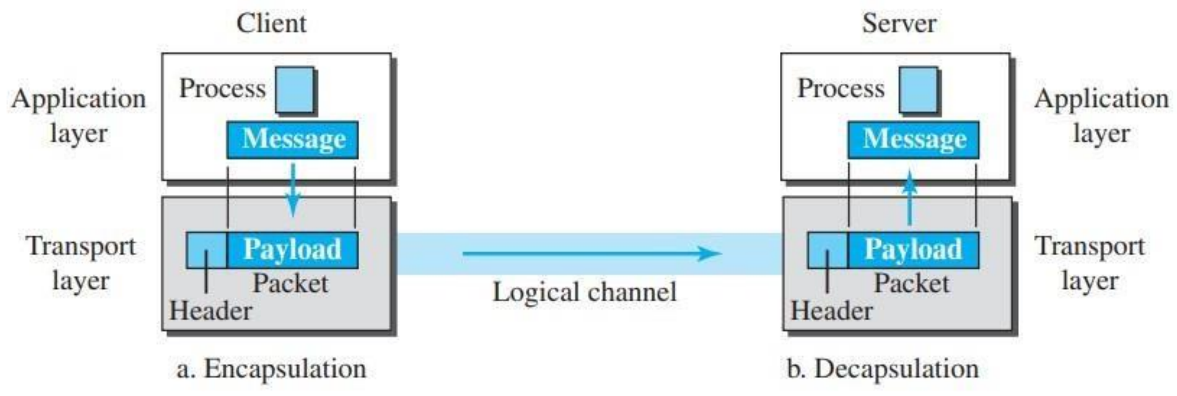


Encapsulation and decapsulation

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).

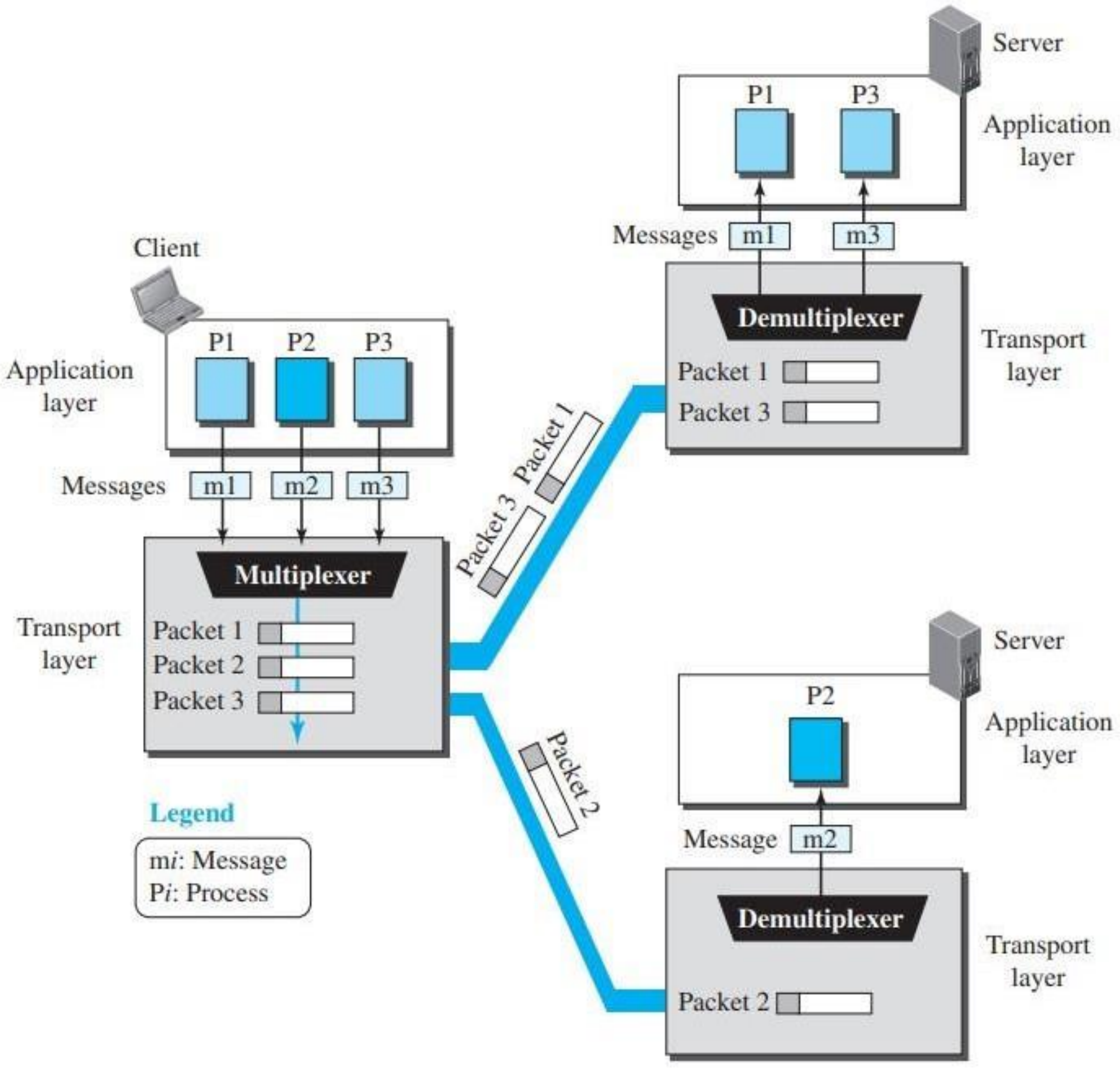
The transport layer at the source performs multiplexing; the transport layer at the destination performs demultiplexing. Figure shows communication between a client and two servers.

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.



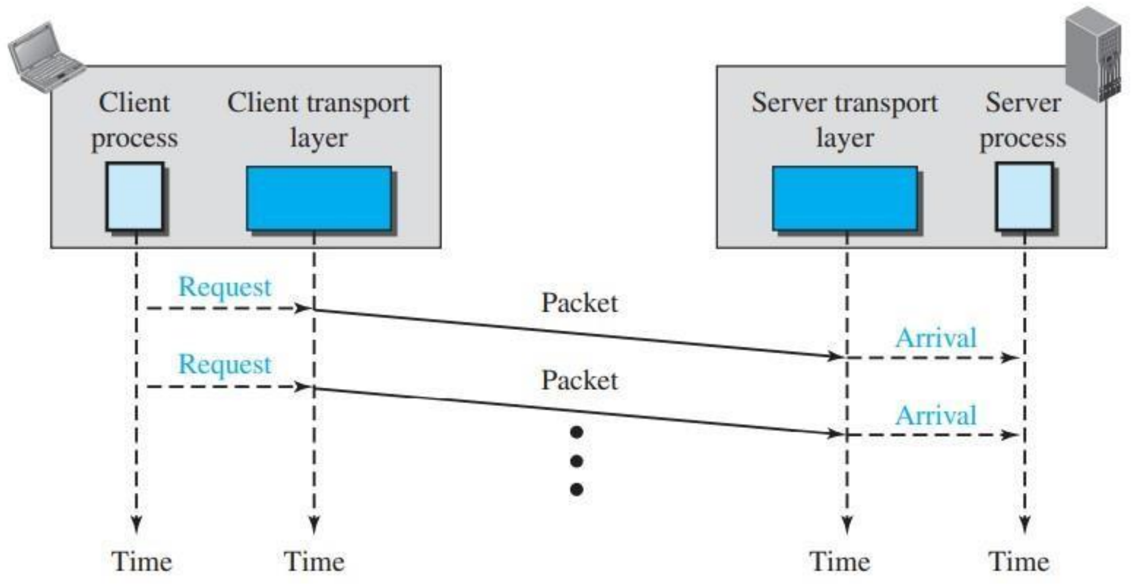
Multiplexing and demultiplexing

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 shows these two types of delivery.



Flow diagram **Stop-and-Wait Protocol**

The second protocol is a connection-oriented protocol called the Stop-and-Wait protocol, which uses both flow and error control.

Both the sender and the receiver use a sliding window of size 1. The sender sends one packet at a time and waits for an acknowledgment before sending the next one.

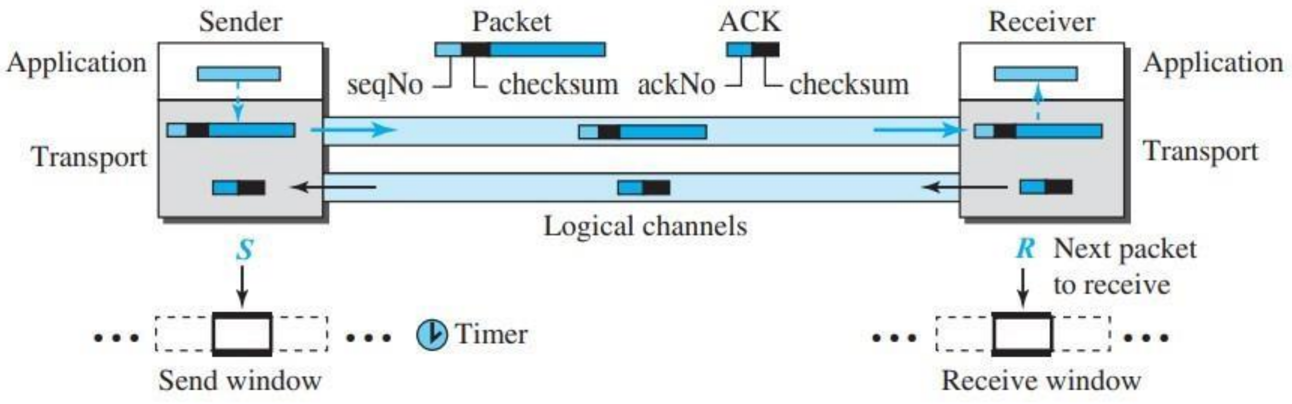
To detect corrupted packets, we need to add a checksum to each data packet. When a packet arrives at the receiver site, it is checked.

If its checksum is incorrect, the packet is corrupted and silently discarded. The silence of the receiver is a signal for the sender that a packet was either corrupted or lost.

Every time the sender sends a packet, it starts a timer. If an acknowledgment arrives before the timer expires, the timer is stopped, and the sender sends the next packet (if it has one to send).

If the timer expires, the sender resends the previous packet, assuming that the packet was either lost or corrupted. This means that the sender needs to keep a copy of the packet until its acknowledgment arrives.

Figure shows the outline for the Stop-and-Wait protocol. Note that only one packet and one acknowledgment can be in the channels at any time.



Stop-and-Wait Protocol

Sequence Numbers

To prevent duplicate packets, the protocol uses sequence numbers and acknowledgment numbers. A field is added to the packet header to hold the sequence number of that packet.

One important consideration is the range of the sequence numbers. Since we want to minimize the packet size, we look for the smallest range that provides unambiguous communication.

Assume we have used x as a sequence number; we only need to use x + 1 after that. There is no need for x + 2.

To show this, 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. The sender resends the packet (numbered x) after the time-out. Note that the packet here is a duplicate. The receiver can recognize this fact because it expects packet x + 1 but packet x was received.

We can see that there is a need for sequence numbers x and x + 1 because the receiver needs to distinguish between case 1 and case 3. But there is no need for a packet to be numbered x + 2.

In case 1, the packet can be numbered x again because packets x and x + 1 are acknowledged and there is no ambiguity at either site.

In cases 2 and 3, the new packet is x + 1, not x + 2. If only x and x + 1 are needed, we can let x = 0 and x + 1 = 1.

This means that the sequence is 0, 1, 0, 1, 0, and so on. This is referred to as modulo 2 arithmetic.

Acknowledgment Numbers

Since the sequence numbers must be suitable for both data packets and acknowledgments, we use this convention: 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).

The sender has a control variable, which we call S (sender), that points to the only slot in the send window. The receiver has a control variable, which we call R (receiver), that points to the only slot in the receive window.

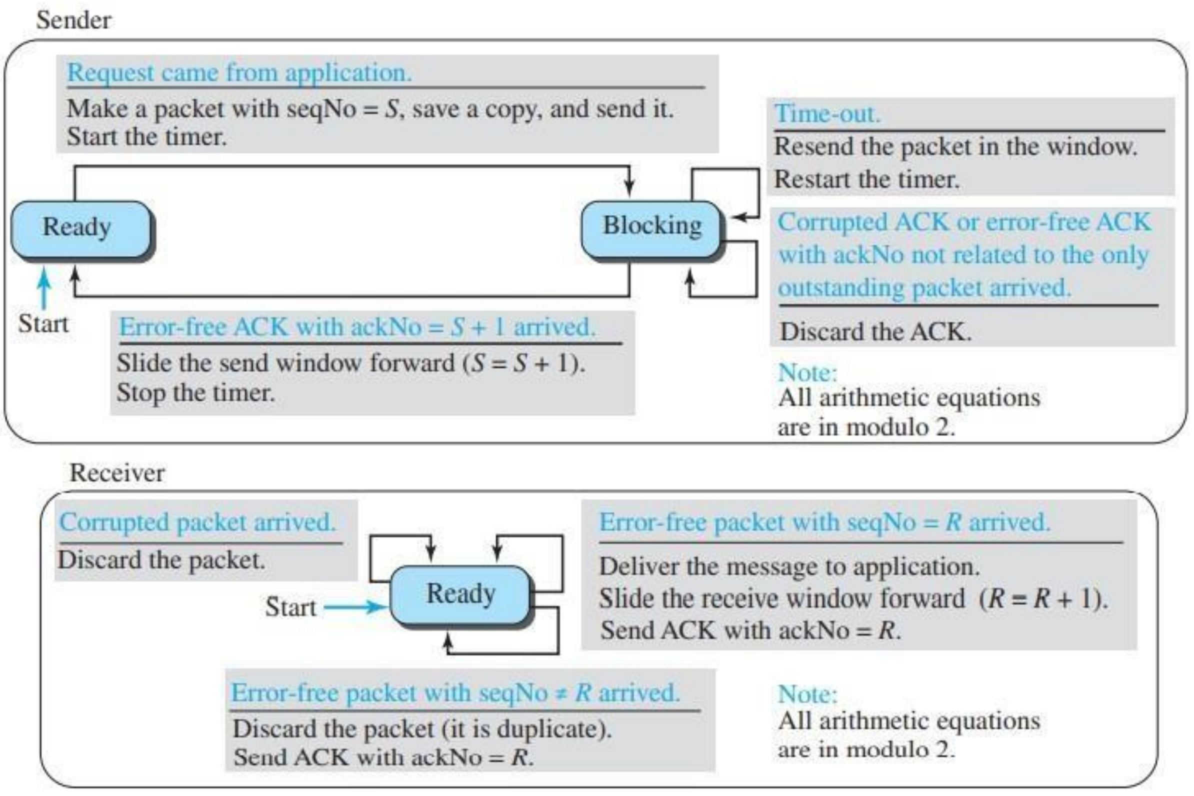
FSMs

Figure shows the FSMs for the Stop-and-Wait protocol. Since the protocol is a connection-oriented protocol, both ends should be in the established state before exchanging data packets. The states are actually nested in the established state.

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.

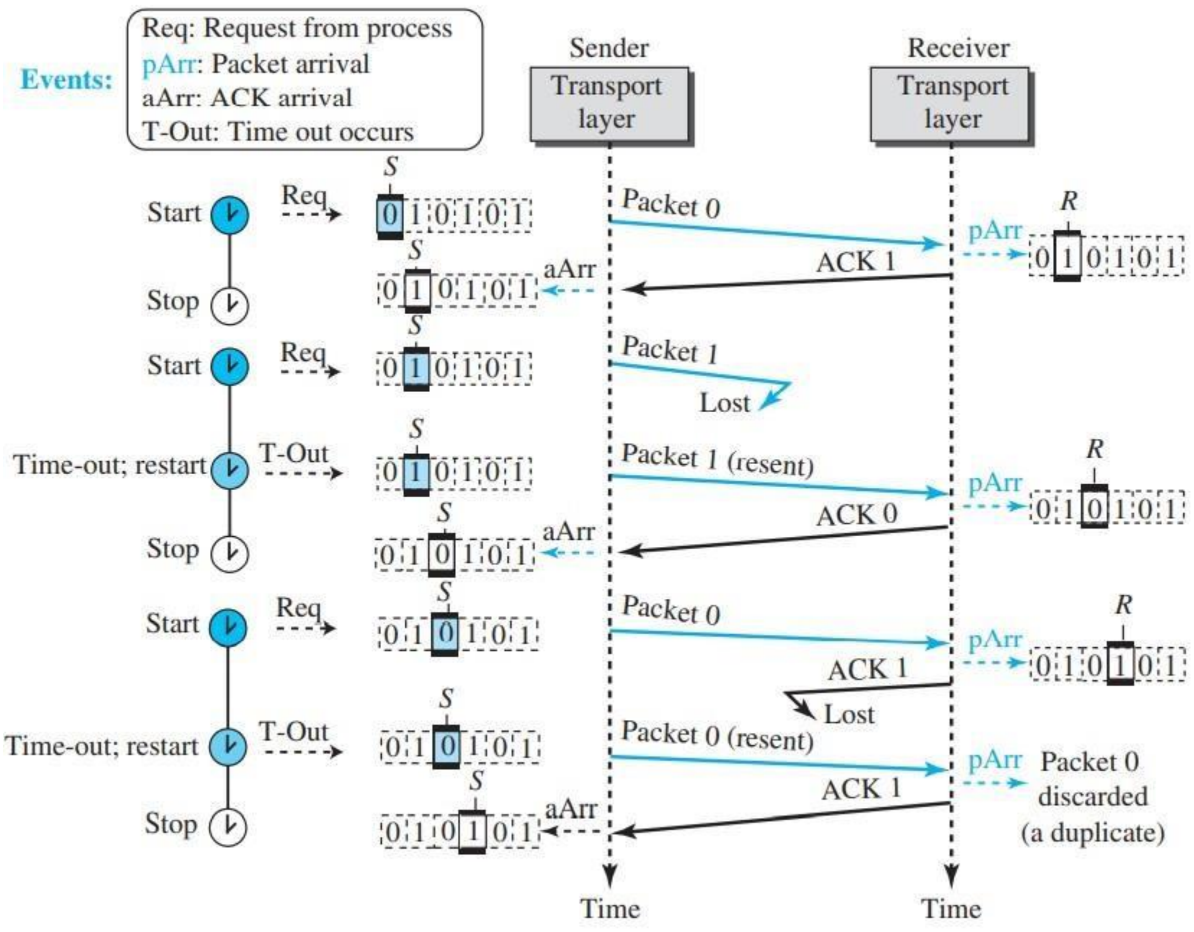
1. Ready state: When the sender is in this state, it is only waiting for one event to occur. If a request comes from the application layer, the sender creates a packet with the sequence number set to S. A copy of the packet is stored, and the packet is sent. The sender then starts the only timer. The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

1. 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.
2. If an error-free packet with seqNo ≠ R arrives, the packet is discarded, but an ACK with ackNo = R is sent.
3. 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
Corrupted ACK or error-free ACK with ackNo not related to the only outstanding packet arrived.
Discard the ACK.
Start
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.
Start
Ready
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.

FSMs for the Stop-and-Wait protocol

Figure shows an example of the Stop-and-Wait protocol. Packet 0 is sent and acknowledged. Packet 1 is lost and resent after the time-out. The resent packet 1 is acknowledged and the timer stops. Packet 0 is sent and acknowledged, but the acknowledgment is lost. The sender has no idea if the packet or the acknowledgment is lost, so after the time-out, it resends packet 0, which is acknowledged.



Flow diagram

Efficiency

The Stop-and-Wait protocol is very inefficient if our channel is thick and long.
By thick, we mean that our channel has a large bandwidth (high data rate); by long, we mean the roundtrip delay is long.
The product of these two is called the bandwidth delay product. We can think of the channel as a pipe. The bandwidth-delay product then is the volume of the pipe in bits. The pipe is always there. It is not efficient if it is not used.
The bandwidth-delay product is a measure of the number of bits a sender can transmit through the system while waiting for an acknowledgment from the receiver.

Example

*Assume that, in a Stop-and-Wait system, the bandwidth of the line is 1 Mbps, and 1 bit takes 20 milliseconds to make a round trip. What is the bandwidth-delay product? If the system data packets are 1,000 bits in length, what is the utilization percentage of the link?

Solution

The bandwidth-delay product is (1 × 106) × (20 × 10−3) = 20,000 bits.

The system can send 20,000 bits during the time it takes for the data to go from the sender to the receiver and the acknowledgment to come back.

However, the system sends only 1,000 bits.

We can say that the link utilization is only 1,000/20,000, or 5 percent.

For this reason, in a link with a high bandwidth or long delay, the use of Stop-and-Wait wastes the capacity of the link

*What is the utilization percentage of the link in above Example if we have a protocol that can send up to 15 packets before stopping and worrying about the acknowledgments?

Solution

The bandwidth-delay product is still 20,000 bits.

The system can send up to 15 packets or 15,000 bits during a round trip.

This means the utilization is 15,000/20,000, or 75 percent.

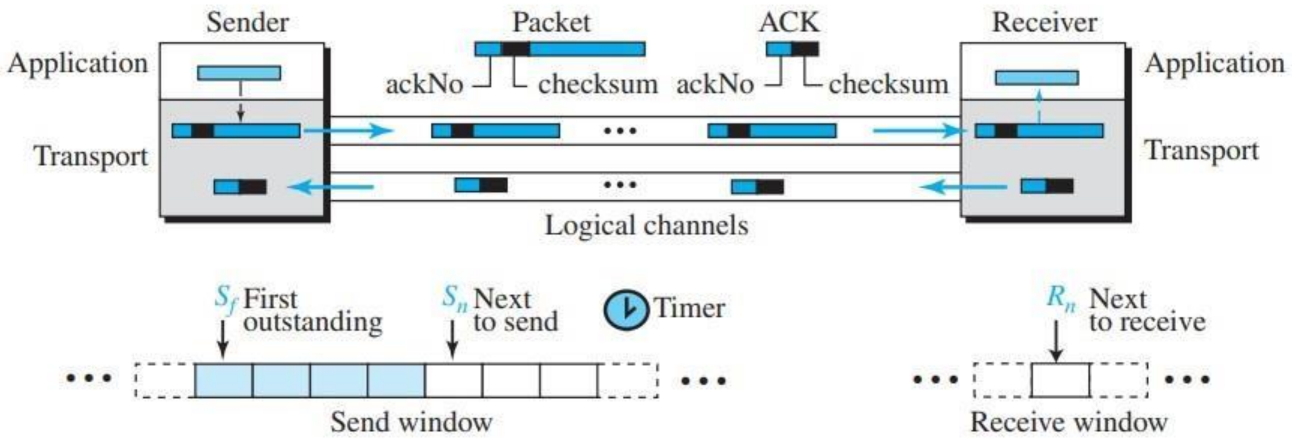
If there are damaged packets, the utilization percentage is much less because packets have to be resent.

Pipelining

In networking and in other areas, a task is often begun before the previous task has ended. This is known as pipelining.
There is no pipelining in the Stop-and-Wait protocol because a sender must wait for a packet to reach the destination and be acknowledged before the next packet can be sent.
Pipelining does apply to our next two protocols because several packets can be sent before a sender receives feedback about the previous packets.
Pipelining improves the efficiency of the transmission if the number of bits in transition is large with respect to the bandwidth delay product

Go-Back-N Protocol (GBN)

To improve the efficiency of transmission (to fill the pipe), multiple packets must be in transition while the sender is waiting for acknowledgment.
We need to let more than one packet be outstanding to keep the channel busy while the sender is waiting for acknowledgment.
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 shows the outline of the protocol. Note that several data packets and acknowledgments can be in the channel at the same time.



Go-Back-N protocol

Sequence Numbers

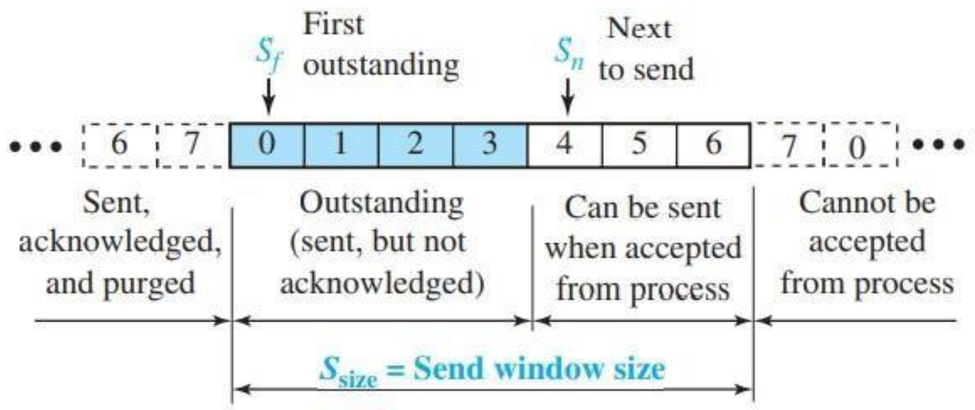
The sequence numbers are modulo 2^m, where m is the size of the sequence number field in bits.

Acknowledgment Numbers

An acknowledgment number in this protocol is cumulative and defines the sequence number of the next packet expected.
For example, if the acknowledgment number (ackNo) is 7, it means all packets with sequence number up to 6 have arrived, safe and sound, and the receiver is expecting the packet with sequence number 7.

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 these 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, but we will see later that some protocols may have a variable window size.
Figure 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

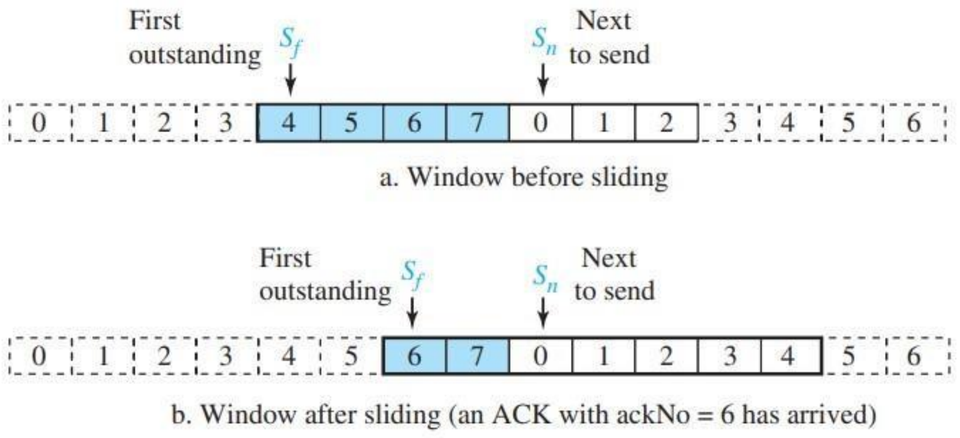
Send window for Go-Back-N

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, coloured, 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. We call these 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.
The window itself is an abstraction; three variables define its size and location at any time. We call these variables S_f (send window, the first outstanding packet), S_n (send window, the next packet to be sent), and S_{size} (send window, size).
The variable S_f defines the sequence number of the first (oldest) outstanding packet. The variable S_n holds the sequence number that will be assigned to the next packet to be sent. Finally, the variable S_{size} defines the size of the window, which is fixed in our protocol.

The send window is an abstract concept defining an imaginary box of maximum size = $2^m - 1$

with three variables: S_f, S_n, and S_{size}.

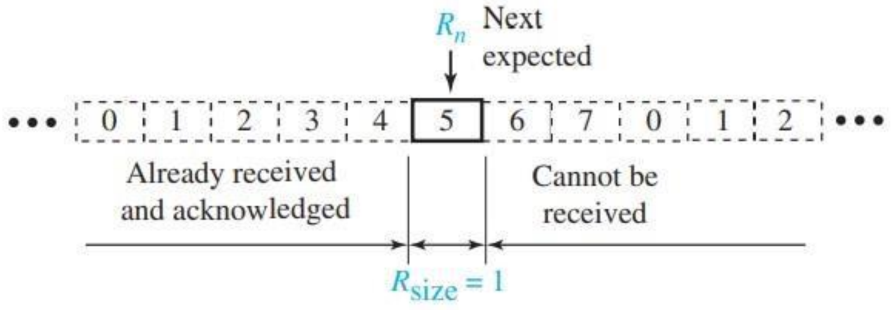
Figure shows how a send window can slide one or more slots to the right when an acknowledgment arrives from the other end. In the figure, an acknowledgment with ackNo = 6 has arrived. This means that the receiver is waiting for packets with sequence number 6.



Sliding the send window *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 shows the receive window. Note that we need only one variable, R_n(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 R_n 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, $R_n = (R_n + 1)$ modulo 2^m.



Receive window for Go-Back-N

Timers

Although there can be a timer for each packet that is sent, in the protocol we use only one. The reason is that the timer for the first outstanding packet always expires first. We resend all outstanding packets when this timer expires.

Resending packets

When the timer expires, the sender resends all outstanding packets.
For example, suppose the sender has already sent packet 6 ($S_n = 7$), but the only timer expires. If $S_f = 3$, this means that packets 3, 4, 5, and 6 have not been acknowledged; the sender goes back and resends packets 3, 4, 5, and 6.
That is why the protocol is called Go-Back-N. On a time-out, the machine goes back N locations and resends all packets.

FSMs

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 ($S_f = S_n = 0$).

a. 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 S_n . 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 S_n is now incremented, ($S_n = S_n + 1$) modulo 2^m. If the window is full, $S_n = (S_f + S_{size})$ 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, the sender slides the window (set S_f = ackNo), and if all outstanding packets are acknowledged (ackNo = S_n), 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 ackNo not related to the outstanding packet arrives, it is discarded.

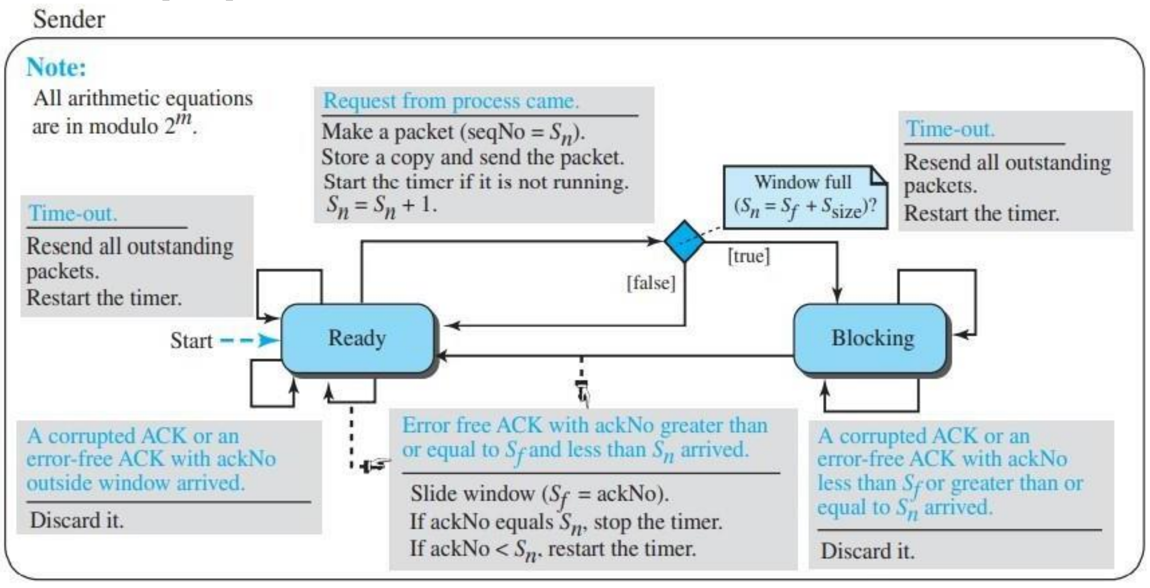
d. If a time-out occurs, the sender resends all outstanding packets and restarts the timer.

b. Blocking state. Three events may occur in this case:

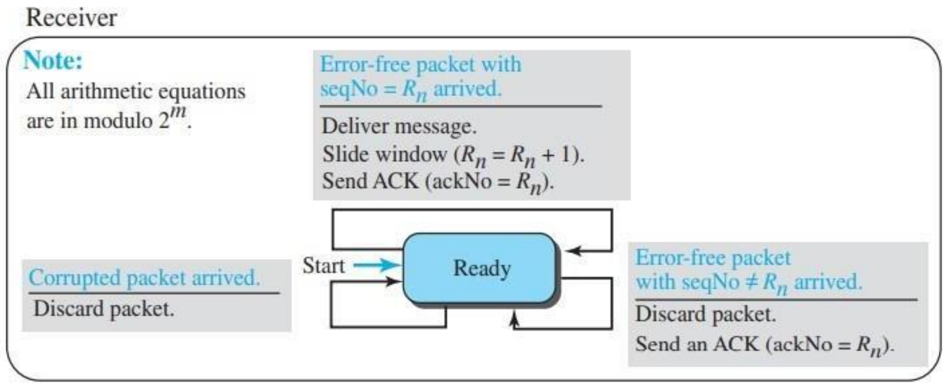
   a. If an error-free ACK arrives with ackNo related to one of the outstanding packets, the sender slides the window (set S_f = ackNo) and if all outstanding packets are acknowledged (ackNo = S_n), then the timer is stopped. If all outstanding packets are not acknowledged, the timer is restarted. The sender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

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

a. If an error-free packet with seqNo = R_n arrives, the message in the packet is delivered to the application layer. The window then slides, $R_n = (R_n + 1)$ modulo 2^m. Finally, an ACK is sent with ackNo = R_n.
b. If an error-free packet with seqNo outside the window arrives, the packet is discarded, but an ACK with ackNo = R_n is sent.
c. If a corrupted packet arrives, it is discarded.



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.

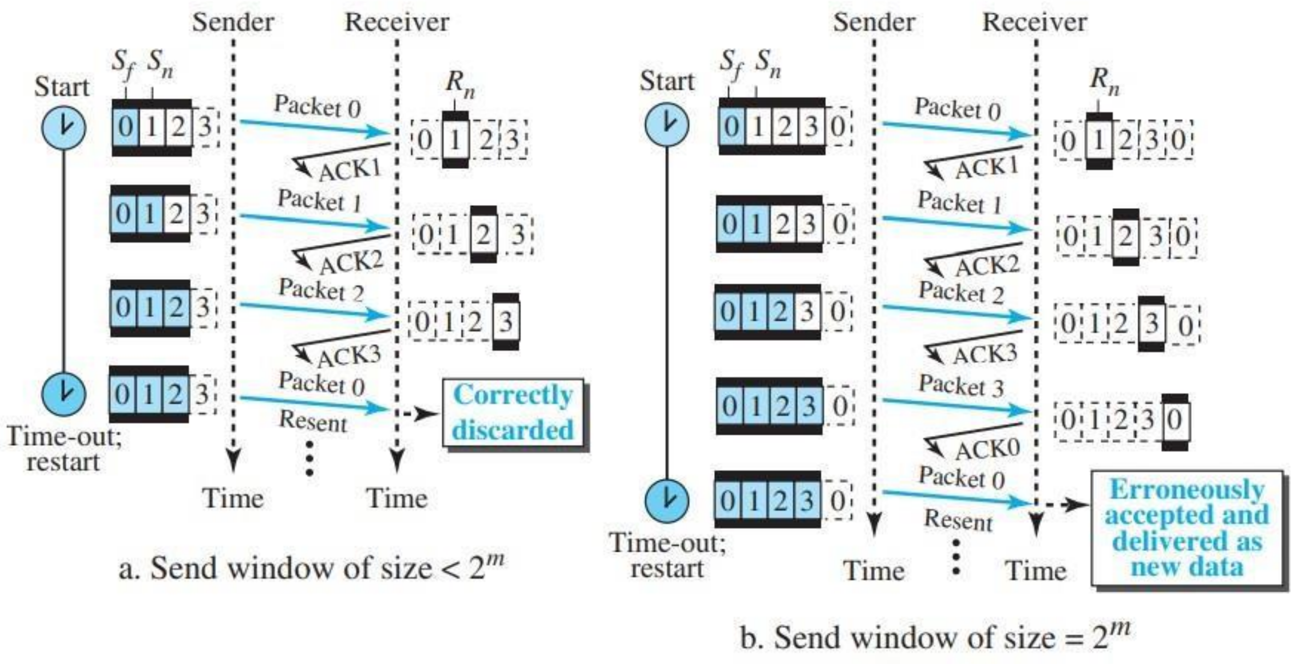


Receiver
Note:
All arithmetic equations are in modulo 2^m.
Error-free packet with seqNo = R_n arrived.
Deliver message.
Slide window ($R_n = R_n + 1$).
Send ACK (ackNo = R_n).
Start
Ready
Corrupted packet arrived.
Discard packet.
Error-free packet with seqNo ≠ R_n arrived.
Discard packet.
Send an ACK (ackNo = R_n).

FSMs for the Go-Back-N protocol

Send Window Size

We can now show why the size of the send window must be less than 2^m. As an example, we choose m = 2, which means the size of the window can be $2^m - 1$, or 3.
Figure 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 2^2) 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.



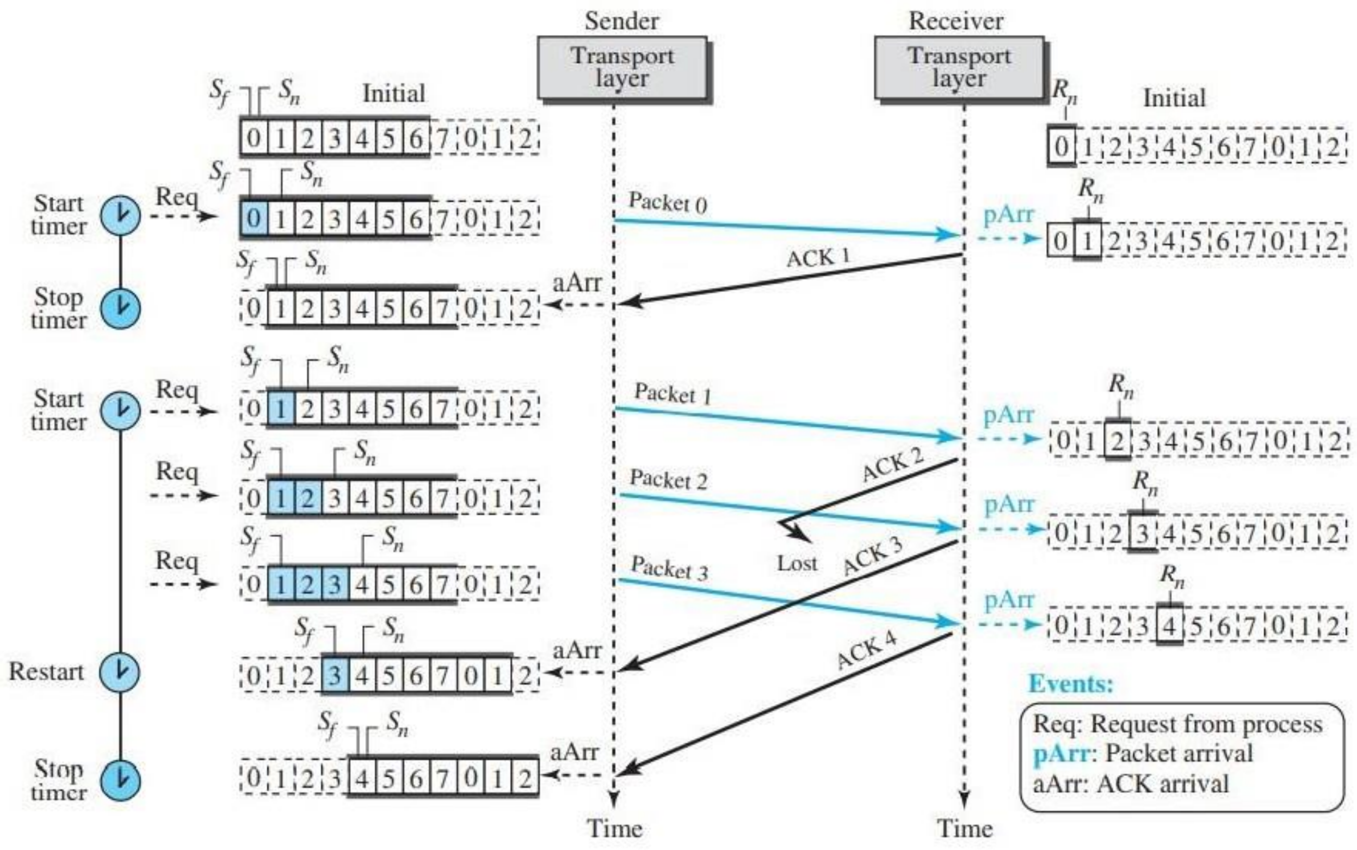
Send window size for Go-Back-N

Figure 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.

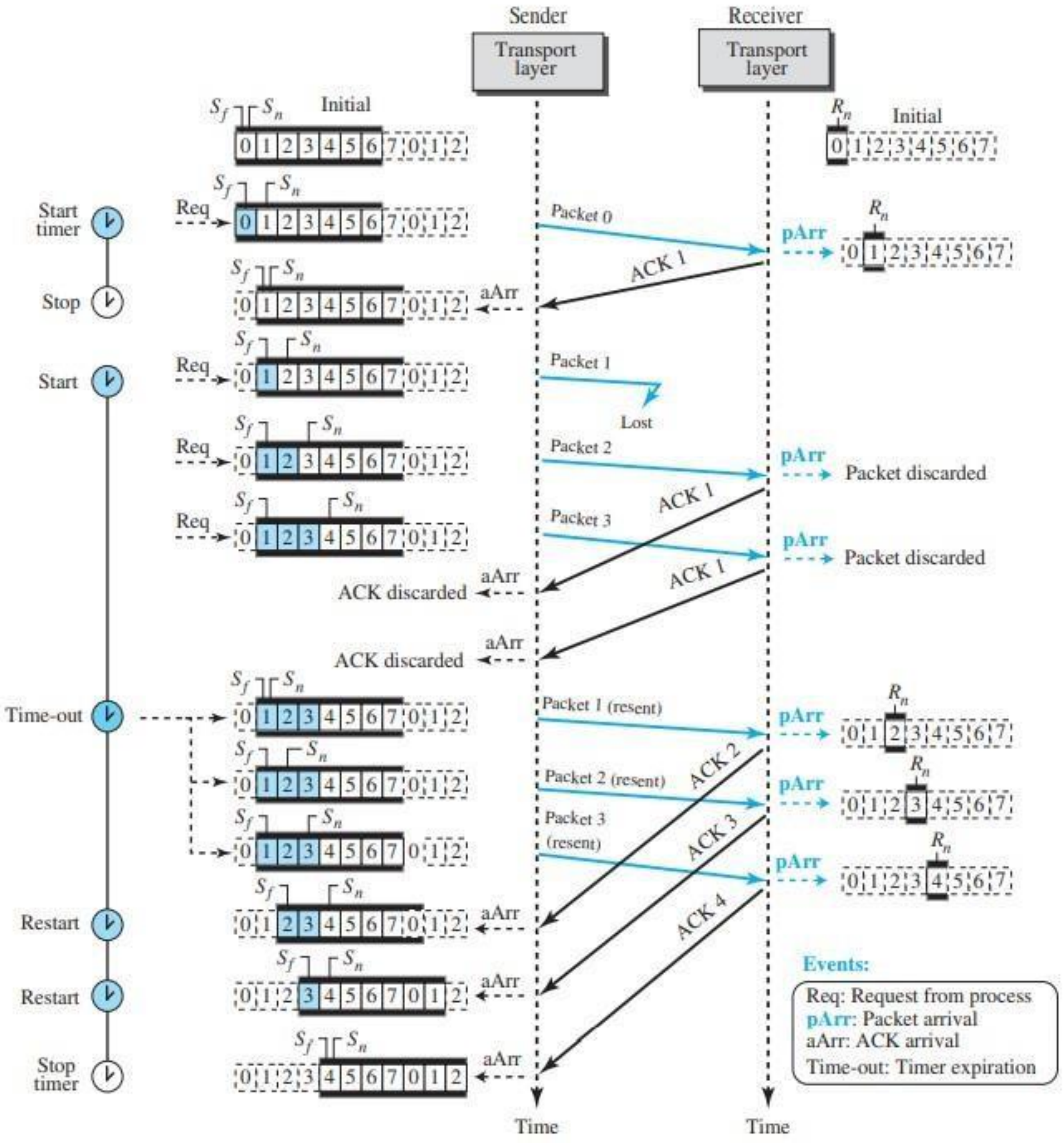
After initialization, there are some sender events. Request events are triggered by message chunks from the application layer; arrival events are triggered by ACKs received from the network layer.

There is no time-out event here because all outstanding packets are acknowledged before the timer expires. Note that 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.



Flow diagram

Figure below 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

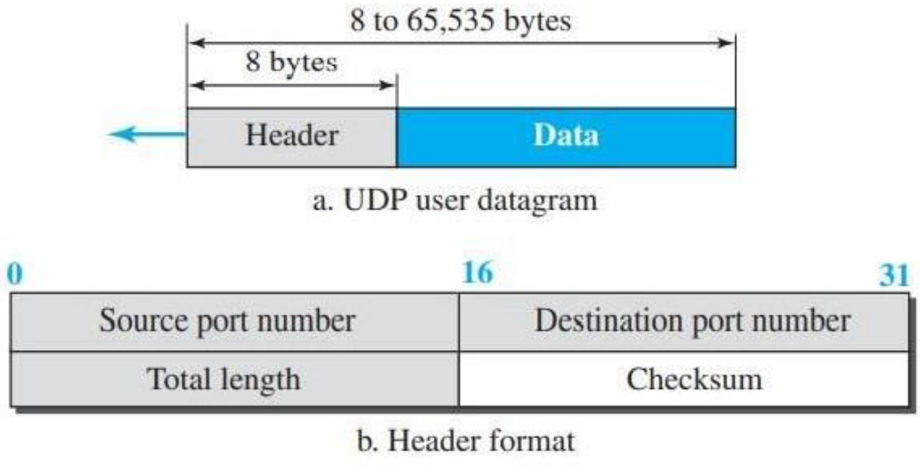
The reader may find that there is a similarity between the Go-Back-N protocol and the Stop-and-Wait protocol.
The Stop-and-Wait protocol is actually a Go-Back-N protocol in which there are only two sequence numbers, and the send window size is 1.

<u>USER DATAGRAM PROTOCOL</u>

The User Datagram Protocol (UDP) is a connectionless, unreliable transport protocol.
It does not add anything to the services of IP except for providing process-to-process communication instead of host-to-host communication.
UDP is a simple protocol using a minimum of overhead. If a process wants to send a small message and does not care much about reliability, it can use UDP.
Sending a small message using UDP takes much less interaction between the sender and receiver than using TCP.

User Datagram

UDP packets, called user datagrams, have a fixed-size header of 8 bytes made of four fields, each of 2 bytes (16 bits). Figure shows the format of a user datagram.
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.
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

Example

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

CB84000D001C001C

i. What is the source port number?
ii. What is the destination port number? iii.
What is the total length of the user
datagram? iv. What is the length of the
data?
v. Is the packet directed from a client to a server or vice versa?
vi. What is the client process?

Solution

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

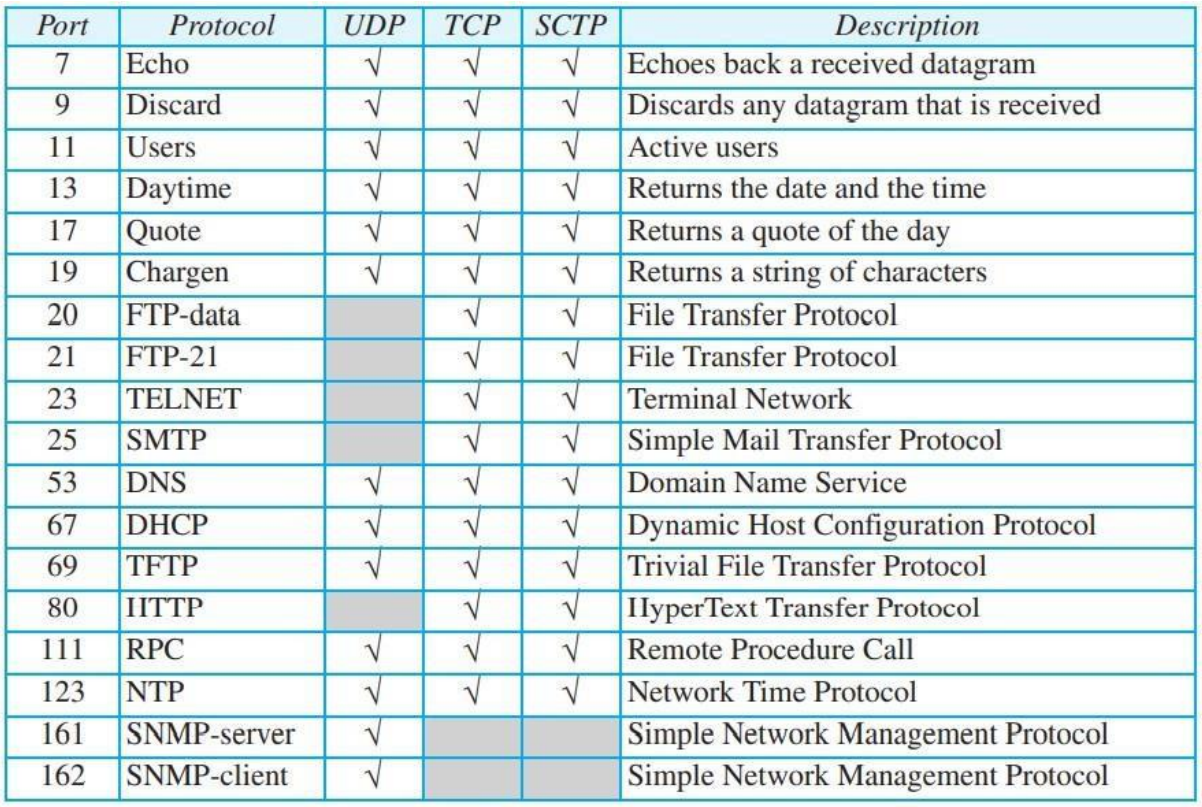
ii. The destination port number is the second four hexadecimal digits (000D)16, which means that the destination port number is 13.

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

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

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

vi. The client process is the Daytime



Port	*Protocol*	*UDP*	*TCP*	*SCTP*	*Description*
7	Echo	√	√	√	Echoes back a received datagram
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
21	FTP-21		√	√	File Transfer Protocol
23	TELNET		√	√	Terminal Network
25	SMTP		√	√	Simple Mail Transfer Protocol
53	DNS	√	√	√	Domain Name Service
67	DHCP	√	√	√	Dynamic Host Configuration Protocol
69	TFTP	√	√	√	Trivial File Transfer Protocol
80	HTTP		√	√	HyperText Transfer Protocol
111	RPC	√	√	√	Remote Procedure Call
123	NTP	√	√	√	Network Time Protocol
161	SNMP-server	√			Simple Network Management Protocol
162	SNMP-client	√			Simple Network Management Protocol

Some well-known ports used with UDP and TCP

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

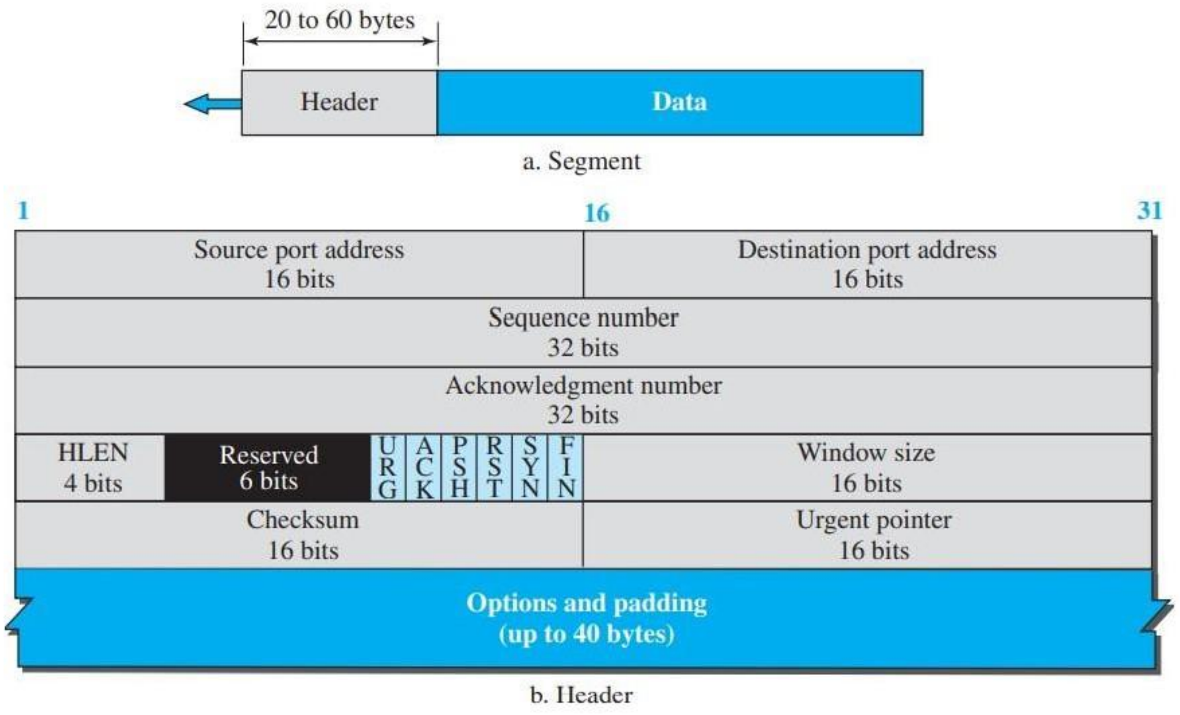
UDP provides a connectionless service. This means that each user datagram sent by UDP is an independent datagram.

Note that this does not mean that the party has received 5642 bytes, because the first byte number does not have to be 0.

Segment

Format

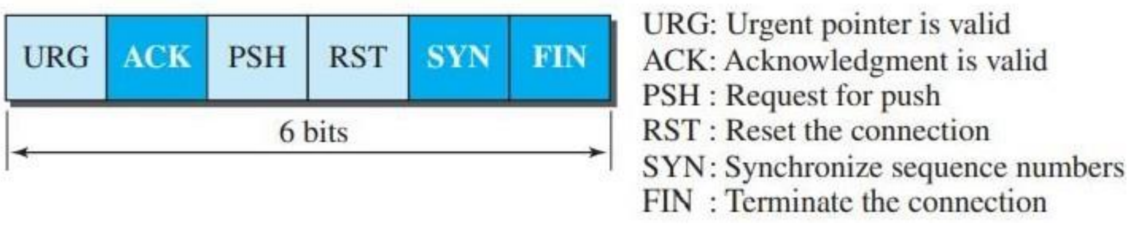
A packet in TCP is called a segment. The format of a segment is shown in Figure. The segment consists of a header of 20 to 60 bytes, followed by data from the application program. The header is 20 bytes if there are no options and up to 60 bytes if it contains options.



TCP segment format

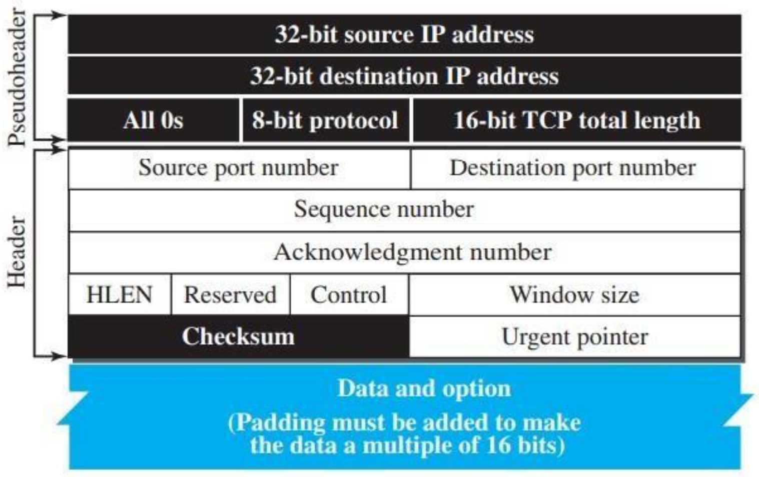
- Source port address: This is a 16-bit field that defines the port number of the application program in the host that is sending the segment.
- Destination port address: This is a 16-bit field that defines the port number of the application program in the host that is receiving the segment.
- Sequence 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 (discussed later) 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.


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

Control field

- Window size: This field defines the window size of the sending TCP in bytes. Note that the length of this field is 16 bits, which means that the maximum size of the window is 65,535 bytes. This value is normally referred to as the receiving window (rwnd) and is determined by the receiver. 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. The same pseudoheader, serving the same purpose, is added to the segment. For the TCP pseudoheader, the value for the protocol field is 6.



Pseudoheader added to the TCP datagram

- 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. We will discuss some of the options used in the TCP header later in the section.

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

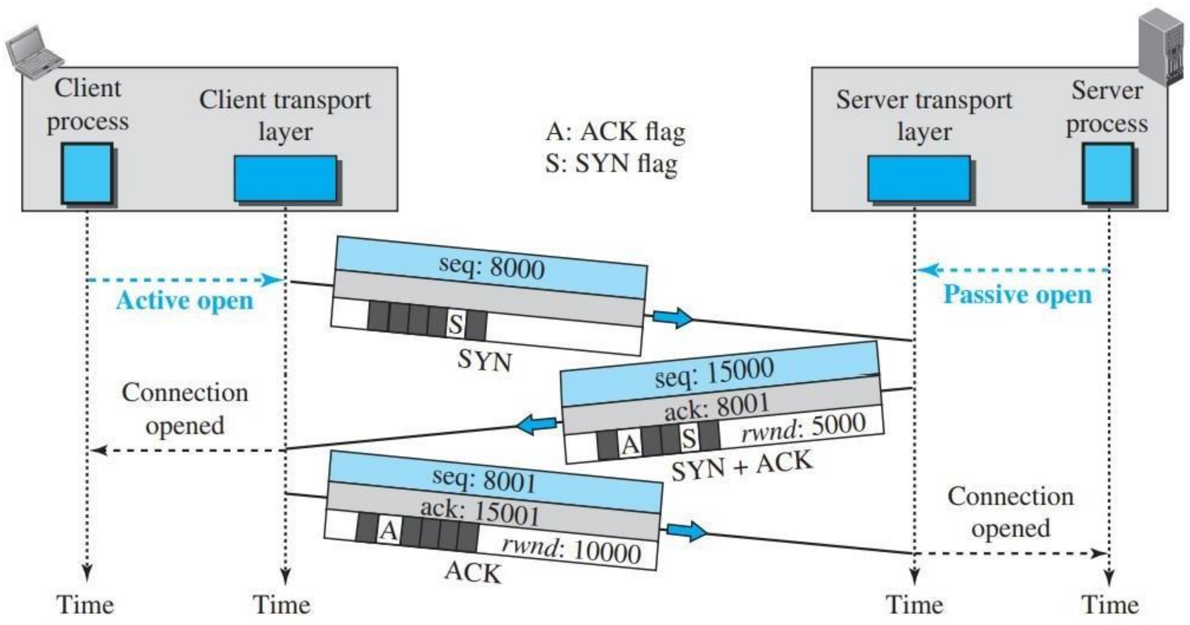
TCP is connection oriented. A connection-oriented transport protocol establishes a logical path between the source and destination.
All of the segments belonging to a message are then sent over this logical path. Using a single logical pathway for the entire message facilitates the acknowledgment process as well as retransmission of damaged or lost frames.
TCP connection is logical, not physical. TCP operates at a higher level. TCP uses the services of IP to deliver individual segments to the receiver, but it controls the connection itself.
If a segment is lost or corrupted, it is retransmitted. Unlike TCP, IP is unaware of this retransmission.
If a segment arrives out of order, TCP holds it until the missing segments arrive; IP is unaware of this reordering.
In TCP, connection-oriented transmission requires three phases: connection establishment, data transfer, and connection termination.

Connection Establishment

TCP transmits data in full-duplex mode. When two TCPs in two machines are connected, they are able to send segments to each other simultaneously. This implies that each party must initialize communication and get approval from the other party before any data are transferred.

Three-Way Handshaking:

The connection establishment in TCP is called three-way handshaking.
In our example, an application program, called the client, wants to make a connection with another application program, called the server, using TCP as the transport-layer protocol.
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



Connection establishment using three-way handshaking

To show the process we use timelines. Each segment has values for all its header fields and perhaps for some of its option fields too. We show only the few fields necessary to understand each phase.
We show the sequence number, the acknowledgment number, the control flags (only those that are set), and window size if relevant. 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 our 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 that we discuss later in the chapter. Note that the SYN segment is a control segment and carries no data. However, it consumes one sequence number because it needs to be acknowledged. We can say that the SYN segment carries one imaginary byte.

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

2. The server sends the second segment, a SYN + ACK segment with two flag bits set 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), as we will see in the flow control section. Since this segment is playing the role of a SYN segment, it needs to be acknowledged. It, therefore, consumes one sequence number.

A SYN 1 ACK segment cannot carry data, but it does consume 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. Note that 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.

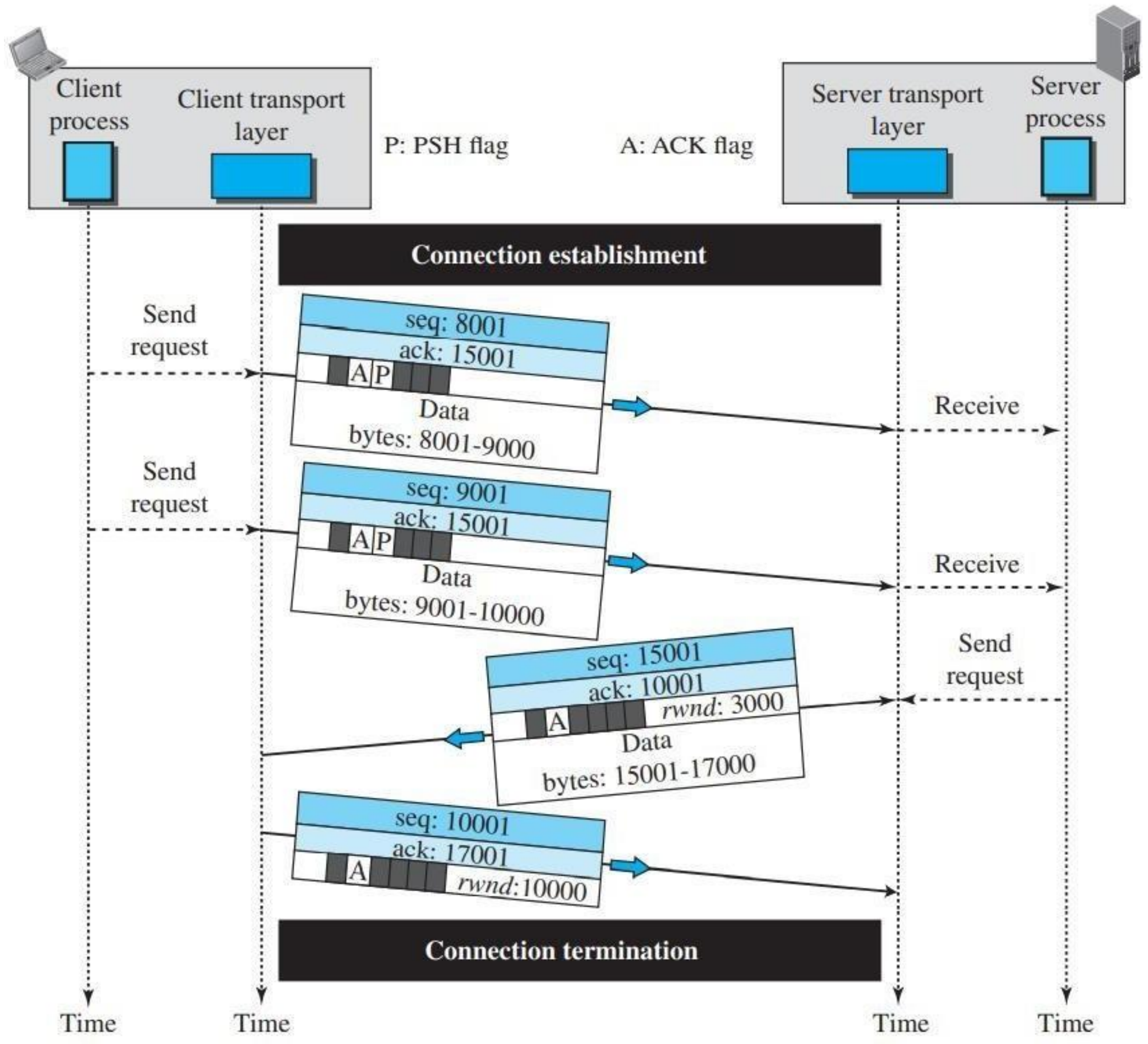
An ACK segment, if carrying no data, consumes no sequence number.

SYN Flooding Attack:

The connection establishment procedure in TCP is susceptible to a serious security problem called SYN flooding attack.
This happens when one or more malicious attackers send a large number of SYN segments to a server pretending that each of them is coming from a different client by faking the source IP addresses in the datagrams.
The server, assuming that the clients are issuing an active open, allocates the necessary resources, such as creating transfer control block (TCB) tables and setting timers.
The TCP server then sends the SYN + ACK segments to the fake clients, which are lost.
When the server waits for the third leg of the handshaking process, however, resources are allocated without being used.
If, during this short period of time, the number of SYN segments is large, the server eventually runs out of resources and may be unable to accept connection requests from valid clients.
This 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.
Some implementations of TCP have strategies to alleviate the effect of a SYN attack. Some have imposed a limit of connection requests during a specified period of time. Others try to filter out datagrams coming from unwanted source addresses.
One recent strategy is to postpone resource allocation until the server can verify that the connection request is coming from a valid IP address, by using what is called a cookie. SCTP, the new transportlayer protocol that we discuss later, uses this strategy.

Data Transfer

After connection is established, bidirectional data transfer can take place.
The client and server can send data and acknowledgments in both directions. The acknowledgment is piggybacked with the data.
Figure shows an example. 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.



Data transfer Pushing

Data

The sending TCP can select the segment size. The receiving TCP also buffers the data when they arrive and delivers them to the application program when the application program is ready or when it is convenient for the receiving TCP.
This type of flexibility increases the efficiency of TCP. There are occasions in which the application program has no need for this flexibility.
For example, consider an application program that communicates interactively with another application program on the other end.
The application program on one site wants to send a chunk of data to the application program at the other site and receive an immediate response. Delayed transmission and delayed delivery of data may not be acceptable by the application program.
TCP can handle such a situation. The application program at the sender can request a push operation.
This means that the sending TCP must not wait for the window to be filled. It must create a segment and send it immediately.
The sending TCP must also set the push bit (PSH) to let the receiving TCP know that the segment includes data that must be delivered to the receiving application program as soon as possible and not to wait for more data to come.

This means to change the byte-oriented TCP to a chunk-oriented TCP, but TCP can choose whether or not to use this feature.

Urgent Data

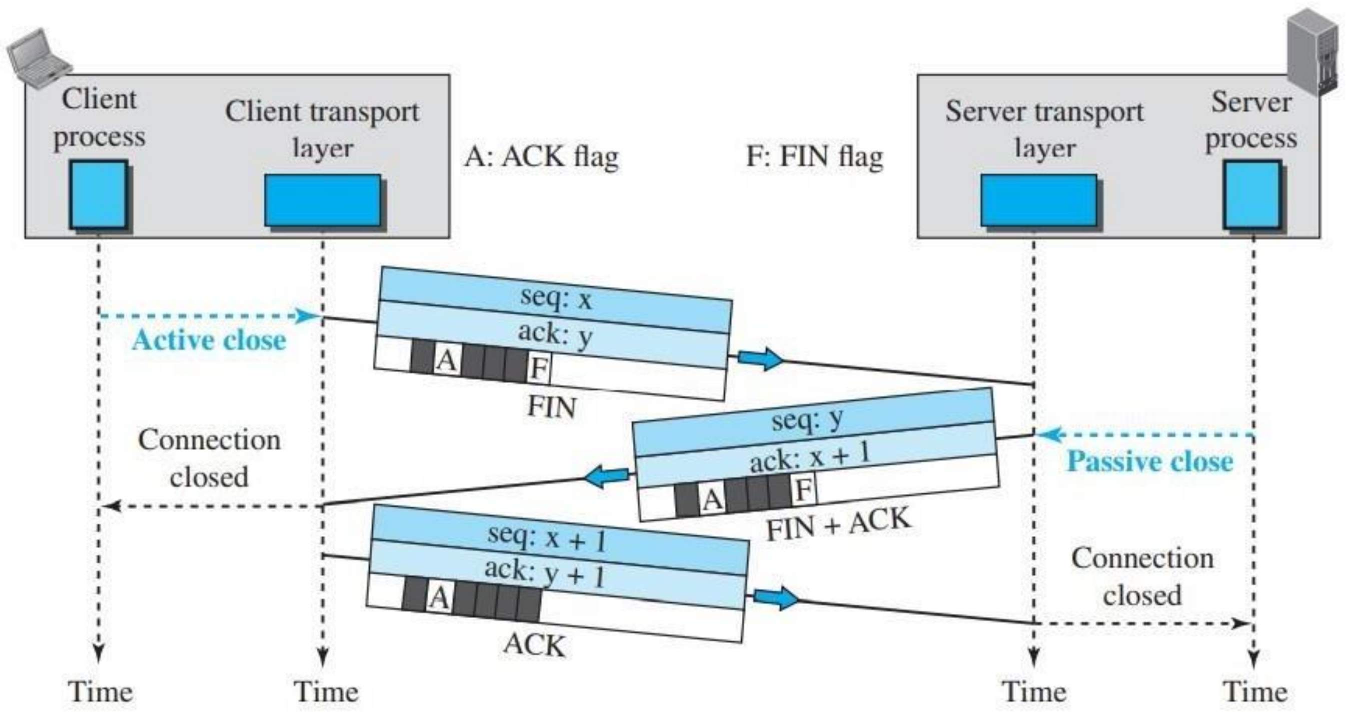
TCP is a stream-oriented protocol. This means that the data is presented from the application program to TCP as a stream of bytes.
Each byte of data has a position in the stream. However, there are occasions in which an application program needs to send urgent bytes, some bytes that need to be treated 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 TC 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. It is important to mention that TCP's urgent data is neither a priority service nor an out-of-band data service as some people think.
TCP urgent mode is a service by which the application program at the sender side marks some portion of the byte stream as needing special treatment by the application program at the receiver side.
The receiving TCP delivers bytes (urgent or nonurgent) to the application program in order but informs the application program about the beginning and end of urgent data. It is left to the application program to decide what to do with the urgent data.

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 today allow two options for connection termination: three-way handshaking and four-way handshaking with a half-close option.

Three-Way Handshaking

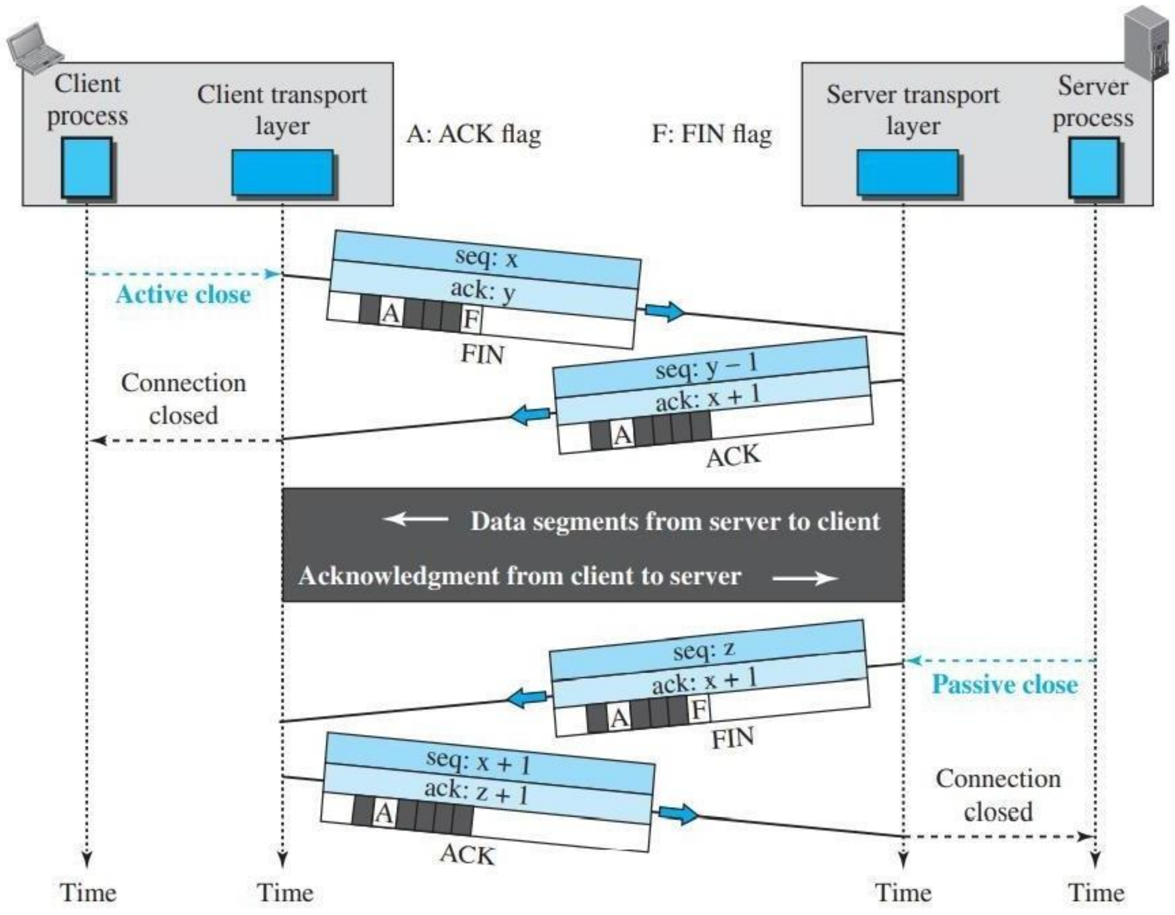
1. In this situation, the client TCP, after receiving a close command from the client process, sends the first segment, a FIN segment in which the FIN flag is set. Note that a FIN segment can include the last chunk of data sent by the client or it can be just a control segment as shown in the figure. If it is only a control segment, it consumes only one sequence number because it needs to be acknowledged. The FIN segment consumes one sequence number if it does not carry data.
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.



Connection termination using three-way handshaking

Half-Close

In TCP, one end can stop sending data while still receiving data. This is called a half-closed.
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.
A good example is sorting.
When the client sends data to the server to be sorted, the server needs to receive all the data before sorting can start.
This means the client, after sending all data, can close the connection in the client-to-server direction.
However, the server-to-client direction must remain open to return the sorted data. The server, after receiving the data, still needs time for sorting; its outbound direction must remain open.
Figure shows an example of a half-close.
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.



Half-close

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

To keep track of all the different events happening during connection establishment, connection termination, and data transfer, TCP is specified as the finite state machine (FSM) as shown in Figure.

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 coloured 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, as explained later in the chapter.