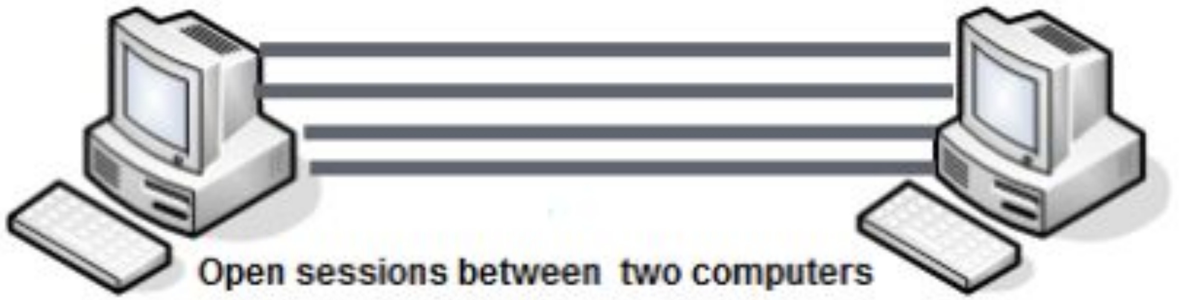
Session Layer and Transport Layer

Module 4

Session Layer

Session Layer

- 5th layer of OSI Reference Model
- Responsible for
  - ✔ Opening session
  - ✔ Managing session
  - ✔ Terminating session



- For two computer to communicate there must be channels – ***open***
- May have **more than one channel**
- Through the channels the messages are transmitted
- **Channels** are nothing but **Sessions**
- Session layers is **not** concerned about **data integrity**
- Associated with the exchange of data
- provide a way for devices or nodes to communicate.

- Session layer provides services like
  - ✔ ***Authentication,***
  - ✔ ***Authorization,***
  - ✔ ***Dialog Control,***
  - ✔ ***Dialog separation***

Authentication

- Authentication is done on the network devices.
- verifying who you are
- Logging on to a server with a username and password is authentication.
- The main purpose of authentication is **security**.
- Without authentication, user's data stored on a server is unsafe on the grounds that everyone can get it from the server.

Authorization

- After being authenticated
- process of verifying what a user is **entitled** to do or access on a given server
- **authorization is configured on every server**
- It is to avoid intrusion on users' personal data

Dialog control

- A successful communication is the one that is well organized and sorted out.
- a third party in a communicative context may effectively lead to a successful communication by regulating things.
- This third party in computer network is called 'Dialog control' and 'Dialog separation'.

 dialog control

- it determines whose turn it is to transfer data in a session
- In a given open session, a device plays dual roles, which is requesting services and replying with services.

- Dialog control determines which role they are playing at any given moment.
- dialog control is critically significant for data transmission.
- provides three different modes or ways of communication, which are

✔ **simplex**

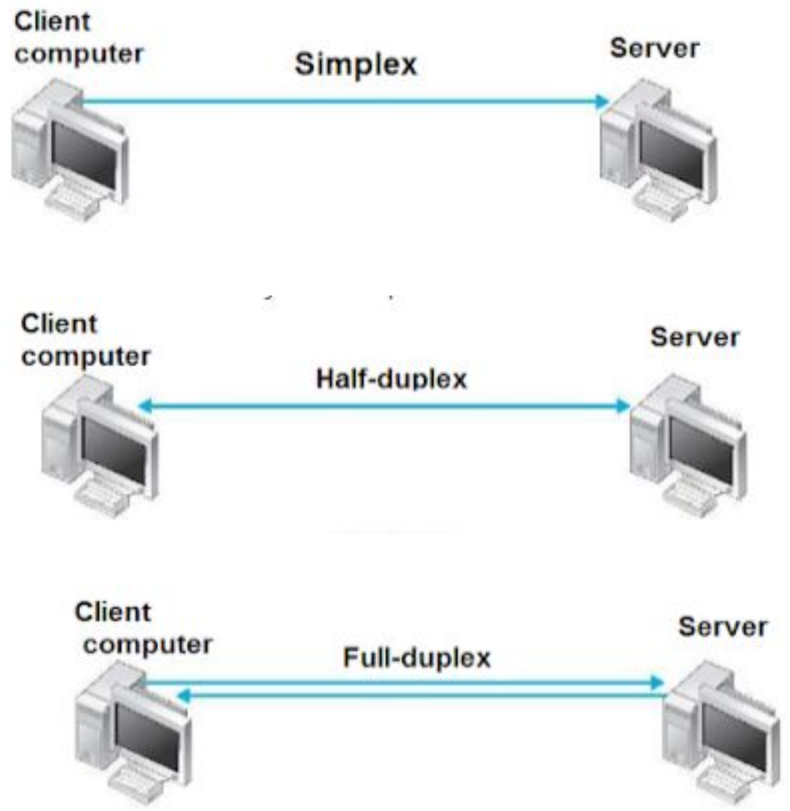
eg: Radio

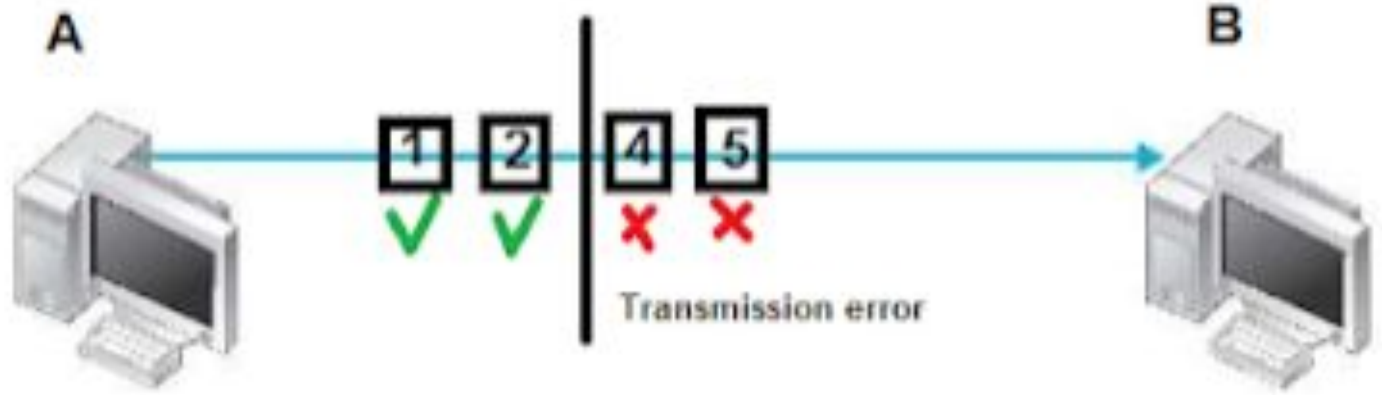
✔ **half-duplex**

eg: Walkie talkie

✔ **full-duplex**

eg: skype

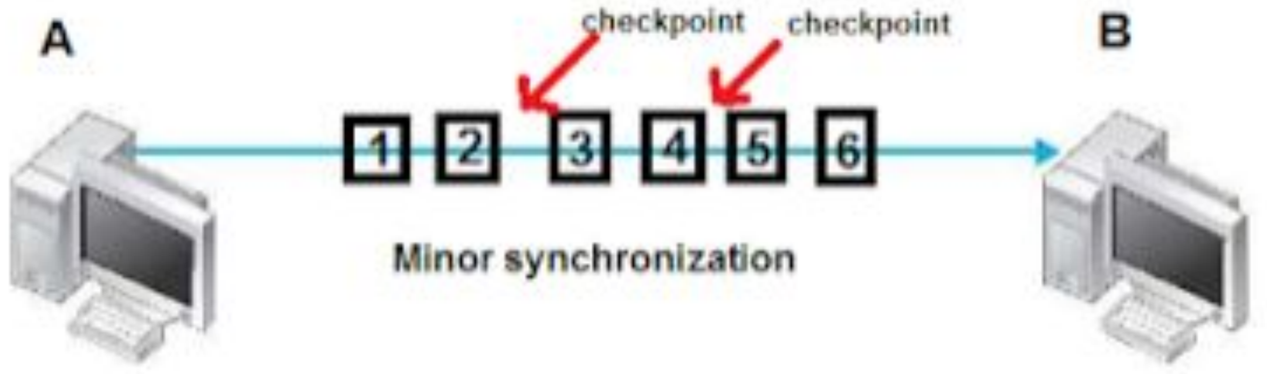




 Dialog Synchronization

- It is concerned with the initiation, termination, and managing of communication.
- error occurred during transmission which led to the loss of data. In this case, computer has to resend the lost data as well as the received ones.
- It is responsible for solving problems that may occur during data transmission. It allows users to resume data transfer from where an error happens without any need to start over.

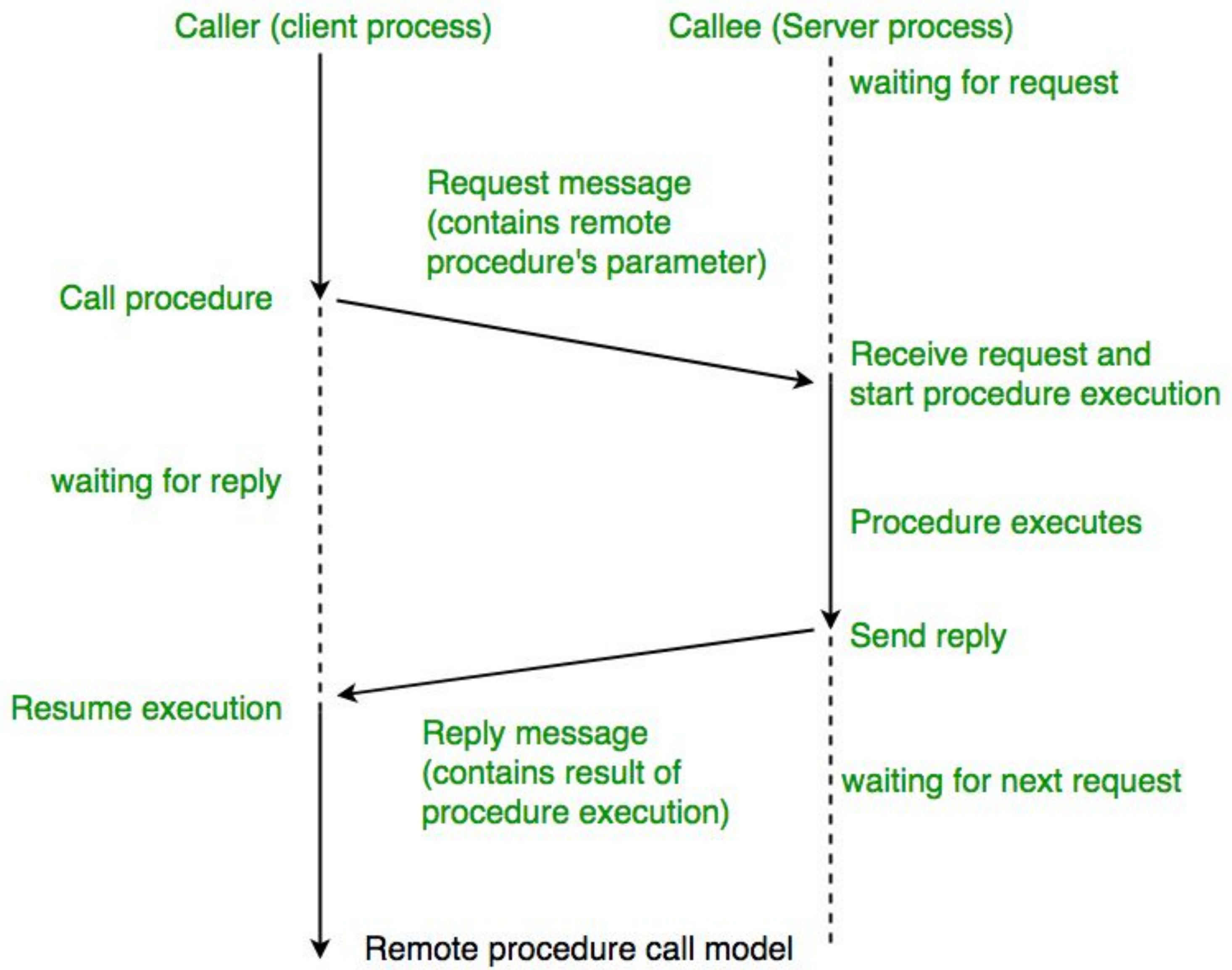
- when an error occurs during transmission, the sender does not need to start sending data from the beginning, it rather start from where it breaks. This is achieved through ***synchronization***.
- If the session is closed by mistake, the dialog separation has the ability to re-establish the connection and start from where it leaves off.
- During the transmission of data, dialog separation allows a process to add ***checkpoints*** to each set of data.



- each piece of data is assigned a ***sequence number***
- when an error occurs, the receiver can re-synchronize the state of the session to a previous synchronization point.

Remote Procedure Call

- Remote Procedure Call (RPC) provides a different paradigm for accessing network services.
- Making a **call to a function** to request for a service, on another machine **without having to know the networking details**
- Session-**layer** services are commonly used in application environments that make use of Remote Procedure Calls (RPCs)
- Instead of accessing remote services by sending and receiving messages, a client invokes services by making a **local procedure call**.
- The local procedure **hides** the details of the **network communication**.
- Message based communication system



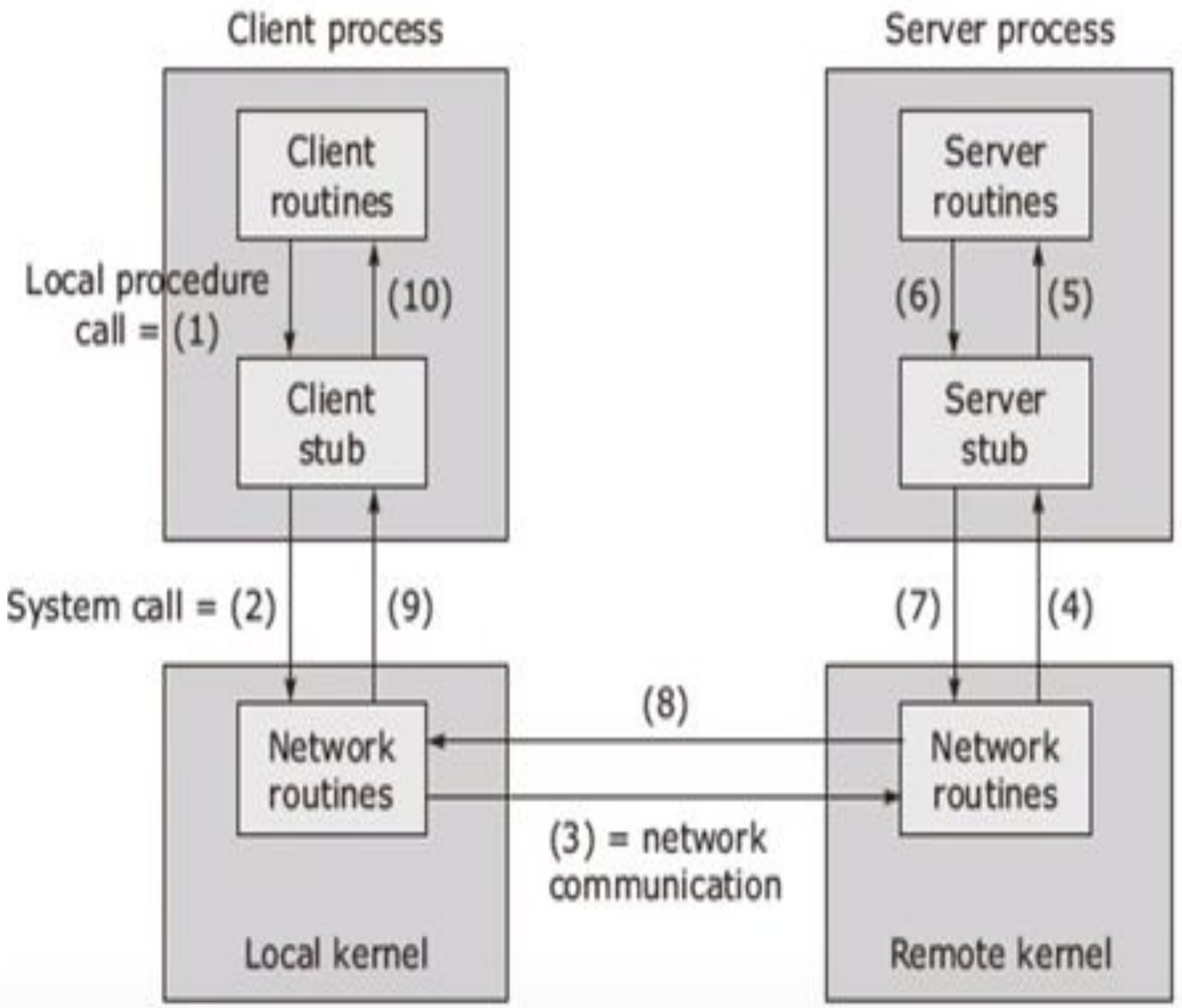
Remote procedure call model

- Client and Server model
- Fundamental idea: –
  - Server process exports an *interface* of procedures or functions that can be called by client programs
    - similar to library API, class definitions, etc.
  - Clients make local procedure/function calls
    - *As if* directly linked with the server process
    - Under the covers, procedure/function call is converted into a message exchange with remote server process

- RPC architecture has mainly five components of the program:
  1) Client
  2) Client Stub
  3)RPC Runtime
  4) Server Stub
  5) Server

RPC Execution

1. Client calls a local procedure on the client stub
2. The client stub acts as a proxy and marshalls the call and the args.
3. The client stub send this to the remote system (via TCP/UDP)
4. The server stub unmarshalls the call and args from the client
5. The server stub calls the actual procedure on the server
6. The server stub marshalls the reply and sends it back to the client



om

Client

- A process, such as a program or task, that requests a service provided by another program. The client process uses the requested service without having to “deal” with many working details about the other program or the service.

Server

- A process, such as a program or task, that responds to requests from a client.

Client Stub

- Module within a client application containing all of the functions necessary for the client to make remote procedure calls using the model of a traditional function call in a standalone application. The client stub is responsible for invoking the marshalling engine and some of the RPC application programming interfaces (APIs).

Server Stub

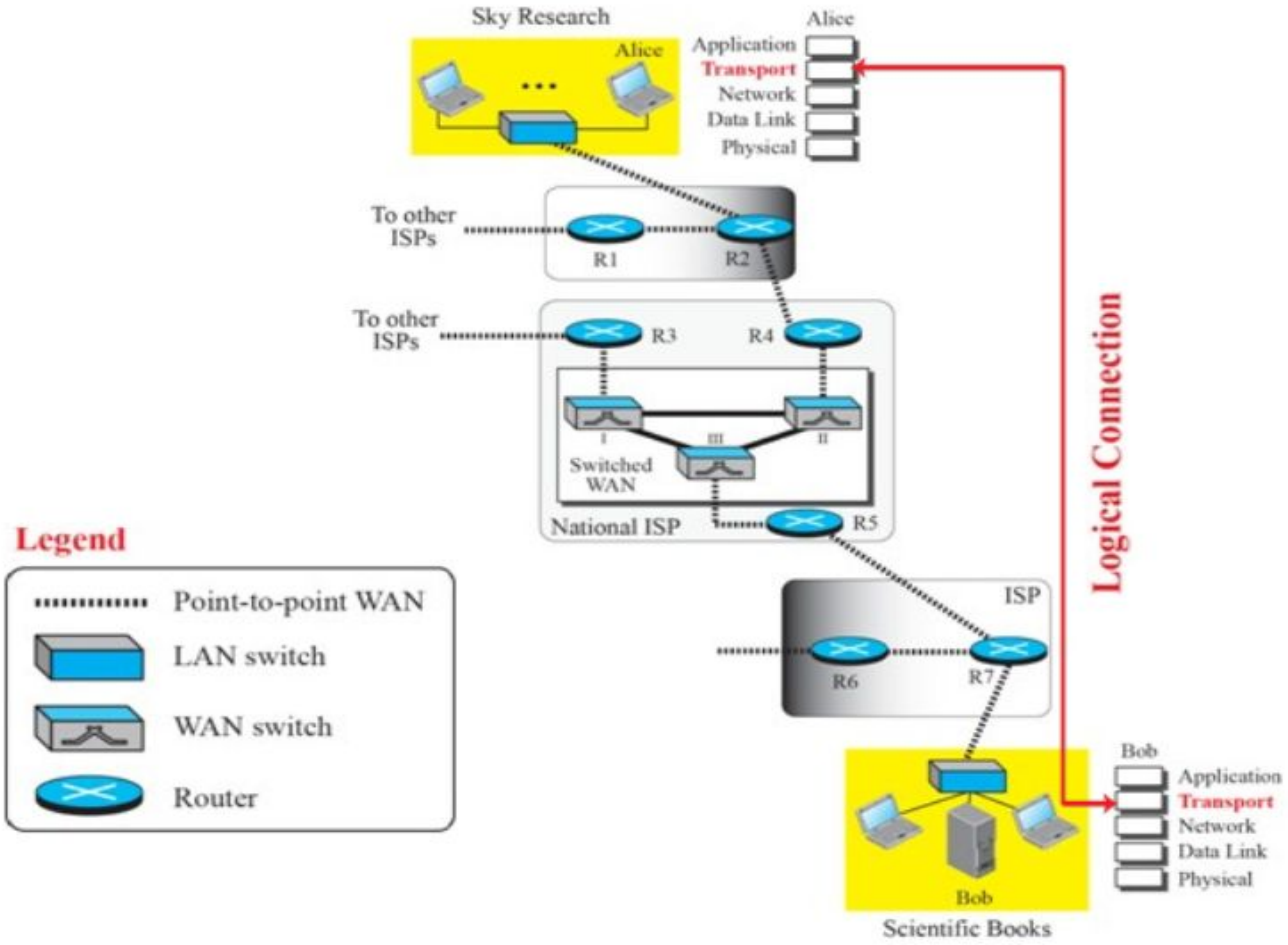
- Module within a server application or service that contains all of the functions necessary for the server to handle remote requests using local procedure calls.

- RPC is supported by no of network protocols – **TCP, UDP**
- Marshalling Arguments
  - *Marshalling* is ***the packing of function*** parameters into a message packet
  - the RPC stubs call ***type-specific functions*** to marshal or unmarshal the parameters of an RPC
    - Client stub marshals the arguments into a message
    - Server stub unmarshals the arguments and uses them to invoke the service function
  - on return:
    - the server stub marshals return values
    - the client stub unmarshals return values, and returns to the client program

Transport Layer

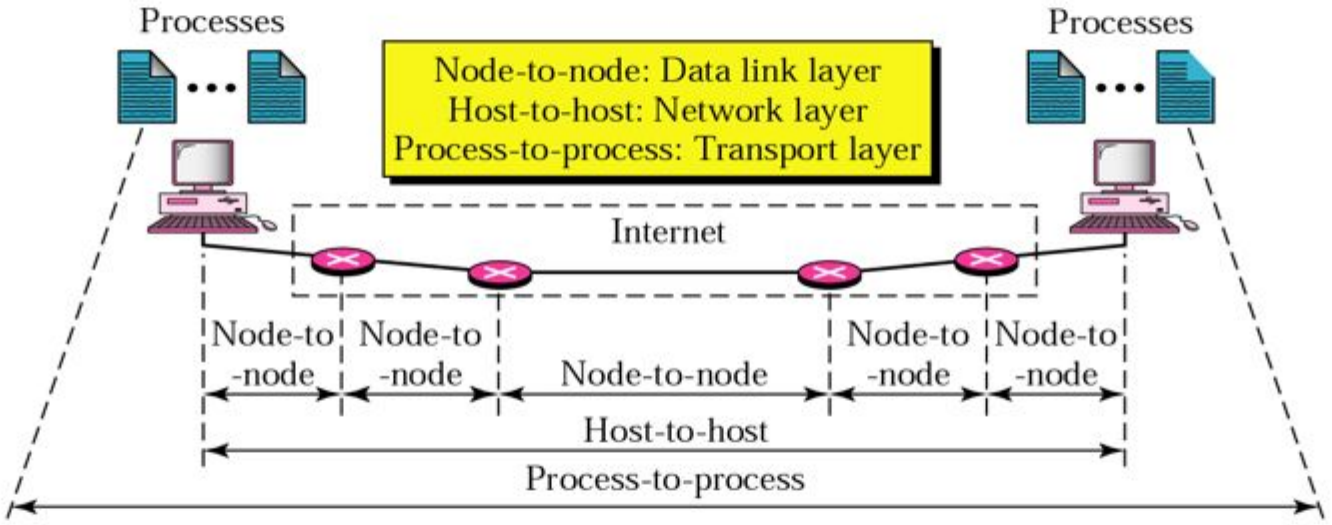
Transport Layer

- The transport layer is located between the application layer and the network layer in TCP reference Model, and between session and network layer in OSI reference Model.
- It provides **end to end and *process-to-process*** communication between two application layers, one at the local host and the other at the remote host.
- Communication is provided using a ***logical connection***.
- The transport layer is responsible for ***providing services to the application layer/ session layer***; it ***receives services from the network layer.***


Sky Research
Alice
Alice
Application
Transport
Network
Data Link
Physical
To other ISPs
R1
R2
To other ISPs
R3
R4
Switched WAN
I
II
III
National ISP
R5
ISP
R6
R7
Logical Connection
Bob
Application
Transport
Network
Data Link
Physical
Bob
Scientific Books
Legend
Point-to-point WAN
LAN switch
WAN switch
Router

Network Layer Vs Transport Layer

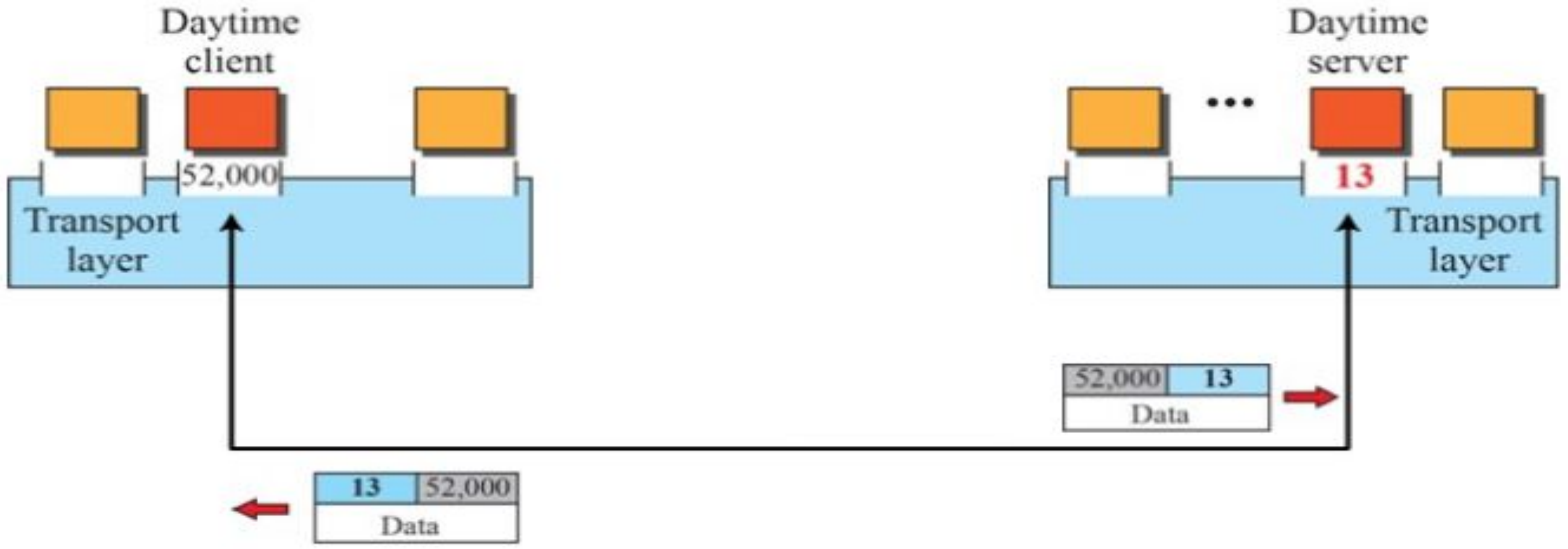
- Network layer only does host to host delivery, later the transport layer takes over to assign to process (port)



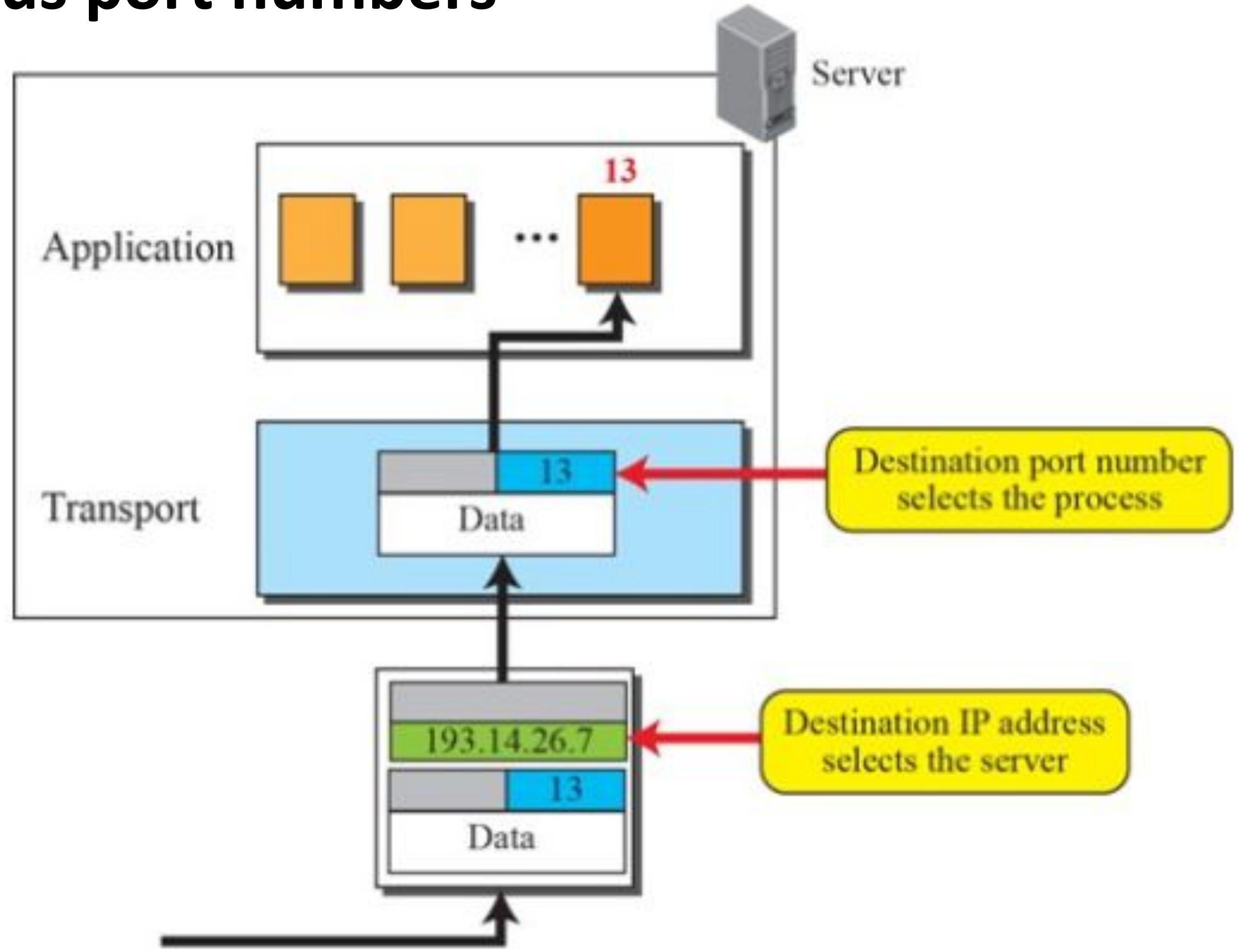
- *The transport layer is responsible for* ***process-to-process*** *delivery*

Port numbers

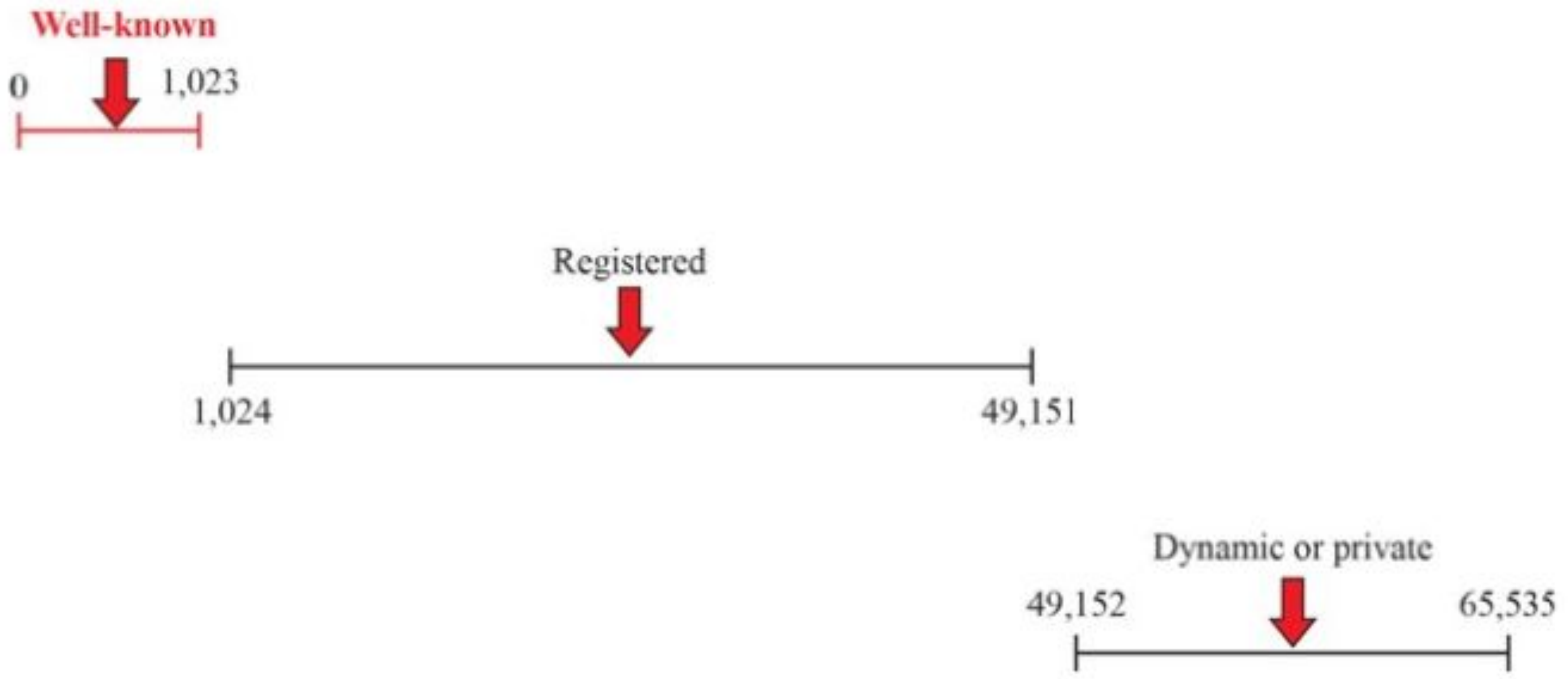
- Both process will have same name
- To find process find port no



IP addresses versus port numbers


Server
Application
13
Transport
13
Data
Destination port number selects the process
193.14.26.7
13
Data
Destination IP address selects the server

- **ICANN** (Internet Corporation for Assigned Names and Numbers)ranges
- Port nos use 16 bits (2^{16})
-



In UNIX, the well-known ports are stored in a file called /etc/services. We can use the *grep* utility to extract the line corresponding to the desired application.

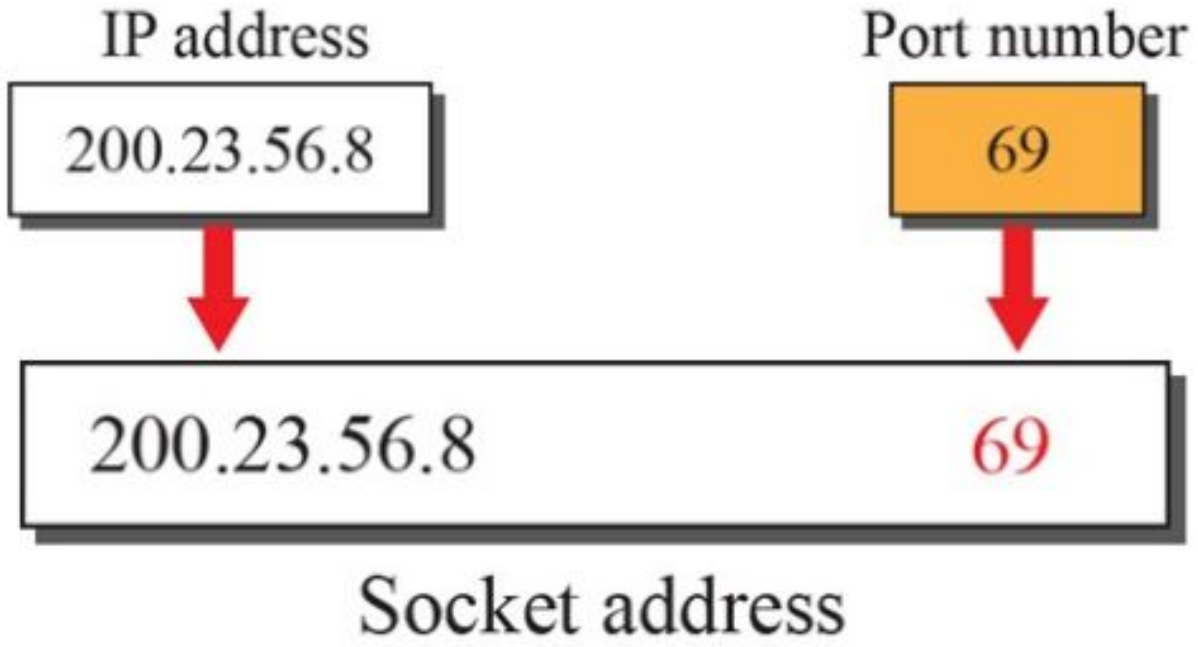
```
$grep    tftp/etc/services
tftp 69/tcp
tftp 69/udp
```

```
$grep    snmp/etc/services
snmp161/tcp#Simple Net Mgmt Proto
snmp161/udp#Simple Net Mgmt Proto
snmptrap162/udp#Traps for SNMP
```

TFTP uses 69
SNMP uses 161 and 162

Socket address

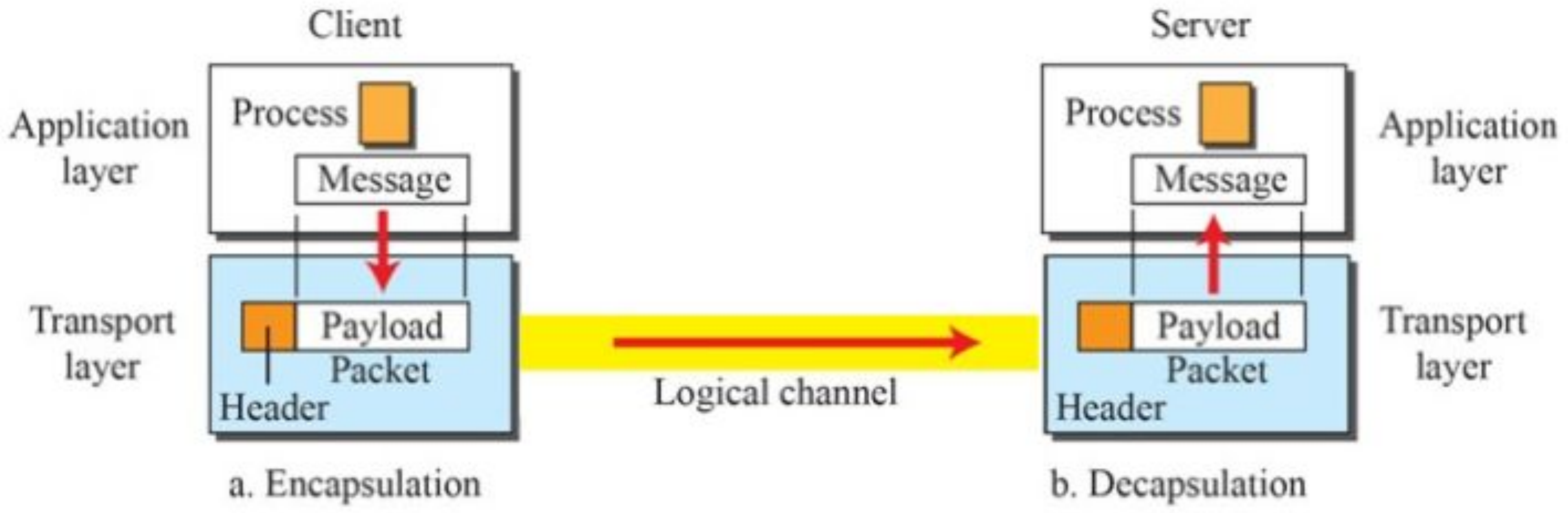
- Transport layer protocol need socket address



Functionality of Transport Layer

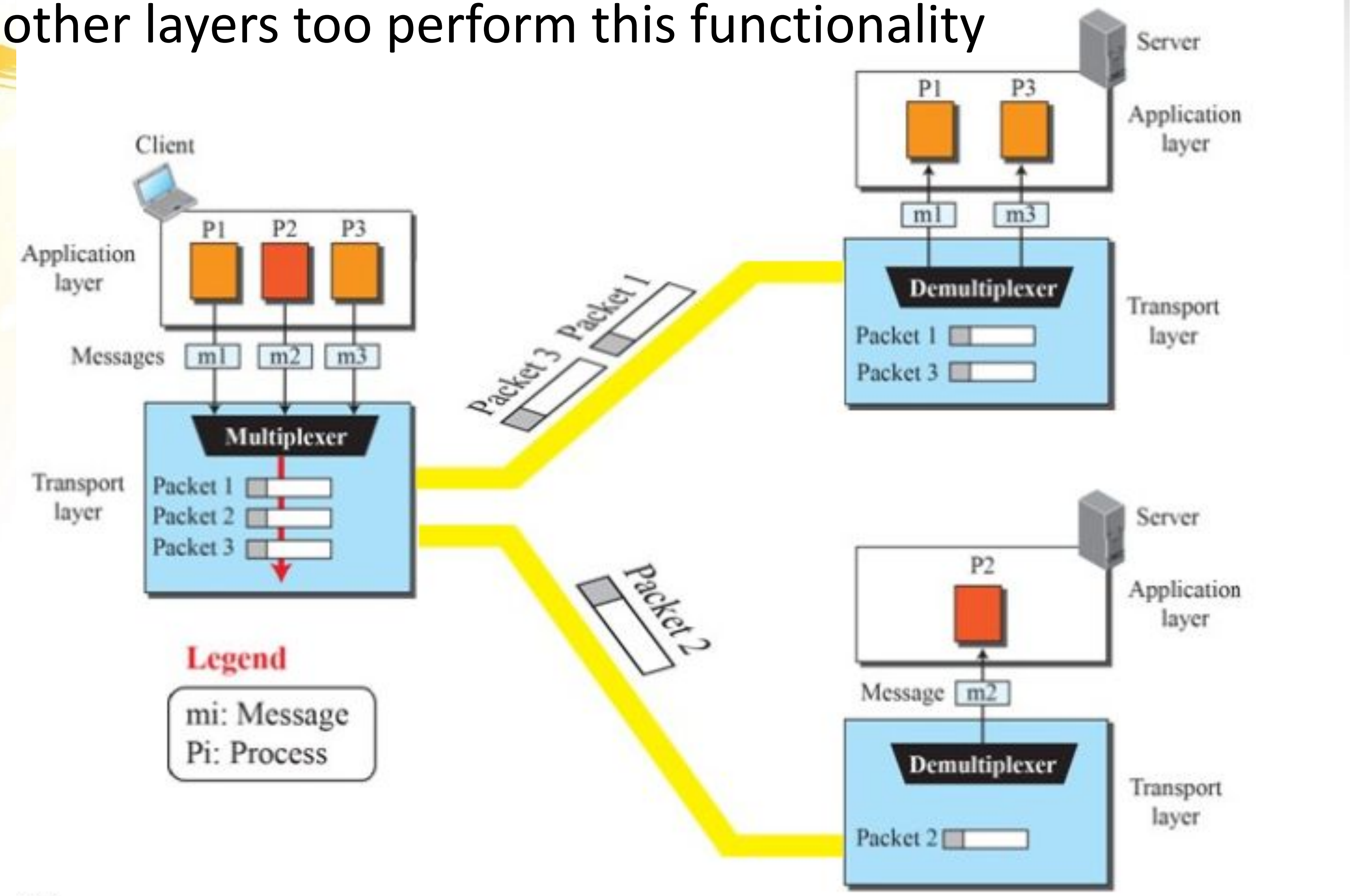
Encapsulation and encapsulation

- User datagram/ segment are encapsulated
- other layers too perform this functionality

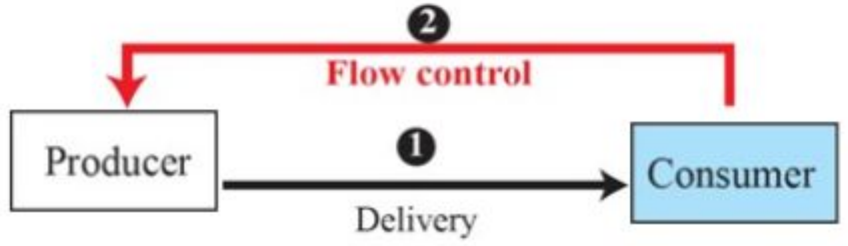


note here TCP model is referred

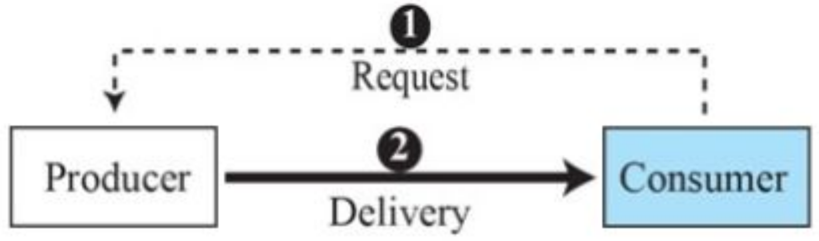
- **Multiplexing and demultiplexing**
- other layers too perform this functionality



Flow Control



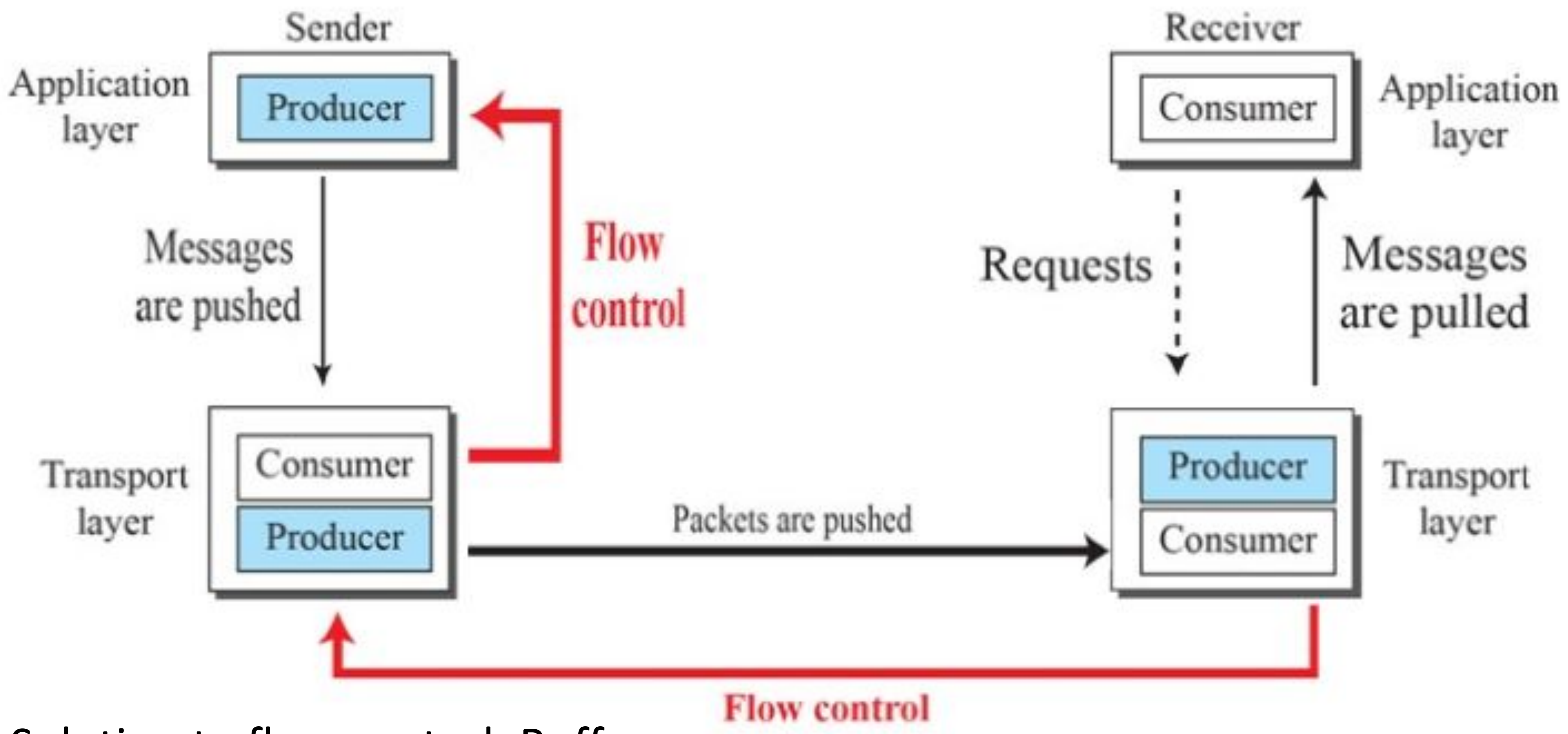
a. Pushing



b. Pulling

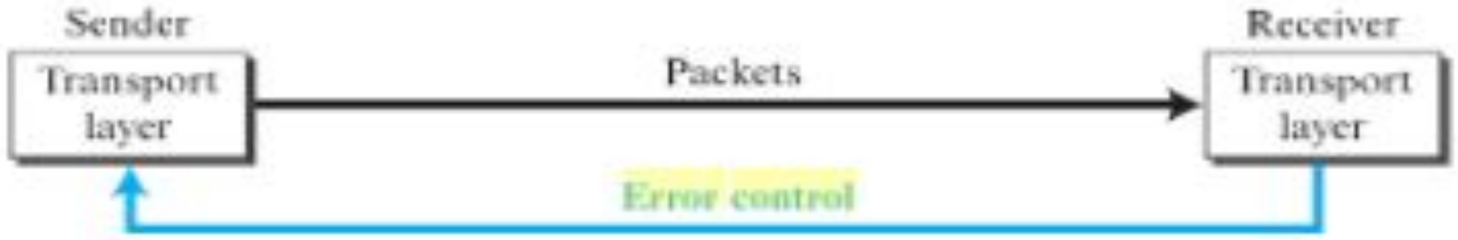
Flow control at the transport layer

- The production and consumption rate must be balanced
- If consumer or producer is overloaded, it has to signal to the other to stop



Solution to flow control: Buffer

- When buffer is full the receiver informs the sender to stop



Error Control

1. Detecting and discarding corrupted packets.
2. Keeping track of lost and discarded packets and resending them.
3. Recognizing duplicate packets and discarding them.
4. Buffering out-of-order packets until the missing packet

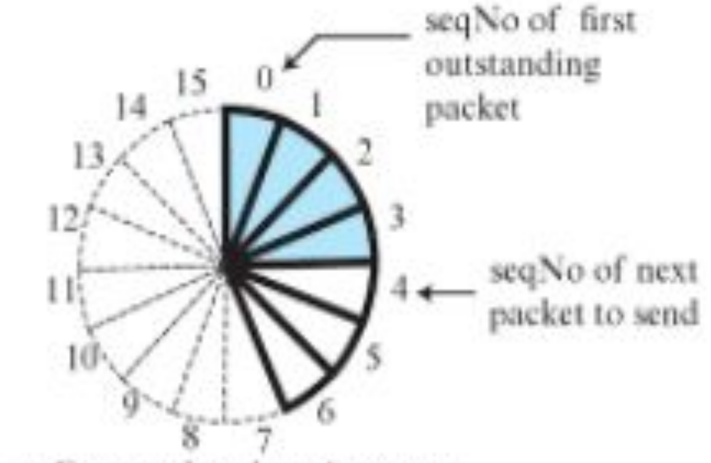
- Assume the data from application layer to transport is error free
- Error control unlike flow control involves only the sending and receiving transport layer

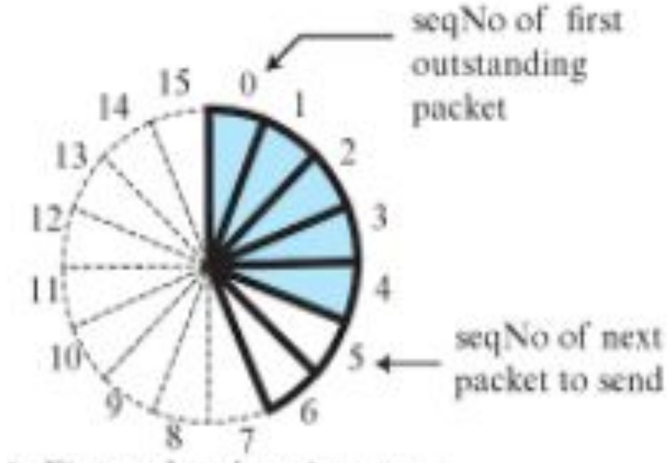
Measures for error control

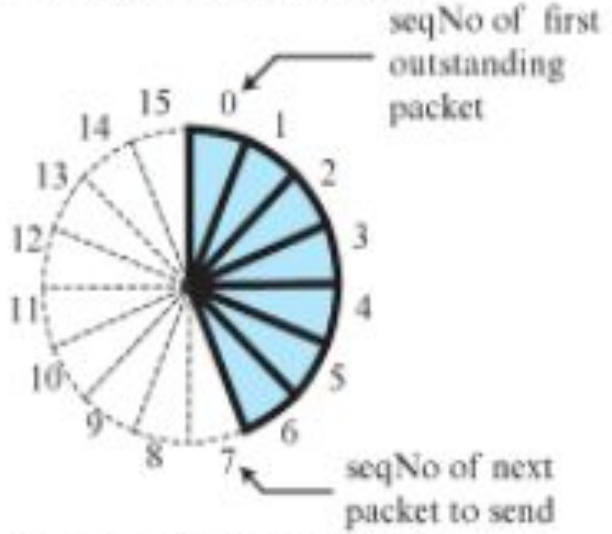
- **Sequence Number** to the packets
  - For m bit 0 to 2^m-1 sequence no are allowed
- **Acknowledgment** is sent from receiver after it receives the packet, and the sender sets a timer to receive the ACK

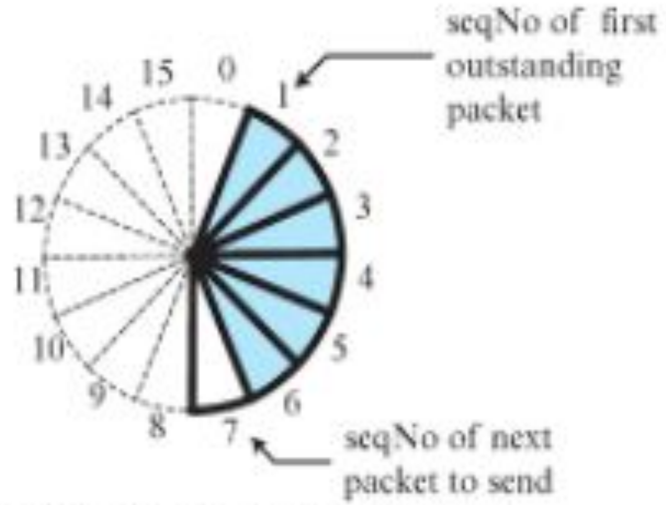
Combination of Error Control and Flow Control

- **SLIDING Window**

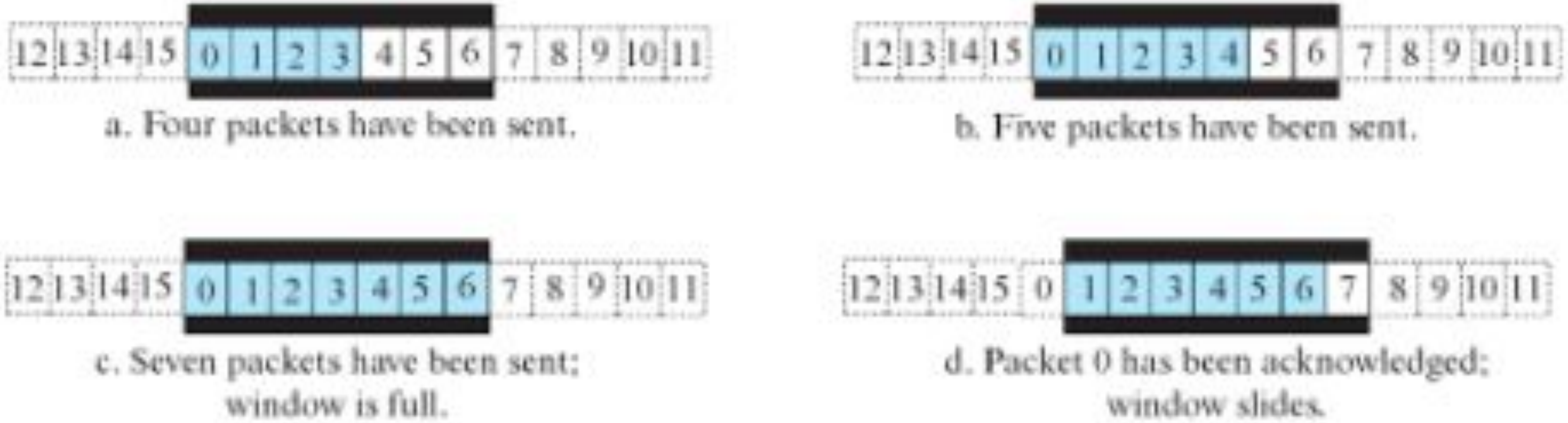

seqNo of first outstanding packet
seqNo of next packet to send
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
a. Four packets have been sent.


seqNo of first outstanding packet
seqNo of next packet to send
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
b. Five packets have been sent.


seqNo of first outstanding packet
seqNo of next packet to send
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
c. Seven packets have been sent; window is full.


seqNo of first outstanding packet
seqNo of next packet to send
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
d. Packet 0 has been acknowledged; window slides.

- Linear form of Sliding window



a. Four packets have been sent.

b. Five packets have been sent.

c. Seven packets have been sent; window is full.

d. Packet 0 has been acknowledged; window slides.

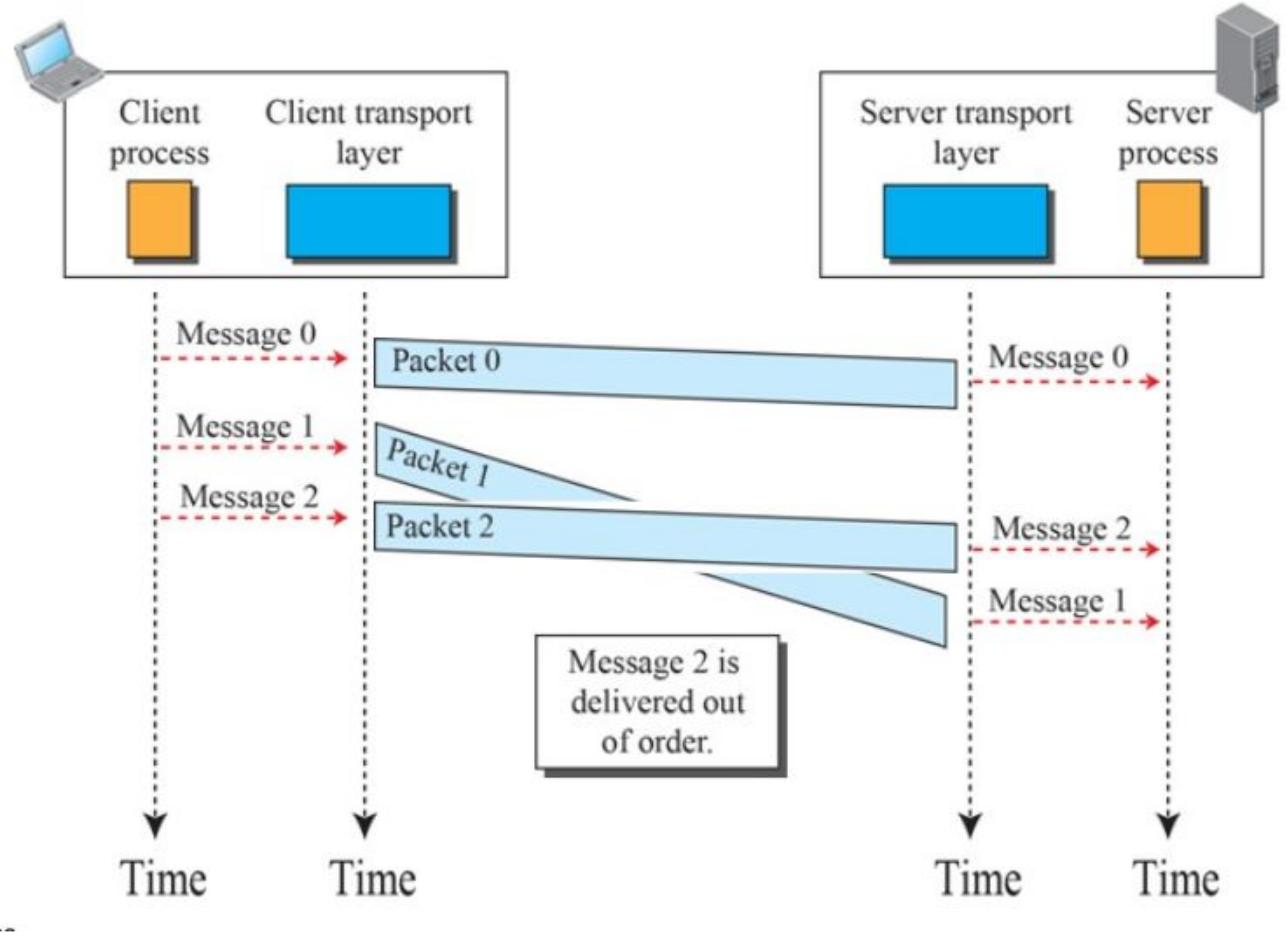
Congestion Control

- Mechanism and technique that controls the congestion and keep the load below the capacity
- Occurs because routers and switches have queues

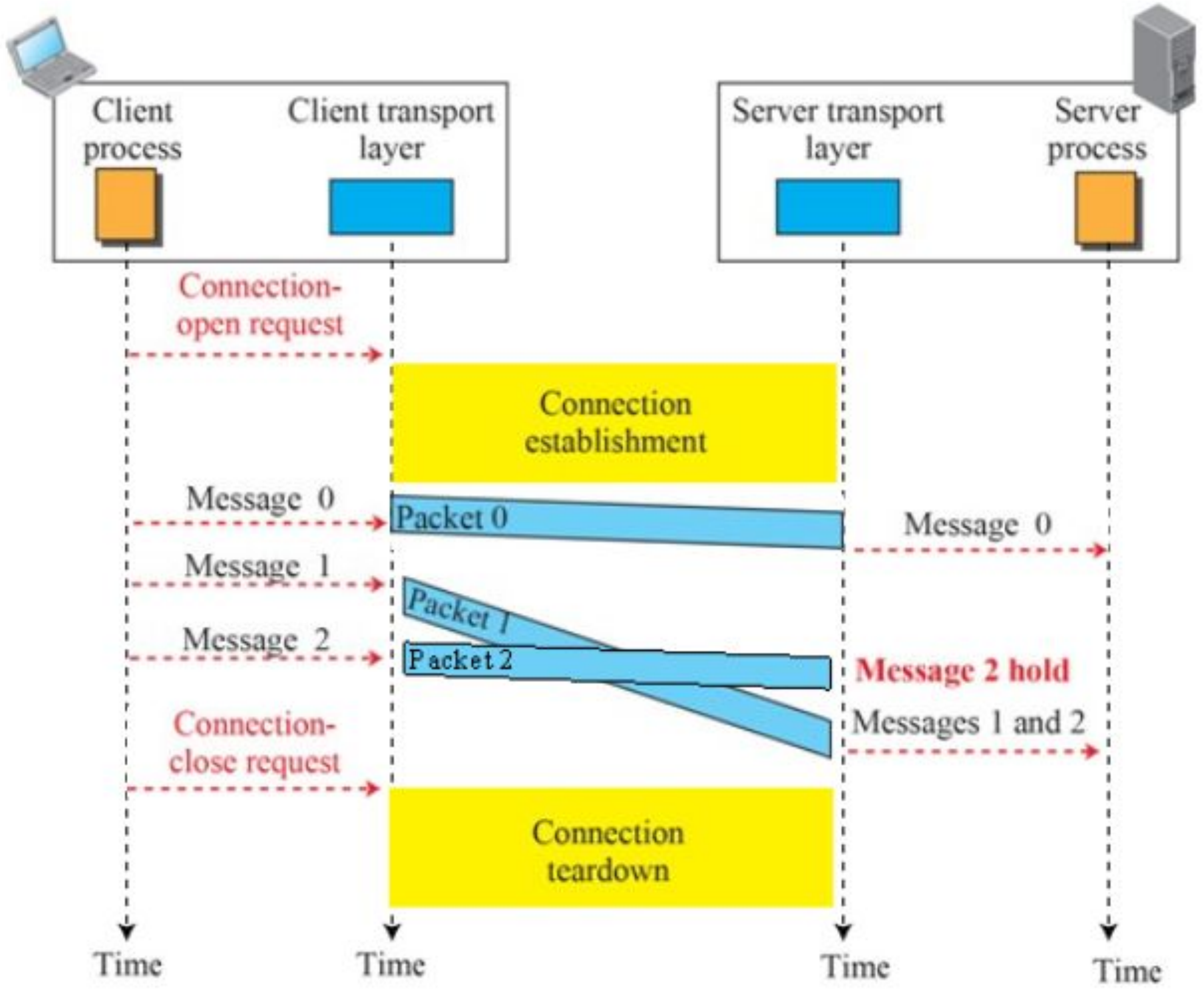
Connectionless and Connection -Oriented Services

- packets are independent of each other
- Packets are dependent on each other

Connectionless



Connection Oriented

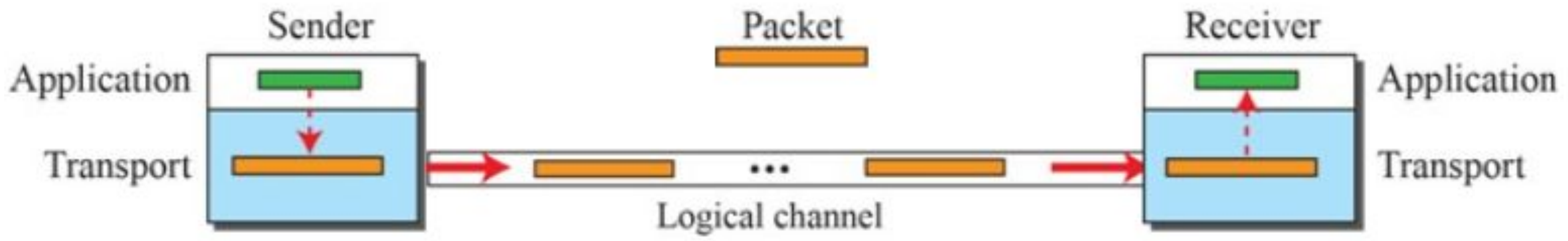


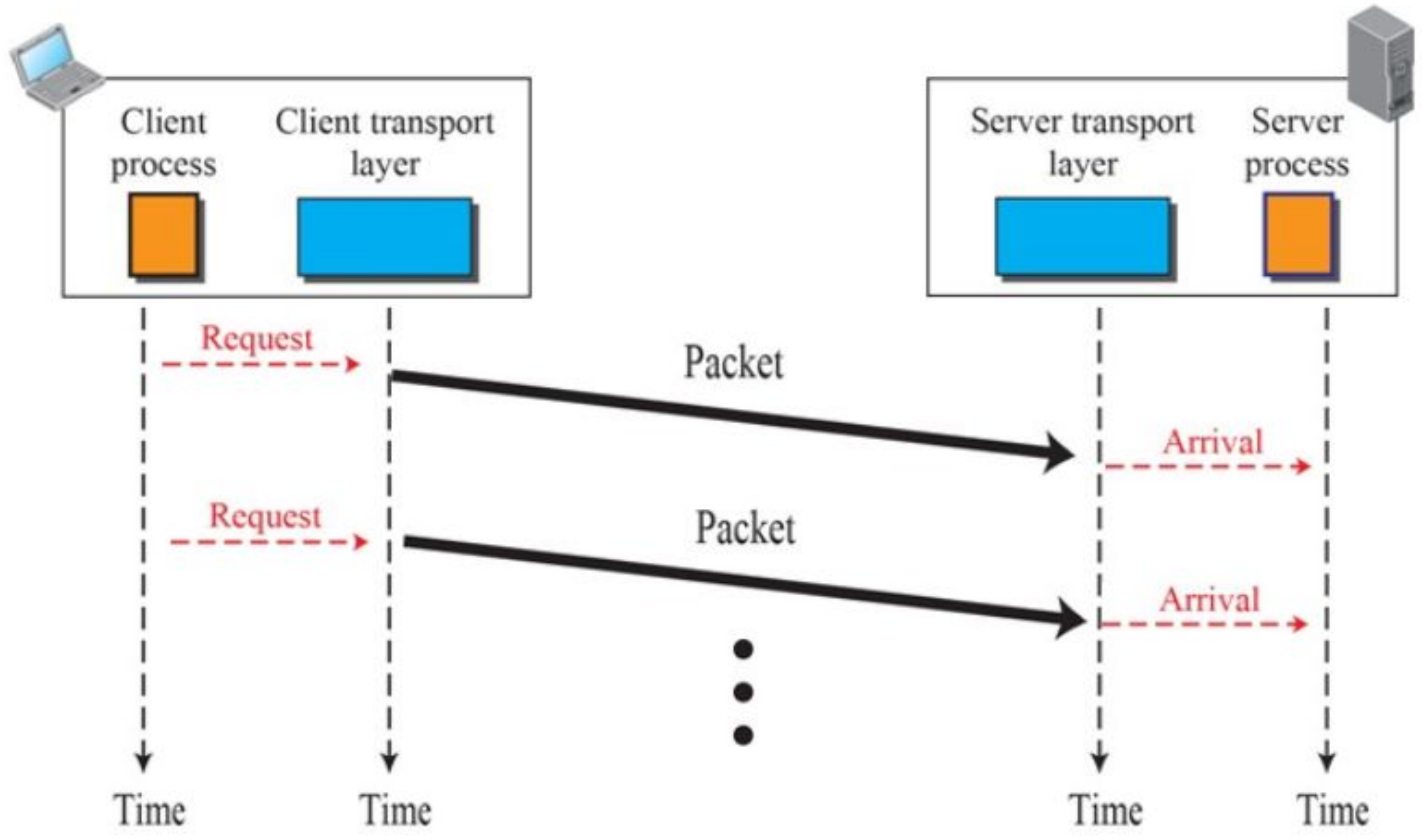
TRANSPORT-LAYER PROTOCOLS

- Simple connectionless protocol
  - No flow control and error control
- Connection-oriented protocol
  - Stop-and-Wait protocol
    - Provide flow and error control
  - Go-Back-N protocol
    - Efficient version of Stop-and-Wait protocol
  - Selective-Repeat Protocol
    - Suited to handle packet loss
  - Piggybacking

1) Simple Protocol

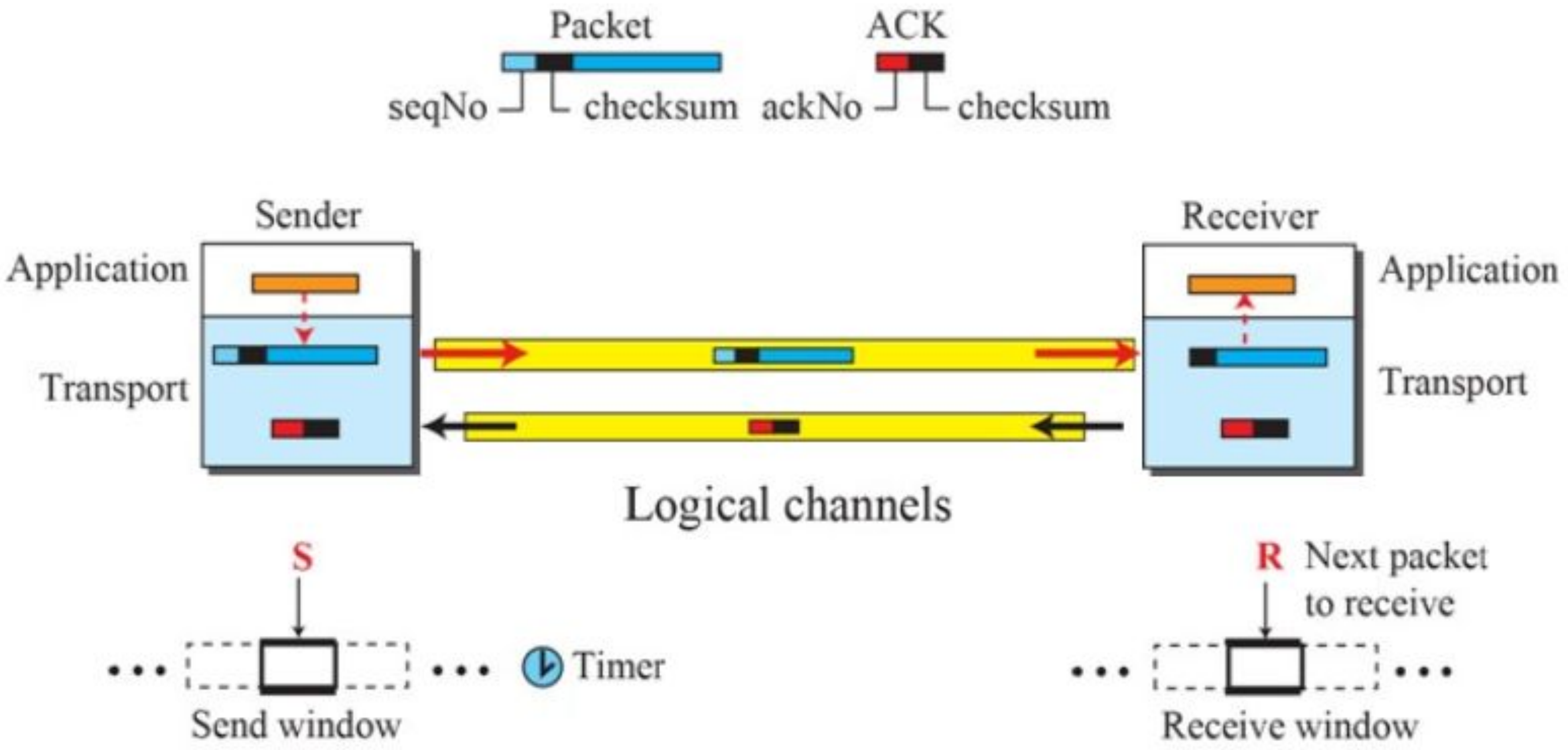
- simple connectionless protocol
- neither flow nor error control
- We assume that the receiver can immediately handle any packet it receives.
- the receiver can never be overwhelmed with incoming packets.




Client process
Client transport layer
Server transport layer
Server process
Request
Packet
Arrival
Request
Packet
Arrival
Time
Time
Time
Time

2) Stop-and-Wait Protocol

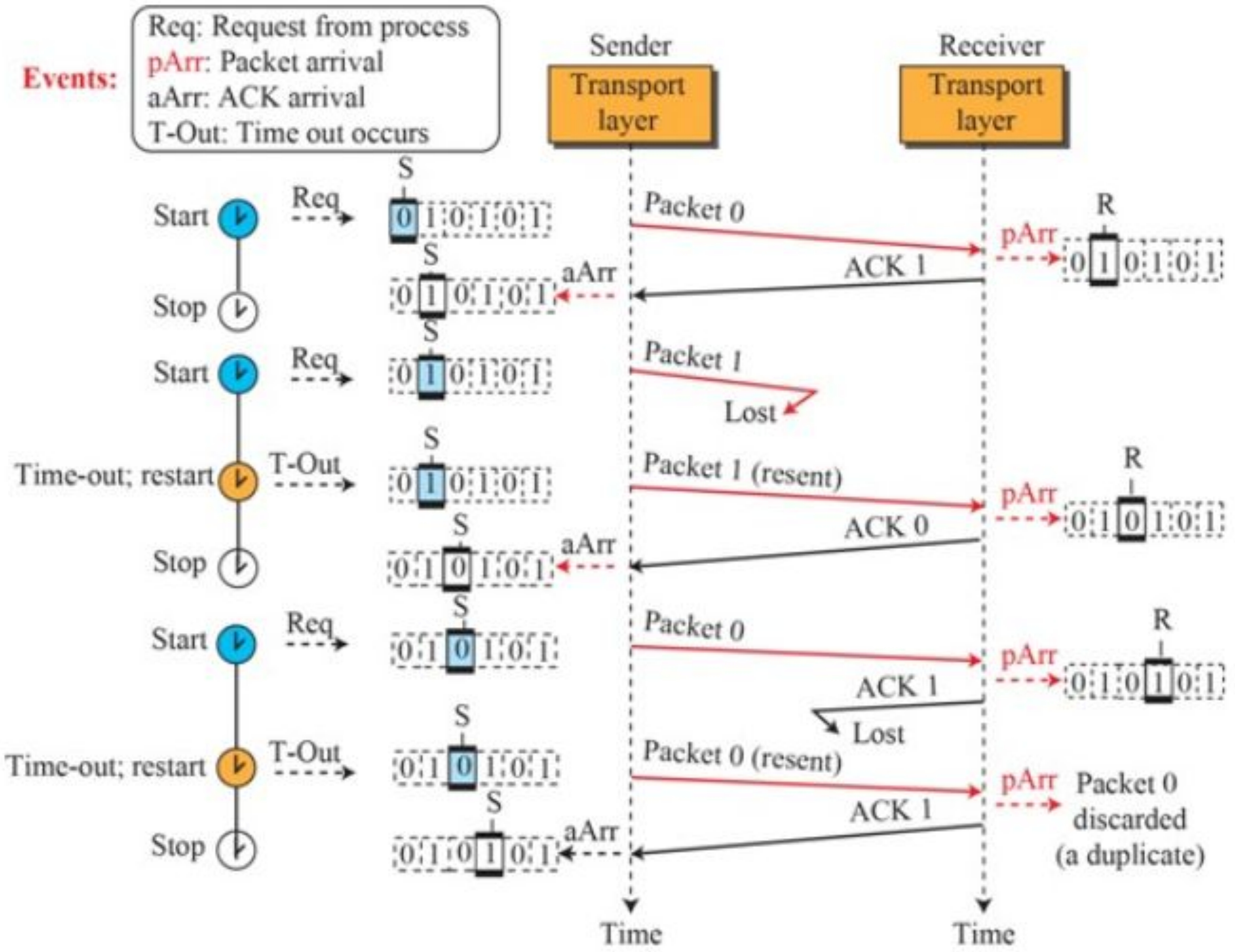
- connection-oriented protocol
- uses both *flow and error control*.
- Both the sender and the receiver use a sliding window size 1 .
- The sender sends one packet at a time and waits for an acknowledgment before sending the next one.
- To detect corrupted packets, we need to add a **checksum** to each 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.


Packet
seqNo
checksum
ACK
ackNo
checksum
Sender
Application
Transport
Receiver
Application
Transport
Logical channels
S
Send window
Timer
R Next packet to receive
Receive window

- Example of the Stop-and-Wait protocol.
- Cases:

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.

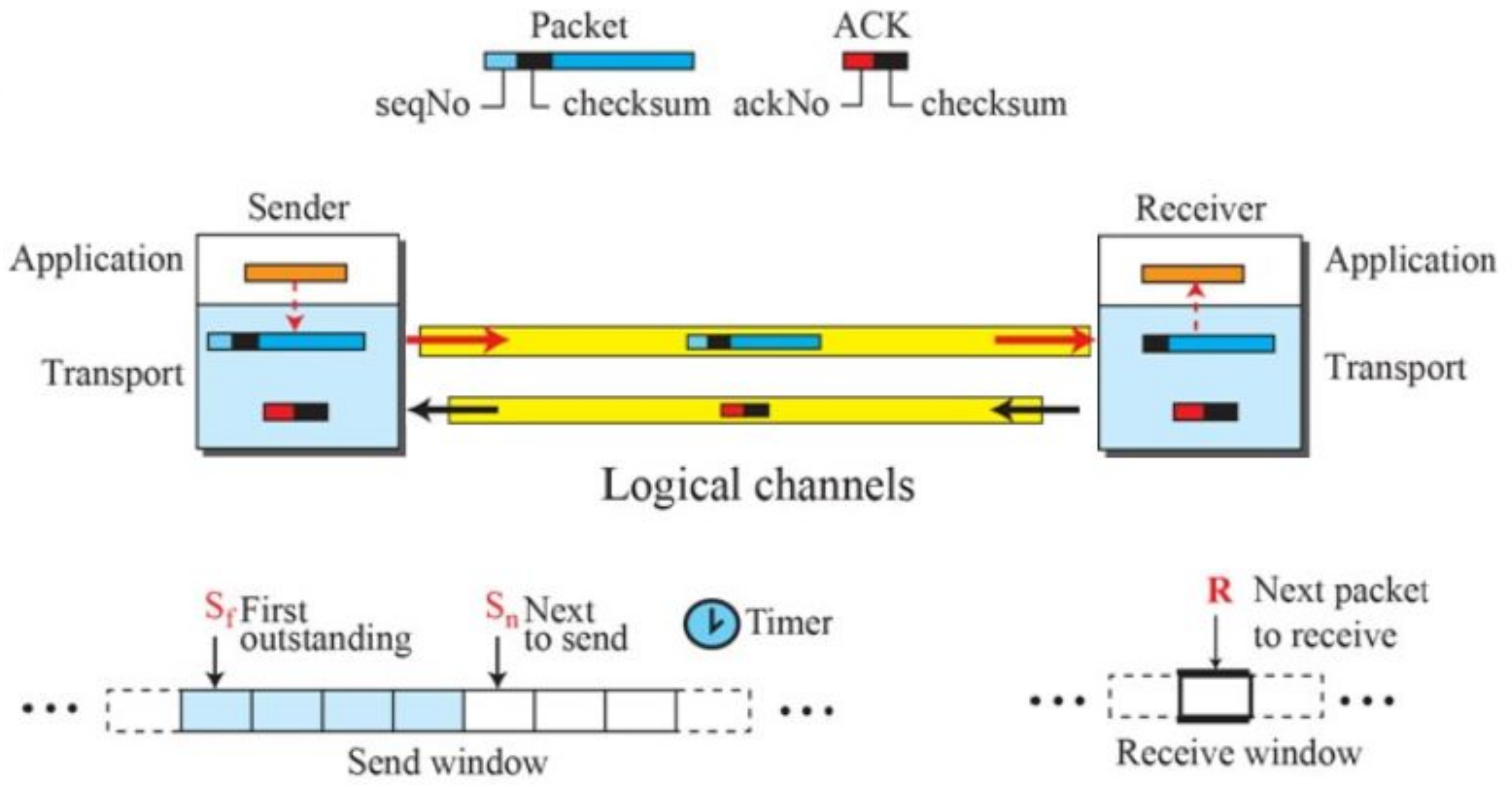

Events:
Req: Request from process
pArr: Packet arrival
aArr: ACK arrival
T-Out: Time out occurs
Sender
Transport layer
Receiver
Transport layer
Start
Req
S
0 1 0 1 0 1
Packet 0
pArr
R
ACK 1
aArr
Stop
Packet 1
Lost
Time-out; restart
T-Out
Packet 1 (resent)
ACK 0
Packet 0
ACK 1
Lost
Packet 0 (resent)
Packet 0 discarded (a duplicate)
Time
Time

- Sequence numbering is used
- Acknowledgement numbering are used
- Efficiency :-
  - Bandwidth delay product
  - Not using the full potential of the channel
- Solution
  - Pipelining

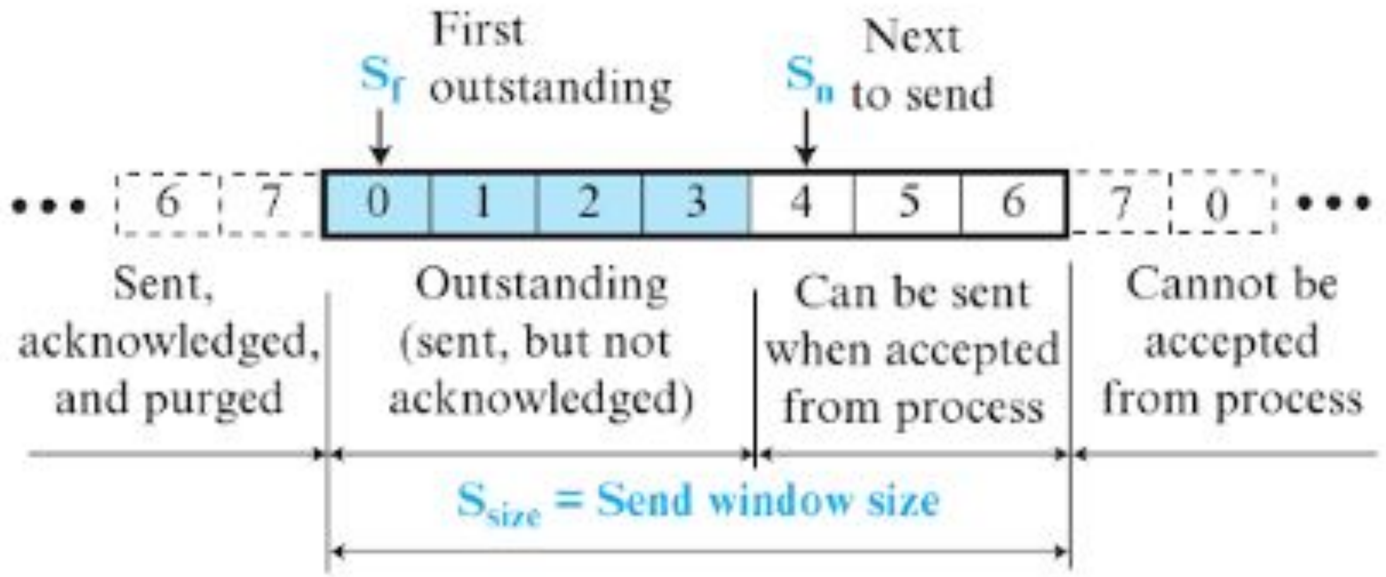
3) Go-Back-N Protocol (GBN)

- To improve the efficiency of transmission (to fill the pipe), multiple packets must be in transition while the sender is waiting for acknowledgment.
- GBN lets more than one packet be outstanding to keep the channel busy while the sender is waiting for acknowledgment.

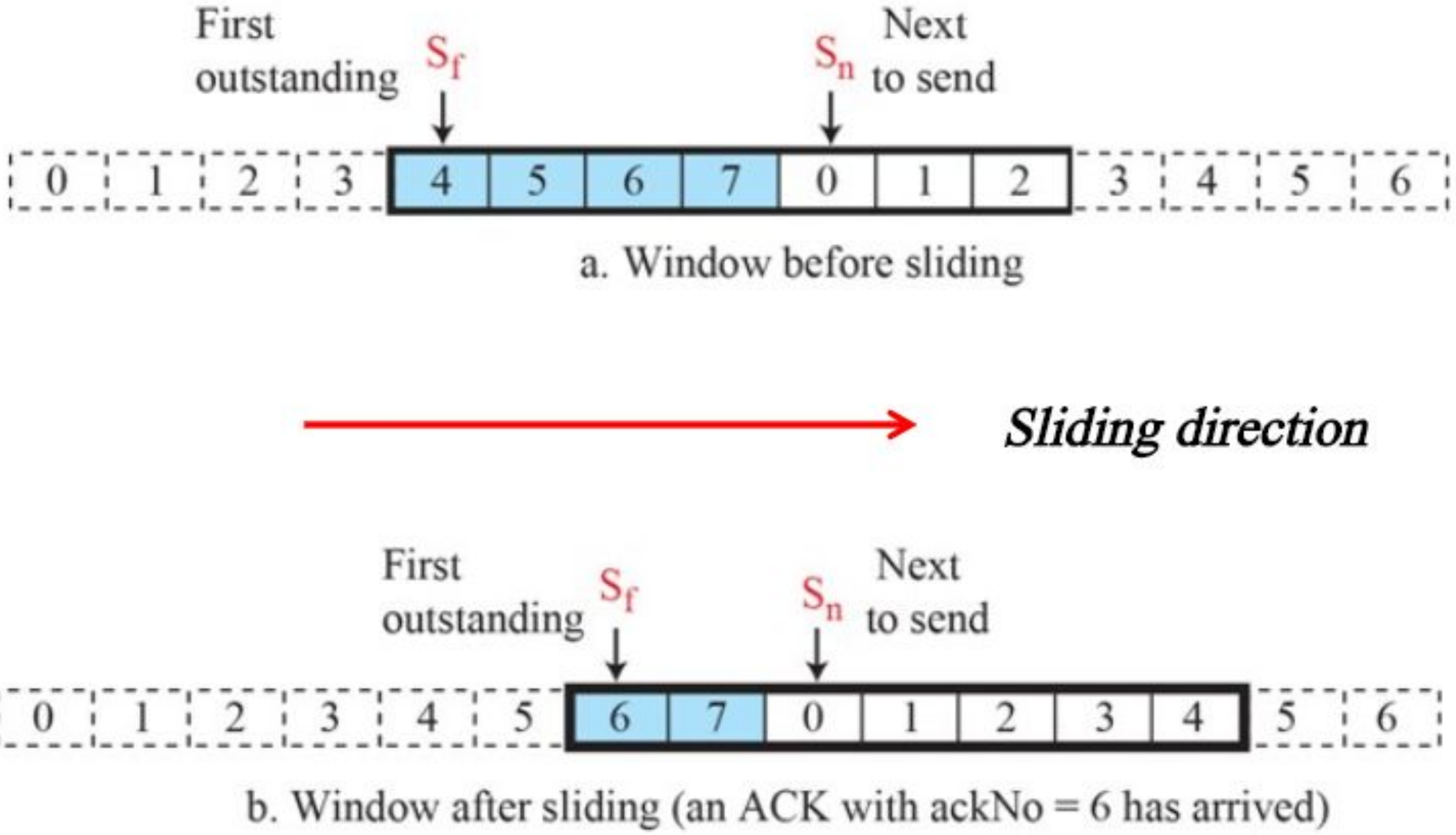
Send window for Go-Back-N



- Sender window for Go-Back-N

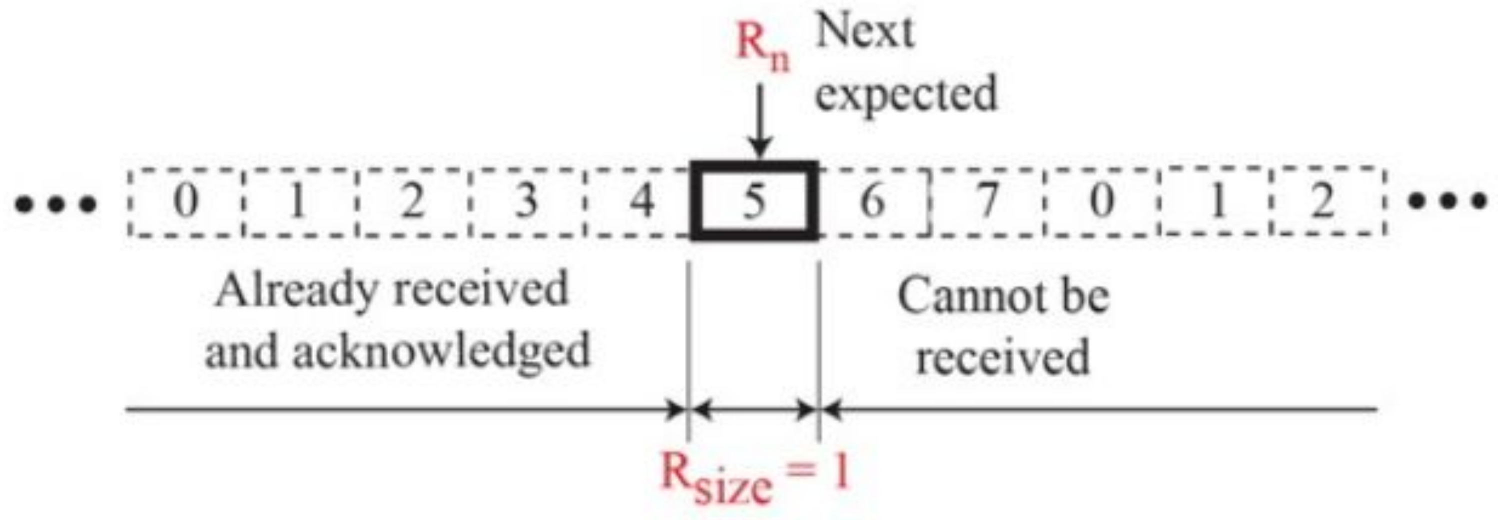

First outstanding
S_f
Next to send
S_n
6
7
0
1
2
3
4
5
6
7
0
Sent, acknowledged, and purged
Outstanding (sent, but not acknowledged)
Can be sent when accepted from process
Cannot be accepted from process
S_{size} = Send window size

Sliding the send window



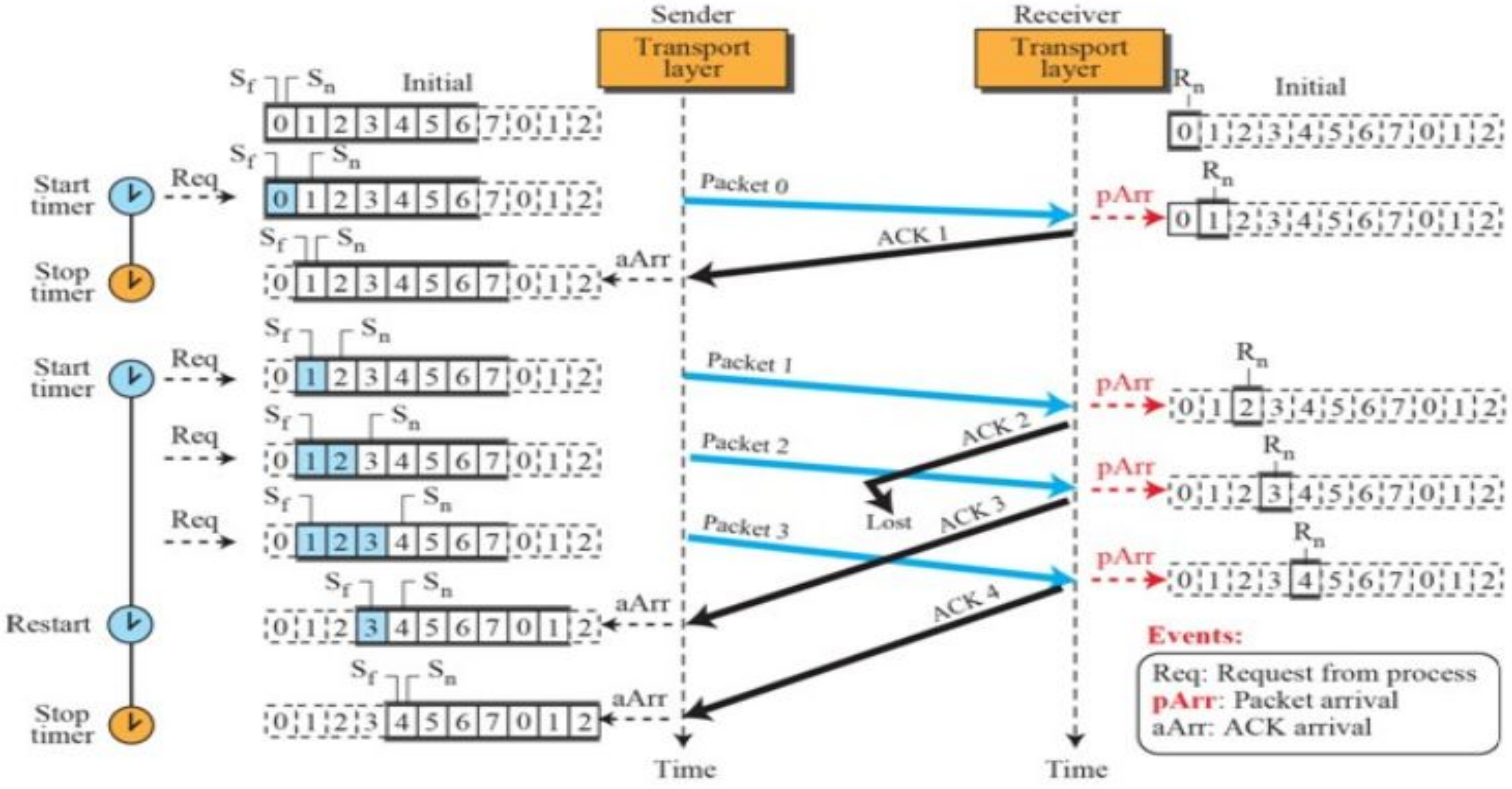
Receive window for Go-Back-N

- Size of the receiving window is always 1
- Any packet coming out of order is discarded and resent (timer)

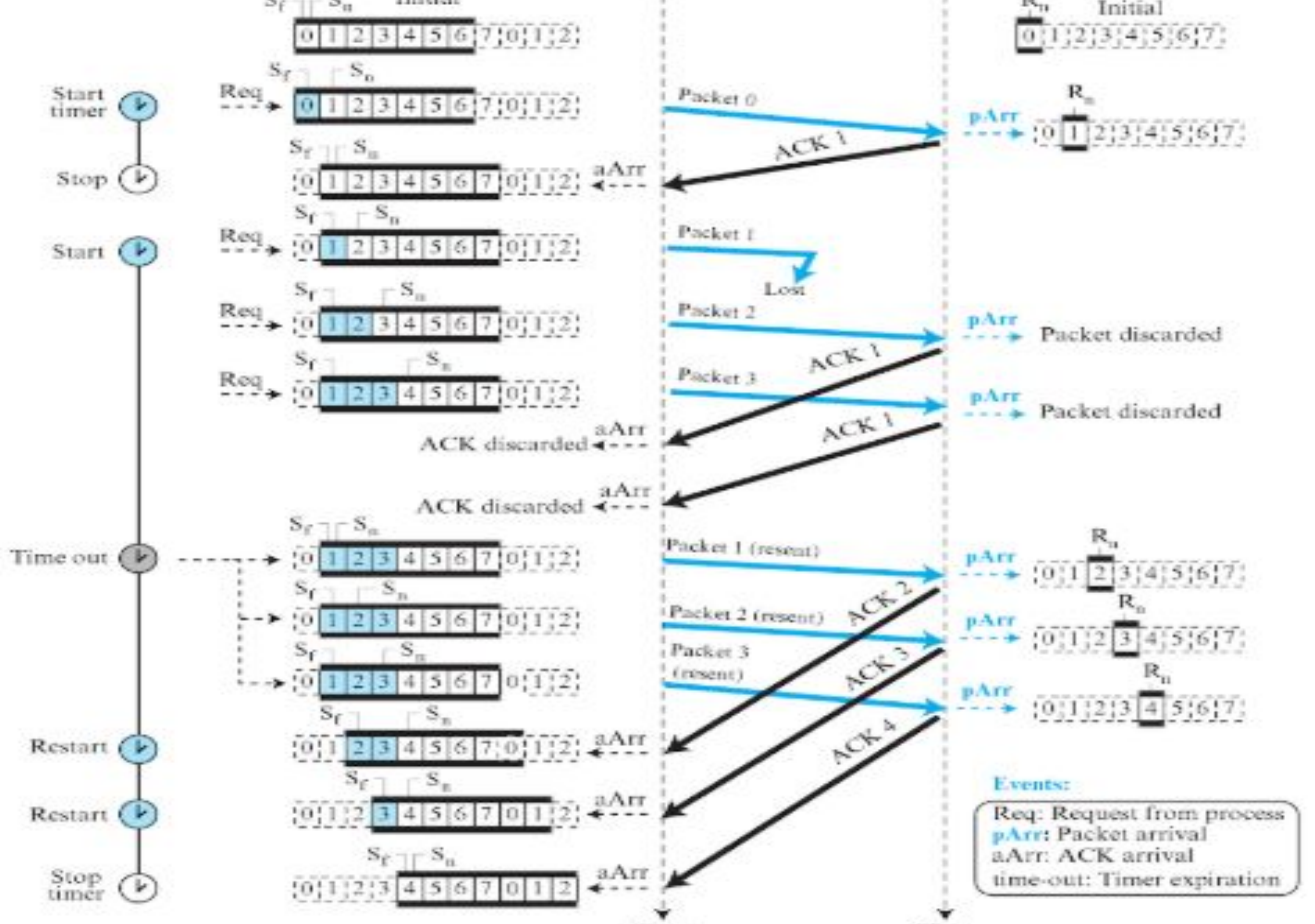


Example of a case where the forward channel is reliable, but the reverse is not.

- 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.


Sender
Transport layer
Receiver
Transport layer
S_f S_n Initial
R_n Initial
Start timer
Stop timer
Restart
Req
Packet 0
Packet 1
Packet 2
Packet 3
ACK 1
ACK 2
ACK 3
ACK 4
Lost
aArr
pArr
Time
Time
Events:
Req: Request from process
pArr: Packet arrival
aArr: ACK arrival

- Next 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 23. However, these ACKs are not useful for the sender because the ackNo is equal to S f, not greater that S f. So the sender discards them. When the time-out occurs, the sender resends packets 1, 2, and 3, which are acknowledged.



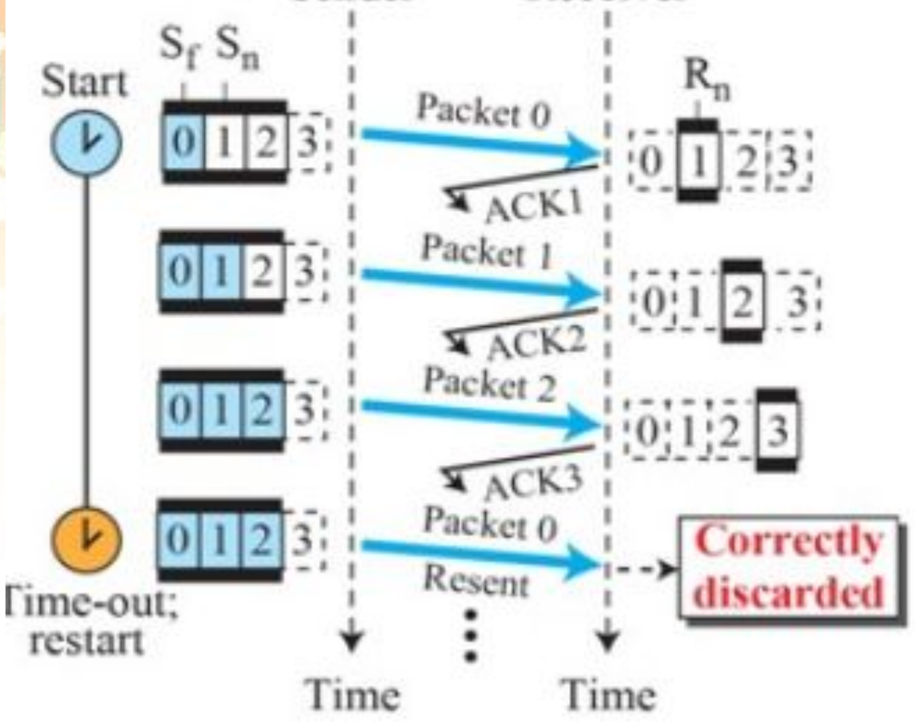
Initial
Start timer
Stop
Start
Time out
Restart
Restart
Stop timer
Req
Packet 0
ACK 1
pArr
aArr
Packet 1
Lost
Packet 2
Packet discarded
Packet 3
ACK discarded
Packet 1 (resent)
Packet 2 (resent)
Packet 3 (resent)
ACK 2
ACK 3
ACK 4
Events:
Req: Request from process
pArr: Packet arrival
aArr: ACK arrival
time-out: Timer expiration

- The Go-Back-N protocol simplifies the process at the receiver.
- The receiver keeps track of only one variable, and there is no need to buffer out-of-order packets
- they are simply discarded. However, this protocol is *inefficient* if the underlying network protocol *loses a lot of packets*.
- Each time a single packet is lost or corrupted, the sender resends all outstanding packets, even though some of these packets may have been received safe and sound but out of order.

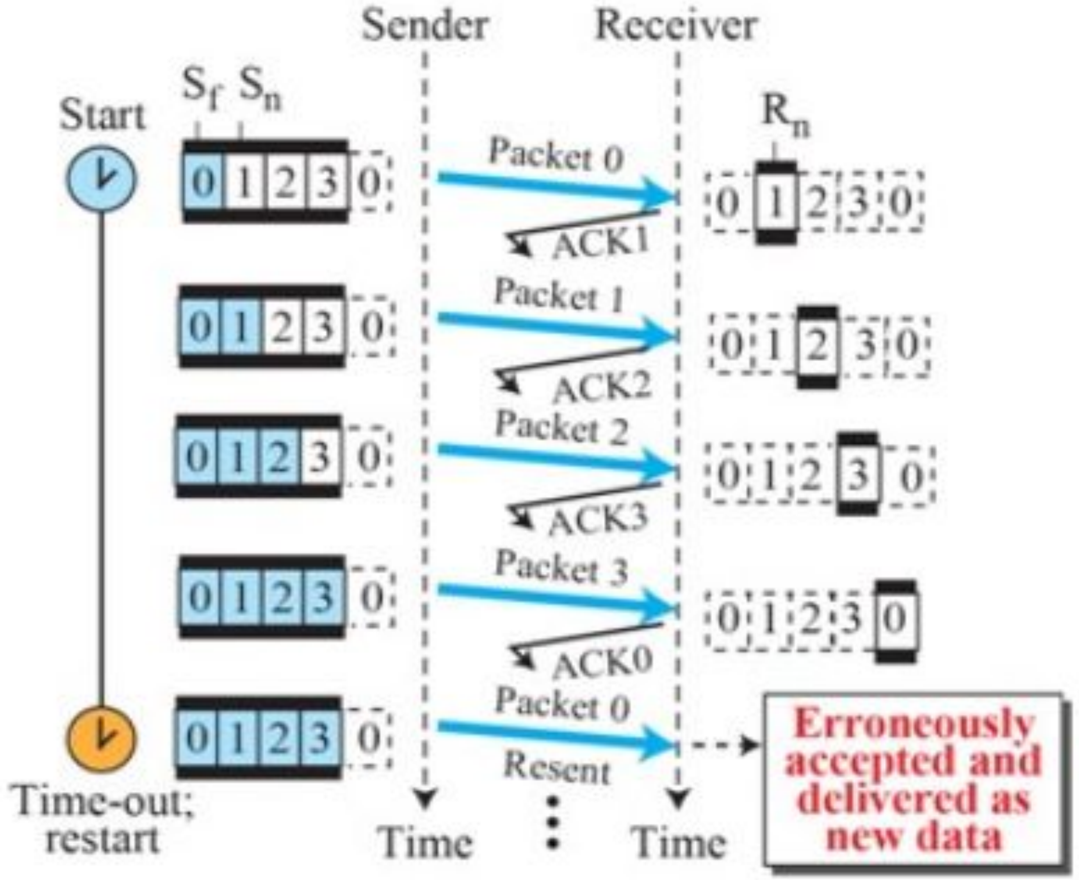
- Acknowledgements are cumulative and defines the sequence number of the next packet to arrive
- Timer: when timer expires the packets are resent
- Resending packet: resend all packets who have not received ack as timer expires. Hence it is called the Go Back N

Send window size for Go-Back-N

- Size of the send window must be $< 2^m$

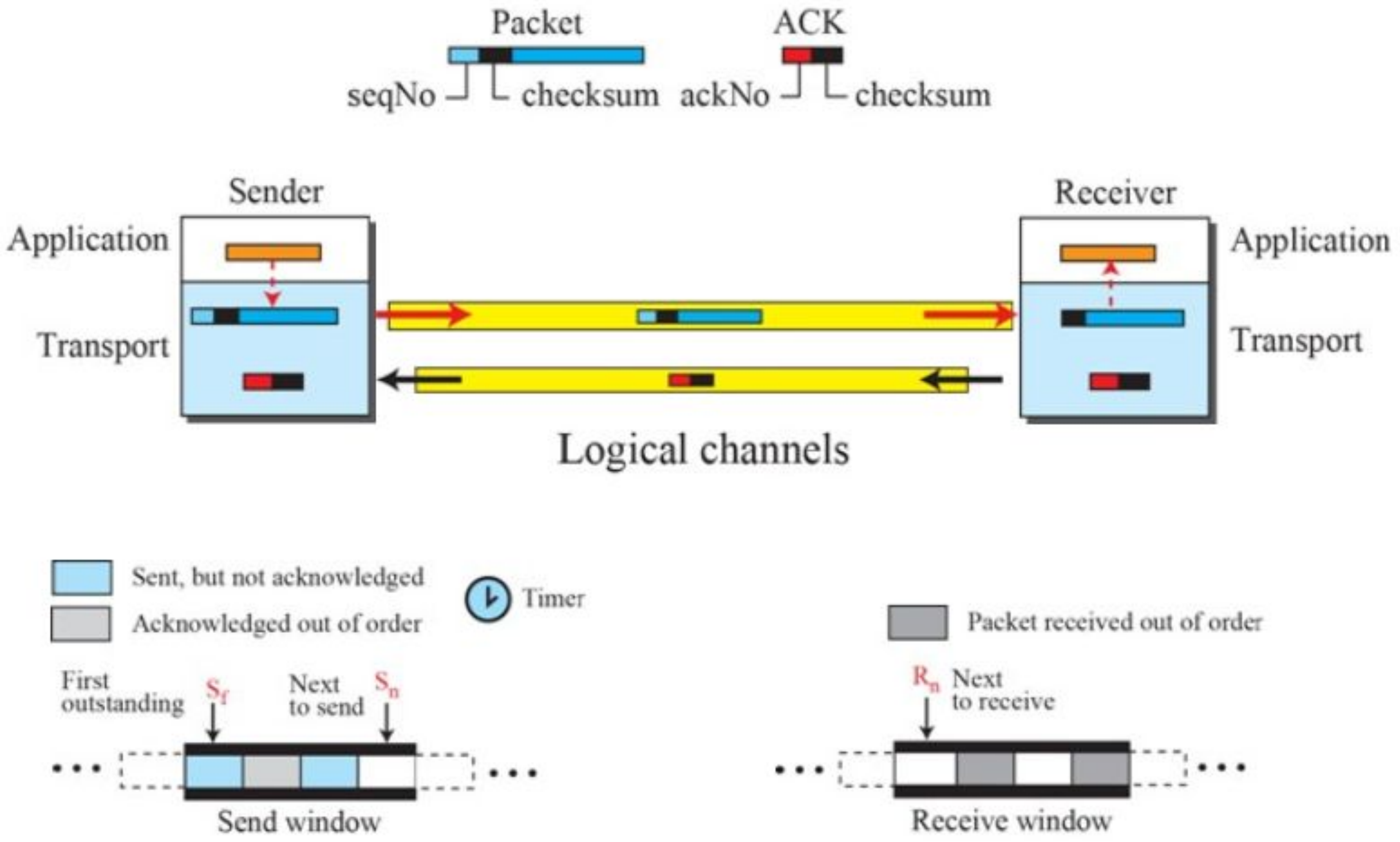


a. Send window of size $< 2^m$

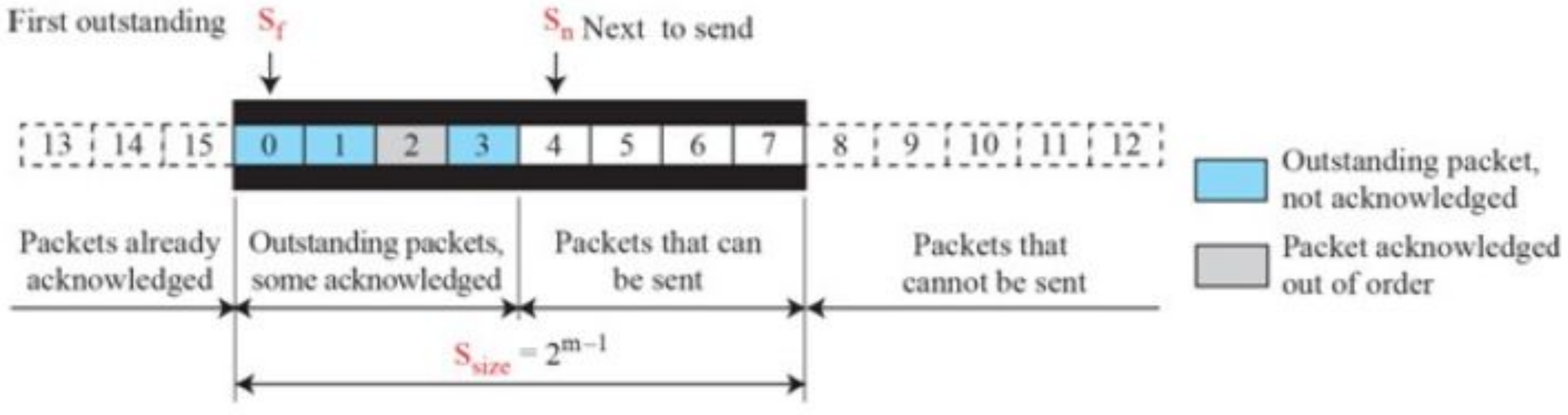


b. Send window of size $= 2^m$

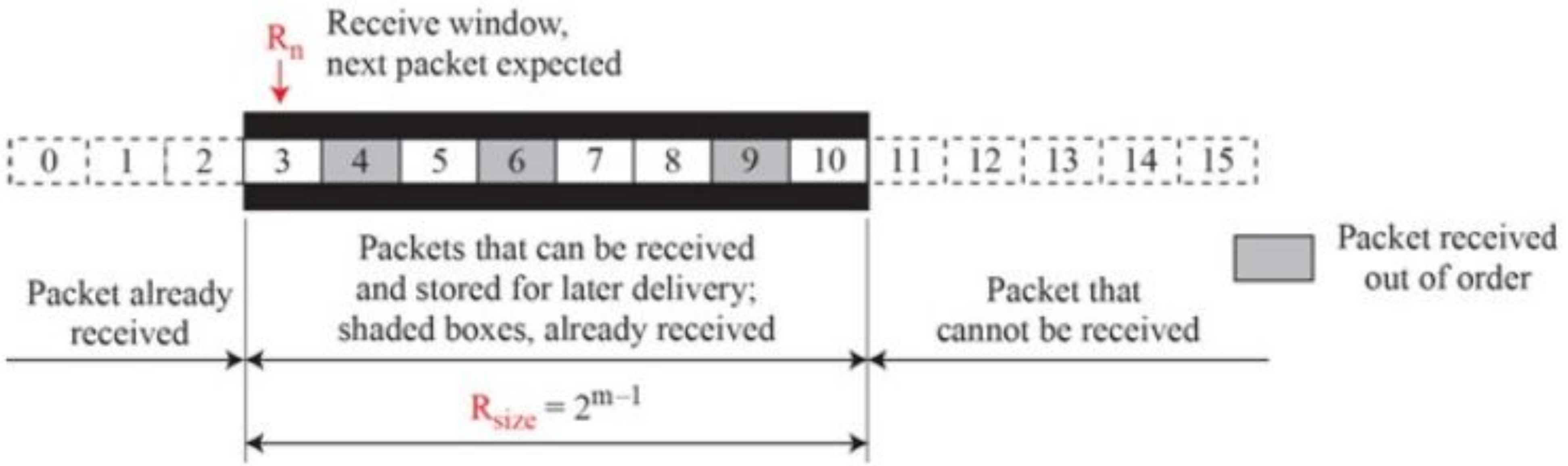
4) Select Repeat Protocol



Send window for Selective-Repeat protocol


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

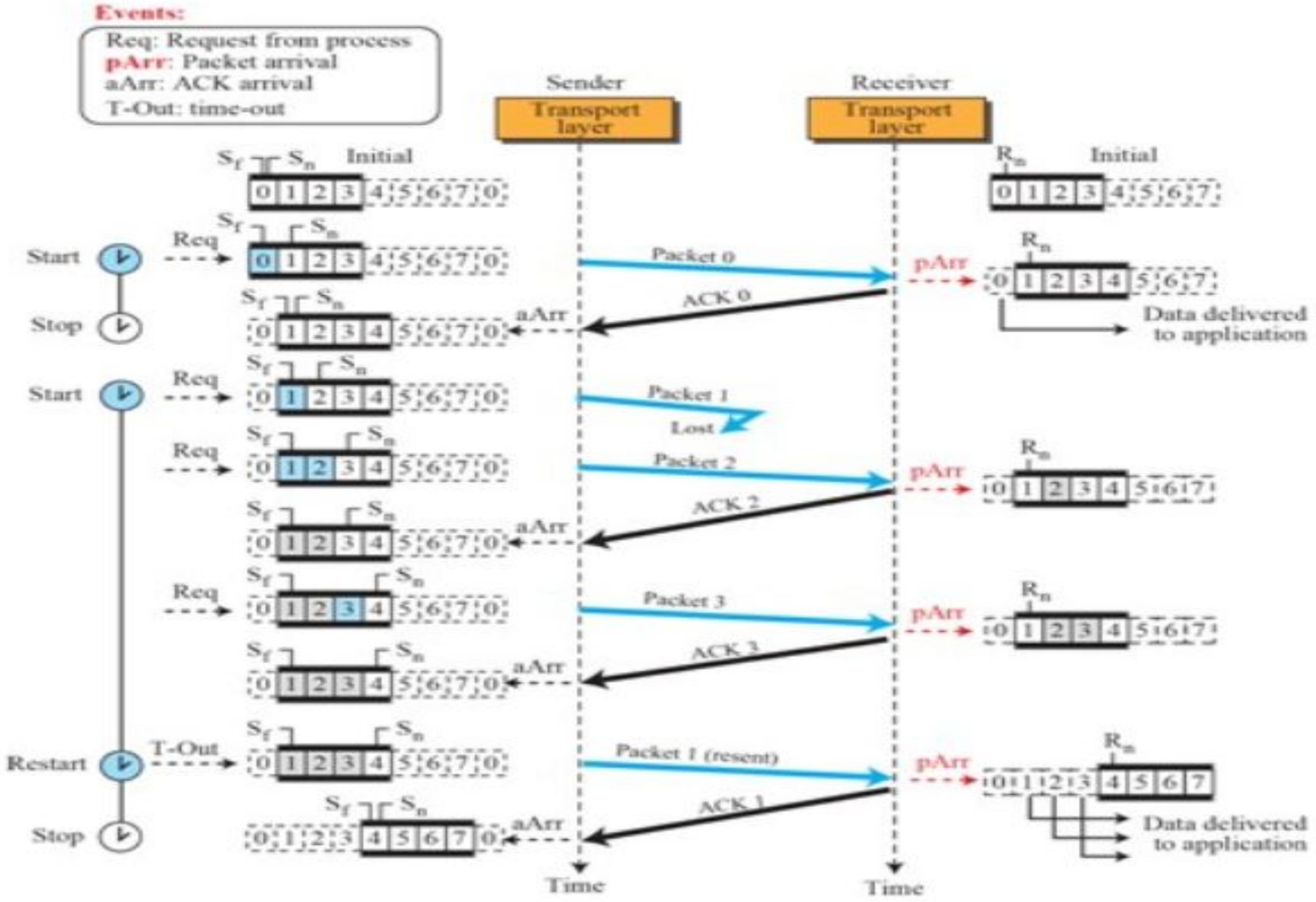
- Receive window for Selective-Repeat protocol


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

In Select repeat , an Acknowledgement defines

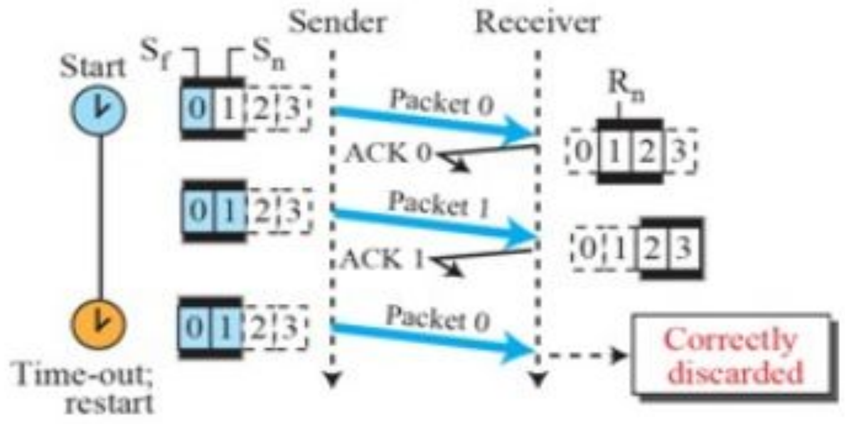
- the sequence no of the error free packet received
- No feedback for any other, like GBN

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

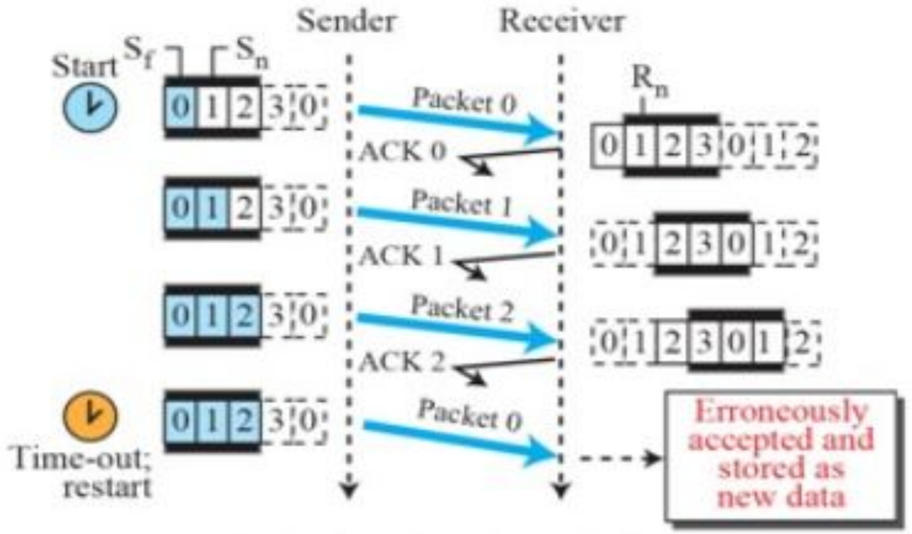

Events:
Req: Request from process
pArr: Packet arrival
aArr: ACK arrival
T-Out: time-out
Sender
Transport layer
Receiver
Transport layer
Initial
Start
Stop
Restart
Req
T-Out
aArr
pArr
Packet 0
ACK 0
Packet 1
Lost
Packet 2
ACK 2
Packet 3
ACK 3
Packet 1 (resent)
ACK 1
Data delivered to application
Time

- At the receiver site we need to distinguish between the acceptance of a packet and its delivery to the application layer.
- At the second arrival, packet 2 arrives and is stored and marked (shaded slot), but it cannot be delivered because packet 1 is missing.
- At the next arrival, packet 3 arrives and is marked and stored, but still none of the packets can be delivered. Only at the last arrival, when finally a copy of packet 1 arrives, can packets 1, 2, and 3 be delivered to the application layer.
- There are *two conditions* for the delivery of packets to the application layer:
  - ***First, a set of consecutive packets must have arrived.***
  - ***Second, the set starts from the beginning of the window.***

Selective-Repeat, window size



a. Send and receive windows of size $= 2^{m-1}$

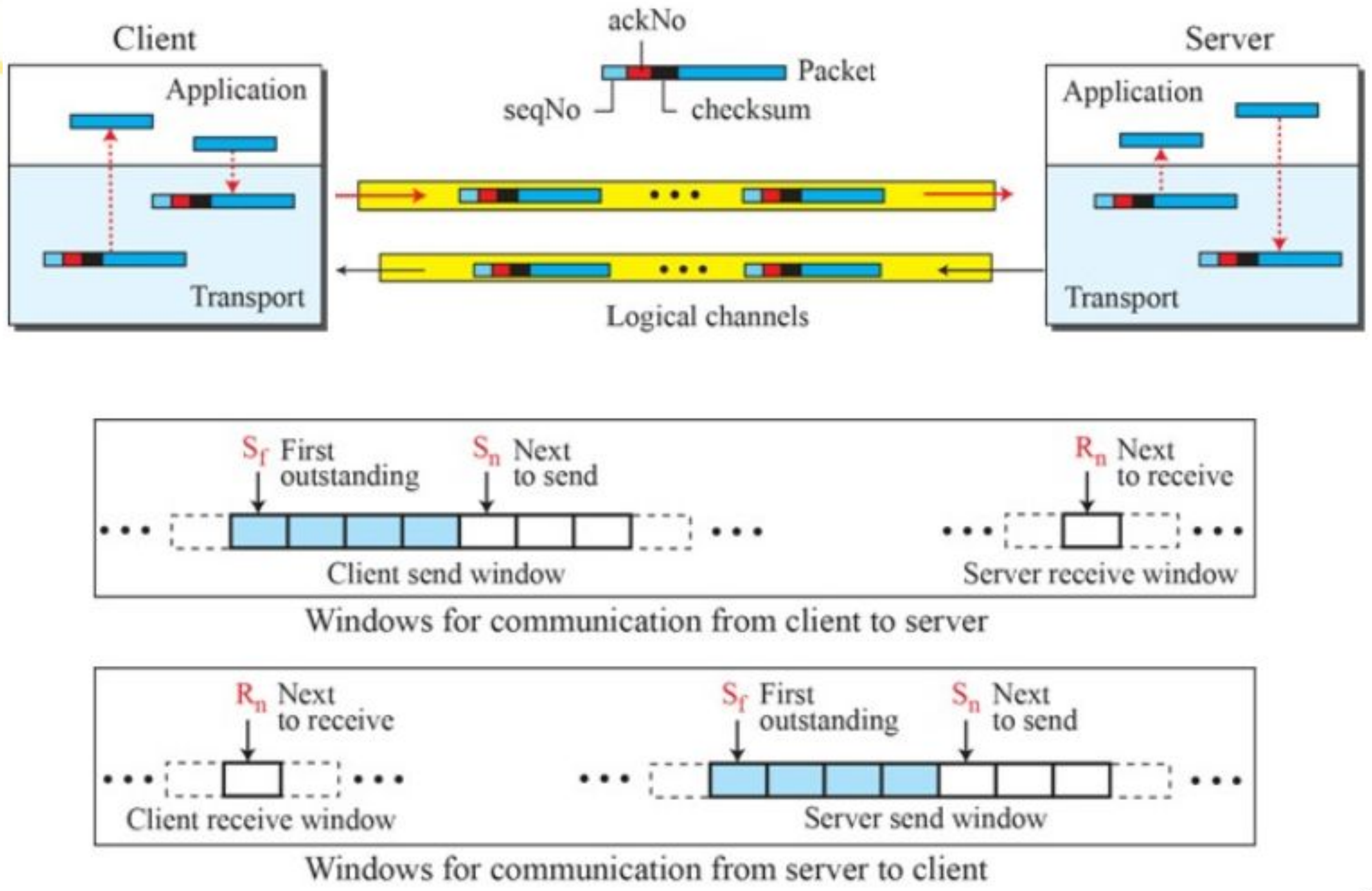


b. Send and receive windows of size $> 2^{m-1}$

5) Bidirectional Protocols : Piggybacking

- All discussed so far are all unidirectional: data packets flow in only one direction and acknowledgments travel in the other direction.
- In real life, data packets are normally flowing in both directions: from client to server and from server to client.
- This means that acknowledgments also need to flow in both directions.
- A technique called ***piggybacking*** is used to improve the efficiency of the bidirectional protocols.
- When a packet is carrying data from A to B, it can also carry acknowledgment feedback about arrived packets from B

Design of piggybacking in Go-Back-N



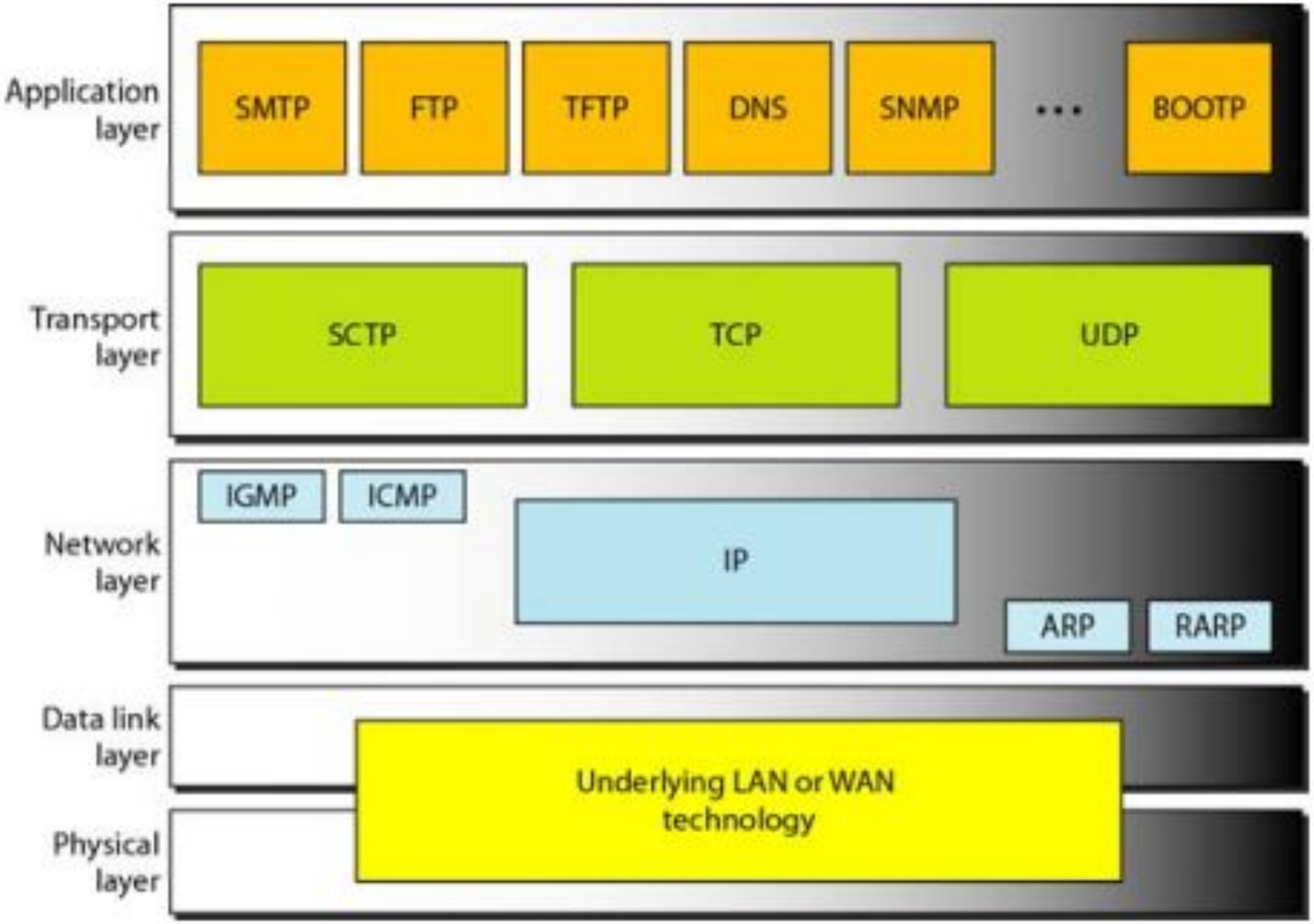
Transport layer duties/ Design Criterion

- **Addressing**
  - Port numbers to identify which network application
- **Reliability**
  - Flow control
  - Error Control
- **Connection control**
  - Connection-oriented
  - Connectionless
- **Packetizing**
  - Sender side: breaks **application messages** into segments, passes them to network layer
  - Transport layer at the receiving host deliver data to the receiving process

- The Internet supports two transport layer protocols:
  - The Transport Control Protocol (**TCP**) for reliable service
  - The Unreliable (User) Datagram Protocol (**UDP**)

Processes communicating across network

- Process is an instance of a program in execution.
- Processes on two hosts communicate with each other by **sending and receiving messages**
- The process receives messages from, and sends messages into the network through its **socket**
- A socket is the **interface** between the **application layer** and the **transport layer** within a host.
- **Sockets** are the **programming interface** used to build network applications over the internet.
- Programmers can select which transport layer protocol (UDP or TCP) to be used by the application and select few transport-layer parameters (maximum buffer size, Maximum segment size, starting sequence number of segment).


Application layer
SMTP
FTP
TFTP
DNS
SNMP
• • •
BOOTP
Transport layer
SCTP
TCP
UDP
Network layer
IGMP
ICMP
IP
ARP
RARP
Data link layer
Physical layer
Underlying LAN or WAN technology

User Datagram Protocol (UDP)

- **Connectionless**
  - **No handshaking** between UDP sender, receiver
  - Each UDP segment handled **independently** of others

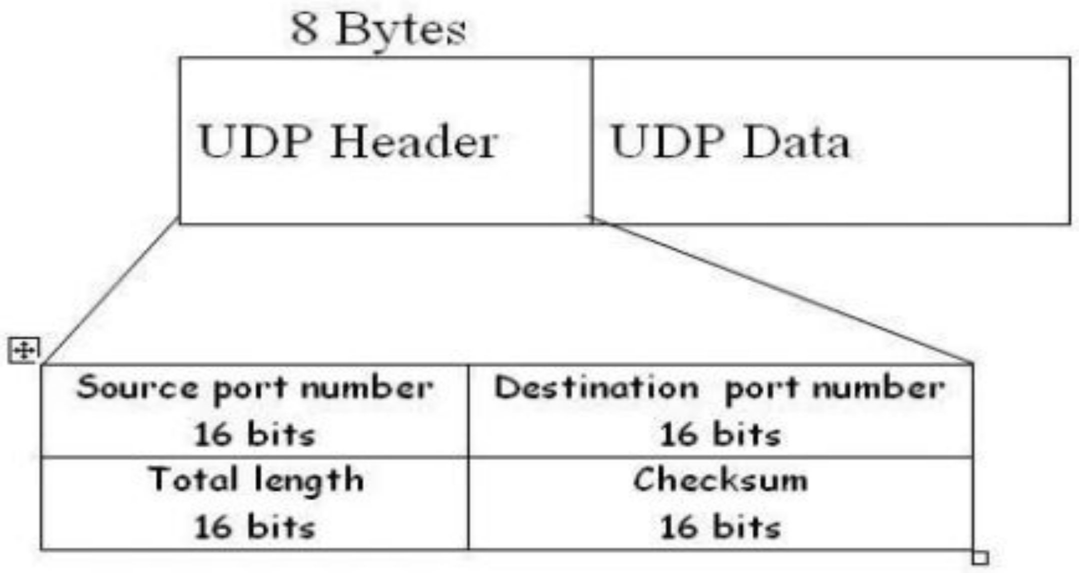
- **<u>Unreliable protocol</u> has no flow and error control**
  - A UDP segment can be **lost**, **arrive out of order**, **duplicated**, or **corrupted**
  - **Checksum field checks error in <u>the entire UDP segment</u>. It is Optional**
  - UDP **doe not do anything to recover** from an error it simply **discard** the segment
  - Application accepts full responsibility for errors

- It **uses port numbers** to multiplex/demultiplex data from/to the application layer.
- Advantages: Simple, **minimum overhead**, **no connection delay**
- **Services provided by UDP:**
    - Process-to-Process delivery
    - Error checking : Checksum(however, if there is an error UDP does NOT do anything to recover from **error. It will just** discard the message)
    - No error control
    - No flow control
    - No congestion control
    - Encapsulation and decapsulation
    - Message-oriented protocol (**datagram**)

User datagram format

- **Header size = 8 bytes**
  Minimum UDP **process data** size 0 bytes
- Maximum UDP **process data** size=

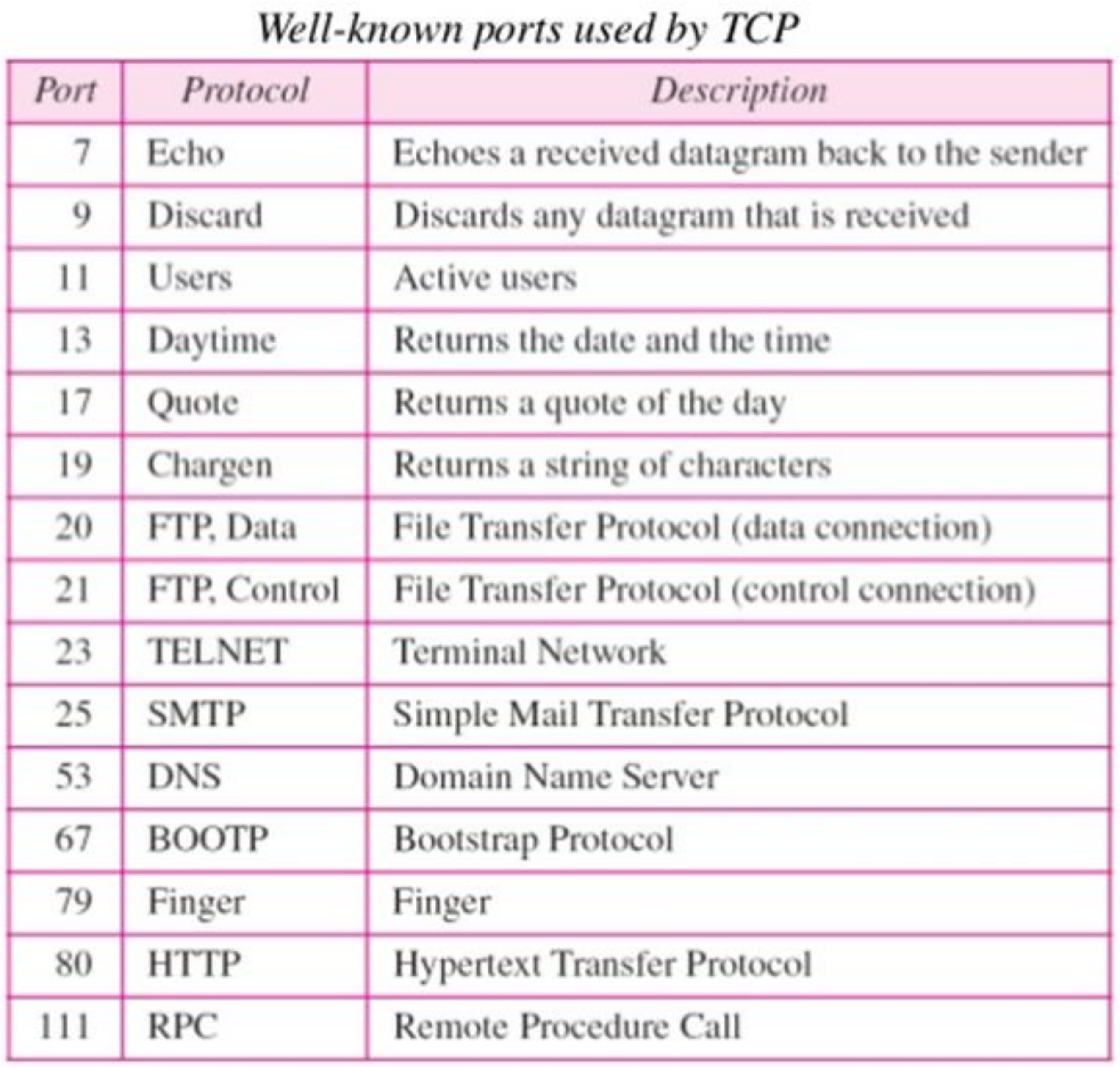
65535 – 20 (network layer headers) - 8 (UDP headers)= **65507** bytes



- Used for applications that can tolerate small amount of packet loss:
  - Multimedia applications,
  - Internet telephony,
  - real-time-video conferencing
  - Domain Name System messages
  - Audio
  - Routing Protocols

Transmission Control Protocol (TCP)

- connection-oriented protocol
- reliable
- flow and error control mechanisms at the transport level

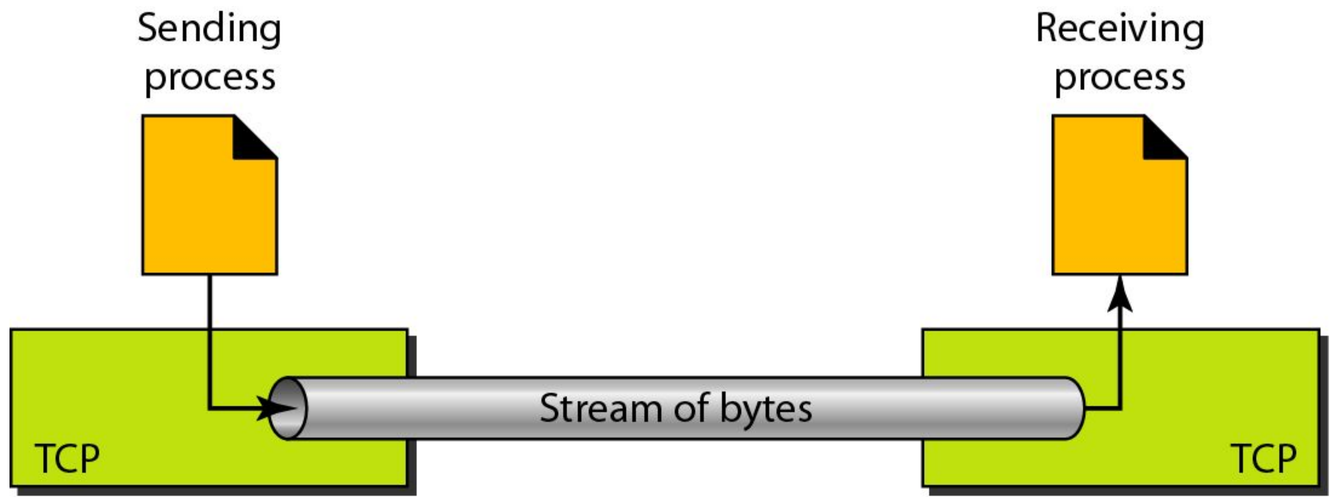


Well-known ports used by TCP

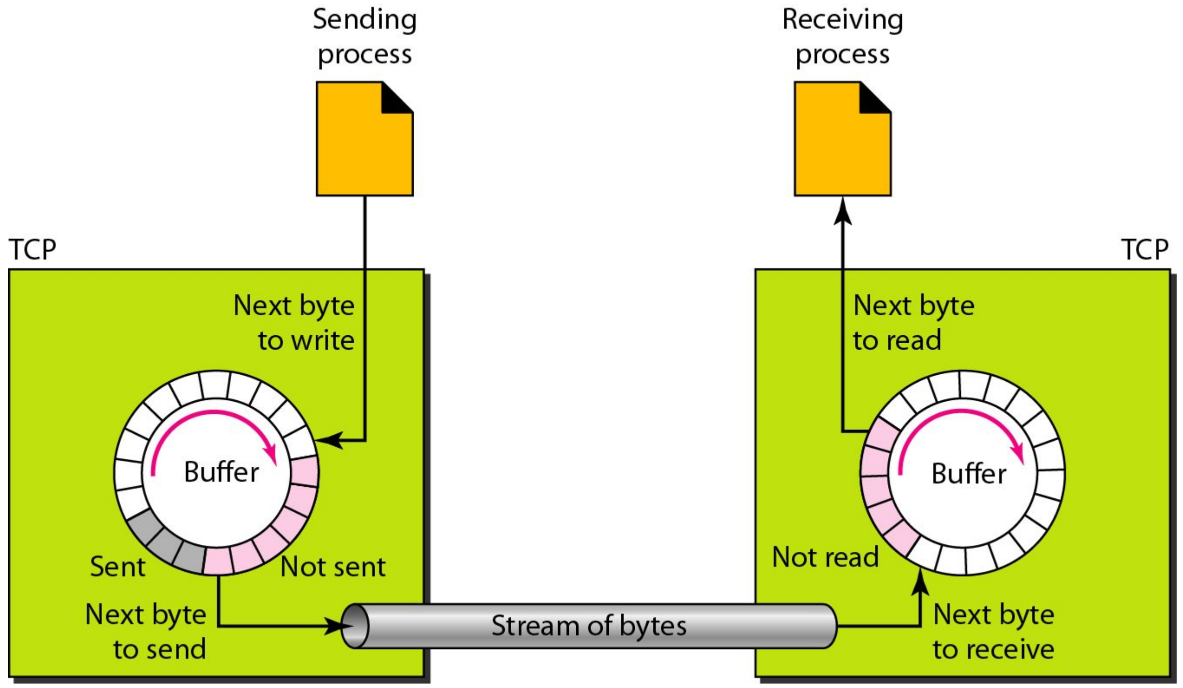
Port	*Protocol*	*Description*
7	Echo	Echoes a received datagram back to the sender
9	Discard	Discards any datagram that is received
11	Users	Active users
13	Daytime	Returns the date and the time
17	Quote	Returns a quote of the day
19	Chargen	Returns a string of characters
20	FTP, Data	File Transfer Protocol (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

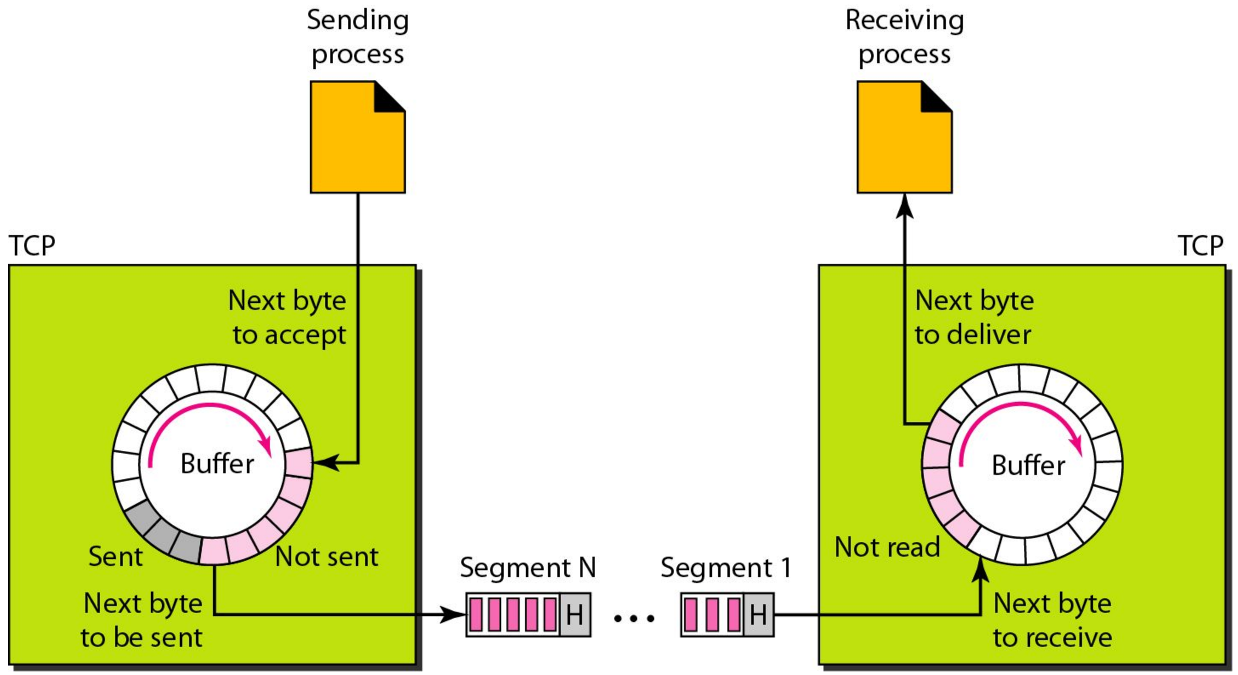
1. Process to Process Communication
2. Stream Delivery Service
   - Deliver/receives data as **stream** of bytes



- Sending and receiving buffers
  - ✔ Sending and receiving process may not operate at same rate
  - ✔ Flow control techniques are used



- Segments
  - ✔ Network layer need data as packets not in stream of bytes
  - ✔ TCP groups of bytes into segments at the transport layer
  - ✔ TCP adds header to each segment for control purpose and delivers to network layer
  - ✔ Segment size can be variable



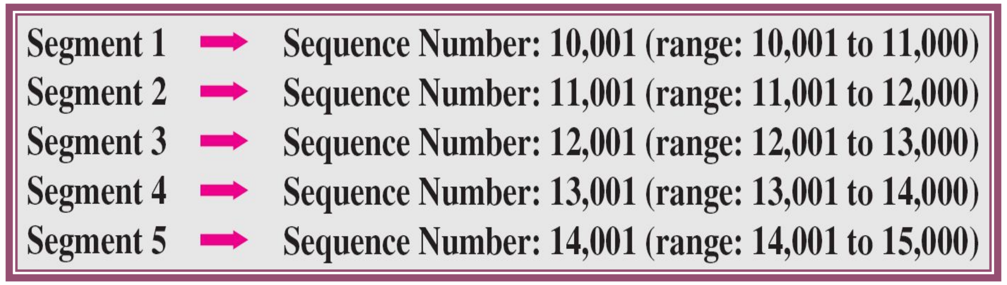
3. Full duplex Communication
4. Multiplexing and Demultiplexing
5. Connection Oriented Service
    a. Establishment of logical connection
    b. Data exchange in both direction
    c. Connection Termination
6. Reliable Service

TCP Numbering System

- ✔ Byte no not segment no
  - ▪ Very octet is numbered
  - ▪ Numbering is independent at sender and receiver
  - ▪ Need not start with 0, it starts with arbitrary no
- ✔ For tracking
  - ▪ Sequence no
    - • To each segment
    - • Sequence no of 1st segment is referred to as ISN (first byte)
    - • Next segment is previous no +1

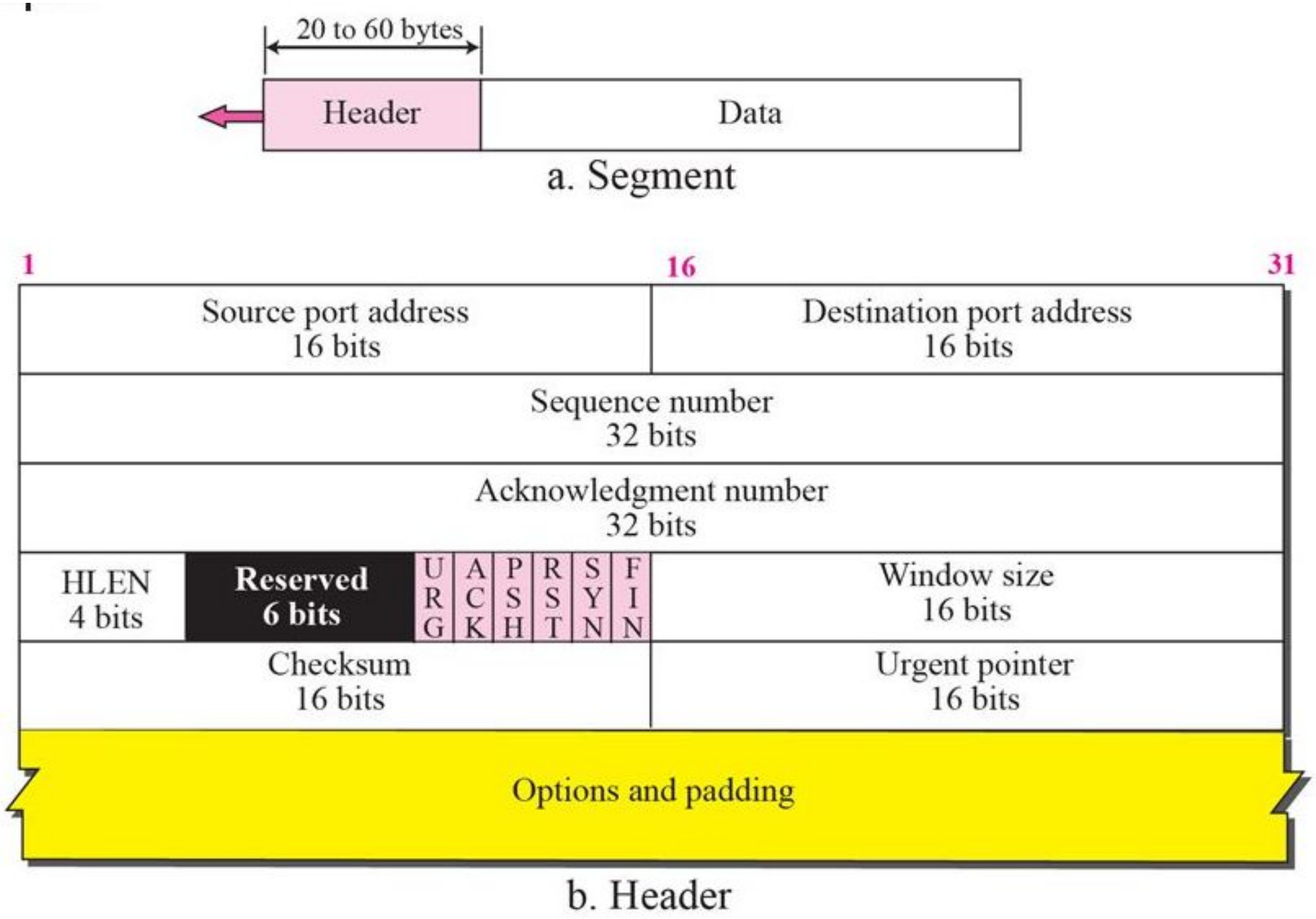
  - ▪ Acknowledgement no
    - • Sequence no +1 *

    (* one block of 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)

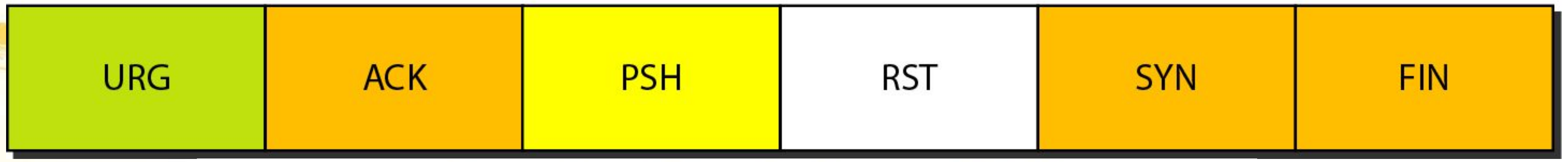
TCP Segment



20 to 60 bytes
Header
Data
a. Segment
1
16
31
Source port address 16 bits
Destination port address 16 bits
Sequence number 32 bits
Acknowledgment number 32 bits
HLEN 4 bits
Reserved 6 bits
URG
ACK
PSH
RST
SYN
FIN
Window size 16 bits
Checksum 16 bits
Urgent pointer 16 bits
Options and padding
b. Header

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



Description of flags in the control field

Flag	*Description*
URG	The value of the urgent pointer field is valid.
ACK	The value of the acknowledgment field is valid.
PSH	Push the data.
RST	Reset the connection.
SYN	Synchronize sequence numbers during connection.
FIN	Terminate the connection.

- TCP segment encapsulates data received from application layer
- itself be encapsulated in an IP datagram at the network layer
- later as frame in data link layer

TCP Connection

Connection oriented

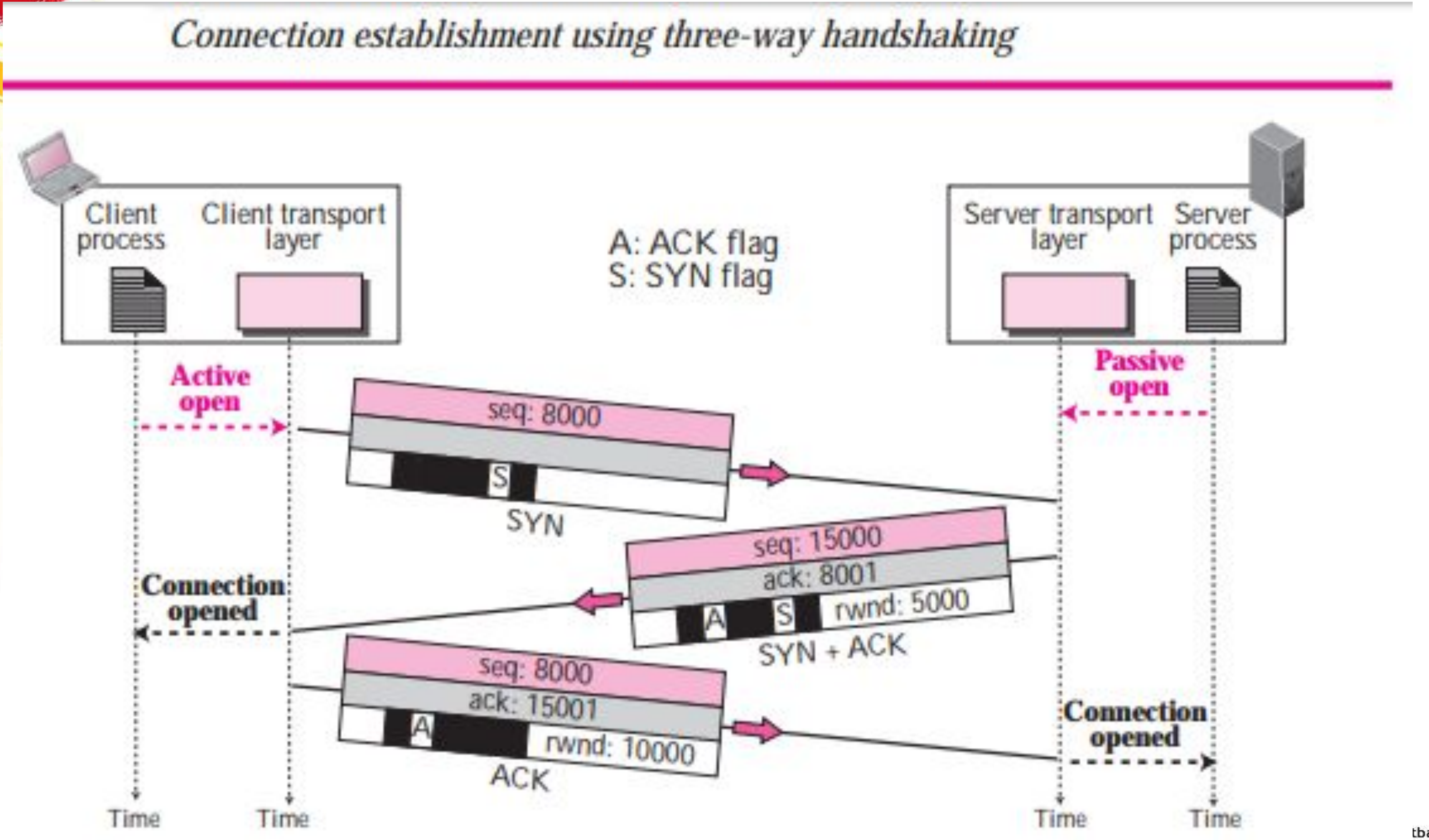
- Both sender and receiver need to establish connection separately

- **Logical path** b/w transport layer facilitates the message and acknowledgement delivery and retransmission of packets.

- 3 phases
  1. ***Establishment*** of logical connection
  2. Data ***exchange*** in both direction
  3. Connection ***Termination***

- **Fully** duplex mode

Step 1: Establishment of logical connection



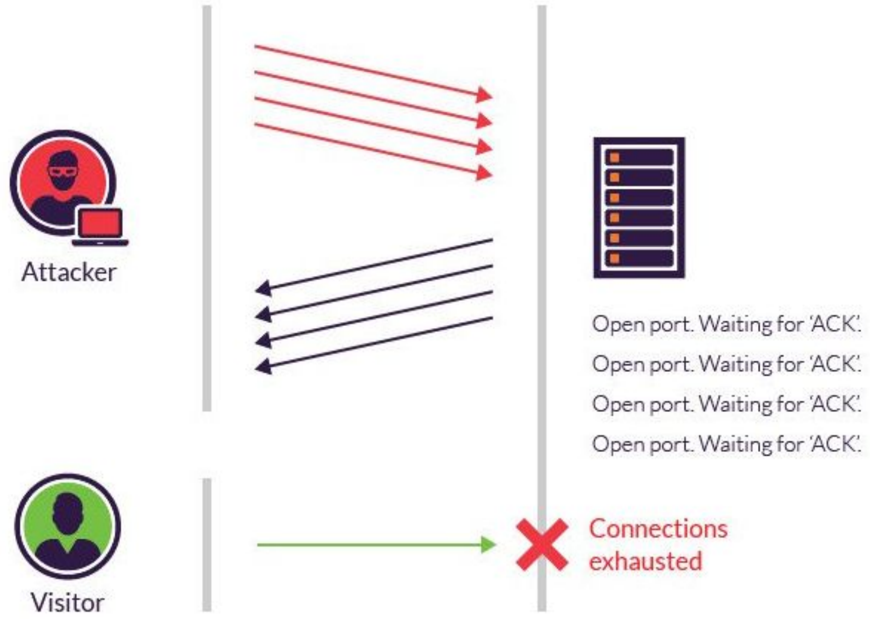
3 way handshake

- Passsive open – server ready to accept connections
- Active open – client that wishes to connect to open server.
- SYN segement – to synchronize sequence nos
  - First no is called ISN (Initial Sequence No)
  - No data
  - No other flags or window size
- SYN + ACK
  - Dual purpose
  - Server sends ISN and ACK to the SYN received
  - As ACK, it need to send the receive window size (***rwnd***)

- A SYN segment cannot carry data, but it consumes one sequence number.
- A SYN + ACK segment cannot carry data, but does consume one sequence number.
- An ACK segment, if carrying no data, consumes no sequence number.

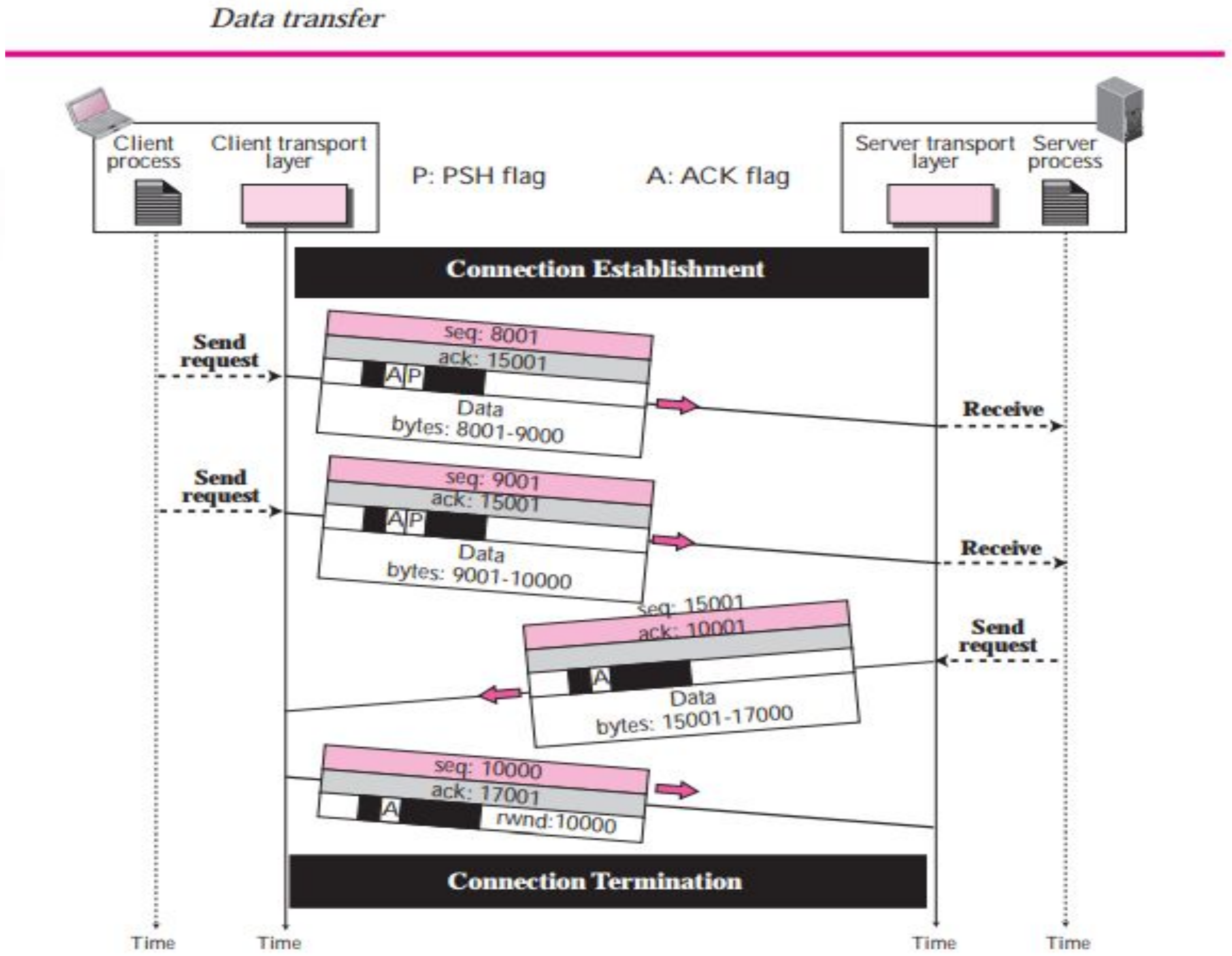
Note: SYN Flooding Attack

- TCP is *vulnerable* to SYN Flooding Attack
- The attacker sends repeated SYN packets to every port on the targeted server, often using a fake IP address.
- The server, unaware of the attack, receives multiple, apparently legitimate requests to establish communication.
- It responds to each attempt with a SYN-ACK packet from each open port.
- The malicious client either does not send the expected ACK, or—if the IP address is spoofed—never receives the SYN-ACK
- Either way, the server under attack will wait for acknowledgement of its SYN-ACK packet for some time.


Attacker
Open port. Waiting for 'ACK'.
Open port. Waiting for 'ACK'.
Open port. Waiting for 'ACK'.
Open port. Waiting for 'ACK'.
Visitor
Connections exhausted

- Now, the server cannot close down the connection
- Before the connection can time out, another SYN packet will arrive. This leaves an increasingly large number of connections half-open
- Also referred to as "***half-open***" attacks.
- Eventually, as the server's connection overflow, service to legitimate clients will be denied, and the server may even ***malfunction or crash***.
- Hence, tries to exhaust network ports, SYN packets can also be used in DDoS attacks that try to clog your pipes with fake packets to achieve network saturation.

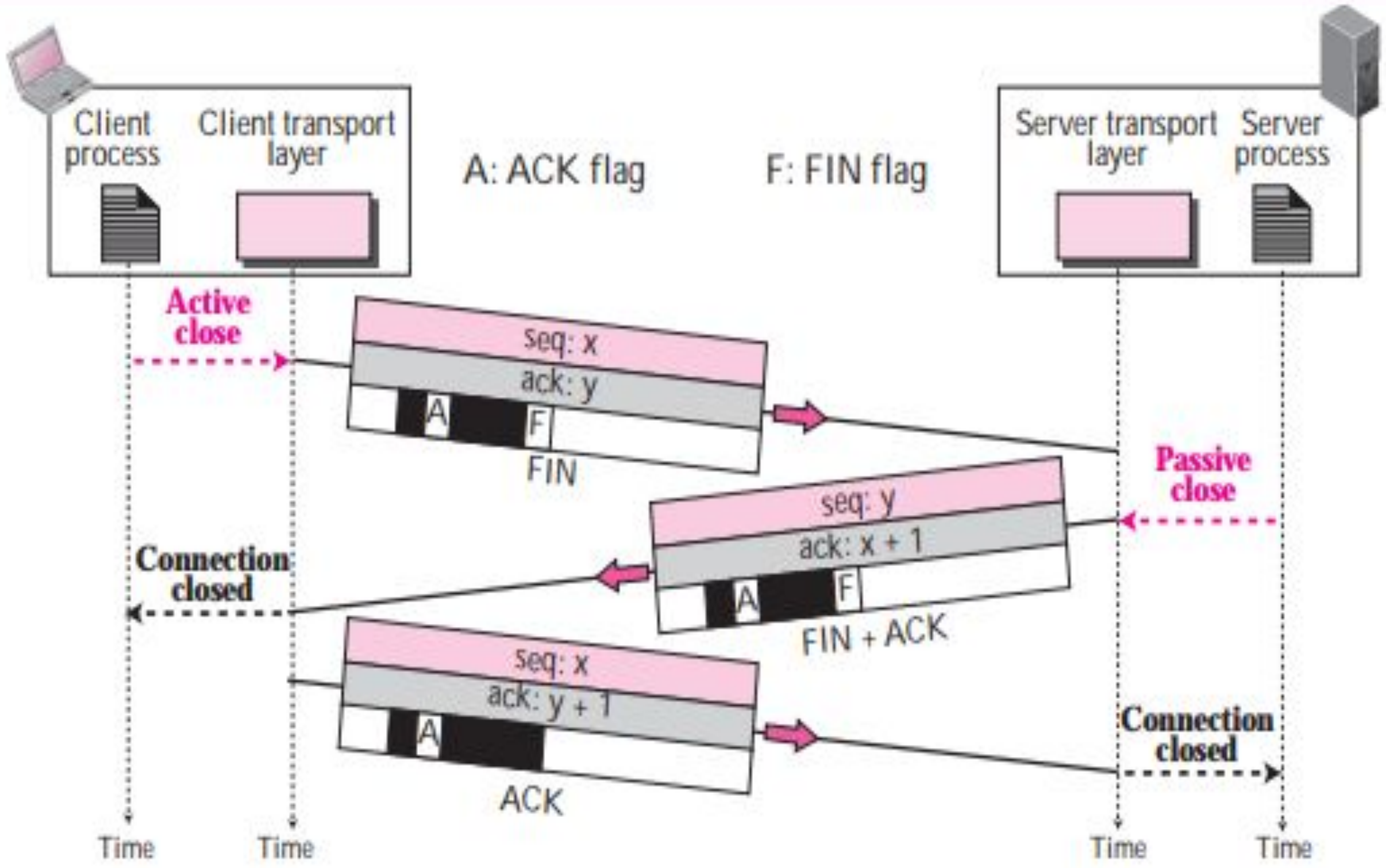
Step 2: Data ***exchange*** in both direction



- bidirectional data transfer can take place
- The client and server can both send data and acknowledgments
- acknowledgment is piggybacked with the data.
- The first three segments carry both data and acknowledgment, but the last segment carries only an acknowledgment because there are no more data to be sent
- ***Pushing Data-***PSH (push) flag set so that the server TCP knows to deliver data to the server process as soon as they are received
- ***Urgent Data-***application program wants a piece of data to be read *out of order* by the receiving application program

Step 3: Connection ***Termination***

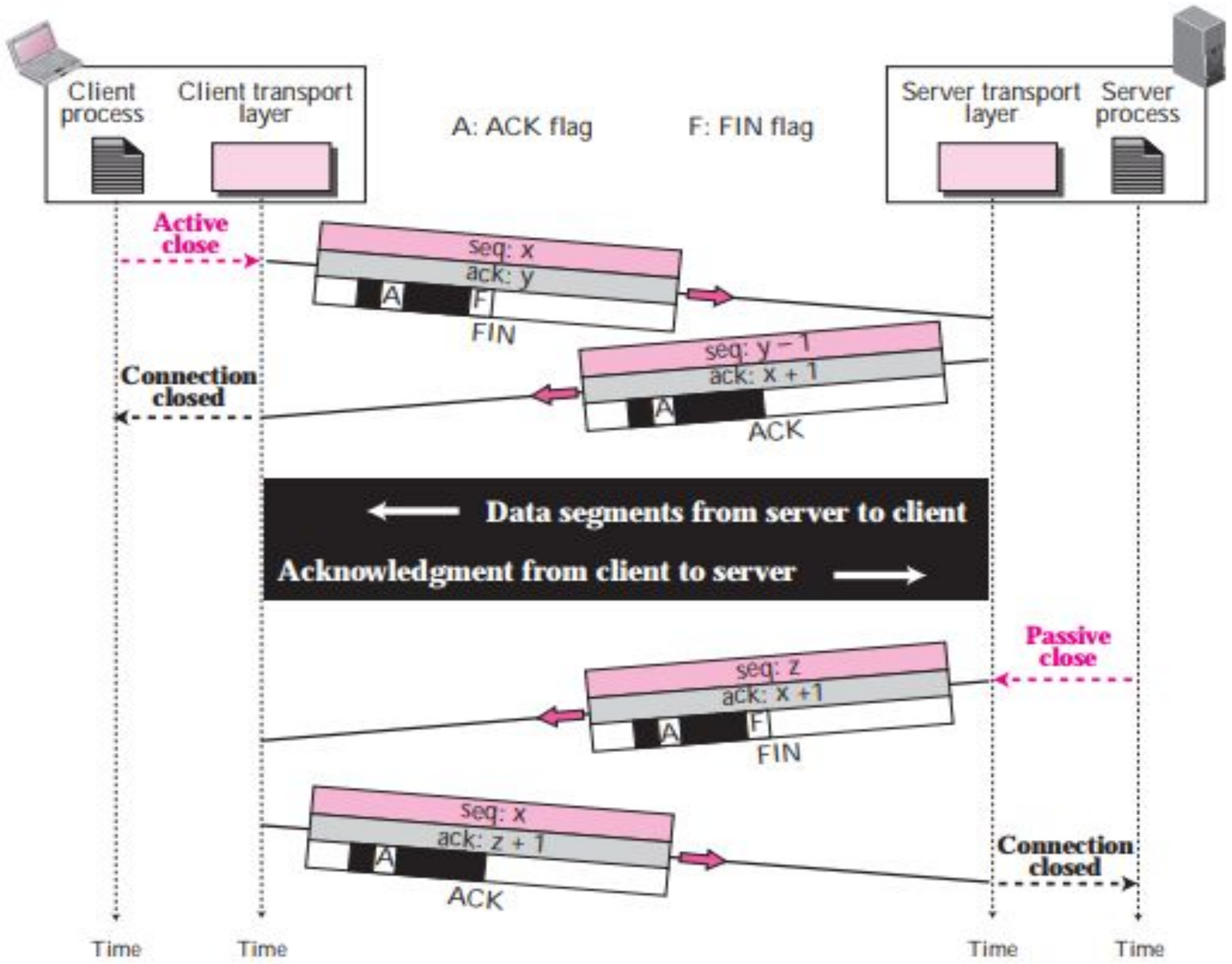
Connection termination using three-way handshaking



- Either can close the connection, although it is usually initiated by the client
- Two options for connection termination:
  - **three-way handshaking**
    - To close a FIN segment in which the FIN flag is set
    - 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
    - last segment, an ACK segment, to confirm the receipt of the FIN segment from the TCP server
  - **four-way handshaking with a half-close option.**

- The FIN segment consumes one sequence number if it does not carry data.
- The FIN + ACK segment consumes one sequence number if it does not carry data.

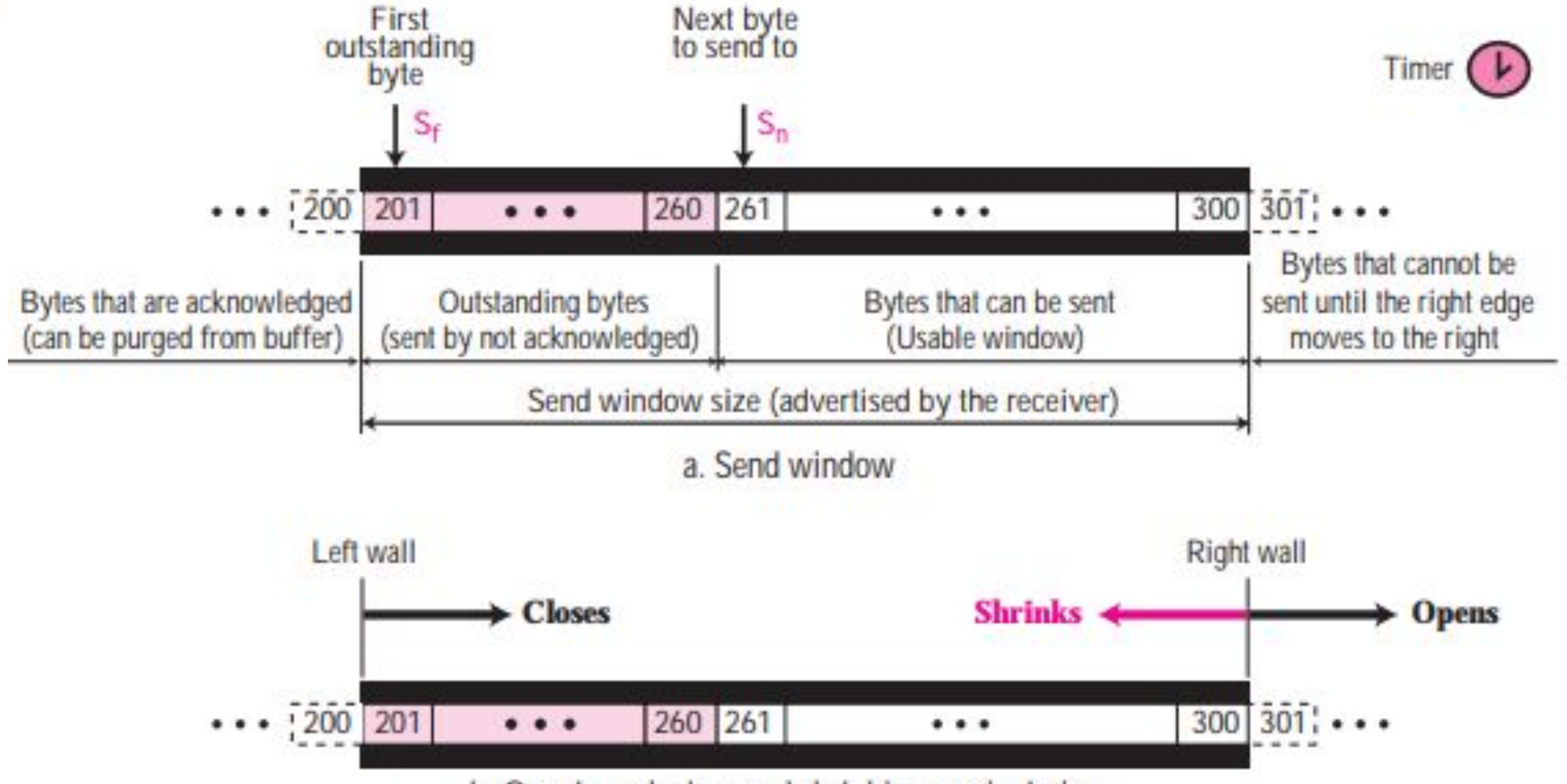
Half-close


Client process
Client transport layer
A: ACK flag
F: FIN flag
Server transport layer
Server process
Active close
seq: x
ack: y
A
F
FIN
seq: y − 1
ack: x + 1
A
ACK
Connection closed
Data segments from server to client
Acknowledgment from client to server
Passive close
seq: z
ack: x +1
A
F
FIN
seq: x
ack: z + 1
A
ACK
Connection closed
Time
Time
Time
Time

- one end can stop sending data while still receiving data. This is called a ***half-close***.
- The client half-closes the connection by sending a FIN segment.
- The server accepts the half-close by sending the ACK segment.
- The data transfer from the client to the server stops. The server, however, can still send data.
- When the server has sent all the processed data, it sends a FIN segment, which is acknowledged by an ACK from the client.

Windows in TCP for Flow Control

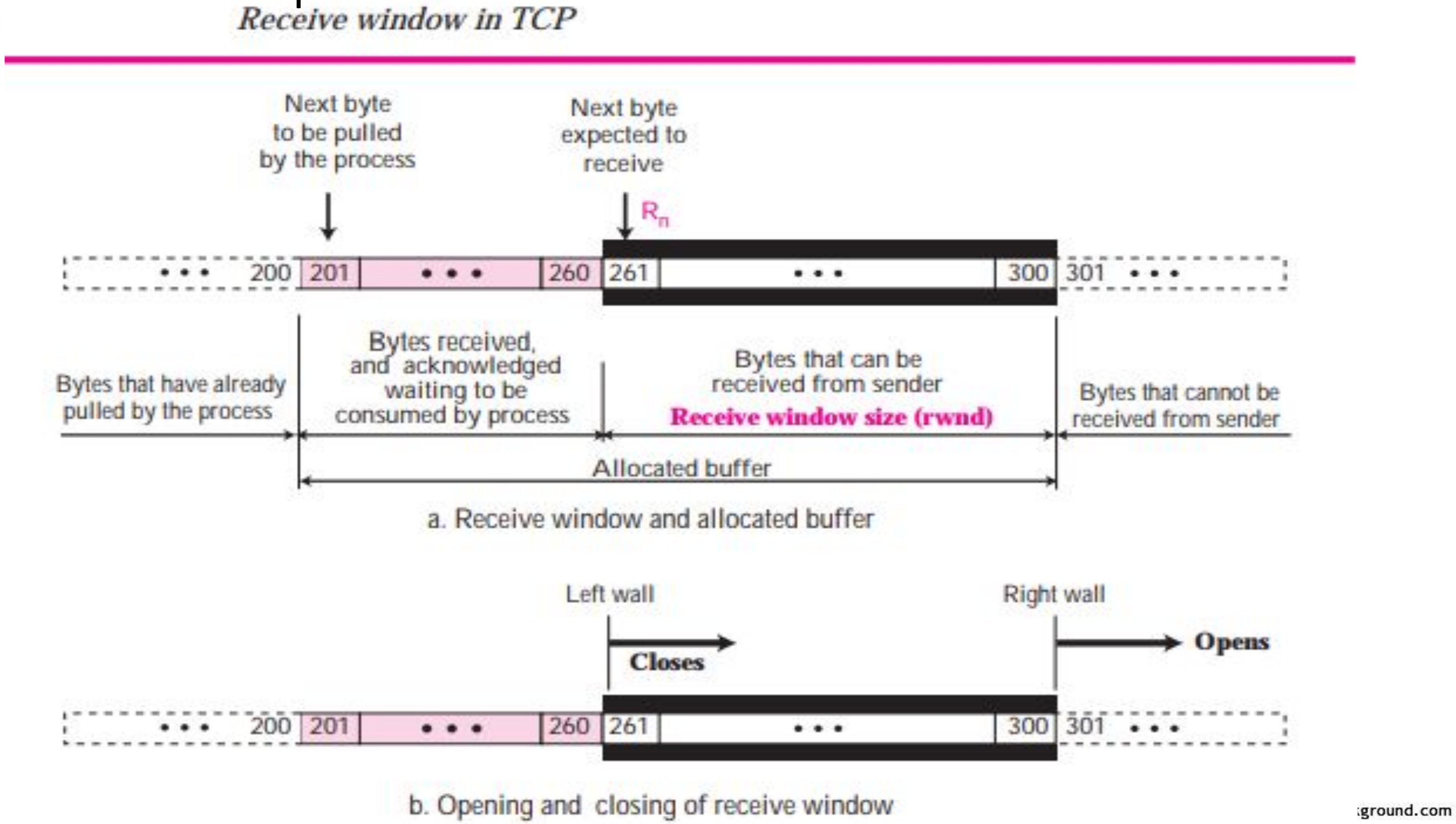
- Send Window
  - send window size is dictated by the receiver (flow control) and the congestion in the underlying network (congestion control).


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

a. Send window

b. Opening, closing, and shrinking send window

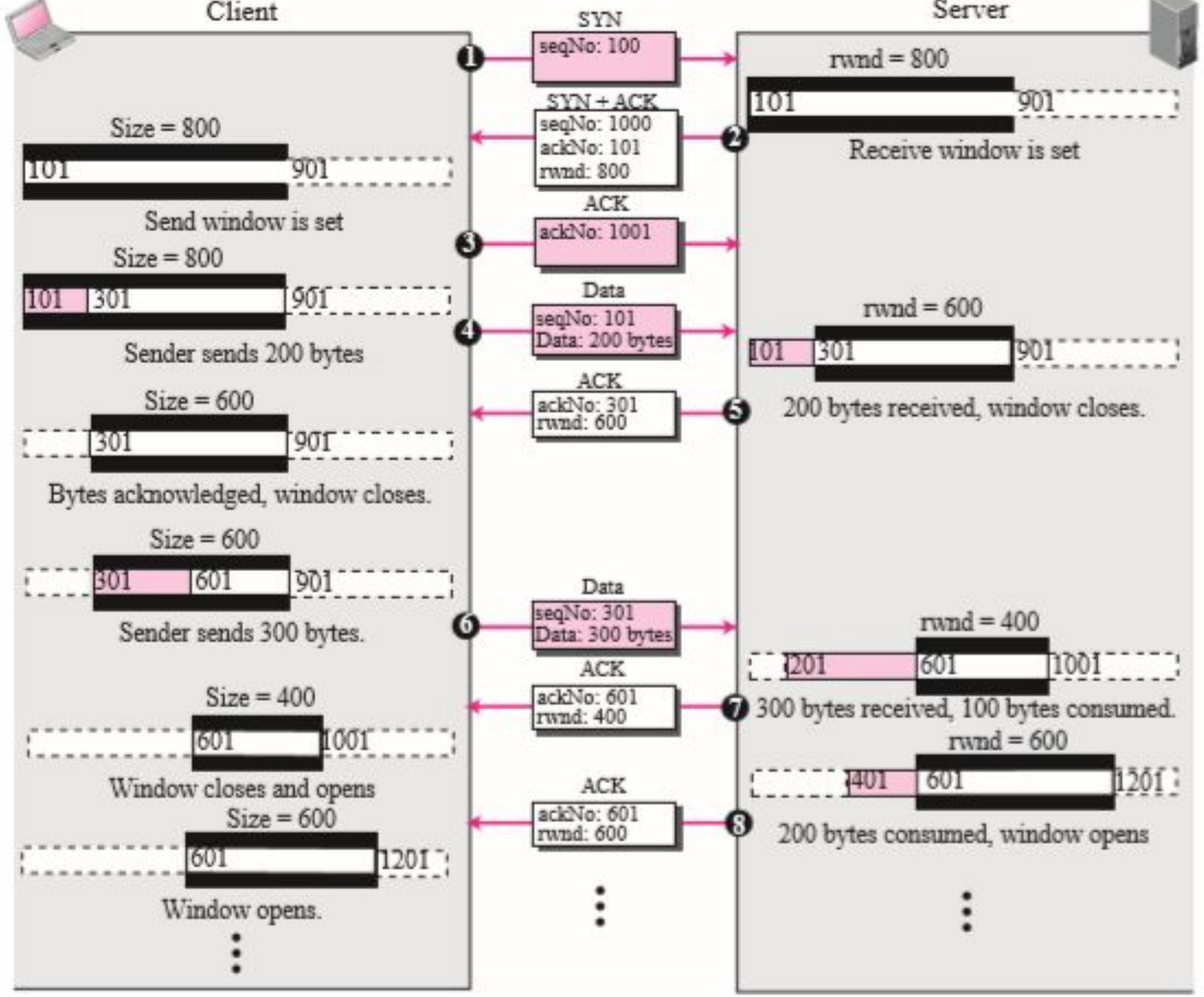
- Receive Window
  - Does not shrink like senders
  - The receiver window size determines the number of bytes that the receive window can accept from the sender



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

Opening and Closing Windows

- To achieve flow control, TCP forces the ***sender and the 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
  - opens (moves its right wal l to the right) when more bytes are pulled by the process
  - Does not shrink
- ***The opening, closing, and shrinking of the send window is controlled by the receiver***
- The send window
  - closes (moves its left wall to the right) when a new acknowledgement allows it to do so.
  - opens (its right wall moves to the right) when the receive window size (rwnd) advertised by the receiver allows it to do so.
  - May shrink

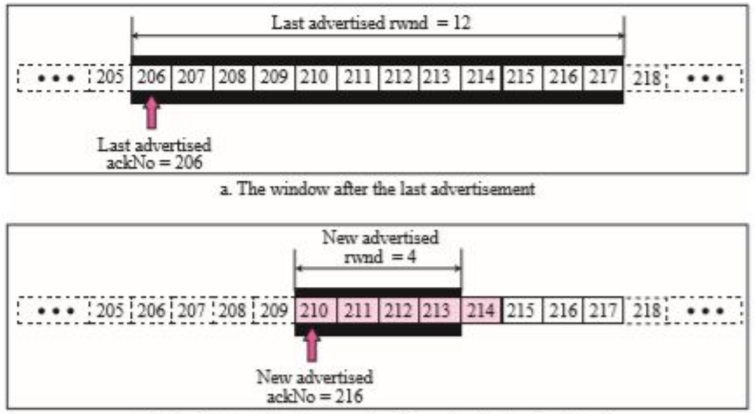

Client
Server
SYN
seqNo: 100
rwnd = 800
101
901
SYN + ACK
seqNo: 1000
ackNo: 101
rwnd: 800
Receive window is set
Size = 800
101
901
Send window is set
ACK
ackNo: 1001
Size = 800
101
301
901
Data
seqNo: 101
Data: 200 bytes
rwnd = 600
101
301
901
Sender sends 200 bytes
ACK
ackNo: 301
rwnd: 600
200 bytes received, window closes.
Size = 600
301
901
Bytes acknowledged, window closes.
Size = 600
301
601
901
Data
seqNo: 301
Data: 300 bytes
Sender sends 300 bytes.
rwnd = 400
201
601
1001
ACK
ackNo: 601
rwnd: 400
300 bytes received, 100 bytes consumed.
Size = 400
601
1001
rwnd = 600
401
601
1201
Window closes and opens
ACK
ackNo: 601
rwnd: 600
200 bytes consumed, window opens
Size = 600
601
1201
Window opens.

Shrinking of Windows

- the receive window cannot shrink. But the send window can shrink if the receiver defines a value for rwnd that results in shrinking the window. Some implements doesn't allow shrinking of the send window

$$\text{new ackNo} + \text{new rwnd} \geq \text{last ackNo} + \text{last rwnd}$$

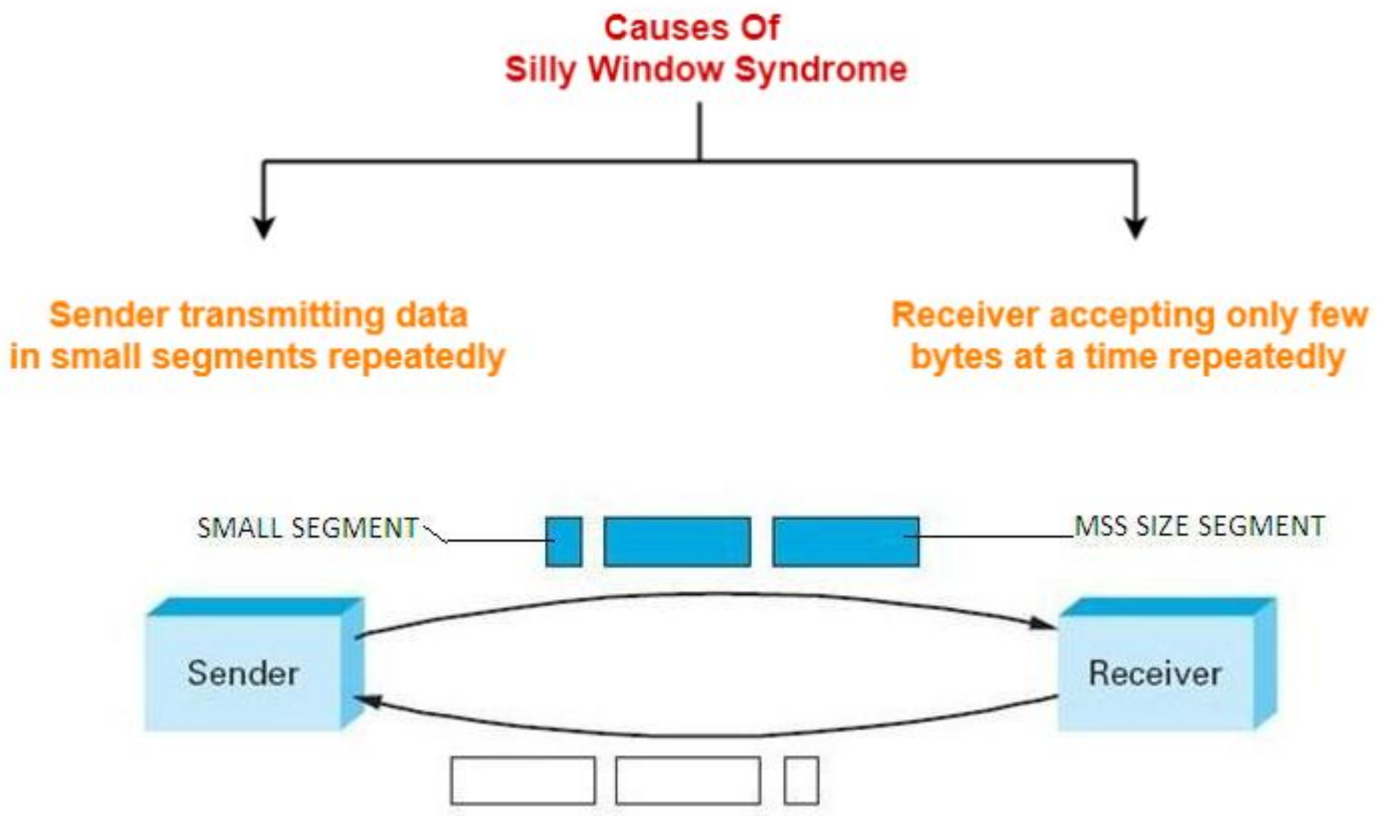
- The left side of the inequality represents the new position of the right wall with respect to the sequence number space; the right side shows the old position of the right wall


Last advertised rwnd = 12
205
206
207
208
209
210
211
212
213
214
215
216
217
218
Last advertised ackNo = 206
New advertised rwnd = 4
New advertised ackNo = 216
a. The window after the last advertisement

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

- Window shutdown
    - If there has to be shrinking then…
    - the receiver can temporarily shut down the window by ***sending a rwnd of 0.***
    - This can happen if for some reason the ***receiver does not want to receive any data from the sender for a while***.
    - In this case, the ***sender*** does not actually shrink the size of the window, but ***stops sending data*** until a new data has arrived

Silly Window Syndrome


Causes Of
Silly Window Syndrome
Sender transmitting data in small segments repeatedly
Receiver accepting only few bytes at a time repeatedly
SMALL SEGMENT
MSS SIZE SEGMENT
Sender
Receiver

- 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.
- For example, if TCP sends segments containing only 1 byte of data, it means that a 41-byte datagram (20 bytes of TCP header and 20 bytes of IP header) transfers only 1 byte of user data. Here the overhead is 41/1, which indicates that we are using the 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***
  - if it is serving an application program that creates data slowly, for example, 1 byte at a time.
  - The application program writes 1 byte at a time into the buffer of the sending TCP.
  - 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 sent.
  - How to prevent the sending TCP from sending the data byte by byte.

    Solution : The sending TCP must be forced to wait and collect data to send in a larger block.
  - Problem ????

    How long should the sending TCP wait? If it waits too long, it may delay the process. If it does not wait long enough, it may end up sending small segments

    Solution : ***Nagle's algorithm***

Nagle's Algorithm

1. Sending TCP sends the first piece of data it receives from the sending application program even if it is only 1 byte.
2. Later, the sending TCP accumulates data in the output buffer and waits until either the receiving TCP sends an acknowledgment or until enough data has accumulated to fill a maximum-size segment (MSS).

   At this time, the sending TCP can send the segment.

3. Step 2 is repeated for the rest of the transmission.

Nagle’s algorithm is simple

- it 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 the maximum segment size).
- Nagle's algorithm has the advantage of increasing the efficiency of the network, but it has the disadvantage of increasing the delay. However, disabling Nagle's algorithm increases the network congestion, which could cause overall throughput degradation.

Syndrome Created by the Receive

- Receiving program that consumes data slowly, like 1 byte at a time.
- Suppose that the sending application program creates data in blocks of 1 kilobyte, but the receiving application program consumes data 1 byte at a time. Also suppose that the input buffer of the receiving TCP is 4 kilobytes. The sender sends the first 4 kilobytes of data. The receiver stores it in its buffer. Now its buffer is full. It advertises a window size of zero, which means the sender should stop sending data. The receiving application reads the first byte of data from the input buffer of the receiving TCP. Now there is 1 byte of space in the incoming buffer. The receiving TCP announces a window size of 1 byte, which means that the sending TCP, which is eagerly waiting to send data, takes this acknowledgment as good news and sends a segment carrying only 1 byte of data. The procedure will continue. One byte of data is consumed and a segment carrying 1 byte of data is sent.

Problem??? an efficiency problem and the silly window syndrome.

Solution :

- ***Clark's Solution***
- ***Delayed Acknowledgment***

Clark's Solution

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

Delayed Acknowledgment

- delay sending the acknowledgment.
- Segment's arrives, it is not acknowledged immediately.
- Receiver waits until there is a decent amount of space in its incoming buffer before acknowledging the arrived segments.
- ***Advantage*****:** it reduces traffic. The receiver does not have to acknowledge each segment
- ***Disadvantage:*** delayed acknowledgment may result in the sender unnecessarily retransmitting the unacknowledged segments
- Timer can be set to balance this situation

ERROR CONTROL

- TCP is a **reliable** transport layer protocol
- 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.***
- Three simple tools:
  - Checksum
  - Acknowledgment
  - Retransmission

Checksum

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

Original Data

10011001	11100010	00100100	10000100
1	2	3	4

k=4, m=8

Sender

```
1            1 0 0 1 1 0 0 1
2            1 1 1 0 0 0 1 0
            -----------------
           (1)0 1 1 1 1 0 1 1
                            1
            -----------------
             0 1 1 1 1 1 0 0
3            0 0 1 0 0 1 0 0
            -----------------
             1 0 1 0 0 0 0 0
4            1 0 0 0 0 1 0 0
            -----------------
           (1)0 0 1 0 0 1 0 0
                            1
            -----------------
Sum:         0 0 1 0 0 1 0 1
            -----------------
CheckSum:    1 1 0 1 1 0 1 0
```

Reciever

```
1              1 0 0 1 1 0 0 1
2              1 1 1 0 0 0 1 0
              -----------------
             (1)0 1 1 1 1 0 1 1
                              1
              -----------------
               0 1 1 1 1 1 0 0
3              0 0 1 0 0 1 0 0
              -----------------
               1 0 1 0 0 0 0 0
4              1 0 0 0 0 1 0 0
              -----------------
             (1)0 0 1 0 0 1 0 0
                              1
              -----------------
               0 0 1 0 0 1 0 1
               1 1 0 1 1 0 1 0
              -----------------
Sum:           1 1 1 1 1 1 1 1
Complement:    0 0 0 0 0 0 0 0
```

Conclusion: Accept Data

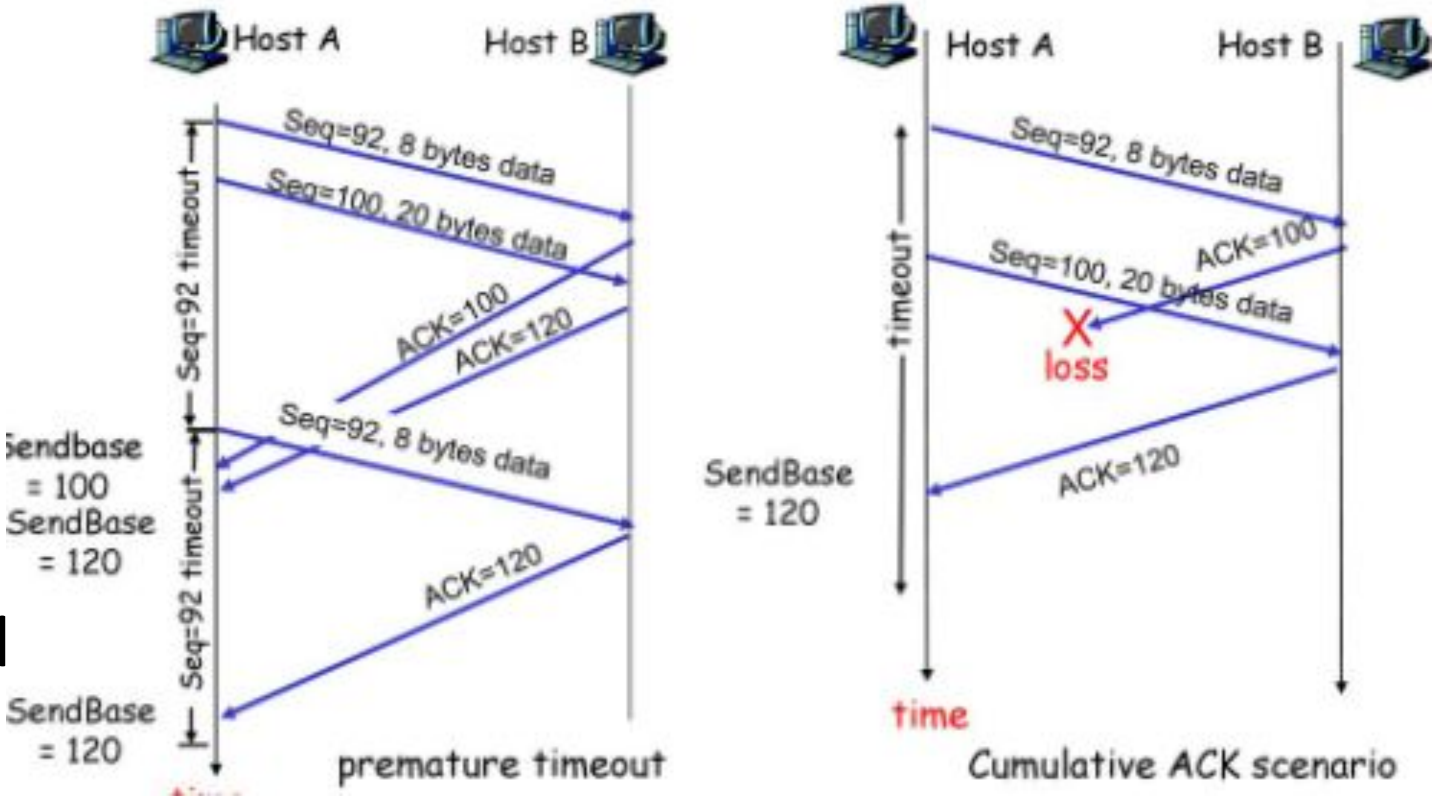
Acknowledgment

- TCP uses acknowledgments to confirm the receipt of data segments.
- Control segments that carry no data, but ***consume a sequence*** number, are also acknowledged.
- ACK segments are never acknowledged.
- There are different ways of ***implementing acknowledgment***
  - cumulative acknowledgment
  - selective acknowledgment.

1. Cumulative Acknowledgment (ACK)

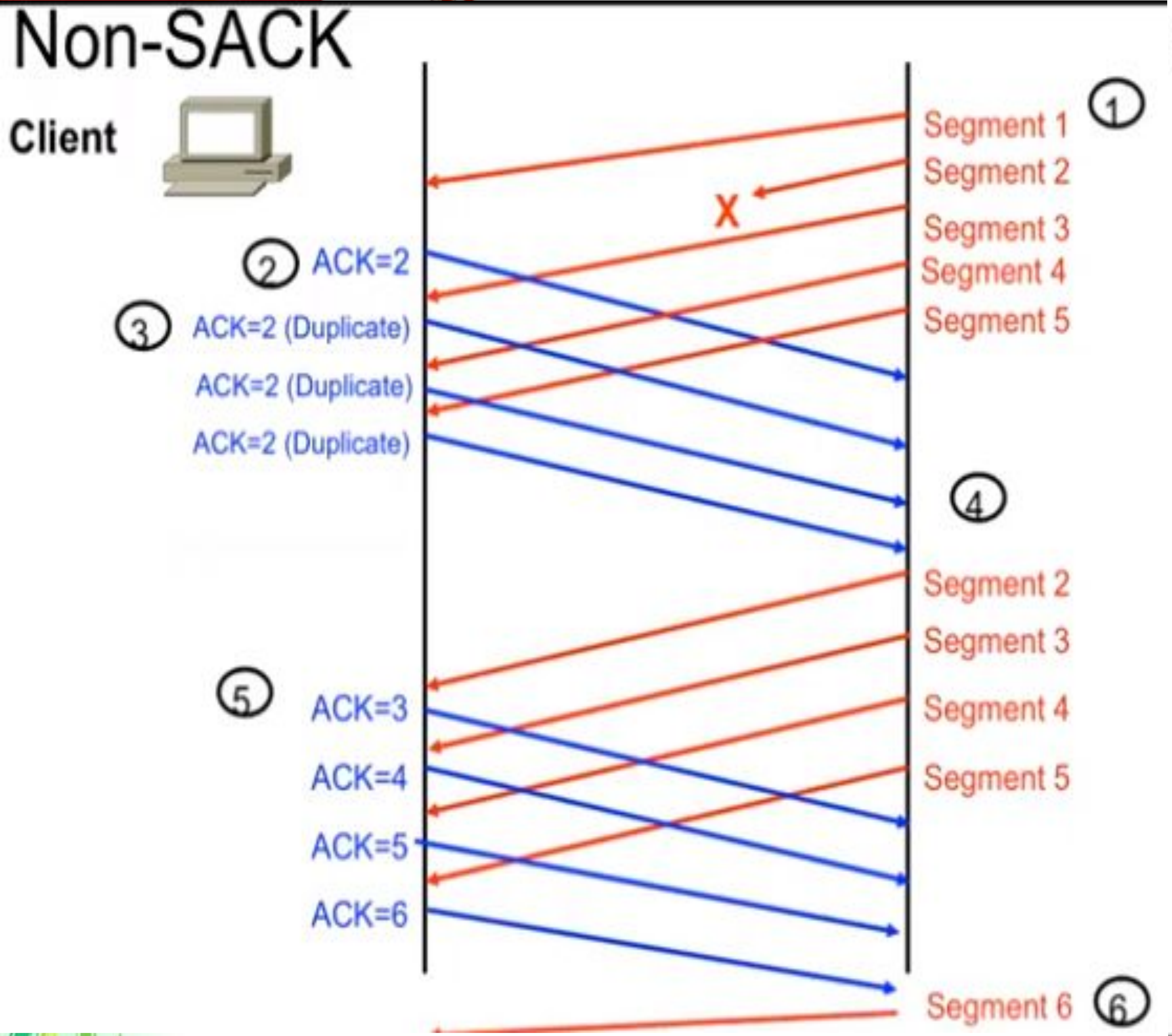
- TCP was originally designed to acknowledge receipt of segments cumulatively.
- The receiver advertises the next byte it expects to receive, ignoring all segments received and stored out of order.
- Also referred to as ***positive cumulative acknowledgment*** or ACK.
- The word “positive” indicates that no feedback is provided for discarded, lost, or duplicate segments.
- The ***32-bit*** ACK field in the TCP header is used for cumulative acknowledgments and its value is valid only when the ***ACK flag bit*** is set to 1.

- TCP can't differentiate between packet loss due to congestion or ACK lost
- Cumulative ACK help to avoid retransmission due to time outs
- With Cumulative ACK Received data buffered, and ACK is incremented

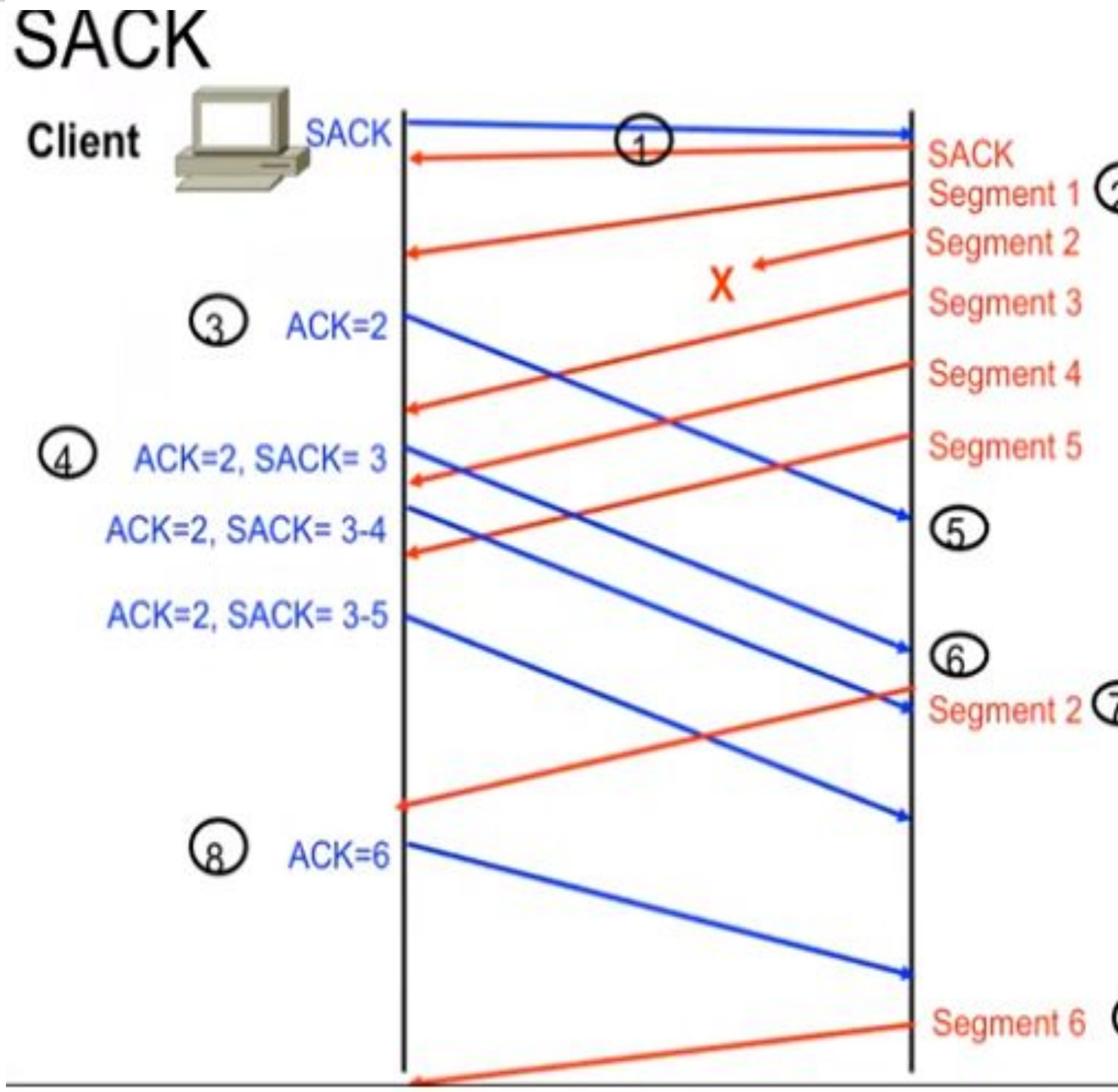


2. Selective Acknowledgment (SACK)

- SACK does not replace ACK
- Reports additional information to the sender.
- A SACK reports a block of data that ***is out of order***, and also a block of segments that is ***duplicated***
- Since no provision in the TCP header for adding this type of information, SACK is implemented as an ***option*** at the end of the TCP header.
- it is an optional feature and is negotiated during 3 way handshake



Non-SACK
Client
Segment 1
Segment 2
X
Segment 3
Segment 4
Segment 5
ACK=2
ACK=2 (Duplicate)
ACK=2 (Duplicate)
ACK=2 (Duplicate)
Segment 2
Segment 3
Segment 4
Segment 5
ACK=3
ACK=4
ACK=5
ACK=6
Segment 6
1
2
3
4
5
6



SACK
Client
SACK
SACK
Segment 1
Segment 2
X
Segment 3
Segment 4
Segment 5
ACK=2
ACK=2, SACK= 3
ACK=2, SACK= 3-4
ACK=2, SACK= 3-5
Segment 2
ACK=6
Segment 6
1
3
4
5
6
8

Summarizing how Acknowledgments are Generating

1. When end A sends a data segment to end B, it must include (***piggyback***) an acknowledgment that gives the next sequence number it expects to receive. This rule decreases the number of segments needed and therefore reduces traffic.

   When the receiver has no data to send and it receives an in-order segment (with expected sequence number) and the previous segment has already been acknowledged, the receiver delays sending an ACK segment until another segment arrives or until a period of time (normally 500 ms) has passed. In other words, the receiver needs to ***delay sending an ACK segment if there is only one outstanding in-order segment***. This rule reduces ACK segment traffic.

3. When a segment arrives with a sequence number that is expected by the receiver, and the previous in-order segment has not been acknowledged, the receiver immediately sends an ACK segment. In other words, there should ***not be more than two in-order unacknowledged segments at any time***. This **prevents the unnecessary retransmission** of segments that may create ***congestion*** in the network.

4. When a segment arrives with an ***out-of-order sequence number*** that is higher than expected, the receiver immediately sends an ACK segment announcing the sequence number of the next expected segment. This leads to the fast ***retransmission of missing segments***.

5. When a ***missing segment*** arrives, the receiver sends an ACK segment to announce the next sequence number expected. This informs the receiver that segments reported missing have been received.

6. If a ***duplicate segment*** arrives, the receiver discards the segment, but immediately sends an acknowledgment indicating the next in-order segment expected. This solves some problems when an ACK segment itself is lost.

Retransmission

- The heart of the error control mechanism is the retransmission of segments
- When a segment is sent, it is stored in a **queue until it is acknowledged.**
- When the retransmission **timer expires or** when the sender receives **three duplicate ACK**s for the first segment in the queue, that segment is retransmitted.

Retransmission after RTO

- The sending TCP maintains one retransmission time-out (RTO) for each connection.
- When the timer matures, TCP sends the segment in the front of the queue (the segment with the smallest sequence number) and restarts the timer.

***Retransmission* after Three Duplicate ACK Segments**

- The previous rule about retransmission of a segment is sufficient if the value of RTO is not large.
- To improve throughput by allowing sender to retransmit sooner than waiting for a time out, most implementations today follow the three duplicate ACKs rule and retransmit the missing segment immediately.
- Also referred to as ***fast retransmission or Reno***.
- In this version, if three duplicate acknowledgments (i.e., an original ACK plus three exactly identical copies) arrives for a segment, the next segment is retransmitted without waiting for the time-out.

CONGESTION CONTROL

- When too many packets are driven on the same link
- The queue overflows
- Packets get dropped

Network is congested!

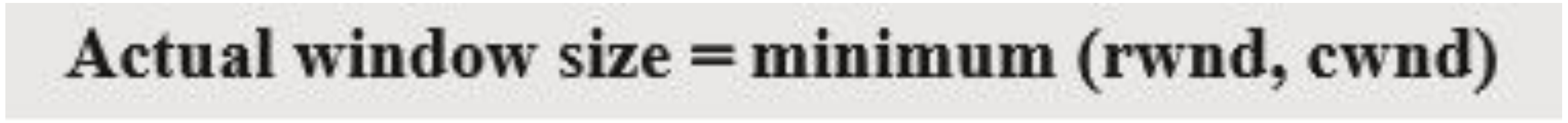
Network should provide a congestion control mechanism to deal with such a situation, TCP uses a

- congestion window
- congestion policy (Control)
    - that avoid congestion and detect and reduce the after effects of congestion

- Congestion control and Resource Allocation
    - Two sides of the same coin
- If the network takes active role in allocating resources
    - The congestion may be avoided
    - No need for congestion control
- Allocating resources with any precision is difficult
    - Resources are distributed throughout the network
- Then recover from the congestion when it occurs
    - Easier approach but it can be disruptive because many packets many be discarded by the network before congestions can be controlled

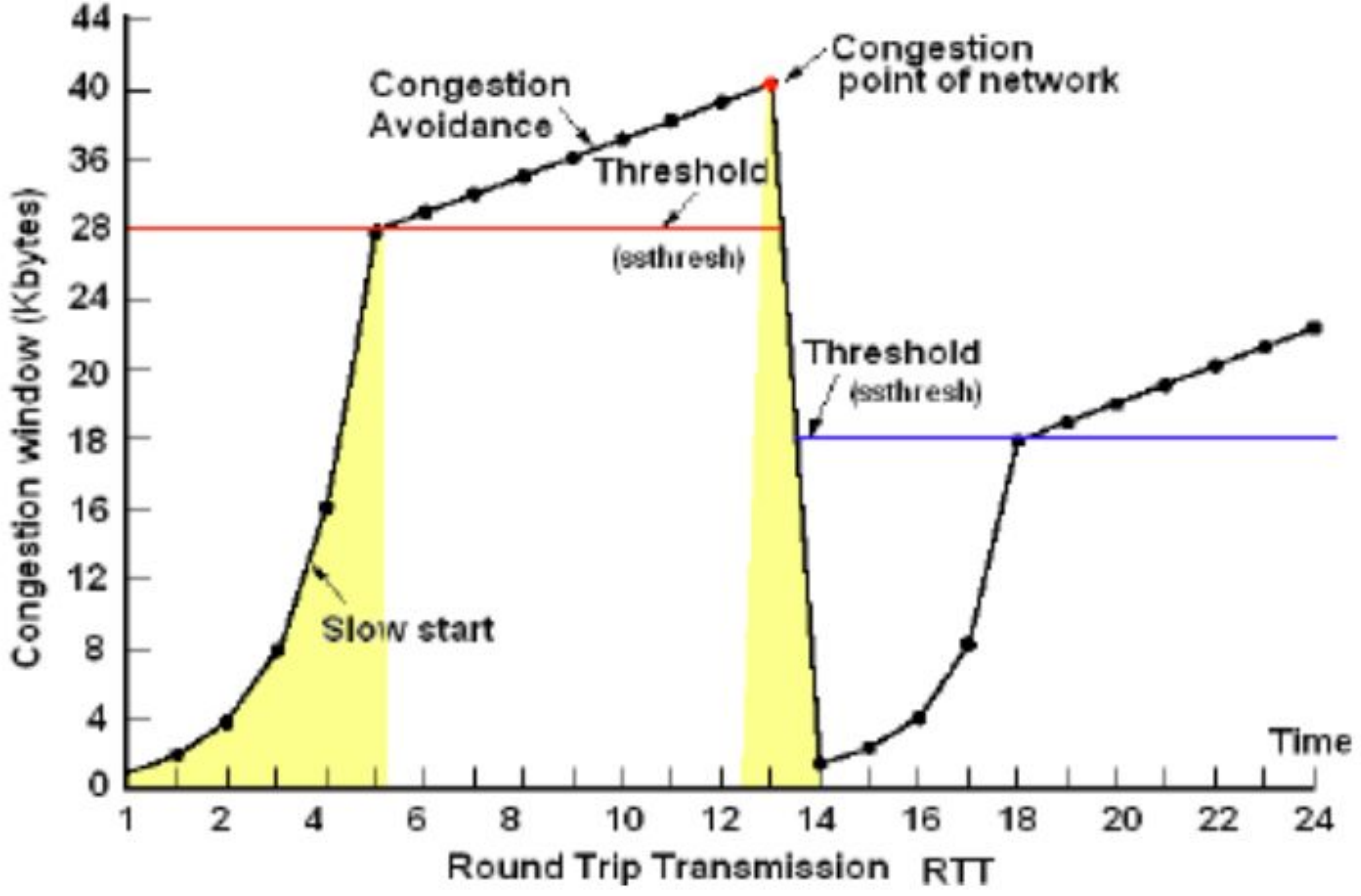
Congestion Window

- Only the receiver can dictate to the sender the size of the sender’s window! Not really
- Another entity that is affected is the network.
- If the network cannot deliver the data as fast as it is created by the sender, it must tell the sender to slow down.
- In addition to the receiver, the network is a second entity that determines the size of the sender’s window.
- The sender has two pieces of information:
  - receiver-advertised window size and
  - congestion window size.
- The actual size of the window is the minimum of these two.



Actual window size = minimum (rwnd, cwnd)

Congestion Policy



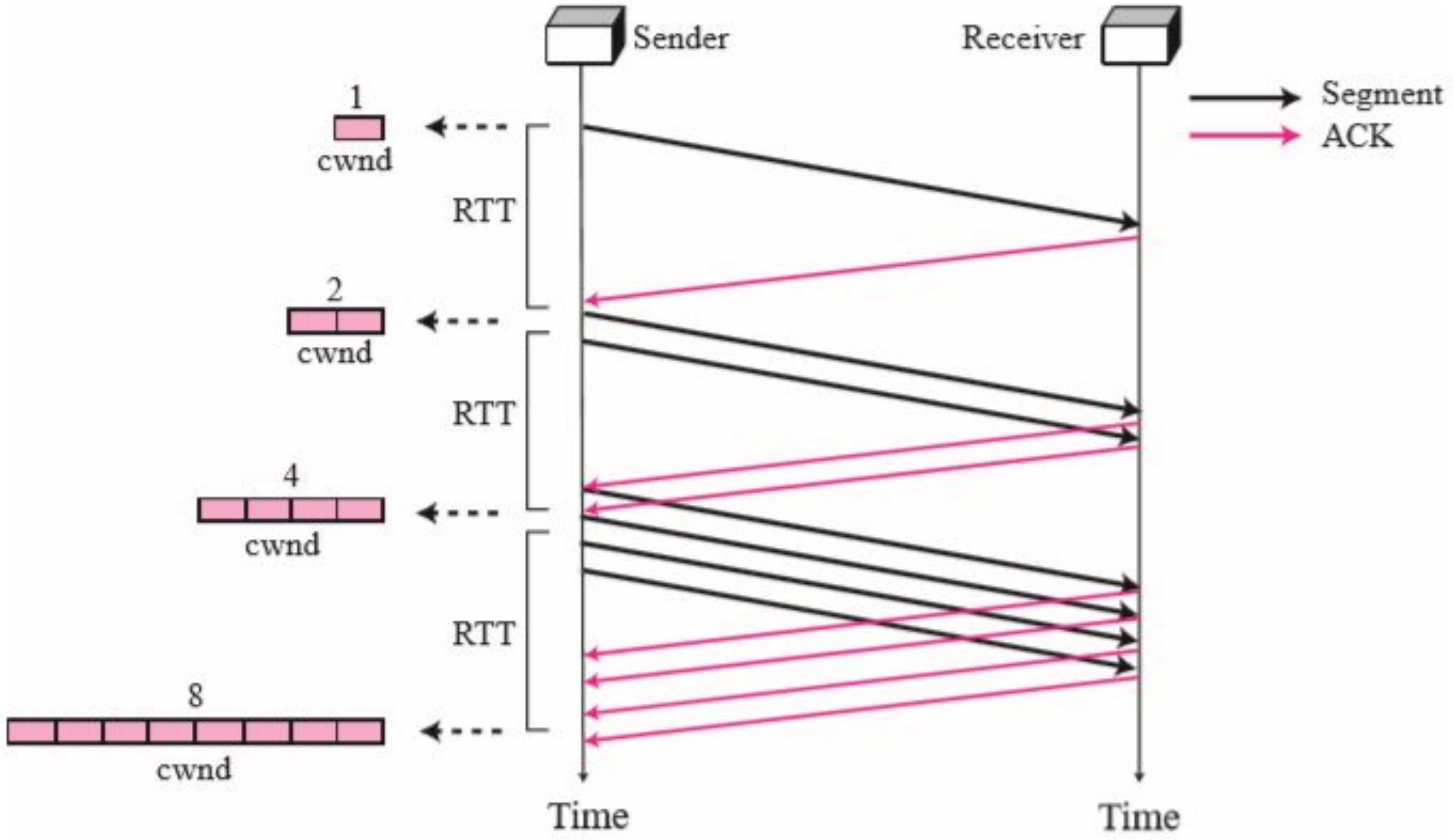
Congestion Policy

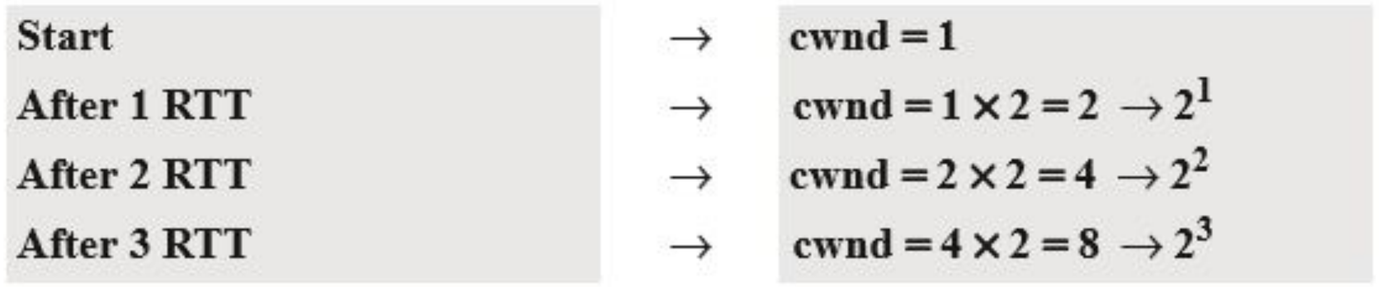
- Three phases:
  - Slow start
  - congestion avoidance
  - congestion detection.

 Slow Start: Exponential Increase

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 is reduced. Finally if ever congestion is detected, the sender goes back to the slow start or congestion avoidance phase

Slow start, exponential increase





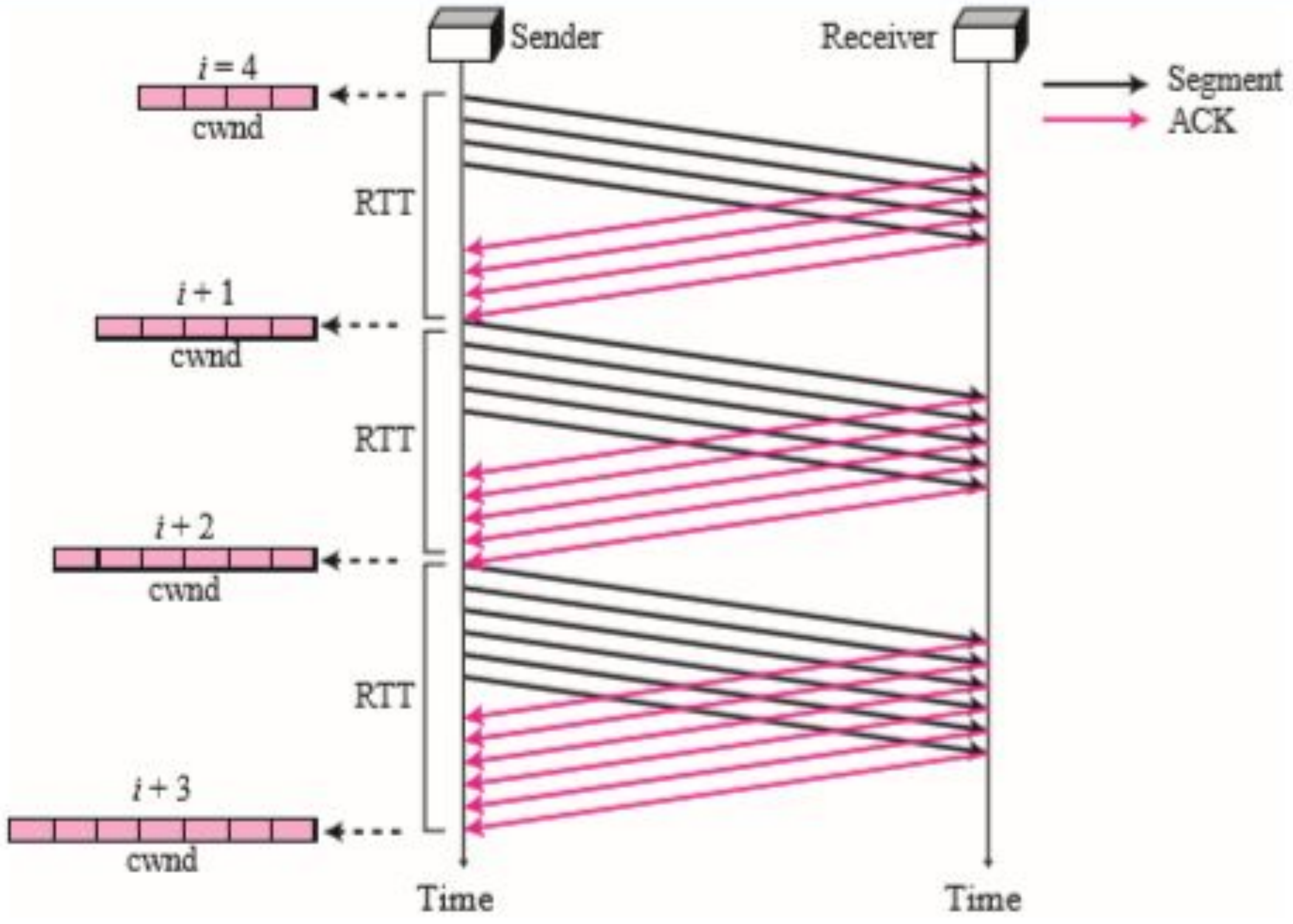
Start	$\rightarrow$	$cwnd = 1$
After 1 RTT	$\rightarrow$	$cwnd = 1 \times 2 = 2 \rightarrow 2^1$
After 2 RTT	$\rightarrow$	$cwnd = 2 \times 2 = 4 \rightarrow 2^2$
After 3 RTT	$\rightarrow$	$cwnd = 4 \times 2 = 8 \rightarrow 2^3$

- starts with one maximum segment size (MSS)
- size of the window increases one MSS each time one acknowledgement arrives
- Slow start cannot continue indefinitely.
- There is a threshold to stop this phase.
- The sender keeps track of a variable named ***ssthresh*** (slow start threshold).
- When the size of window in bytes reaches this threshold, slow start stops and the next phase starts.

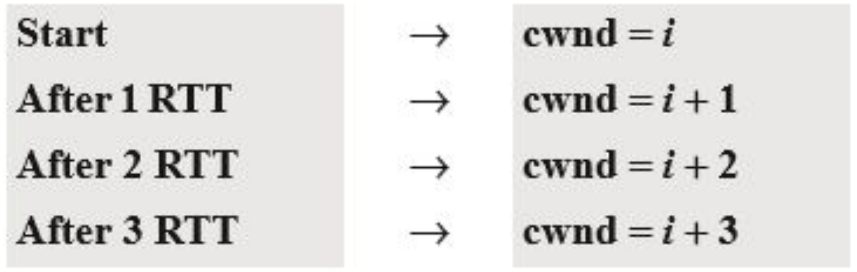
Congestion Avoidance: Additive Increase

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

Congestion avoidance, additive increase



- each time the whole “window” of segments is acknowledged, the size of the congestion window is increased by one.
- A window is the number of segments transmitted during RTT.
- The increase is based on RTT, not on the number of arrived ACKs as in slow start
- Start with slow start, after the sender has received acknowledgments for a complete window-size of segments, the size of the window is increased one segment



Start	$\rightarrow$	$\text{cwnd} = i$
After 1 RTT	$\rightarrow$	$\text{cwnd} = i + 1$
After 2 RTT	$\rightarrow$	$\text{cwnd} = i + 2$
After 3 RTT	$\rightarrow$	$\text{cwnd} = i + 3$

Congestion Detection: Multiplicative Decrease

- If congestion occurs, the congestion window size must be decreased.
- The only way a sender can guess that congestion has occurred is the need to retransmit a segment. This is a 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 cases: when the RTO timer times out or when three duplicate ACKs are received.
- In both cases, the size of the threshold is dropped to half (multiplicative decrease). Most TCP implementations have two reactions:

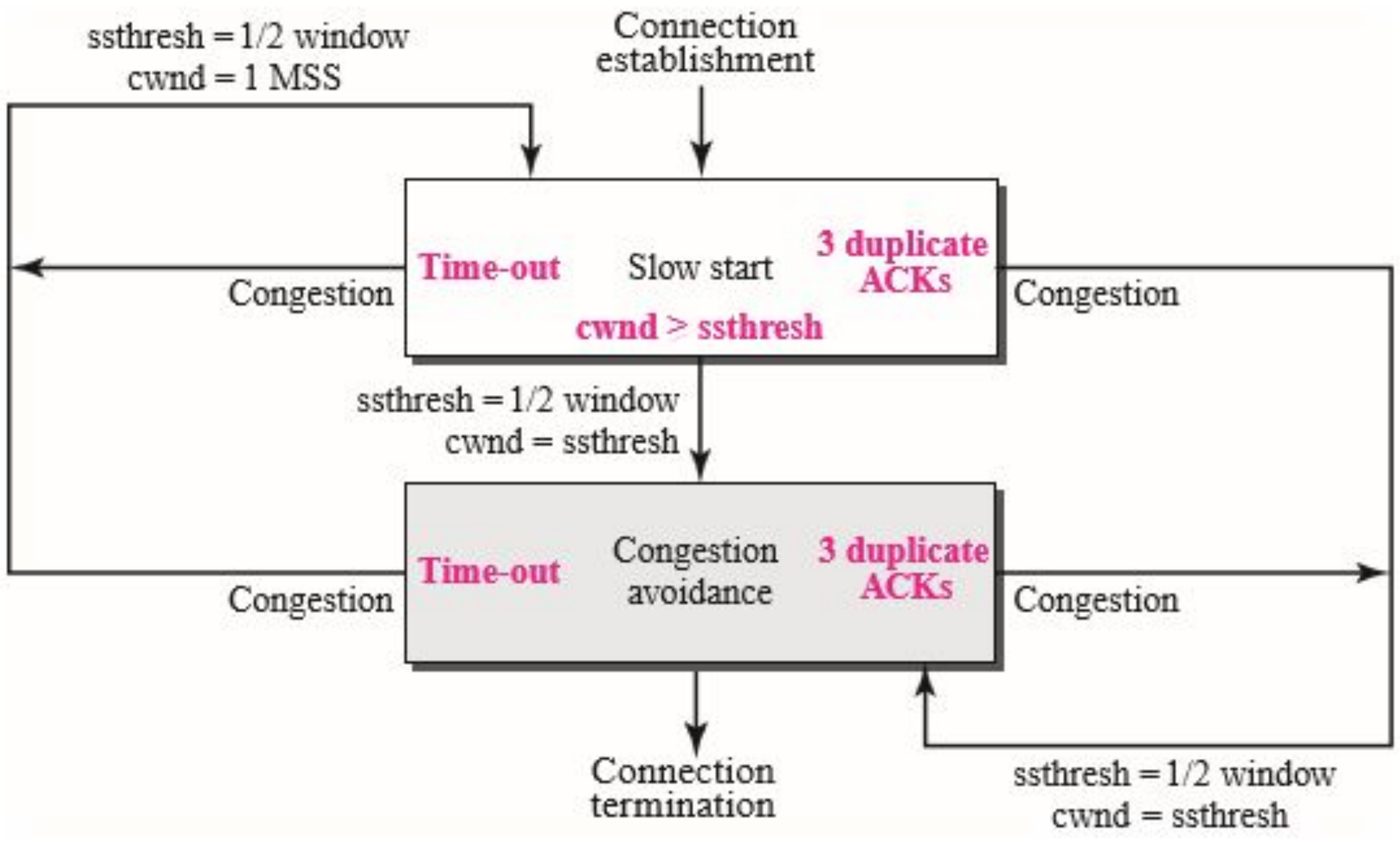
1. If a time-out occurs, there is a stronger possibility of congestion;

- a segment has probably been dropped in the network and there is no news about the following sent segments.
- In this case TCP reacts strongly:
  a. It sets the value of the threshold to half of the current window size.
  b. It reduces cwnd back to one segment.
  c. It starts the slow start phase again.

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

a. It sets the value of the threshold to half of the current window size.
b. It sets cwnd to the value of the threshold (some implementations add three segment sizes to the threshold).
c. It starts the congestion avoidance phase.

TCP Congestion Policy Summary



TCP Timers

- 4 types of timers
  - TIME-WAIT Timer
  - Retransmission Timer
  - Persistence Time
  - Keepalive Timer

 TIME WAIT Timer

  - ➢ This timer during connection termination
  - ➢ The timer starts after sending the last Ack for 2nd FIN and closing the connection.

- **Retransmission Timer**

  To retransmit lost segments, TCP employs one retransmission timer (for the whole connection period) that handles the retransmission time-out (RTO), the waiting time for an acknowledgment of a segment.

- **Persistence Timer**
  - To deal with a ***zero-window-size*** advertisement, TCP needs another timer.
  - If the receiving TCP announces a window size of zero, the sending TCP stops transmitting segments until the receiving TCP sends an ACK segment announcing a nonzero window size.
  - This ACK segment can be lost. (ACK segments are not acknowledged nor retransmitted in TCP)

- If this acknowledgment is lost, the receiving TCP thinks that it has done its job and waits for the sending TCP to send more segments.
- There is **no retransmission timer for a segment containing only an acknowledgment.**
- The sending TCP has not received an acknowledgment and waits for the other TCP to send an acknowledgment advertising the size of the window.
- Both TCPs might continue to wait for each other forever (a deadlock).
- 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 sends a special segment called a ***probe***.
- This segment contains only 1 byte of new data.
- It has a sequence number, but its sequence number is never acknowledged
- 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
- After that the sender sends one probe segment every 60 s until the window is reopened.

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.
- Left so……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 time-out is usually 2 hours. If the server does not hear from the client after 2 hours, it sends a probe segment.
- If there is no response after 10 probes, each of which is 75 s apart, it assumes that the client is down and terminates the connection.