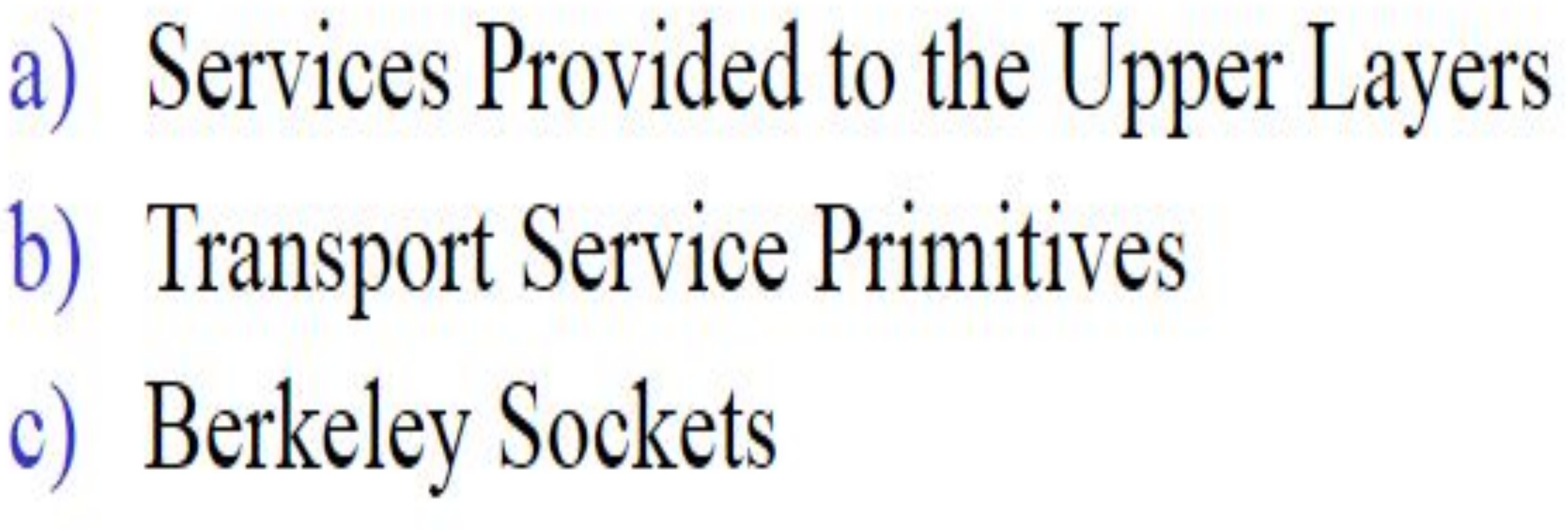
THE TRANSPORT LAYER

Ms. Pallavi Gangurde

5.1 THE TRANSPORT LAYER

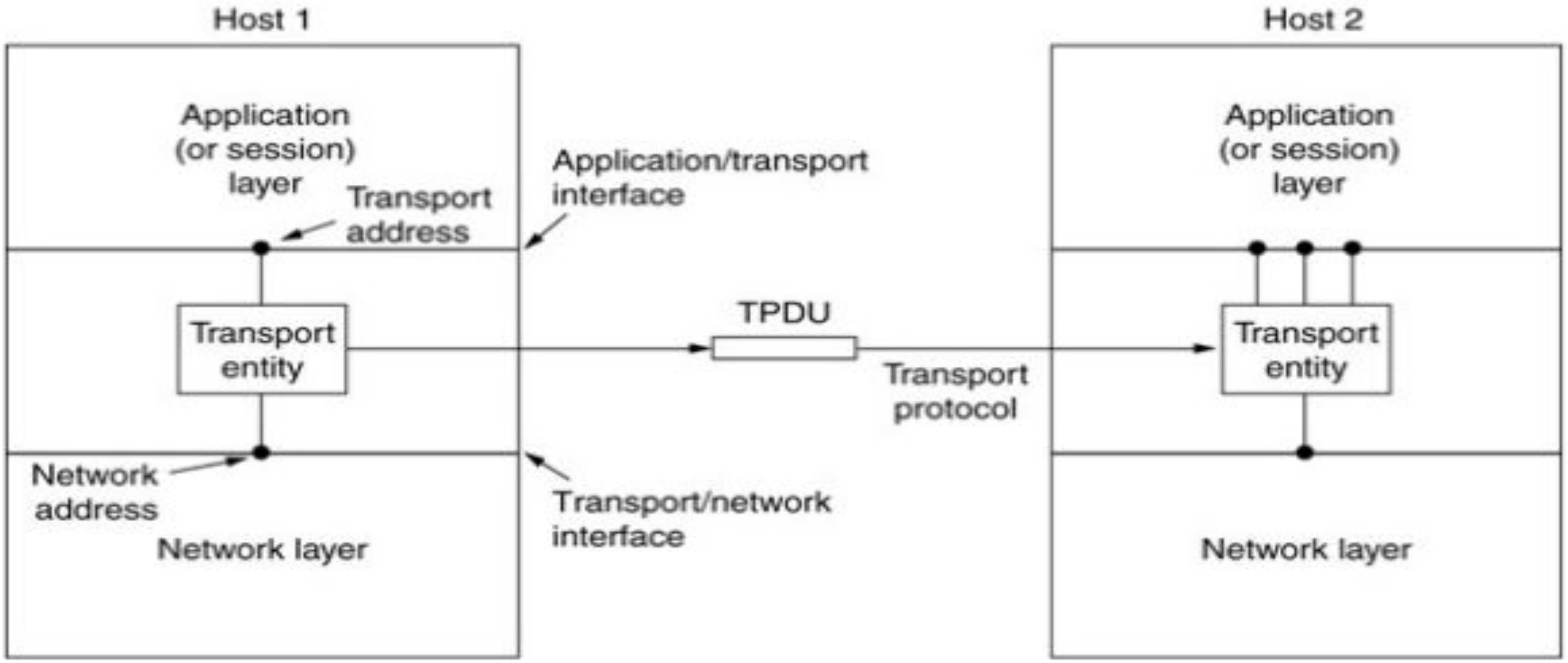
- The transport layer is not just another layer.
- It is the heart of the whole protocol hierarchy.
- Its task is to provide reliable, cost-effective data transport from the source machine to the destination machine, independently
- Without transport layer, the whole concept of layered protocols would make little sense.
- In this chapter we will study the transport layer in detail, including its services, design, protocols,and performance.

5.1 SERVICES PROVIDED

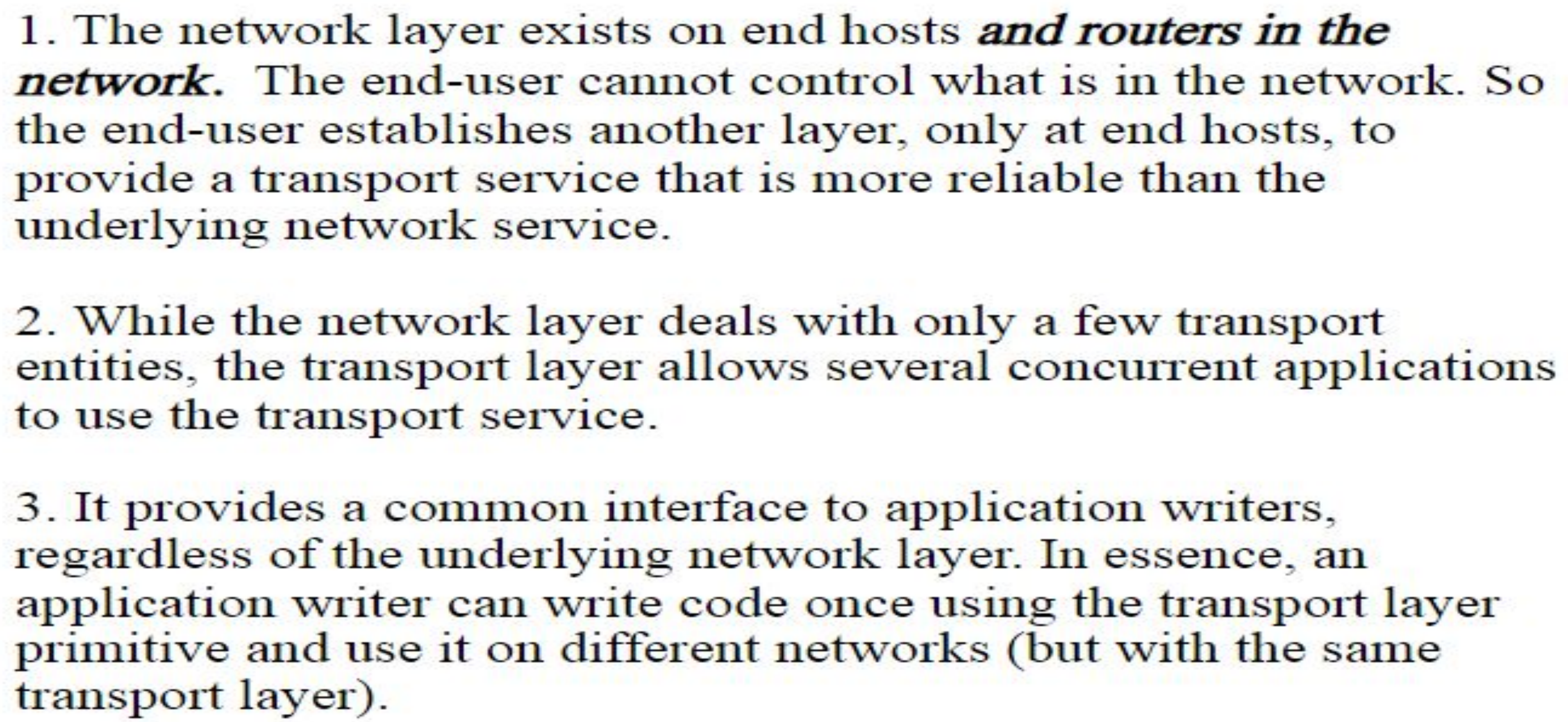


a) Services Provided to the Upper Layers
b) Transport Service Primitives
c) Berkeley Sockets

5.1 SERVICES PROVIDED TO UPPER LAYER



WHY TRANSPORT LAYER



1. The network layer exists on end hosts ***and routers in the network.*** The end-user cannot control what is in the network. So the end-user establishes another layer, only at end hosts, to provide a transport service that is more reliable than the underlying network service.

2. While the network layer deals with only a few transport entities, the transport layer allows several concurrent applications to use the transport service.

3. It provides a common interface to application writers, regardless of the underlying network layer. In essence, an application writer can write code once using the transport layer primitive and use it on different networks (but with the same transport layer).

INTERNET TRANSPORT PROTOCOL (TCP & UDP)

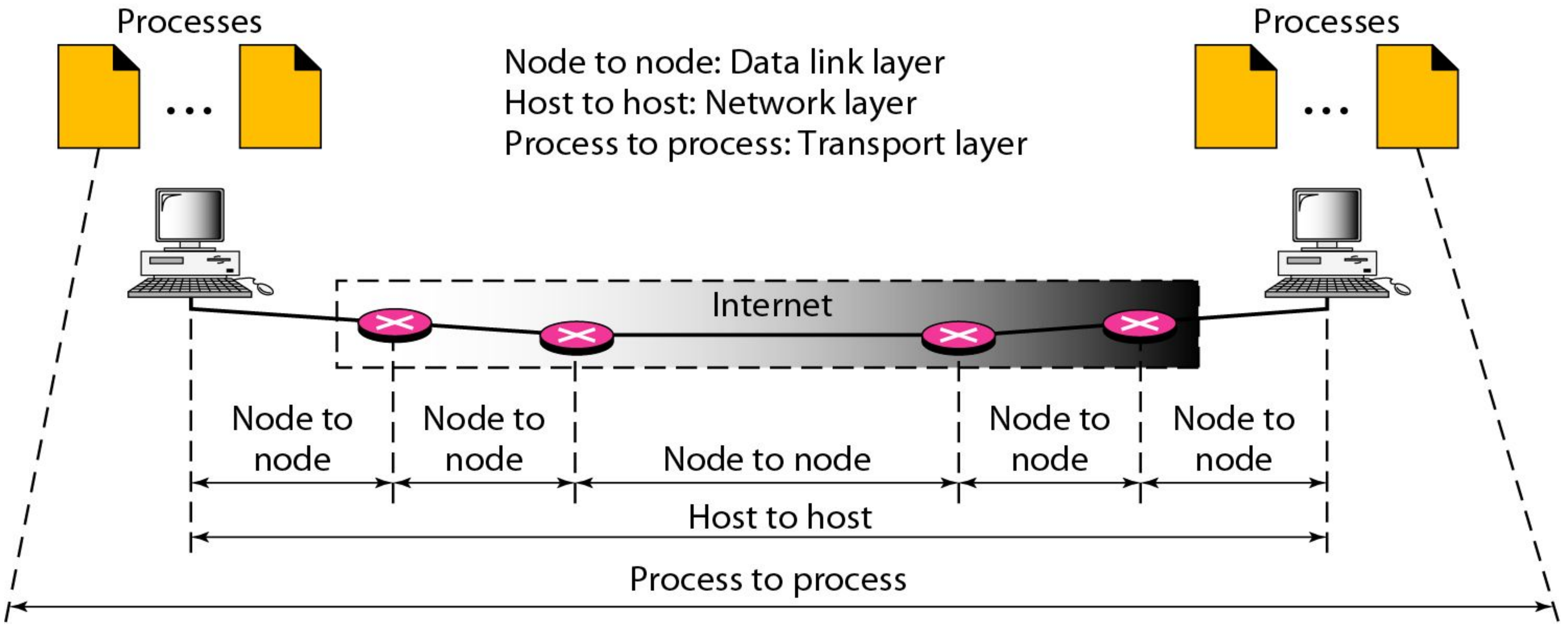
- Internet has two main protocols in the transport layer
- One is connection oriented and other is connectionless services.
- Transmission control protocol (TCP) is a connection oriented and UDP is connectionless protocol
- UDP is just IP with an additional short header

USER DATAGRAM PROTOCOL (UDP)

INTRODUCTION

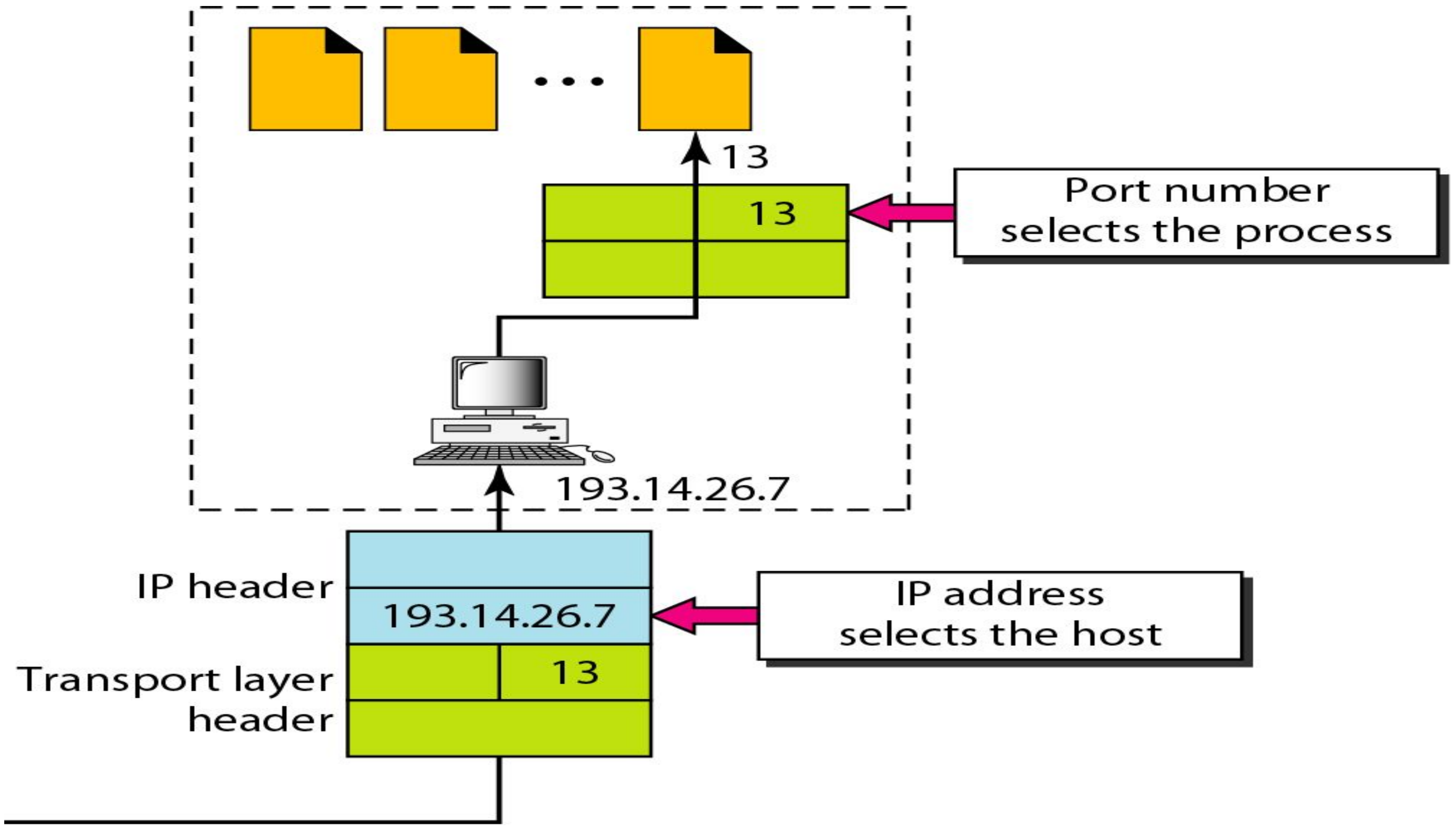
- UDP is located between the application layer and the IP layer
- It serves as the intermediary between the application programs and the network operations.
- Connectionless, unreliable transport protocol.
- It does not add anything to the services of IP except to provide process-to-process communication instead of host-to-host communication.
- It 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.

PROCESS TO PROCESS DELIVERY



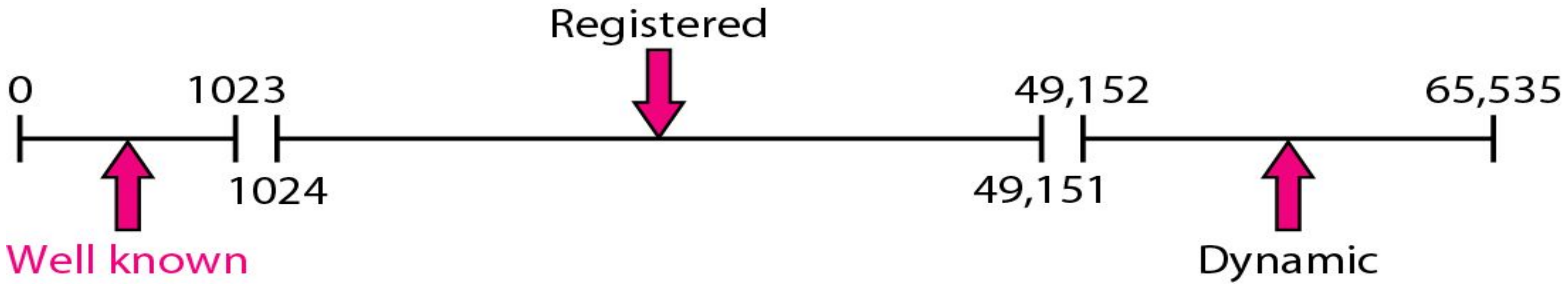
The transport layer is responsible for **process-to-process delivery—the delivery of a packet**, part of a message, from one process to another.

IP ADDRESS AND PORT NUMBERS

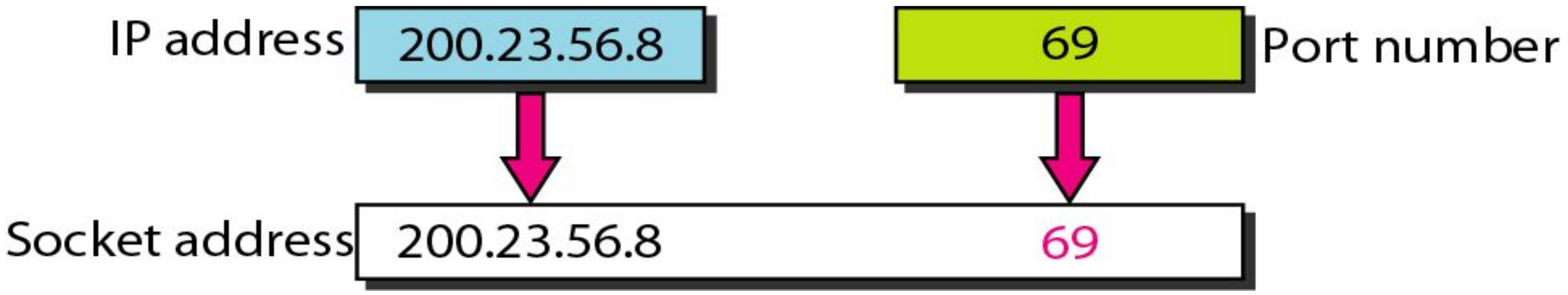


SOCKET ADDRESS

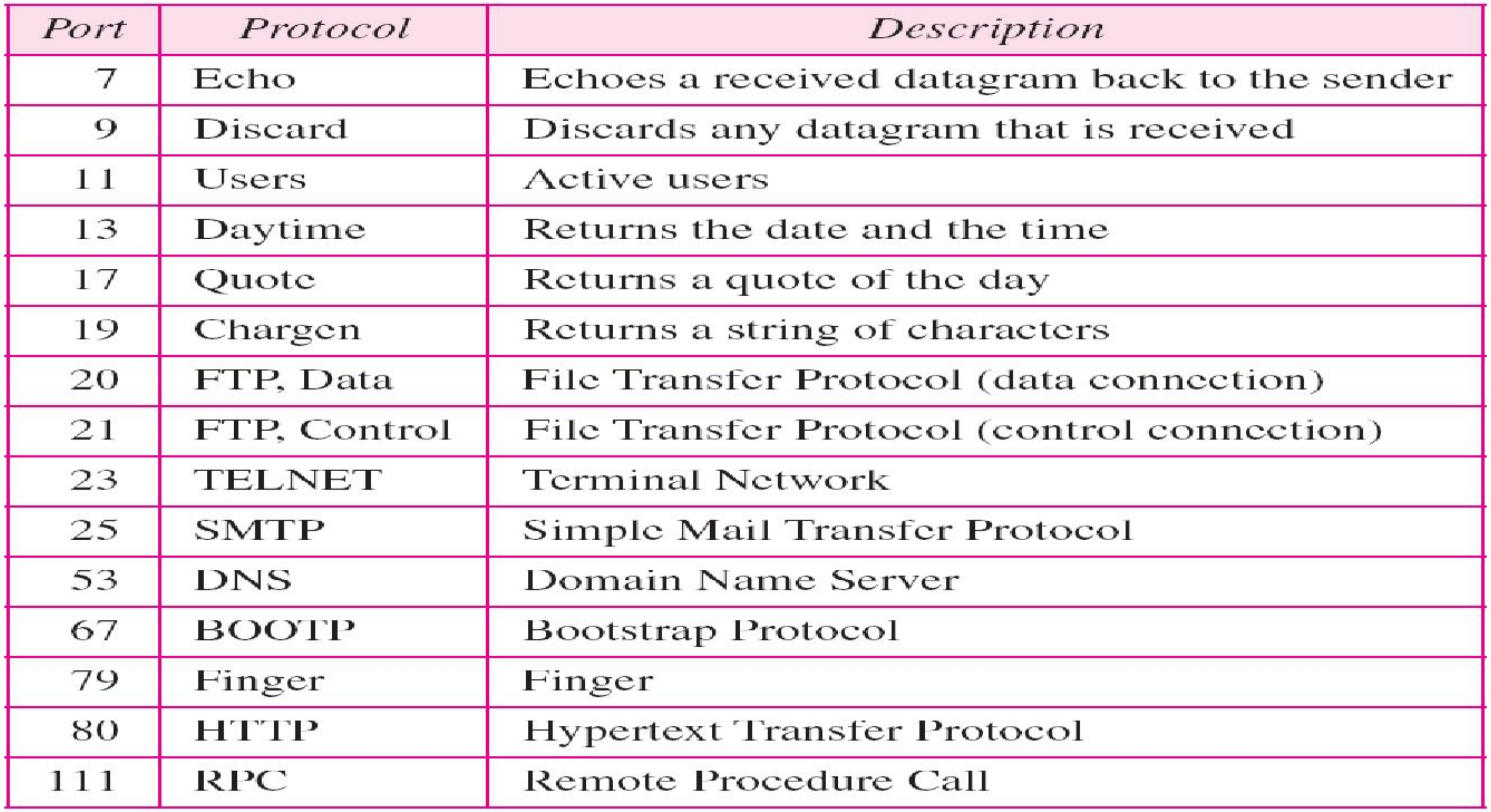
IANA ranges (*International Assigned Number Authority*) :



IP address and port number makes a **Socket address**

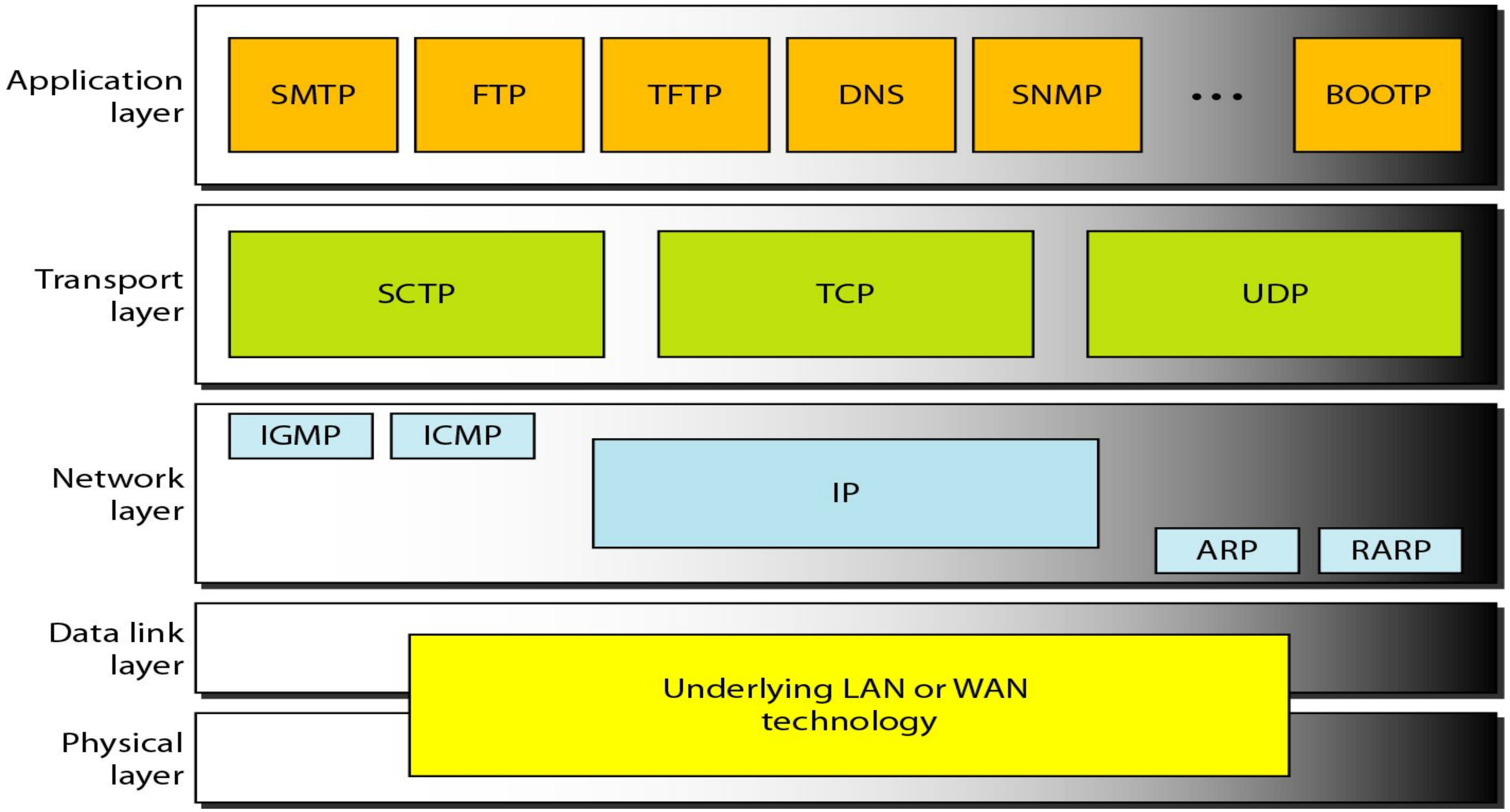


WELL-KNOWN PORTS USED BY UDP



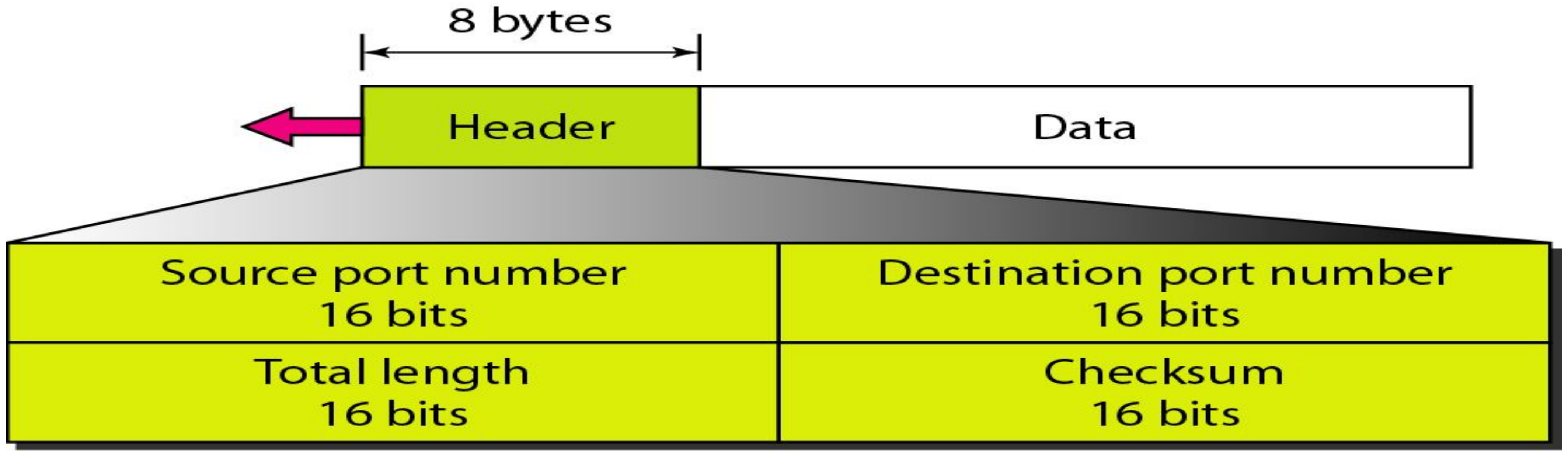
Port	*Protocol*	*Description*
7	Echo	Echoes a received datagram back to the sender
9	Discard	Discards any datagram that is received
11	Users	Active users
13	Daytime	Returns the date and the time
17	Quote	Returns a quote of the day
19	Chargen	Returns a string of characters
20	FTP, Data	File Transfer Protocol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

POSITION OF UDP AND TCP/IP PROTOCOL SUITE



USER DATAGRAM

- UDP packet is called as User datagram.



1. **Source Port Number**:

✔ Is the port number used by the process running on the source host.

✔ It is 16 bit long, means port numbers can range from 0 to 65535

✔ If the source host is **Client**, the port number is an **ephemeral (means Short lived) port number** requested by the process and chosen by the UDP software running on the source host.

✔ If the source host is **Server**, the port number is a **well known port number**.

USER DATAGRAM

2. Destination Port Number:

- ✔ Is a port number **used by the process** running on the destination host.
- ✔ It is also **16 bit** long.
- ✔ If the destination host is **server,** the port number is a **well known port number.**
- ✔ If the destination host is **client**, the port number is an **ephemeral port number.** In this case, the server copies the ephemeral port number it has received in the request packet.

3. Length:

- ✔ Is 16 bit field, defines length (of 0 to 65535 bytes) of the user datagram, header plus data.
- ✔ **UDP length = IP length – IP header's length**
- ✔ When IP software delivers the UDP user datagram to the UDP layer, it has already dropped the IP header.

EXAMPLE 1

- The following is a dump of a UDP header in hexadecimal format.

CB84000D001C001C

a. What is the source port number?
b. What is the destination port number?
c. What is the total length of the user datagram?
d. What is the length of the data?
e. Is the packet directed from a client to a server or vice versa?
f. What is the client process?

EXAMPLE 1

CB84-000D-001C-001C

- Solution

a. The source port number is the 1st four hexadecimal digits ($CB84_{16}$), which means that the source port number is 52100

b. The destination port number is the 2nd 4 hexadecimal digits ($000D_{16}$), which means destination port number is 13.

c. The 3rd 4 hexadecimal digits ($001C_{16}$) define the length of the whole UDP packet as 28 bytes.

d. The length of the data is = length of whole packet – length of header i.e 28-8 =20 bytes.

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

f. The client process is the Daytime. (refer table of well known ports used with UDP)

E2 A7 00 0D 00 20 74 9E 0E FF 00 00 00

a] Source port no $\rightarrow (E2A7)_{16}$

b] Dest " " $\rightarrow (000D)_{16}$

c] Total length $\rightarrow (0020)_{16} \rightarrow (\underline{32})_{10}$ bytes

d] Total length of data = Total length - header length

$32 - 8 = 24$ bytes

UDP SERVICES

- **Process to process communication:**
  - ✔ UDP provides process to process communication using sockets, a combination of IP addresses and port numbers.
  - ✔ Several port numbers used by UDP are shown in table of well known ports
- **Connectionless Services:**
  - ✔ Each user datagram sent by UDP is an independent datagram.
  - ✔ No relationship between different datagram even if they are coming from same source or going to the same destination.
  - ✔ The user datagrams are not numbered.
  - ✔ **ramification-** process that uses UDP cannot send a stream of data to UDP and except UDP to chop them into different related user datagram.
  - ✔ Instead each request must b small enough to fit into one user datagram.
  - ✔ Only those processes sending short messages, message less than 65,507 bytes (65,535-8 bytes for UDP header – 20 bytes for IP header) can use UDP.

UDP SERVICES

- **Flow Control:**

✔ No flow control in UDP 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 this service, if needed.

- **Error Control:**

✔ no error control mechanism in UDP except for the checksum.

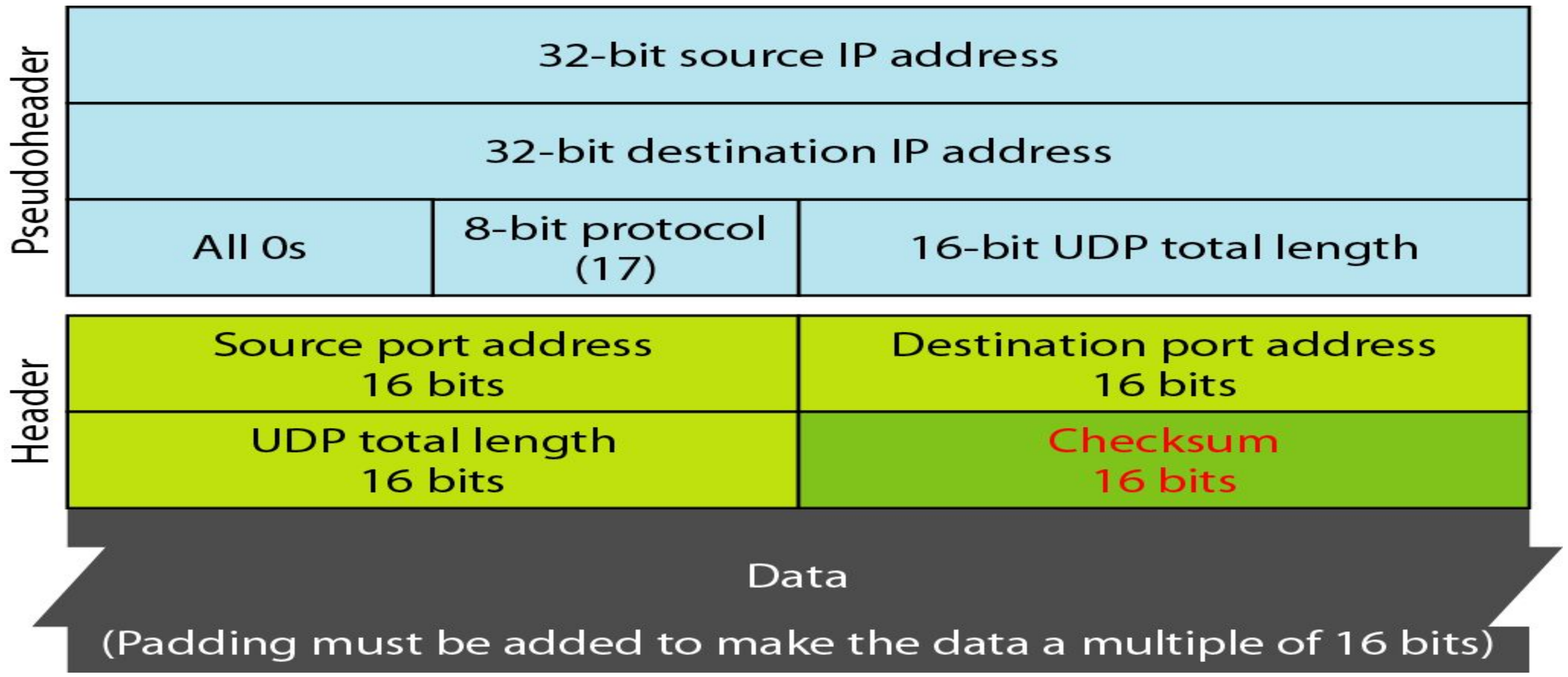
✔ This means that sender does not know if a message has been lost or duplicated.

✔ When the receiver detects an error through the checksum, the user datagram is silently discarded.

✔ The lack of error control means that process using UDP should provide for this service if needed.

CHECKSUM

- UDP checksum calculation is different from that of IP.
- UDP checksum includes 3 sections: a **pseudoheader, the UDP header and data coming from the application layer**
- The **pseudoheader** is the part of header of IP packet in which the user datagram is to be encapsulated with some fields with all 0s.


Pseudoheader
32-bit source IP address
32-bit destination IP address
All 0s
8-bit protocol (17)
16-bit UDP total length
Header
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)

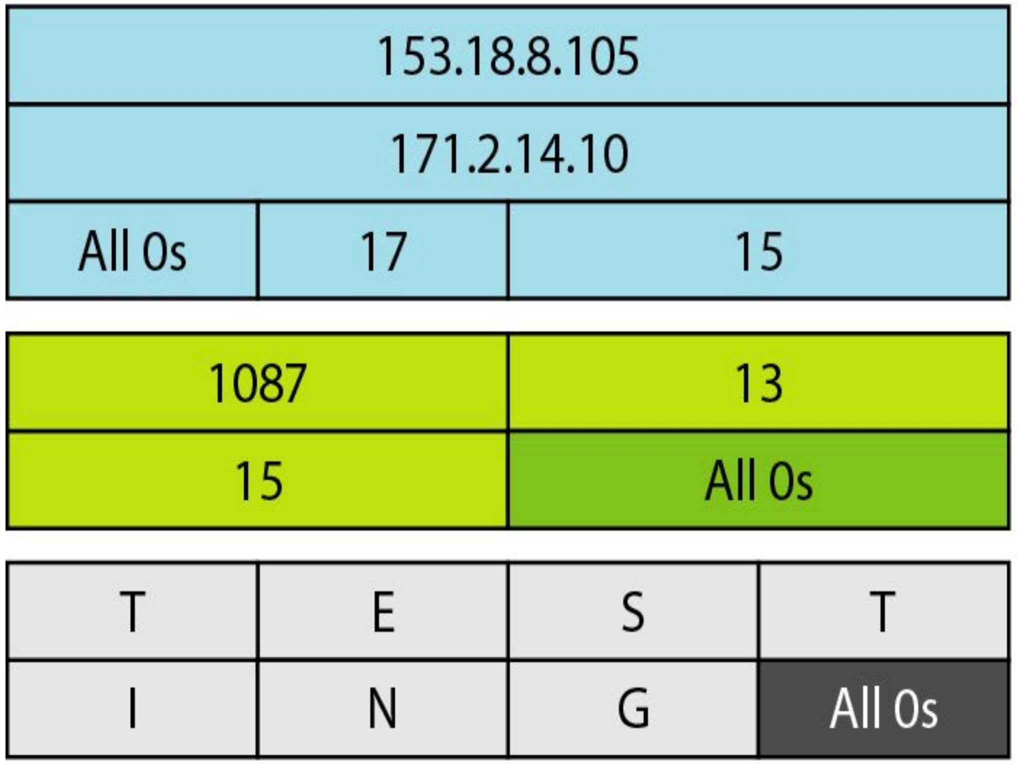
CHECKSUM

- If the checksum does not include the pseudoheader, a user datagram may arrive safe and sound.
- If the datagram 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.
- The destination port number can be same for UDP or TCP.
- The value of protocol field=17 for UDP.
- If the 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.
- There are similarities between the pseudoheader fields and last 12 bytes of IP header.

EXAMPLE

- Figure 23.11 shows the checksum calculation for a very small user datagram with only 7 bytes of data. Because the number of bytes of data is odd, padding is added for checksum calculation. The pseudoheader as well as the padding will be dropped when the user datagram is delivered to IP.

CHECKSUM CALCULATION OF A SIMPLE UDP USER DATAGRAM



10011001 00010010	⟶	153.18
00001000 01101001	⟶	8.105
10101011 00000010	⟶	171.2
00001110 00001010	⟶	14.10
00000000 00010001	⟶	0 and 17
00000000 00001111	⟶	15
00000100 00111111	⟶	1087
00000000 00001101	⟶	13
00000000 00001111	⟶	15
00000000 00000000	⟶	0 (checksum)
01010100 01000101	⟶	T and E
01010011 01010100	⟶	S and T
01001001 01001110	⟶	I and N
01000111 00000000	⟶	G and 0 (padding)
10010110 11101011	⟶	Sum
01101001 00010100	⟶	Checksum

EXAMPLE

What value is sent for the checksum in one of the following hypothetical situations?

a. The sender decides not to include the checksum.

b. The sender decides to include the checksum, but the value of the sum is all 1s.

c. The sender decides to include the checksum, but the value of the sum is all 0s.

Solution

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

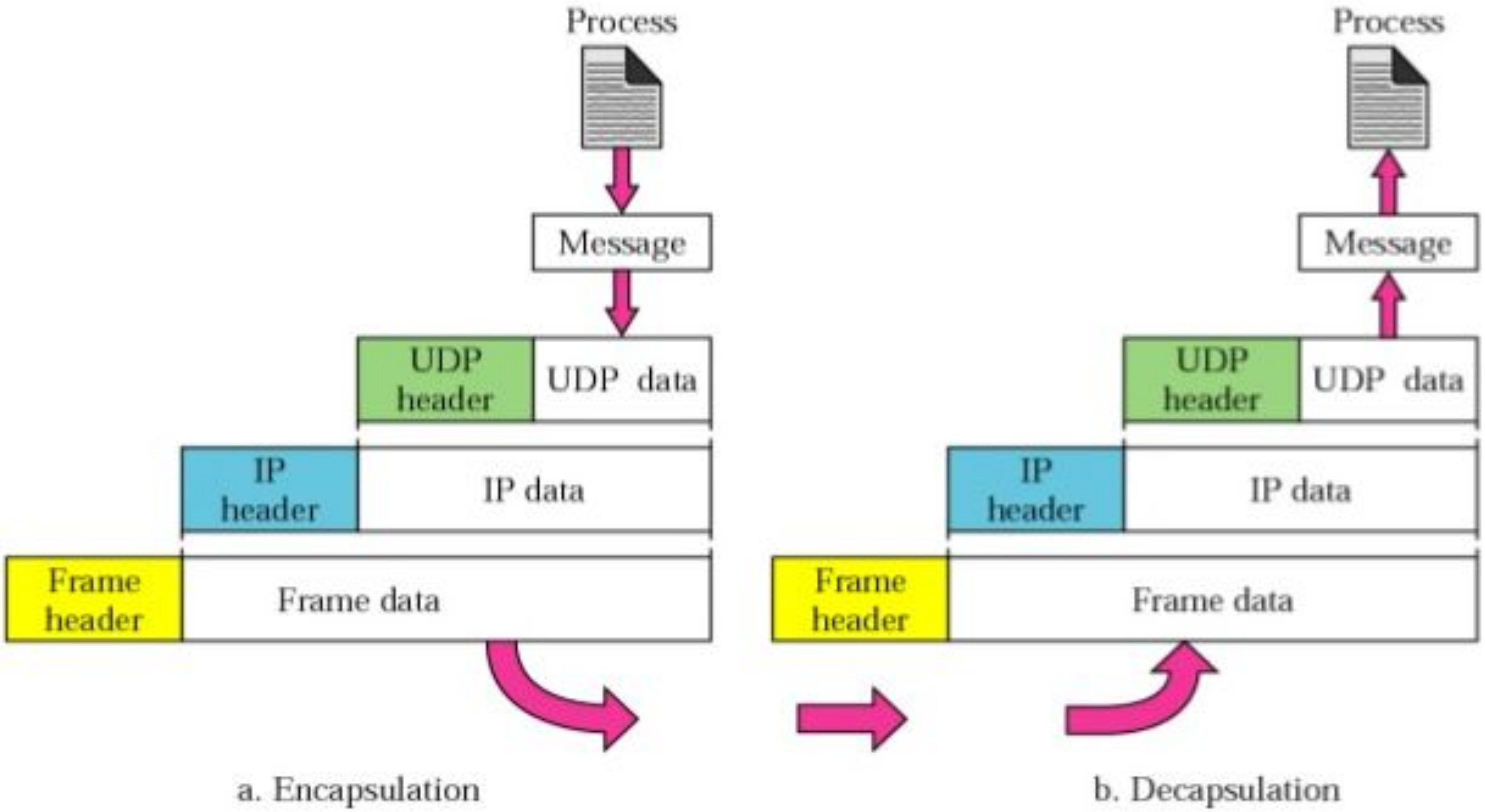
b. When the sender complements the sum, the result is all 0s; the sender complements the results again before sending. The value sent for the checksum is all 1s. The second complement option is needed to avoid confusion with the case in part a.

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

CONGESTION CONTROL

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

ENCAPSULATION AND DECAPSULATION



ENCAPSULATION

- When a process has a message to send through UDP, it passes the message to UDP along with a pair of socket addresses and the length of data.
- UDP receives the data and adds the UDP header.
- UDP then passes the user datagram to IP with the socket addresses.
- IP adds its own header, using the value 17 in the protocol field, indicating that the data has come from the UDP protocol.
- IP datagram is then passed to the data link layer.
- The data link layer receives the IP datagram adds its own header (and possibly a trailer) and passes it to the physical layer.
- The physical layer encodes the bits into electrical or optical signals and sends it to the remote machine.

DECAPSULATION

- When the message arrives at the destination host, the physical layer decodes the signals into bits and passes it to the data link layer.
- The data link layer uses the header (and the trailer) to check the data.
- If there is no error, the header and trailer are dropped and datagram is passed to UDP with the sender and receiver IP addresses.
- UDP uses the checksum to check the entire user datagram.
- If there is no error, the header is dropped and the application data along with the sender socket address is passed to the process.
- The sender socket address is passed to the process in case it needs to respond to the message received.

TRANSMISSION CONTROL PROTOCOLS (TCP)

TCP/IP PROTOCOL SUITE

- TCP is a connection-oriented protocol
- It creates a virtual connection between two TCPs to send data.
- It uses flow and error control mechanisms at the transport level.

TCP SERVICES:

1. **Process to Process communication:**

✔ As with UDP, TCP provides process to process communication using port numbers

✔ Next table lists some well known port numbers used by TCP.

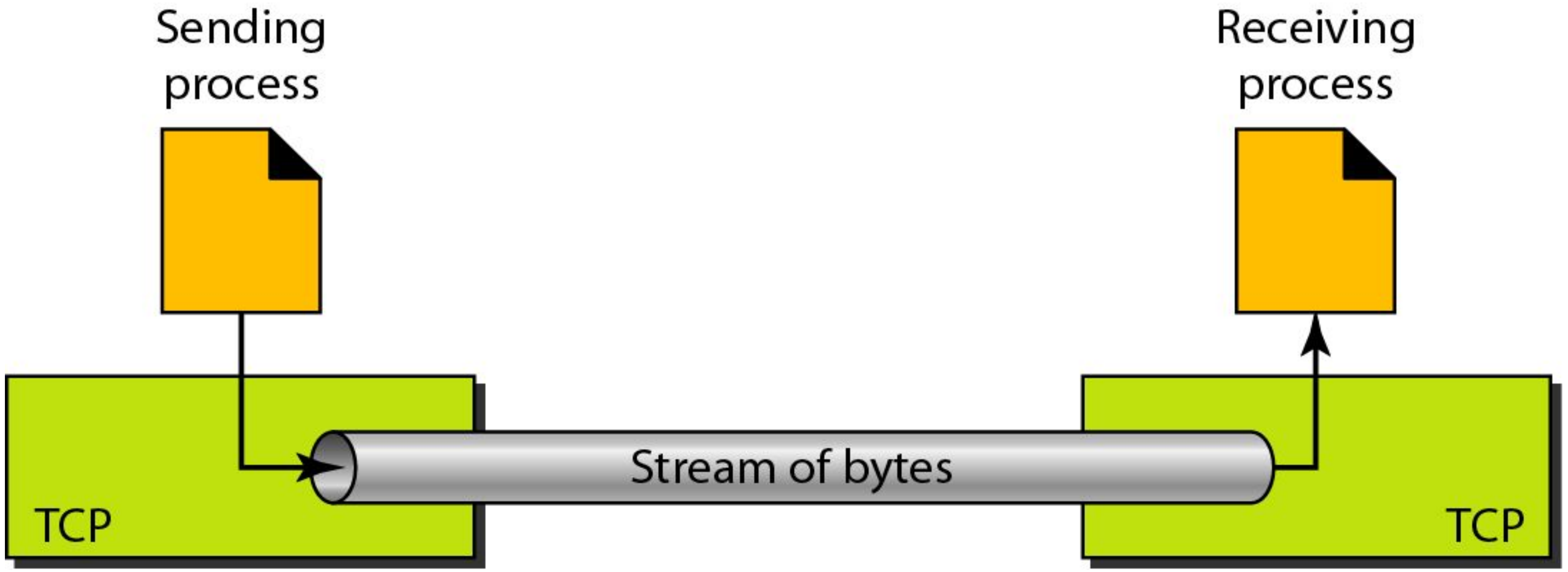
TABLE: WELL KNOWN PORTS USED BY TCP

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

TCP SERVICES

2. **Stream Delivery services:**

- ✔ TCP 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 two processes seem to be connected by an imaginary **"tube"** that carries their bytes across the internet.

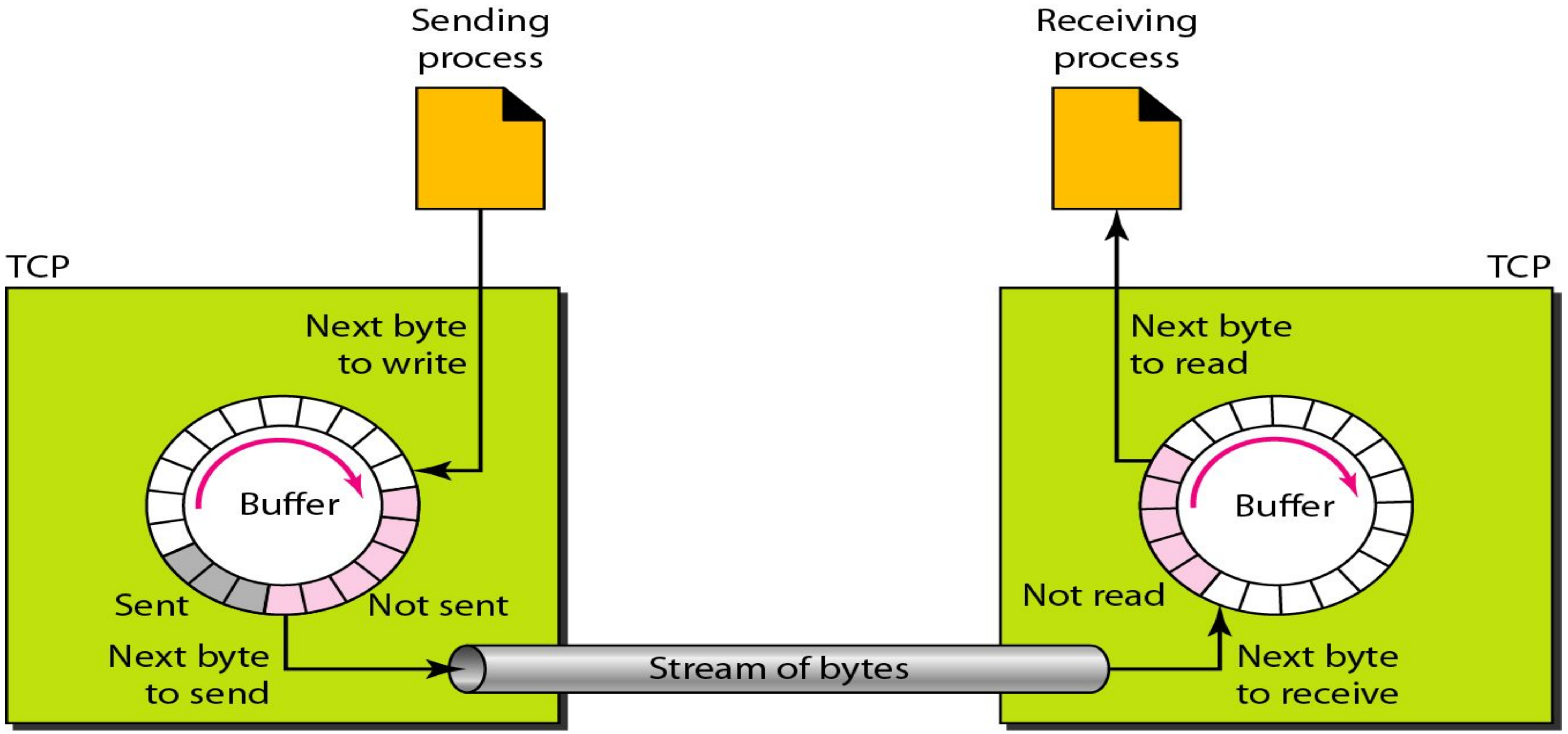


SENDING & RECEIVING BUFFERS

- ✔ Because the sending and receiving processes may not necessarily write or read data at the same time, TCP needs buffers for storage.
- ✔ These buffers are also necessary for flow and error control mechanisms used by TCP.
- ✔ One way to implement a buffer is to use a circular array of 1 byte locations as shown.
- ✔ Two buffers of 20 bytes each are shown for simplicity.
- ✔ Normally the buffers are hundreds or thousands of bytes, depending on the implementation.

SENDING & RECEIVING BUFFERS

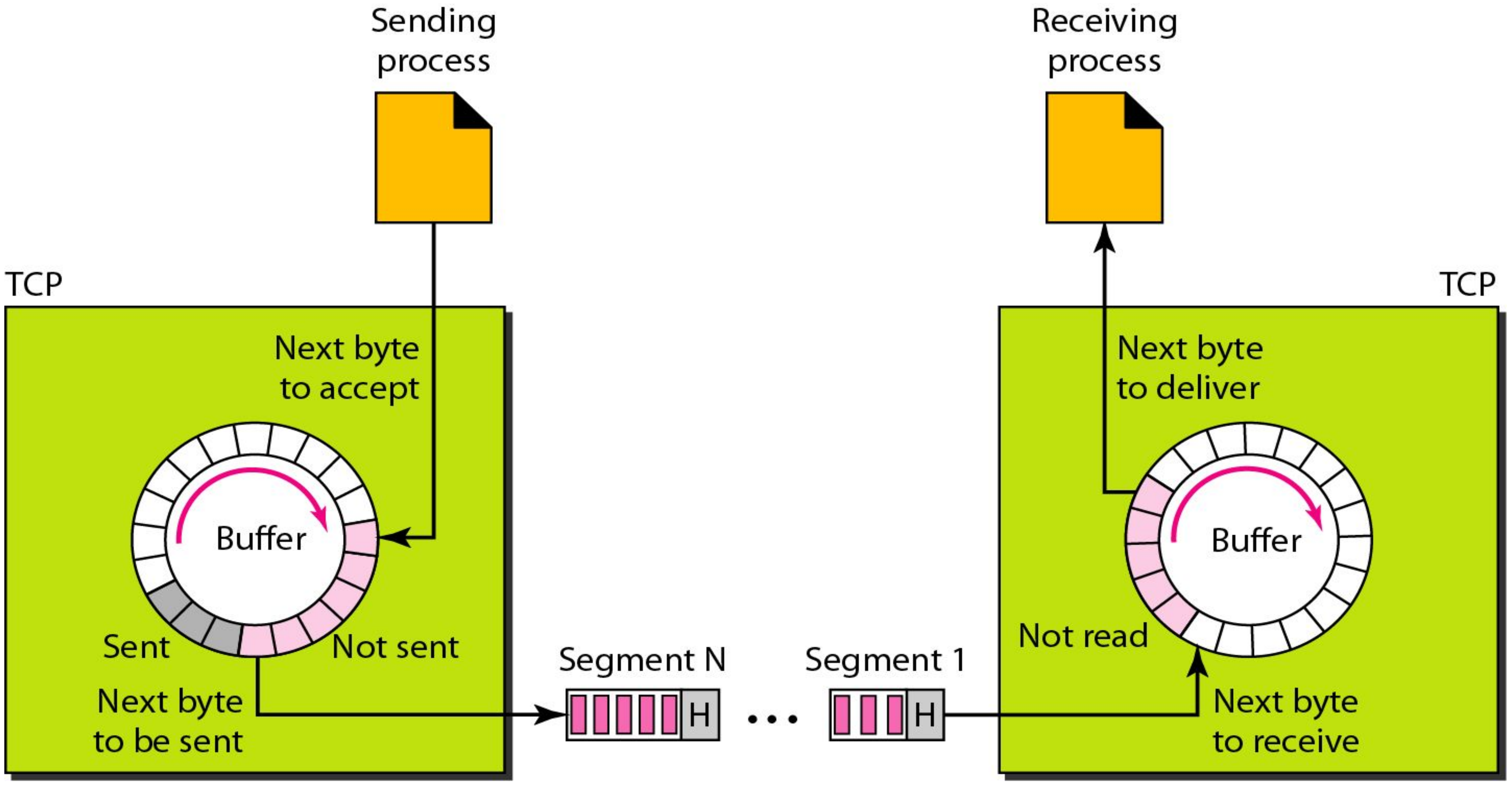
Figure shows the movement of the data in one direction



SEGMENTS

- ✔ 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 purpose) and delivers the segment to the IP layer for transmission.
- ✔ The segments are encapsulated in an IP datagram and transmitted.
- ✔ The entire operation is transparent to the receiving process.
- ✔ Next figure shows how segments are created from the bytes in the buffers.
- ✔ In figure we show one segment carrying 3 bytes and other carrying 5 bytes. In reality, segments carry hundreds, if not thousands of bytes.

SEGMENTS


Sending process
Receiving process
TCP
TCP
Next byte to accept
Next byte to deliver
Buffer
Buffer
Sent
Not sent
Not read
Next byte to be sent
Next byte to receive
Segment N
Segment 1
H
H

3. 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.

4. Multiplexing & Demultiplexing:

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

5. CONNECTION ORIENTED SERVICES

- ✔ 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 TCPs established a virtual connection between them.
2. Data are exchanged in both directions.
3. The process is terminated.

- ✔ This is a virtual connection and not a physical connection.
- ✔ TCP segment is encapsulated in an IP datagram and can be sent out of order, or lost or corrupted and then resent.
- ✔ There is no physical connection, it's a stream oriented environment.

6. Reliable Services:

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

TCP FEATURES

❑ **Numbering System:**

✔ There is no field for a segment number value in the segment header.

✔ There are two fields- sequence number and acknowledgment number

✔ These two fields refer to a byte number and not a segment number.

❑ **Byte Number:**

✔ TCP numbers all data types (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.

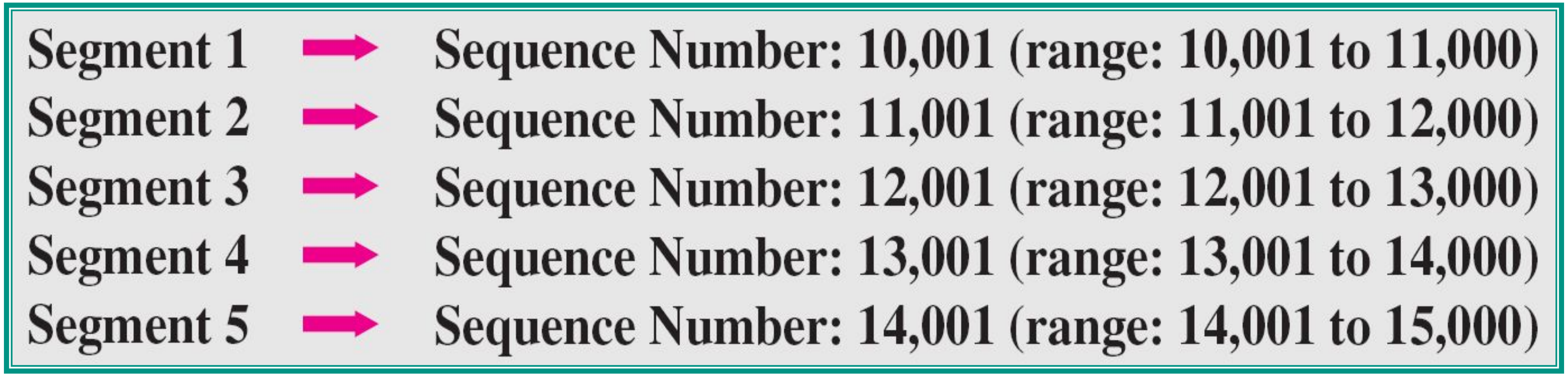
✔ The numbering does not necessarily starts from 0.

✔ TCP chooses an arbitrary number between 0 & 2^{32}-1 for the number of 1st byte.

EXAMPLE 1

- Suppose a TCP connection is transferring a file of 5,000 bytes. The first byte is numbered 10,001. What are the sequence numbers for each segment if data are sent in five segments, each carrying 1,000bytes.?

The following shows the sequence number for each segment:



Segment 1	➡	Sequence Number: 10,001 (range: 10,001 to 11,000)
Segment 2	➡	Sequence Number: 11,001 (range: 11,001 to 12,000)
Segment 3	➡	Sequence Number: 12,001 (range: 12,001 to 13,000)
Segment 4	➡	Sequence Number: 13,001 (range: 13,001 to 14,000)
Segment 5	➡	Sequence Number: 14,001 (range: 14,001 to 15,000)

SEQUENCE NUMBER

- **Sequence Number:**
- ✔ After the bytes have been numbered, TCP assigns a sequence number to each segment that is being sent.
- ✔ The sequence number for each segment is the **number of first byte of data carried in that segment.**
- ✔ When a segment carries a combination of data and control information, it uses a sequence number.
- ✔ If segment does not carry user data, it does not logically define a sequence number. The field is there but the value is not valid.
- ✔ When a segment carries only control information, need a sequence number to allow an acknowledgement from the receiver. These segments are used for connection establishment, termination or abortion.
- ✔ Each of these segments consume one sequence number as though it carries one byte, but there are no actual data.

ACKNOWLEDGEMENT NUMBER

- Connection in TCP is full duplex.
- Each party numbers the byte usually with a different starting byte number.
- Each party uses an acknowledgement number to confirm the **bytes** it has **received.**
- The acknowledgement number defines the **number of next byte that party expects to receive.**
- Acknowledgement number is **cumulative**, which means that the party **takes the number of last byte that it has received, safe and sound, adds 1 to it and announces this sum as the acknowledgement number.**
- E.g. if the party uses 5,643 as an acknowledgement number, it has received all bytes from the beginning up to 5,642. It does not mean that the party has received 5,642 bytes because the first byte number does not have to start from 0.

A TCP CONNECTION

- TCP is a connection oriented transport protocol establishes a virtual path between source and destination.
- Using virtual path way for the entire message facilitates the acknowledgement process as well as retransmission of damaged or lost frames. TCP operates at higher level.
- TCP uses the service of IP to deliver individual segments to the receiver but it controls the connection itself.
- TCP handles lost or corrupted segments by retransmitting (Unlike IP).
- If segments arrives out of order, TCP holds segments until the missing segment arrive (unlike IP).
- In TCP connection oriented transmission requires 3 phases: connection establishment, data transfer and connection termination.

CONNECTION ESTABLISHMENT

- **Three-way Handshaking:**

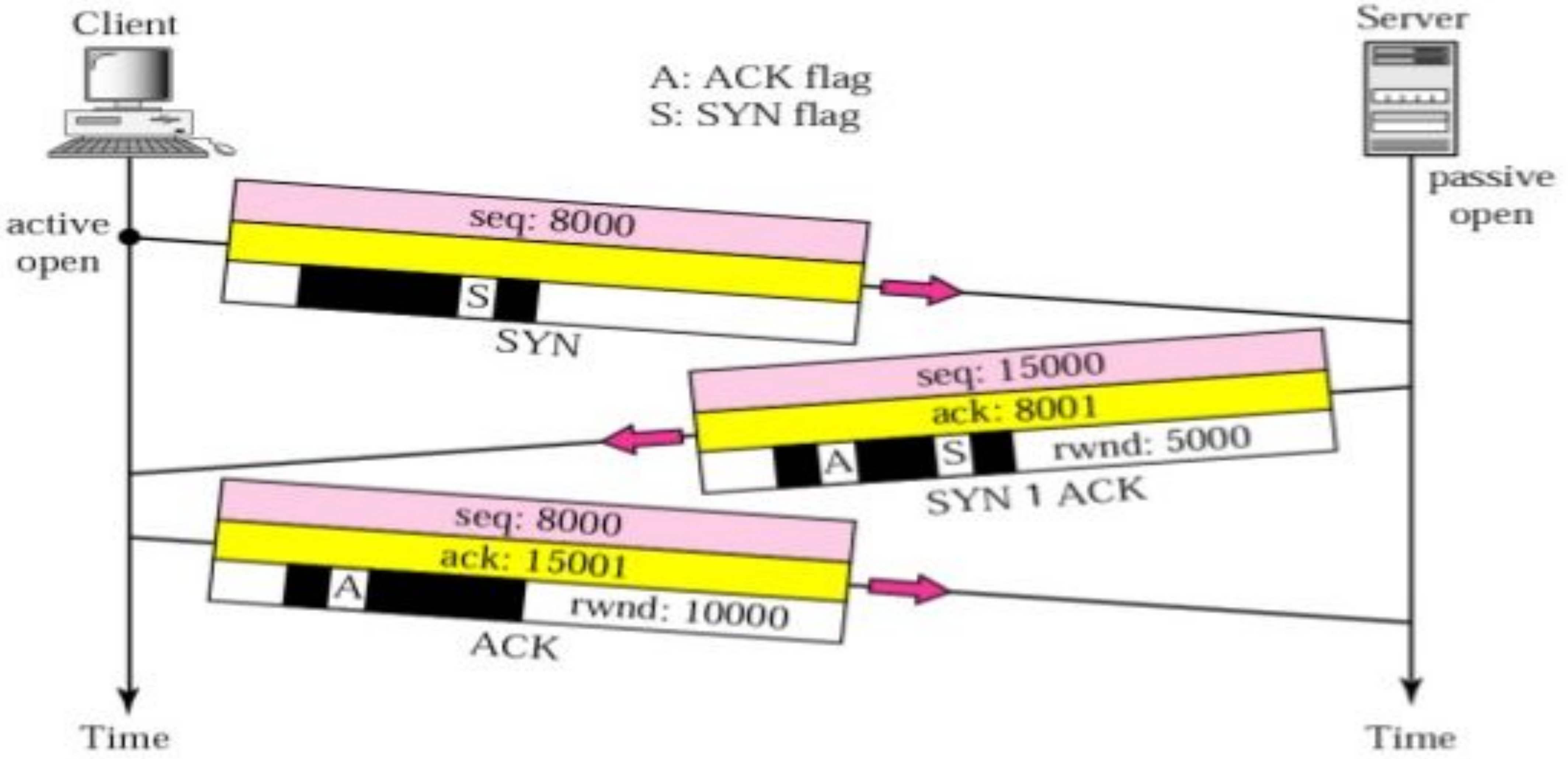
✔ The process starts with **Server**. The server program tells its TCP that it is ready to accept a connection. This request is called a **'Passive Open'.**

✔ The client program issues a request for an **'active open'**.

✔ A client that wishes to connect to an open server tells its TCP to connect to a particular server.

✔ TCP can now start the three-way handshaking process as shown in next figure.

CONNECTION ESTABLISHMENT USING 3 WAY HANDSHAKING



THREE STEPS

Step 1. The client sends the first segment: SYN segment-

- ✔ SYN flag is set (used for synchronization of sequence numbers.
- ✔ The client chooses random number as the first sequence number and send this number to the server.
- ✔ This sequence number is called as the **first sequence number** and sends this number to the server- **'Initial Sequence number '(ISN)**.
- ✔ This segment does not contain an acknowledgement number
- ✔ It does not define window size either, a window size definition makes sense only when a segment includes an acknowledgment.
- ✔ SYN segment is **control segment and carries no data**.
- ✔ It consumes one sequence number.
- ✔ When data transfer starts, the ISN is incremented by 1.

THREE STEPS

Step 2. The sender sends the second segment: SYN + ACK segment:

- ✔ **SYN + ACK** segment with two flags bits set: SYN and ACK.
- ✔ This segment has a dual purposes-
- ✔ 1) It is a SYN segment for the communication in the other direction. The server uses this segment to initialize a sequence number for numbering the bytes sent from server to the client.
- ✔ 2) Server also acknowledges the receipt of the SYN segment from the client by setting the ACK flag and displaying the next sequence number it expects to receive from the client. Because it contains an acknowledgement, it also needs to define the receiver window size, rwnd (to be used by the client).

THREE STEPS

Step 3. the client send the third segment.

- ✔ This is just an ACK segment
- ✔ It acknowledges the receipt of the second segment with the flag and acknowledgment number field.
- ✔ The sequence number in this segment is the same as the one in SYN segment, the ACK segment does not consume any sequence numbers.
- ✔ The client must also define the server window size.
- ✔ Some implementations allow this third segment in the connection phase to carry the first chunk of data flow the client.
- ✔ In this case, the third segment must have a new sequence number showing the byte number of the first byte in the data.
- ✔ The third segment usually does not carry data and consumes no sequence numbers.

CONNECTION TERMINATION

- Any of the two parties involved in exchanging data (client and server) can close the connection, although it is usually initiated by the client.
- Most implementations today allow **two options** for connection termination:

1. Three way handshaking
2. Four way handshaking with half close option

THREE-WAY HANDSHAKING

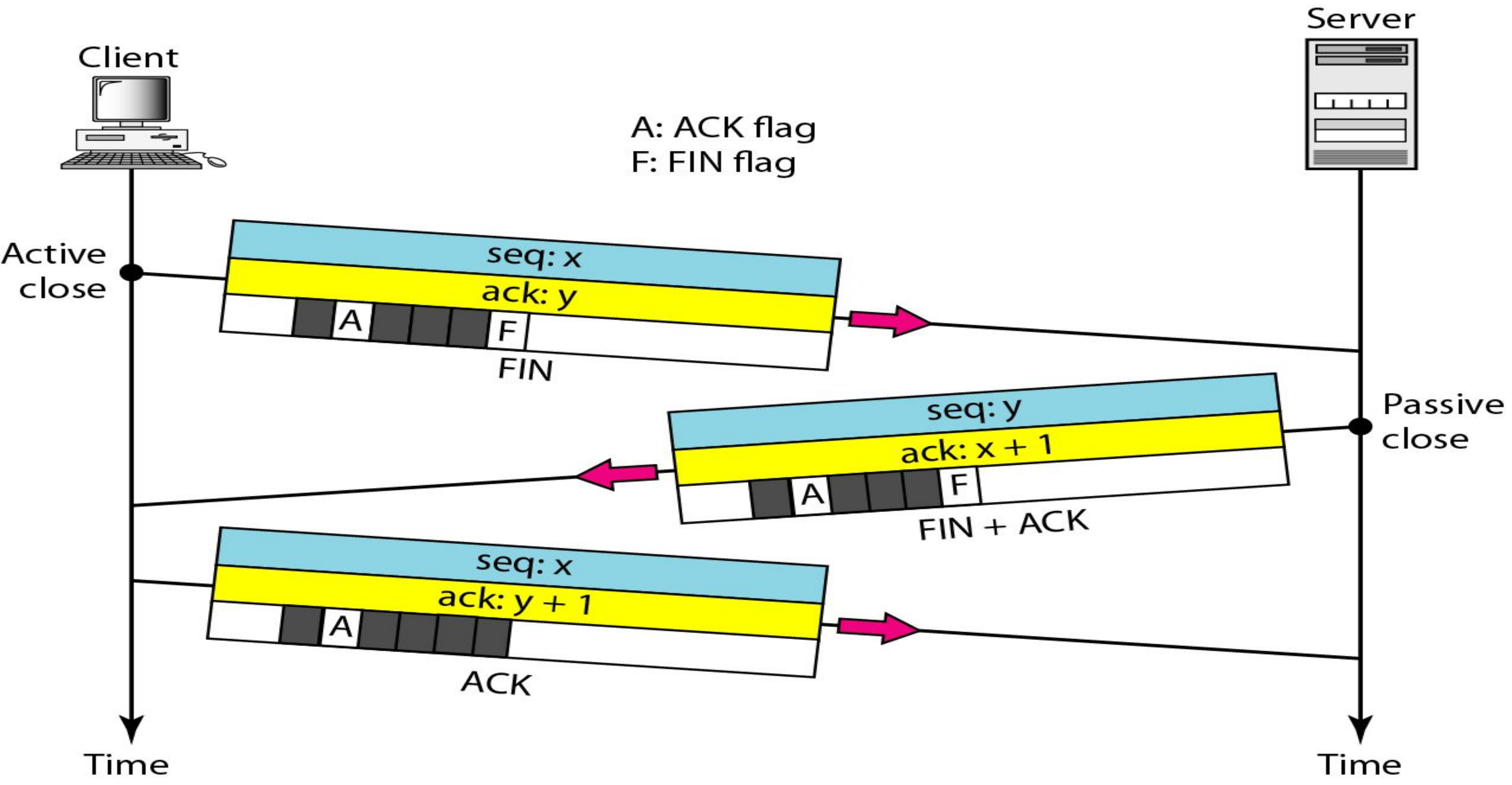
Step 1:

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

Step 2:

- ✔ The server TCP, after receiving the FIN request, 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.
- ✔ At the same time to announce the closing of the connection in the other directions.
- ✔ 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.

THREE-WAY HANDSHAKING



THREE-WAY HANDSHAKING

- **Step 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 acknowledgement number, which is one plus the sequence number received in FIN segment from the server.
- ✔ This segment cannot carry data and consumes no sequence numbers.

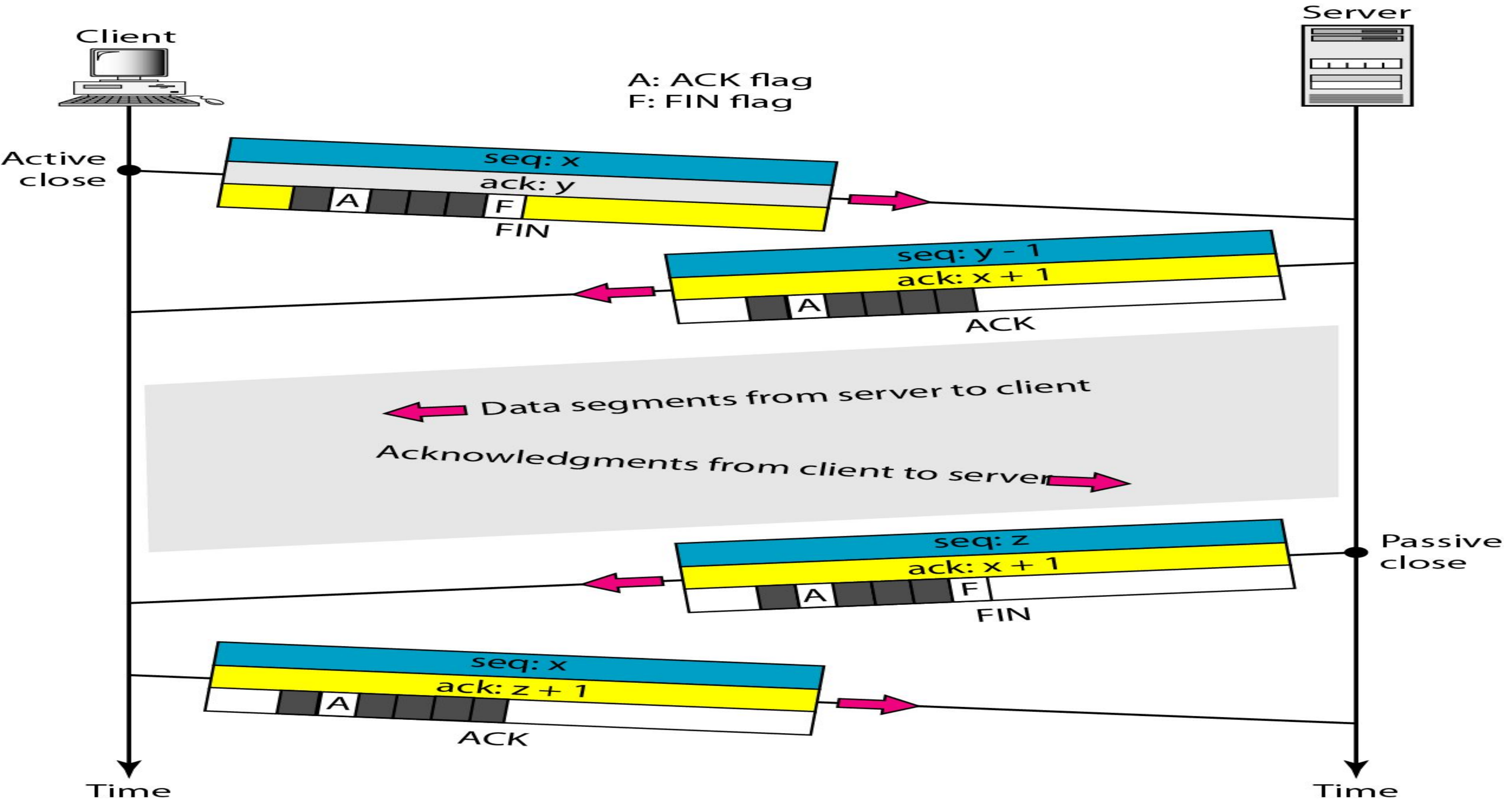
HALF CLOSE

- In TCP, one end can stop sending data while still receiving data. This is called a **Half-close.**
- 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 send data to the server to be sorted, the server needs to receive all the data before sorting can start.
- This means that the client, after sending all data, can close the connection in the client-to-sever 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.

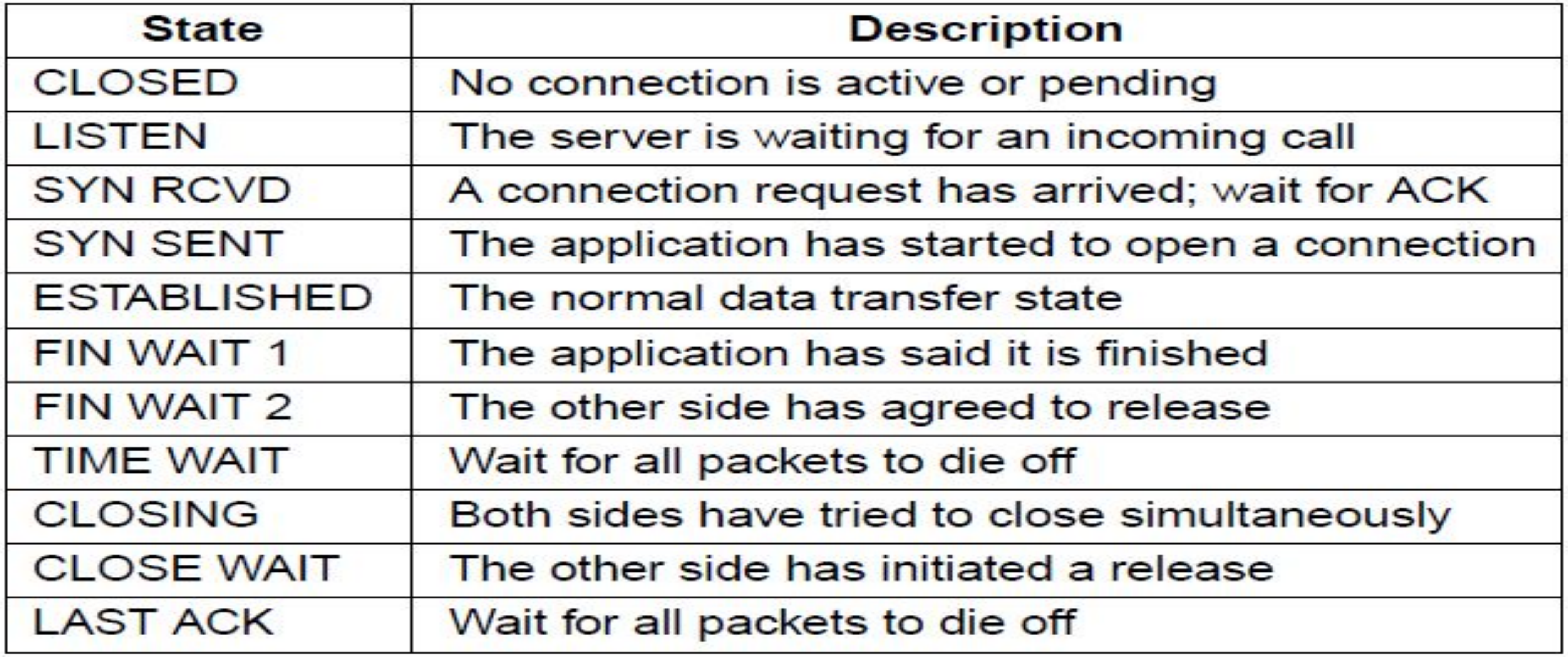
HALF CLOSE

- Next 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, can still send data.
- When the server has sent all of the processes data, it sends a FIN segment, which is acknowledged by an ACK from client.
- After half closing the connection, data can travel from server to the client and acknowledgement can travel from the client to the server.
- The client cannot send any more data with server.
- The second segment (ACK) consumes no sequence number.
- Although the client has received sequences number y-1 and is expecting y, the server sequence number is still y-1.
- When the connection finally close, the sequence number of the last ACK segment is still x, because no sequence numbers are consumed during data transfer in that direction.

HALF CLOSE

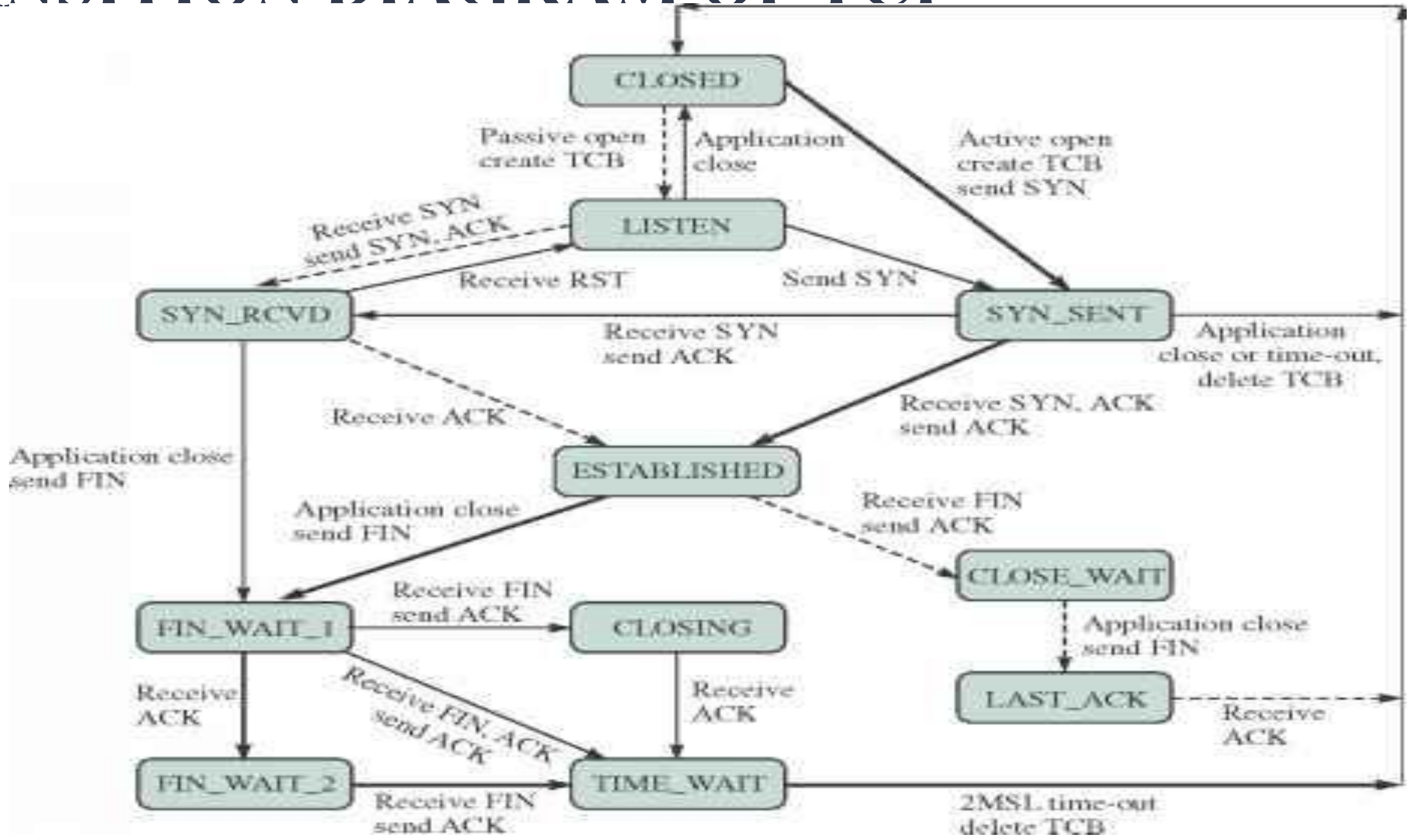

Client
Server
A: ACK flag
F: FIN flag
Active close
seq: x
ack: y
A
F
FIN
seq: y - 1
ack: x + 1
A
ACK
Data segments from server to client
Acknowledgments from client to server
Passive close
seq: z
ack: x + 1
A
F
FIN
seq: x
ack: z + 1
A
ACK
Time
Time

TCP CONNECTION MANAGEMENT MODELING



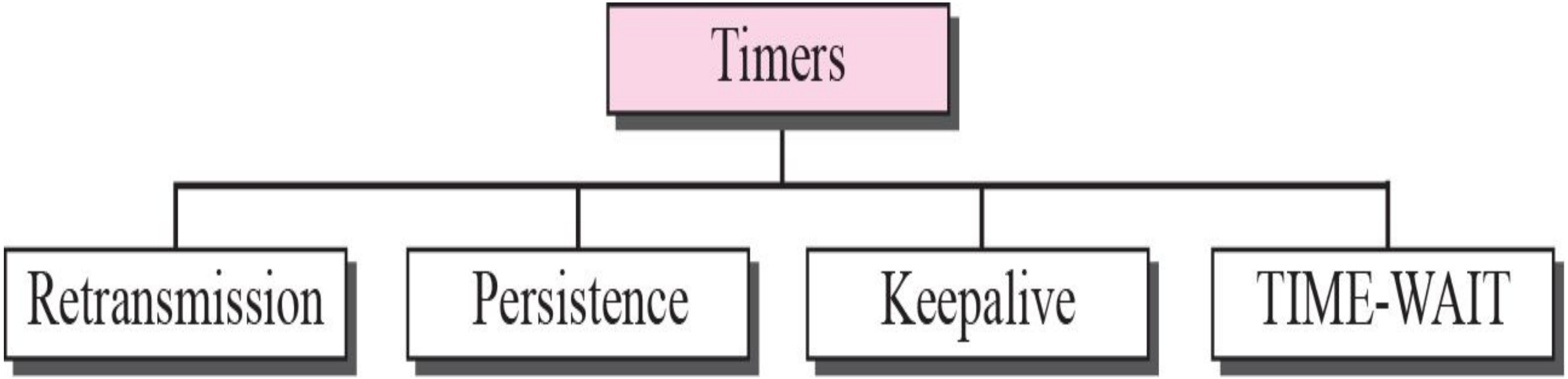
State	Description
CLOSED	No connection is active or pending
LISTEN	The server is waiting for an incoming call
SYN RCVD	A connection request has arrived; wait for ACK
SYN SENT	The application has started to open a connection
ESTABLISHED	The normal data transfer state
FIN WAIT 1	The application has said it is finished
FIN WAIT 2	The other side has agreed to release
TIME WAIT	Wait for all packets to die off
CLOSING	Both sides have tried to close simultaneously
CLOSE WAIT	The other side has initiated a release
LAST ACK	Wait for all packets to die off

STATE TRANSITION DIAGRAM OF TCP



TCP TIMERS

- To perform its operation smoothly, most TCP implementations use at least four timers as shown in figure.



- ✓ Persistence Timer
- ✓ Keepalive Timer
- ✓ TIME-WAIT Timer
- ✓ Retransmission Timer

1. RETRANSMISSION TIMER

- To decide when to retransmit Retransmission timer is used

- There are two conditions to decide when to retransmit:

1. Timer is time out
2. Three duplicate acknowledgement are received.

2. PERSISTENCE TIMER

- To correct this deadlock, TCP uses a **persistence timer for each connection.**
- When the sending TCP receives an acknowledgment with a window size of zero, it starts a persistence timer .
- When the persistence timer goes off, the sending TCP send a special segment called a **Probe.**
- This segment contains only 1 byte of new data. It has a sequence number but sequence number is never acknowledged.
- It is even ignored in calculating the sequence number for the rest of the data.
- The probe causes the receiving TCP to resend the acknowledgment.
- The value of the persistence timer is set to the value of the retransmission time.
- If a response is not received from the receiver, another probe segment is sent and the value of the persistence timer is doubled and reset.
- The sender continues sending the probe segments and doubling and resetting the value of the persistence timer until the value reaches a threshold (usually 60s).
- After that the sender sends one probe segment every 60s until the window is reopened.

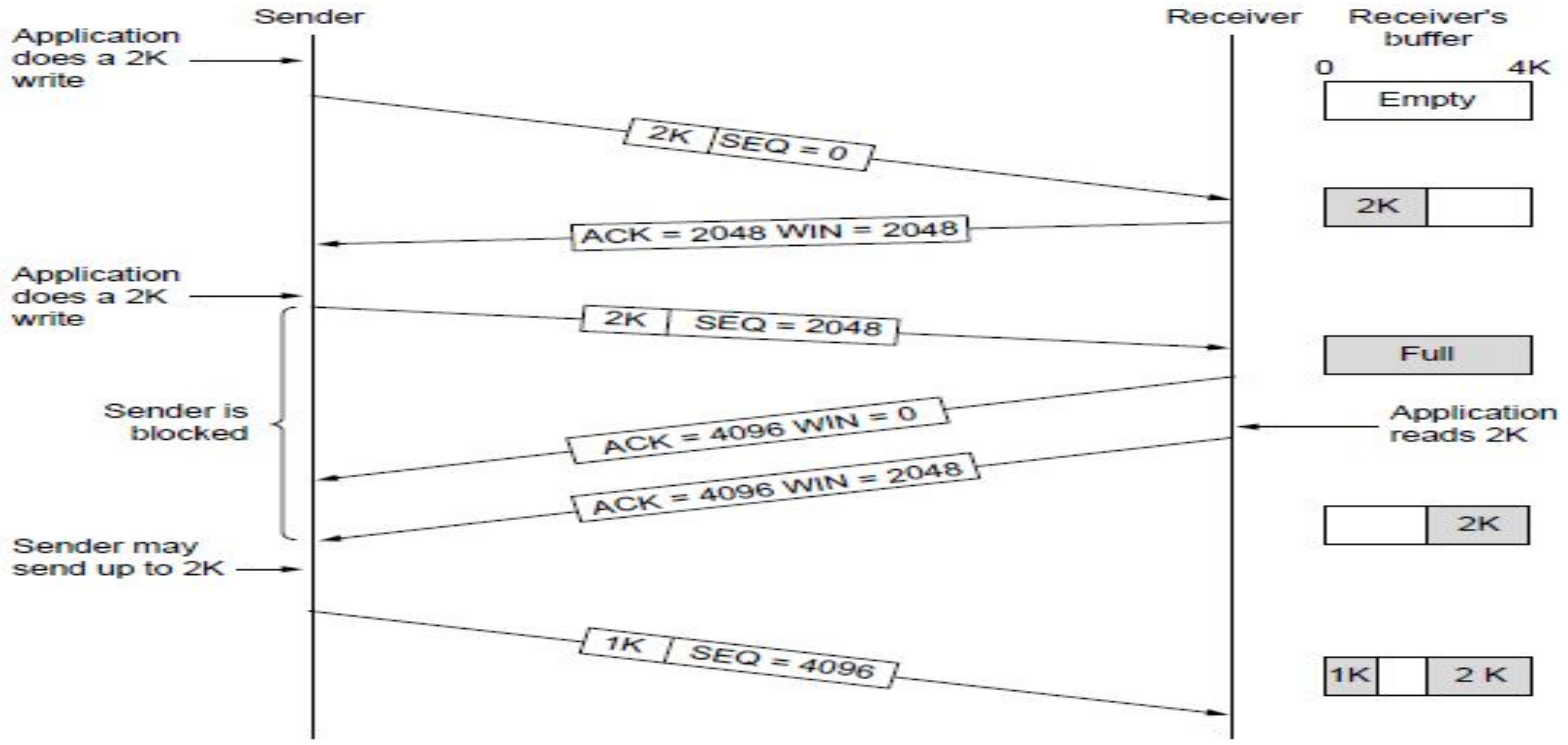
3. KEEPALIVE TIMER

- A keepalive timer is used in some implementations to prevent a long idle connection between two TCPs.
- Suppose that a client opens a TCP connection to a server, transfers some data, and becomes silent.
- Perhaps the client has crashed. In this case, the connection remains open forever.
- To remedy this situation, most implementations equip a server with a keepalive timer.
- Each time the server hears from a client, it resets this timer.
- The timeout is usually 2 hours.
- If the server does not hear from the client after 2 hrs, it sends a probe segment.
- If there is no response after 10 probes, each of which is 75s apart, it assumes that the client is down and terminates the connection.

4. TIME-WAIT timer:

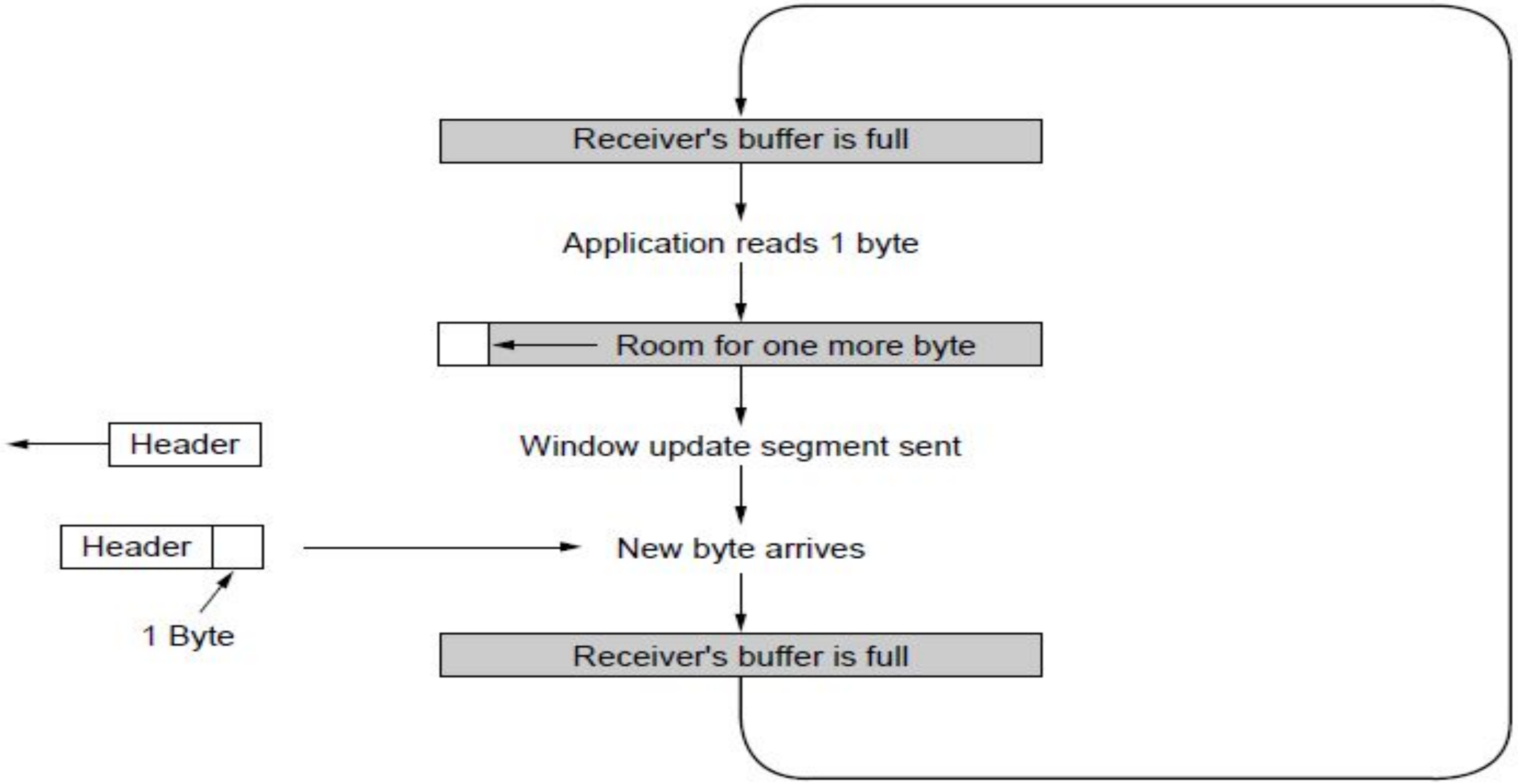
- The TIME-WAIT (2MSL) timer is used during connection termination.

TCP SLIDING WINDOW



Window management in TCP

TCP SLIDING WINDOW



Silly window syndrome

TCP TRANSMISSION POLICY: SILLY WINDOW SYNDROME

- A serious problem can arise in the sliding window operation when either the sending application program creates data slowly or the receiving application program consumes data slowly, or both.
- Any of these situations results in the sending of data in very small segments, which reduces the efficiency of the operation.
- E.g. if TCP sends the segments containing only 1 byte of data, it means that a 41-byte datagram (20 bytes of TCP header and 20 bytes of IP header) transfers only 1 byte of user data. Here the overhead is 41/1, which indicates that we are **using capacity** of the network very **inefficiently**.
- The inefficiency is even worse after accounting for the data link layer and physical layer overhead. This problem is called the **Silly Window Syndrome.**

SYNDROME CREATED BY THE SENDER

- The sending TCP may create a silly window syndrome if it is serving an application program that creates data slowly. Eg, 1 byte at a time.
- If the sending TCP does not have any specific instructions, it may create segments containing 1 byte of data. The result is a lot of 41 byte segments that are travelling through an internet.
- The solution is to prevent the sending TCP from sending the data byte by byte. The sending TCP must be forced to wait and collect data to send in larger block.
- Sending TCP waits too long, it may delay the process.
- If it does not wait long enough , it may end up sending small segment.
- Nagle found an elegant solution.

NAGLE’S ALGORITHM

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

Step 2: After sending the first segment the sending TCP accumulates data in the output buffer and waits until either the receiving TCP send an acknowledgment or until enough data has accumulated to fill a maximum size segment. At this time, the sending TCP is send the segment.

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

- ✔Nagle algorithm takes into account the speed of the application program that creates the data and the speed of the network that transports the data.
- ✔If the application program is faster than the network, the segments are larger. (maximum size segments).
- ✔If the application program is slower than the network, the segments are smaller. (less than maximum size segments).

SYNDROME CREATED BY THE RECEIVER

- The receiving TCP may create a silly window syndrome if it is serving an application program that consumes data slowly.eg, 1 byte at a time.
- The receiving TCP announces a window size of 1 byte, which means that the sending TCP, which is eagerly waiting to send data, take thus advertisement as good news and sends a segment carrying only 1 byte of data, the procedure will continue.
- One byte of consumed and a segment carrying 1 byte of data is sent. Again we have an efficiency problem and silly window syndrome.
- **Two solutions** have been proposed to prevent the silly window syndrome created by an application program that consumes data slower that they arrive.

1. Clark's solution
2. Delayed Acknowledgement

CLARK’S SOLUTION

- Clark’s solution is to send an acknowledgement as soon as the data arrive, but to announce a window size of zero until **either there is enough space** to accommodate a segment of maximum size or **until at least half of the receive buffer is empty.**

DELAYED ACKNOWLEDGMENT

- This means that when a segment arrives, it is not acknowledged immediately.
- The receiver waits until there is a decent amount of space in its incoming buffer before acknowledging the arrived segments.
- The delayed acknowledgement **prevents the sending TCP from sliding its window.**
- After sending TCP has sent the data in the window, it stops. This kills the syndrome.

Advantage:

✔ It reduces traffic.

✔ The receiver does not have to acknowledge each segment.

Disadvantage:

✔ delayed acknowledgement may result in the sender unnecessarily retransmitting the unacknowledged segments.

- TCP balances the advantages and disadvantages.
- It now defines that the acknowledgement should not be delayed by more that 500ms.

FLOW CONTROL ,ERROR CONTROL AND CONGESTION CONTROL

- **Flow control:**
- Unlike UDP, TCP provides flow control.
- The sending TCP controls how much data can be accepted from the sending process.
- The receiving TCP controls how much data can be sent by the sending TCP.
- This is done to prevent the receiver from being overwhelmed with the data.
- The numbering system allows TCP to use a byte oriented flow control.

- **Error Control:**
- To provide reliable service, TCP implements an error control mechanism.
- Although error control considers a segment as the unit for error detection (loss or corrupted), error control is byte oriented.

- **Congestion control:**
- Unlike UDP, TCP takes into account congestion in the network.
- The amount of data sent by a sender is not only controlled by the receiver (flow control) but is also determined by the level of congestion, if any, in the network.

ERROR CONTROL

- TCP **provides reliability** using error control.
- Error control includes
  - ✓ Mechanisms for detecting and resending corrupted segments
  - ✓ Resending lost segments
  - ✓ Storing out of order segments until missing segments arrive
  - ✓ Detecting and discarding duplicated segments.

- Error control achieved through the use of **three simple tools**:

1. Checksum
2. Acknowledgment
3. Time-out

ERROR CONTROL TOOLS

1. Checksum:

- Each segment includes checksum field which is used to check for a corrupted segments.
- If a segment is corrupted as deleted by an invalid checksum, the segment is discarded by the destination TCP and is considered as lost.
- TCP uses a 16-bit checksum that is mandatory in every segment.

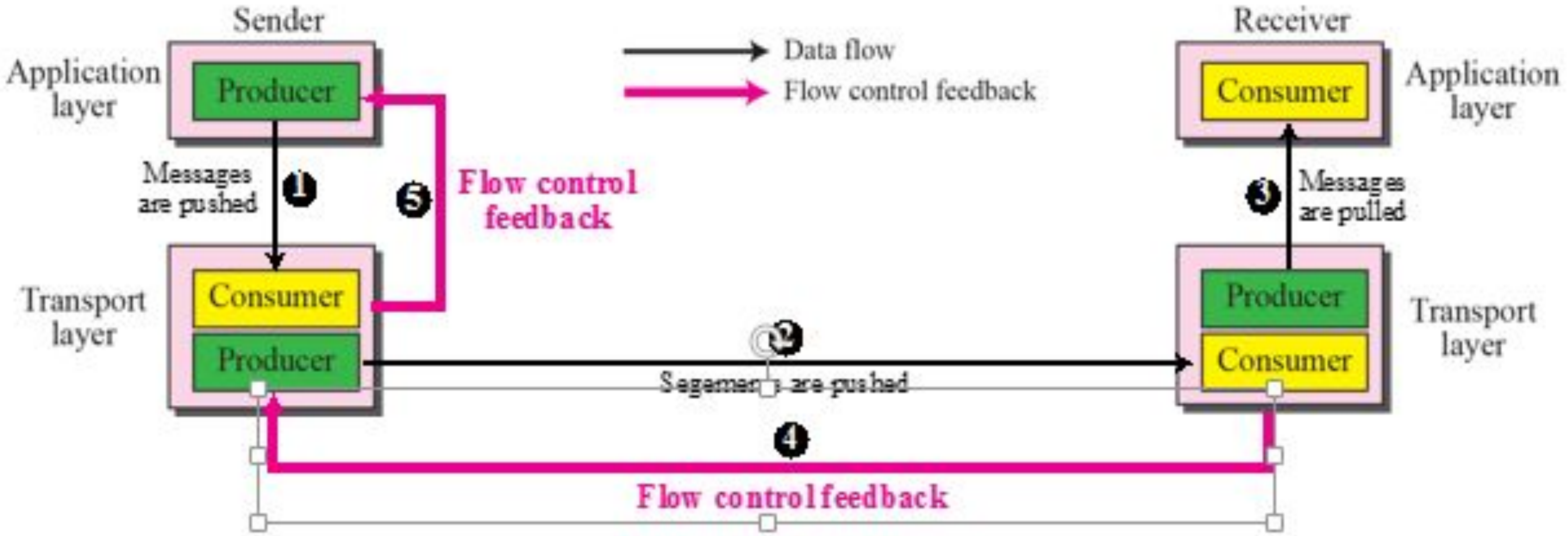
2. Acknowledgment:

- TCP uses acknowledgement to confirm the receipt of data segments.
- Control segments that carry no data, but consumes a sequence number are also acknowledged.
- ACK segments are never acknowledged.

FLOW CONTROL

- Flow control balances the rate a producer creates data with the rate a consumer can use the data.
- TCP separates flow control from error control.
- Next figure shows the unidirectional data transfer between a sender and a receiver.
- The receiving TCP controls the sending TCP; the sending TCP controls the sending process.
- Flow control feedback from the sending TCP to the sending process (path 5) is achieved through simple rejection of data by sending TCP when its window is full.

FLOW CONTROL

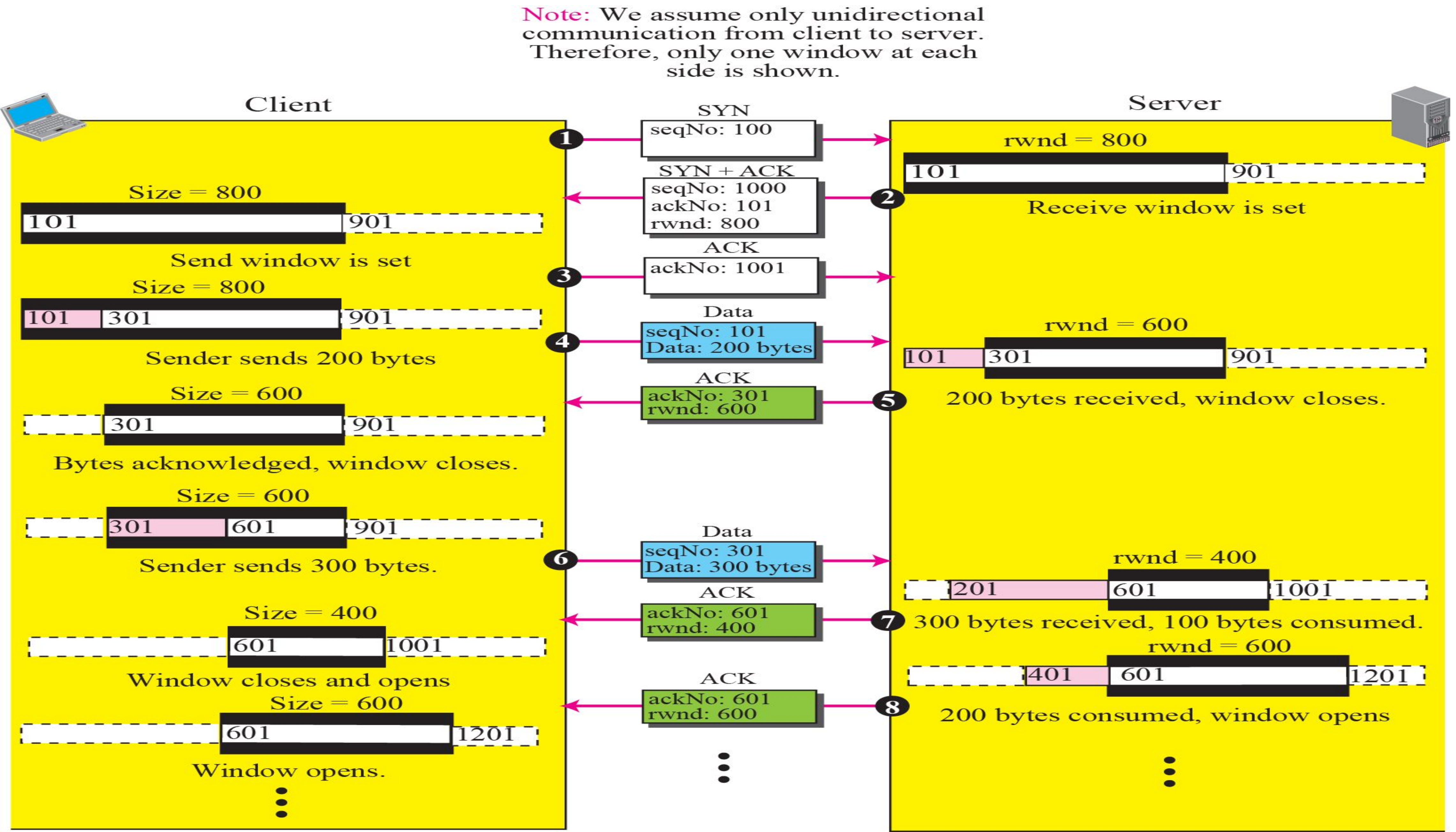


✔Path1: data travel from the sending process down to the sending TCP
✔Path 2: data travel from the sending TCP to the receiving TCP
✔Path 3: data travel from receiving TCP up to the receiving process
✔Path 4: flow control feedback travelling from the receiving TCP to the sending TCP
✔Path 5: flow control feedback travelling from the sending TCP to sending process

OPENING AND CLOSING WINDOWS

- To achieve flow control, TCP forces the sender and receiver to adjust their window sizes, although the size of the buffer for both parties is fixed when the connection is established.
- The receive window closes (moves its left wall to the right)when more bytes arrive from the sender; it opens (moves its right wall to the right) when more bytes are pulled by the process.
- Assuming that is does not shrink (the right wall does not move to the left.)
- The opening, closing and shrinking of the send window is controlled by the receiver.
- The send window closes (moves its left wall to right) when a new acknowledgement allows it to do so.
- The send window opens (its right wall moves to the right) when the receive window size (rwnd) advertised by the receiver allows it to do so.
- The send window shrinks on occasion.

A EXAMPLE OF FLOW CONTROL



A EXAMPLE OF FLOW CONTROL

- Eight segments are exchanged between the client and server:

1. First segment is from the client to the server (a SYN segment) to request connection. Client announces its sequence number= 100, rwnd=800, next byte to arrive is 101.
2. Second segment is from the server to client. This is an ACK+SYN segment, uses ack no=101 to show that it expects to receive bytes starting from 101.it also announces that the client can set a buffer size of 800 bytes.
3. Third segment is the ACK segment from the client to server.
4. After the client has set its windows with the size (800) dictated by the server, the process pushes 200 bytes of data. The TCP client numbers these bytes 101 to 300. It then creates a segment and sends it to the server. Starting byte 101 and carries 200 bytes. When this segment is received at the server, the bytes are stored and receive window closes to show that the next byte expected is byte 301; the stored bytes occupy 200 bytes of buffer.
5. Fifth segment is feedback from server to the client. The server acknowledges bytes up to and including 300 (expecting receive byte 301). The segment also carries the size of the receive window after decrease 600. the client after receiving this segment, purges the acknowledged bytes from its window and closes its window to show that the next byte to send is byte 301. the window size, decreases to 600 bytes. Although the allocated buffer can store 800 bytes, the window cannot open (moving its right wall to the right) because the receiver does not let it.

A EXAMPLE OF FLOW CONTROL

6. Segment 6 is sent by the client after its process pushes 300 more bytes. The segment defines seqNo as 301 and contains 300 bytes. When this segment arrives at the server, the server stores them, but it has to reduce its window size.

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

8. Segment 8 is also from the server after its process has pulled another 200 bytes. Its window size increases. The new rwnd value is now 600. the segment informs the client that the server still expects byte 601, but the server window size has expanded to 600. the sending of this segment depends on the policy imposed by the implementation. Some implementation may not allow advertisement of the rwnd at this time; the server then needs to receive some data before doing so. After this segment arrives at the client, the client opens its window by 200 bytes without closing it. The result is that its window size increases to 600 bytes.

CONGESTION CONTROL

- Congestion control in TCP is based on both open loop and closed loop mechanisms.
- TCP uses a **congestion window** and a **congestion policy** that avoid congestion and detect and alleviate congestion after it has occurred.

CONGESTION WINDOW

- The sender window size is determined by the available buffer space in the receiver (rwnd).
- It is only the receiver that can dictate to the sender the size of the sender’s window, ignoring another entity- network.
- If the network cannot deliver t he data as fast as it is created by the sender, it must tell the sender to slow down.
- In other words, in addition to the receiver, the network is the second entity that determines the size of the sender’s window.
- The sender has two pieces of information-
  1. Receiver advertised window size
  2. Congestion window size

The actual size of the window is the minimum of these two.

Actual window size = minimum (rwnd, cwnd)

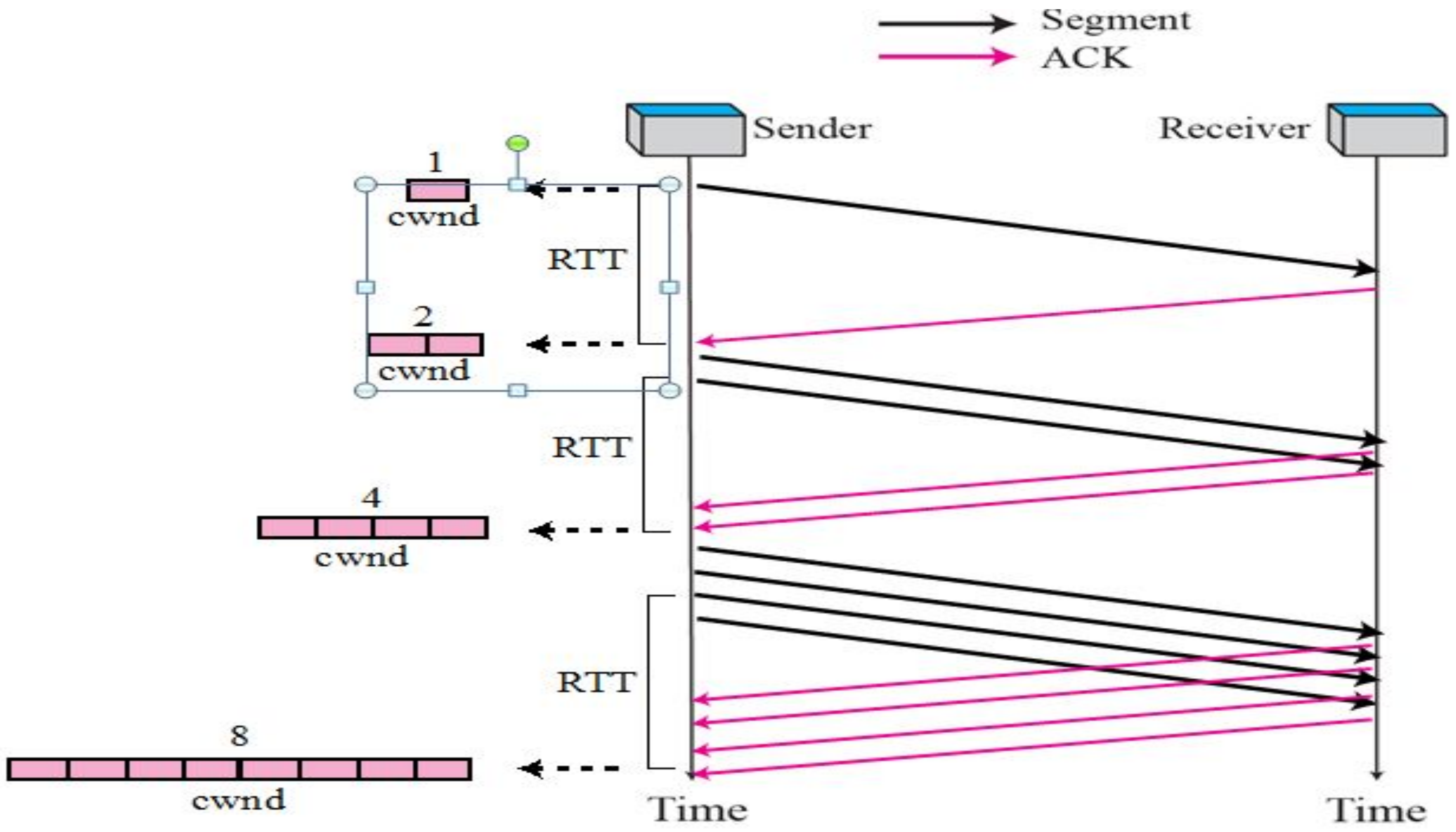
CONGESTION POLICY

- TCP’s general policy for handling congestion is based on three phases: **slow start, congestion avoidance and congestion detection**.
- In the slow start phase, the sender starts with a slow rate of transmission, but increases the rate rapidly to reach a threshold.
- When the threshold is reached , the rate of increase is reduced.
- Finally if ever congestion is detected, the sender goes back to the slow start or congestion avoidance phase based on how the congestion is detected.

SLOW START: EXPONENTIAL INCREASE

- The slow start algorithm is based on the idea that size of the congestion (cwnd) starts with one maximum segment size (MSS).
- The MSS is determined during connection establishment using an option of the same name.
- The size of the window increases one MSS each time one acknowledgement arrives.
- As the name implies, the algorithm starts slowly, but grow exponentially.
- The idea is shown in next figure.
- We assume that rwnd is much longer than cwnd, so that the sender window size always equals cwnd.
- For simplicity, we ignore delayed ACK policy and assume that each segment is acknowledged individually.

SLOW START: EXPONENTIAL INCREASE



SLOW START: EXPONENTIAL INCREASE

- The sender starts with cwnd=1 MSS.
- This means that the sender can send only one segment.
- After the first ACK arrives, the size of the congestion window is increased by 1, which means that cwnd is now 2. now 2 segments can be sent.
- When two more ACKs arrive, the size of the window is increased by 1 MMS for each ACK, which means cwnd is now 4. now 4 segments can be sent.
- When four ACK arrive, the size of the window increases by 4, which means that cwnd is now 8.

SLOW START: EXPONENTIAL INCREASE

- The size of cwnd in terms of round trip timers (RTTs), the growth rate is exponential.

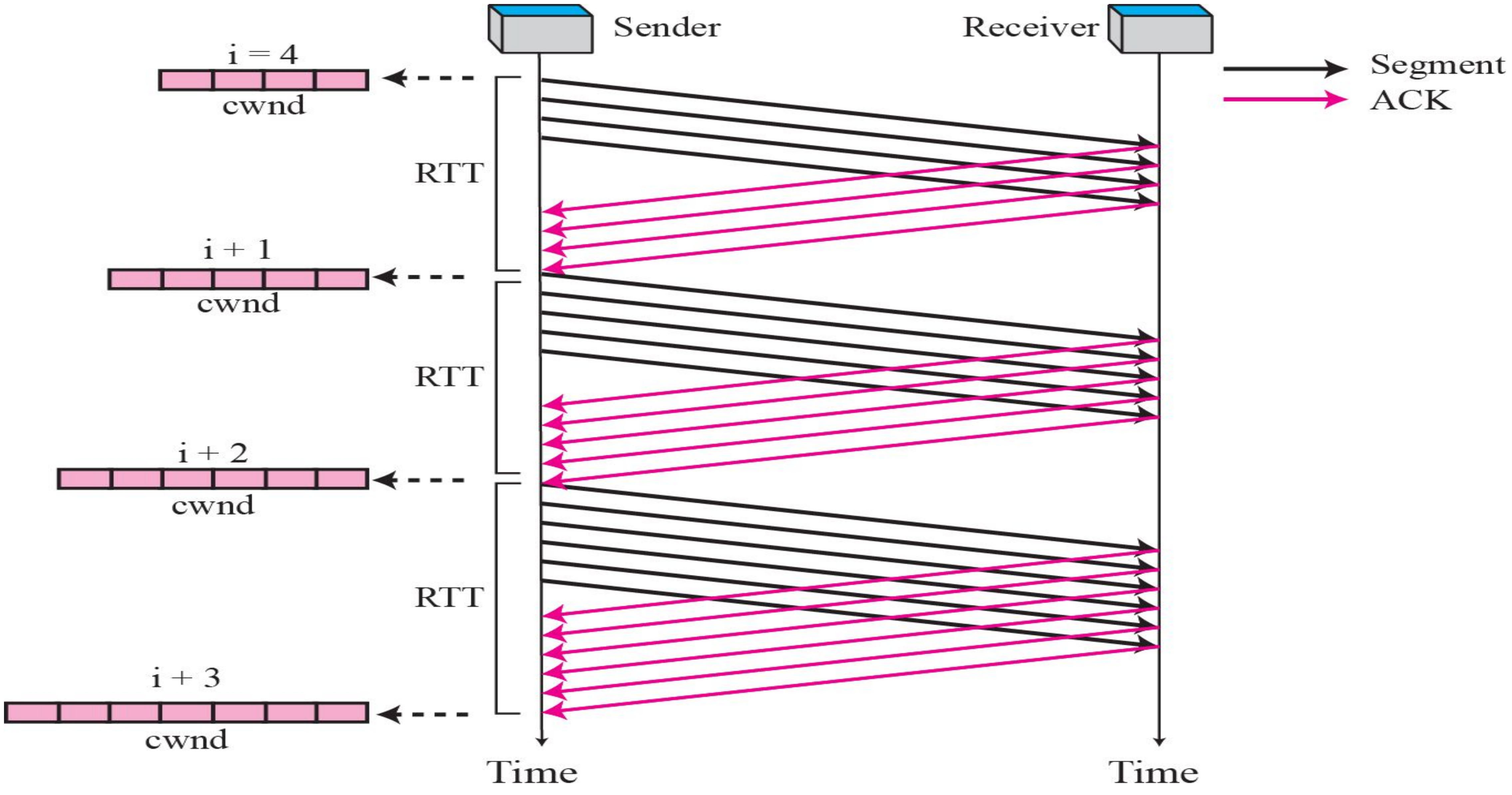
Start	Cwnd=1
After 1 RTT	Cwnd = 1*2=2 = 2^1
After 2 RTT	Cwnd = 2* 2= 4= 2^2
After 3 RTT	Cwnd = 4* 2= 8

- To mention that the slower in the case of delayed acknowledgment.
- For each ACK, the cwnd is increased by only 1 MSS.
- If three segments are acknowledged accumulatively, the size of the cwnd increases by only 1 MSS, not 3 MSS.
- The growth is still exponential but it is not a power of 2.
- With one ACK for every 2 segments, the power is closer to 2.
- Slow start cannot continue indefinitely. There must be threshold to stop this phase.
- The sender keeps track of a variable named **sshresh (slow start threshold) .**
- When the size of window in bytes reaches this threshold, slow stops and the next phase starts.

CONGESTION AVOIDANCE: ADDITIVE INCREASE

- To avoid congestion before it happens, we must slow down the exponential growth occurring in slow start algorithm.
- TCP defines another algorithm, called **Congestion avoidance,** which increase the cwnd additively instead of exponentially.
- Next figure shows the idea.
- When the size of the congestion window reaches the slow start threshold in the case where cwnd=1, the slow start phase stops and the additive phase begins.
- In this algorithm, each time the whole ‘window’ of segment is acknowledged, the size of the congestion window is increased by one.
- A window is the number of segments transmitted during RTT.
- The increase is based on RTT, not on the number of arrived ACKs.

CONGESTION AVOIDANCE: ADDITIVE INCREASE



CONGESTION AVOIDANCE: ADDITIVE INCREASE

- To show the idea, we apply this algorithm to the same scenario as slow start.
- In this case, after the sender has received acknowledgments for a complete window size of the segments, the size of the window is increased one segment.
- If we look at the size of cwnd in terms of RTT, we find the rate is additive as shown below.

Start	cwnd=i
After 1 RTT	cwnd = i+1
After 2 RTT	cwnd = i+2
After 3 RTT	cwnd = i+3

In the congestion avoidance algorithm the size of the congestion window increases additively until congestion is detected.

CONGESTION DETECTION: MULTIPLICATIVE DECREASE

- If the congestion occurs, the congestion window size must be decreased.
- The only way a sender can guess that congestion has occurred is the need to retransmit a segment. This is the major assumption made by TCP.
- Retransmission is needed to recover a missing packet which is assumed to have been dropped (i.e. lost) by a router that had so many incoming packets, that had to drop the missing segment, i.e., the router/network became overloaded or congested.
- However, retransmission can occur in one of two case: 1. when the RTO timer times out or 2. when three duplicates ACKs are received.
- In both cases, the size of the threshold is dropped to half (**multiplicative decrease**).
- Most TCP implementation have two reactions.

CONGESTION DETECTION: MULTIPLICATIVE DECREASE

- Two Reactions of TCP implementations:

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

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

b) It reduces cwnd back to one segment.

c) It starts the slow start phase again.

2. If three duplicate ACKs are received, there is a weaker possibility of congestion; a segment may have been dropped out some segments after that have arrived safely since three duplicates ACKs are received. This is called Fast transmission and fast recovery. In this case, TCP has a weaker reaction as shown below.

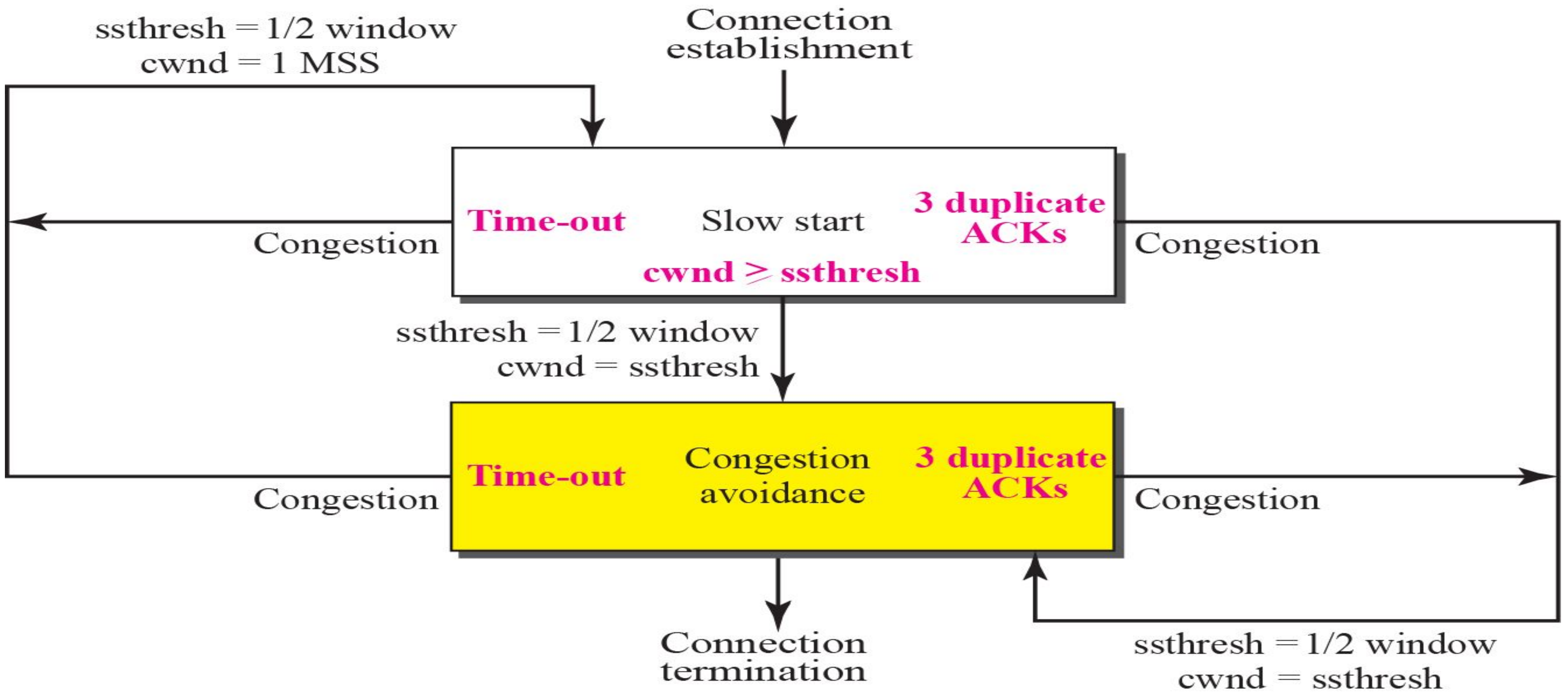
a) It sets the value of the threshold to half of the current window size.

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

c) It starts the congestion avoidance phase.

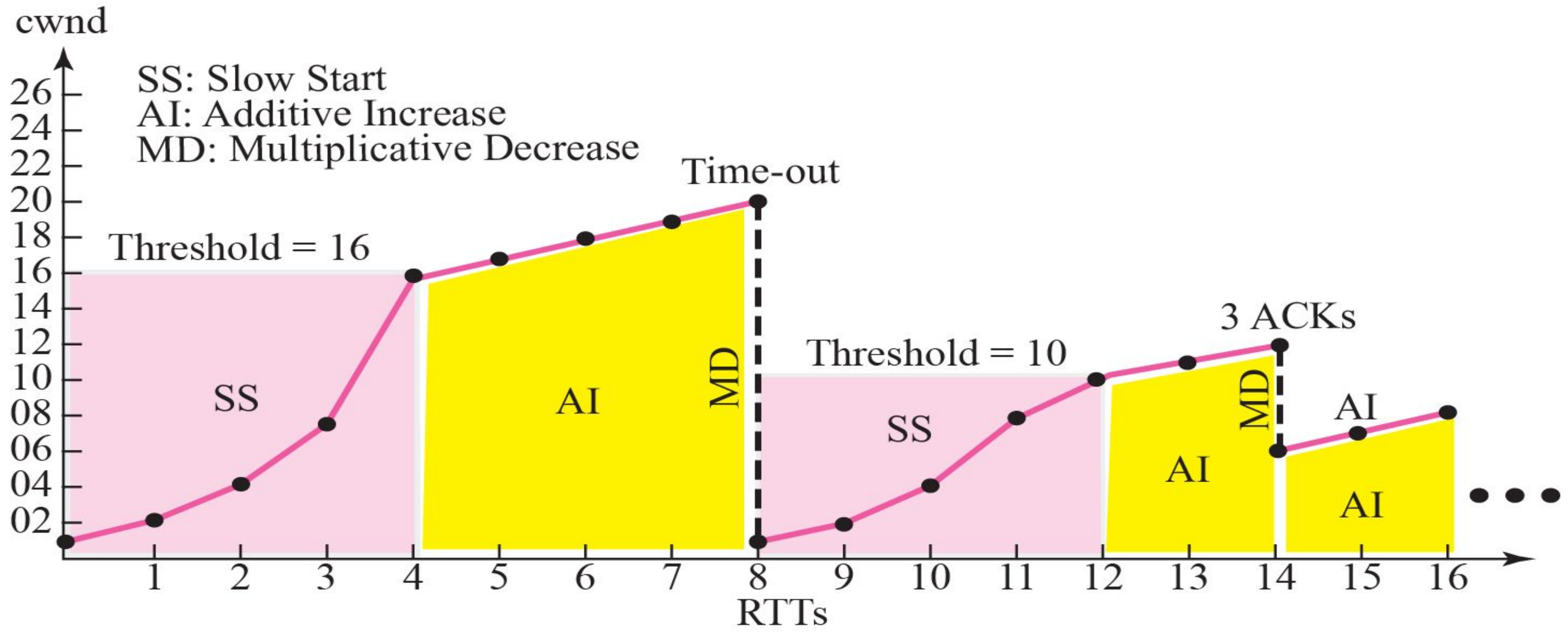
SUMMARY

- Next figure shows summary of the congestion policy of TCP and the relationships between the three phases.



CONGESTION EXAMPLE

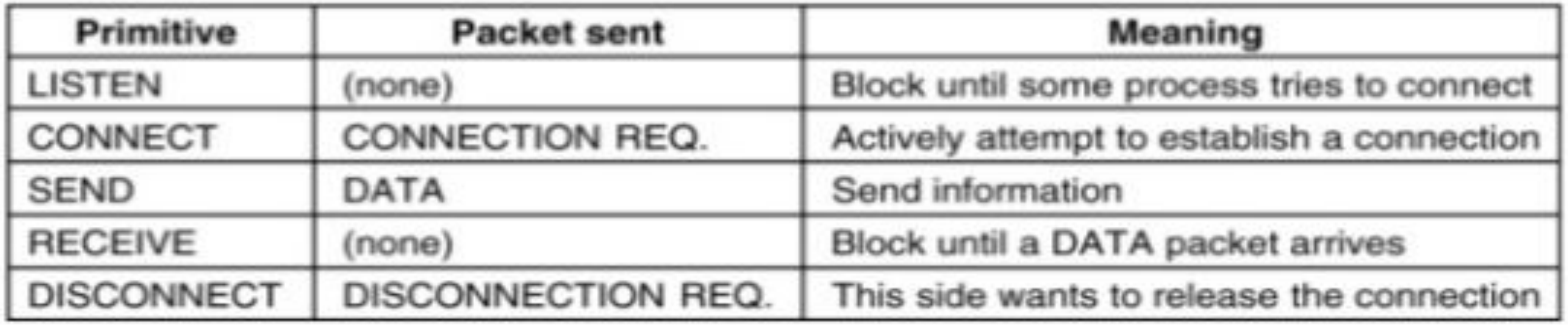
- Assuming maximum window size initially as 32 segments.
- The threshold is initially set to 16 segments (half of the maximum window size).
- In the slow start phase the window size starts from 1 and grows exponentially until it reaches the threshold.
- After reaching the threshold, the congestion avoidance (additive increase) procedure allows the window size to increase linearly until a time out occurs or the maximum window size is reached.



CONGESTION EXAMPLE

- In the figure, a time out occurs when the window size is 20.
- At this moment, the multiplicative decrease procedure takes over and reduces the threshold to half of the window size.
- The window size was 20 when the timeout happened so the new threshold is now 10.
- TCP moves to slow start again and starts with a window size of 1 and moves to additive increase when the new threshold is reached.
- When the window size is 12, a three ACKs event happens.
- The multiplicative decrease procedure takes over again.
- The threshold and window size set to 6 and TCP enters the additive increase phase this time.
- TCP remains in this phase until another timeout or another three ACKs event happens.

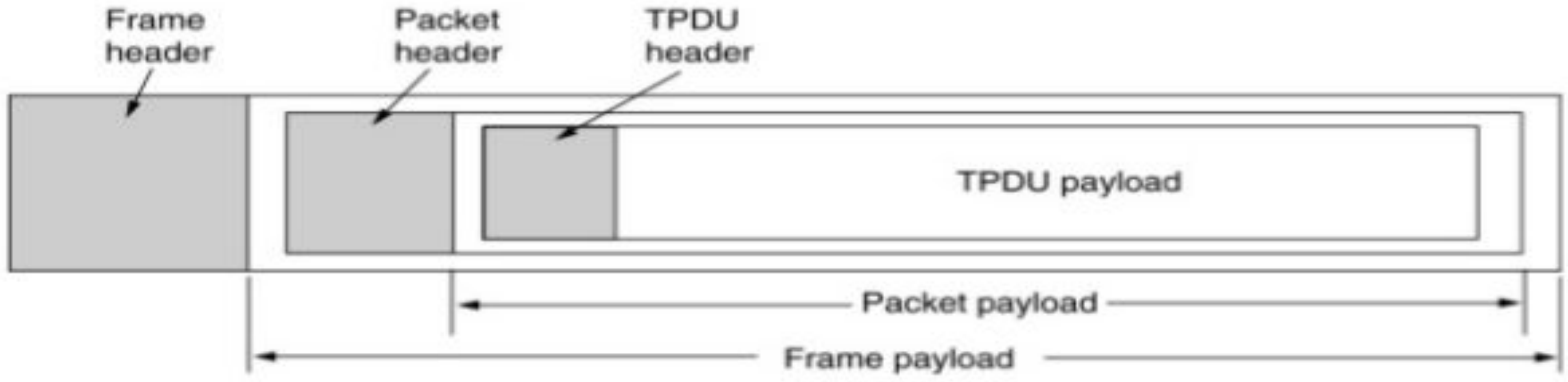
5.2 TRANSPORT SERVICES PRIMITIVES



Primitive	Packet sent	Meaning
LISTEN	(none)	Block until some process tries to connect
CONNECT	CONNECTION REQ.	Actively attempt to establish a connection
SEND	DATA	Send information
RECEIVE	(none)	Block until a DATA packet arrives
DISCONNECT	DISCONNECTION REQ.	This side wants to release the connection

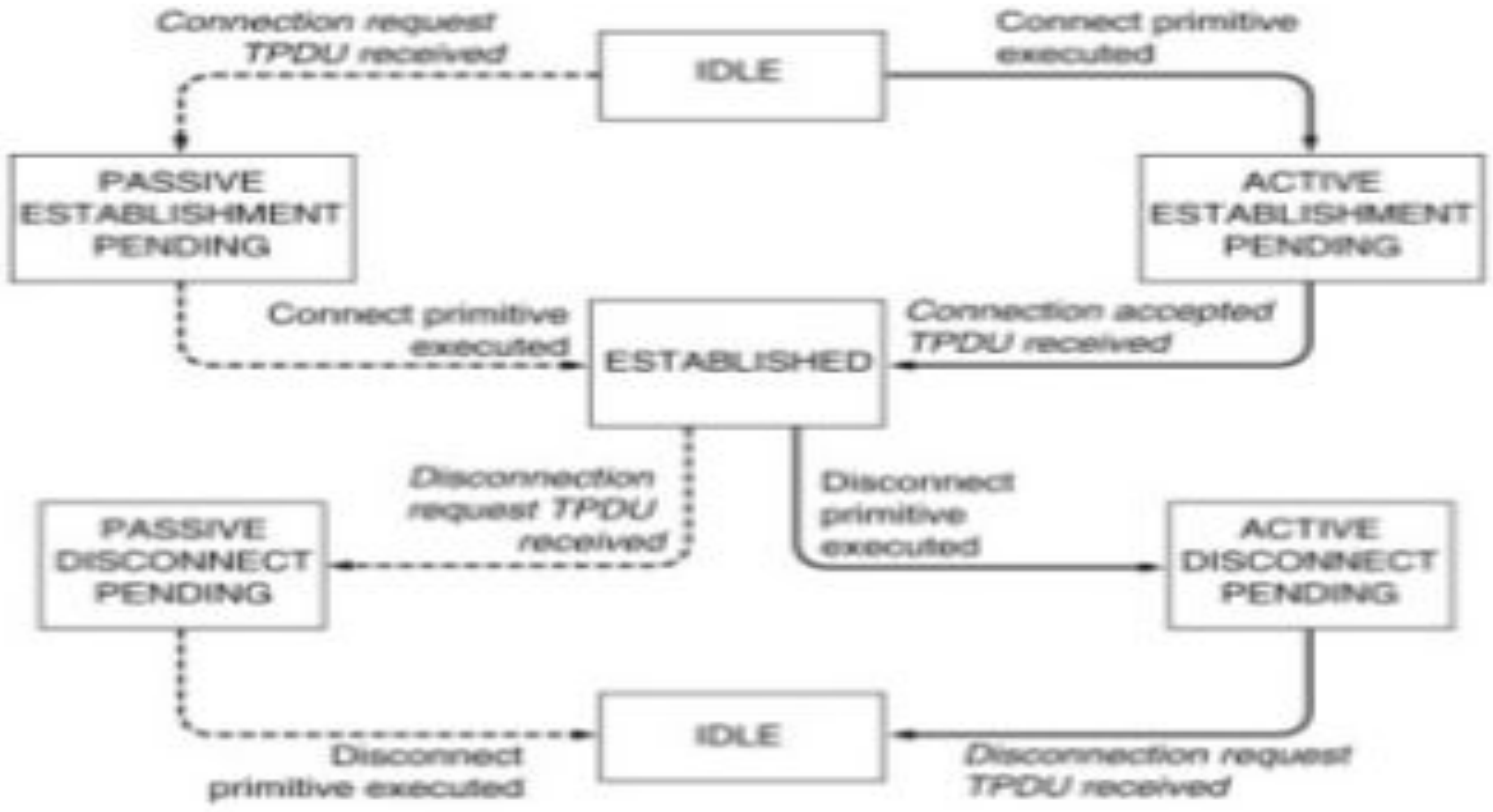
The primitives for a simple transport service.

TRANSPORT SERVICES PRIMITIVES



The nesting of TPDUs, packets, and frames.

5.1 BERKELEY SOCKETS



Connection request TPDU received
IDLE
Connect primitive executed
PASSIVE ESTABLISHMENT PENDING
ACTIVE ESTABLISHMENT PENDING
Connect primitive executed
ESTABLISHED
Connection accepted TPDU received
Disconnection request TPDU received
Disconnect primitive executed
PASSIVE DISCONNECT PENDING
ACTIVE DISCONNECT PENDING
IDLE
Disconnect primitive executed
Disconnection request TPDU received

Thank you