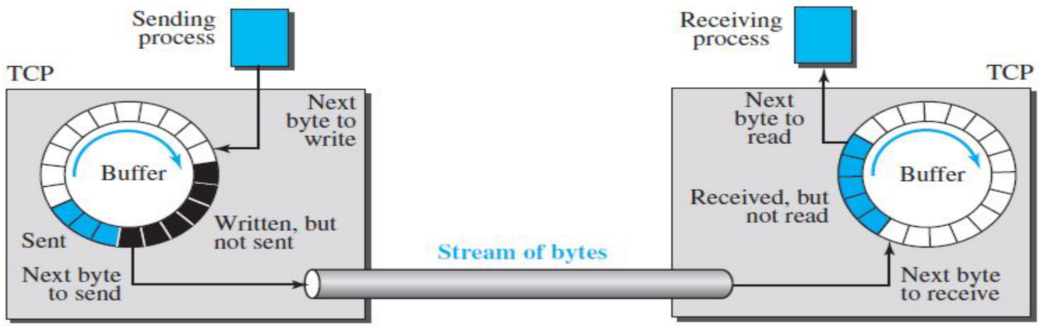


Figure 22: Sending and receiving buffers

- **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.

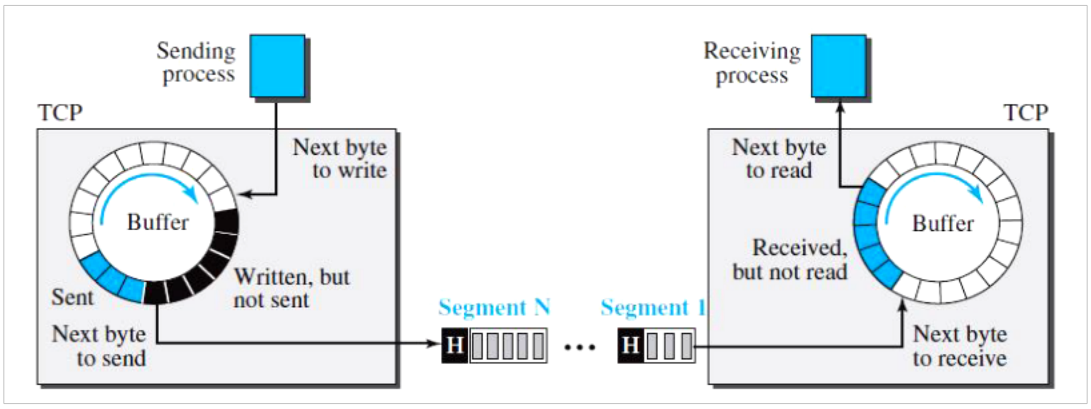


Figure 23: TCP segments

- **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.

6. With a neat diagram explain 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.

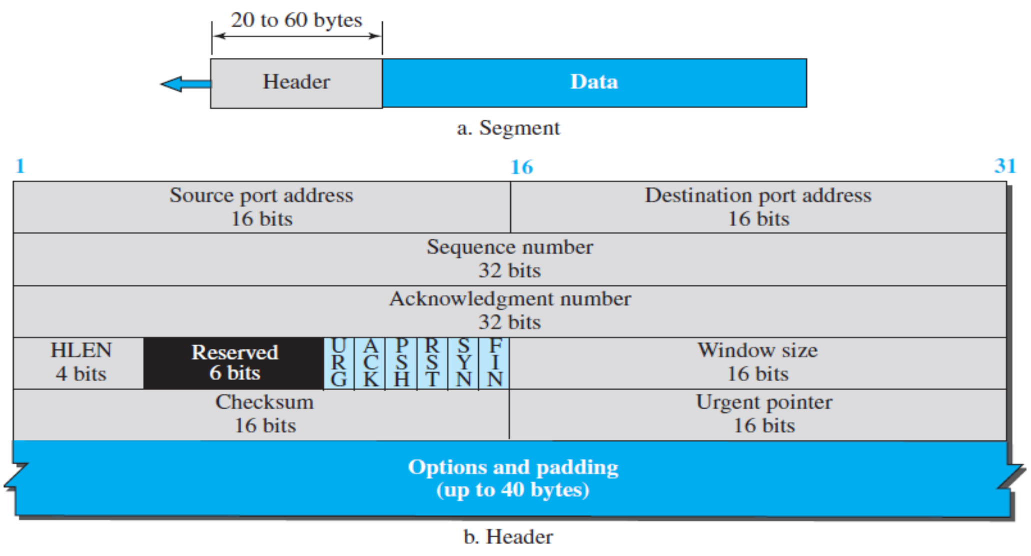


Figure 23.1: TCP segment format

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.

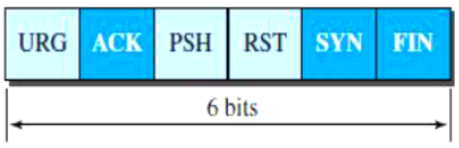
Sequence number. This 32-bit field defines the number assigned to the first byte of data contained in this segment. As we said before, TCP is a stream transport protocol. To ensure connectivity, each byte to be transmitted is numbered. The sequence number tells the destination which byte in this sequence 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. 24

Figure 24. *Control field*



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.

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.