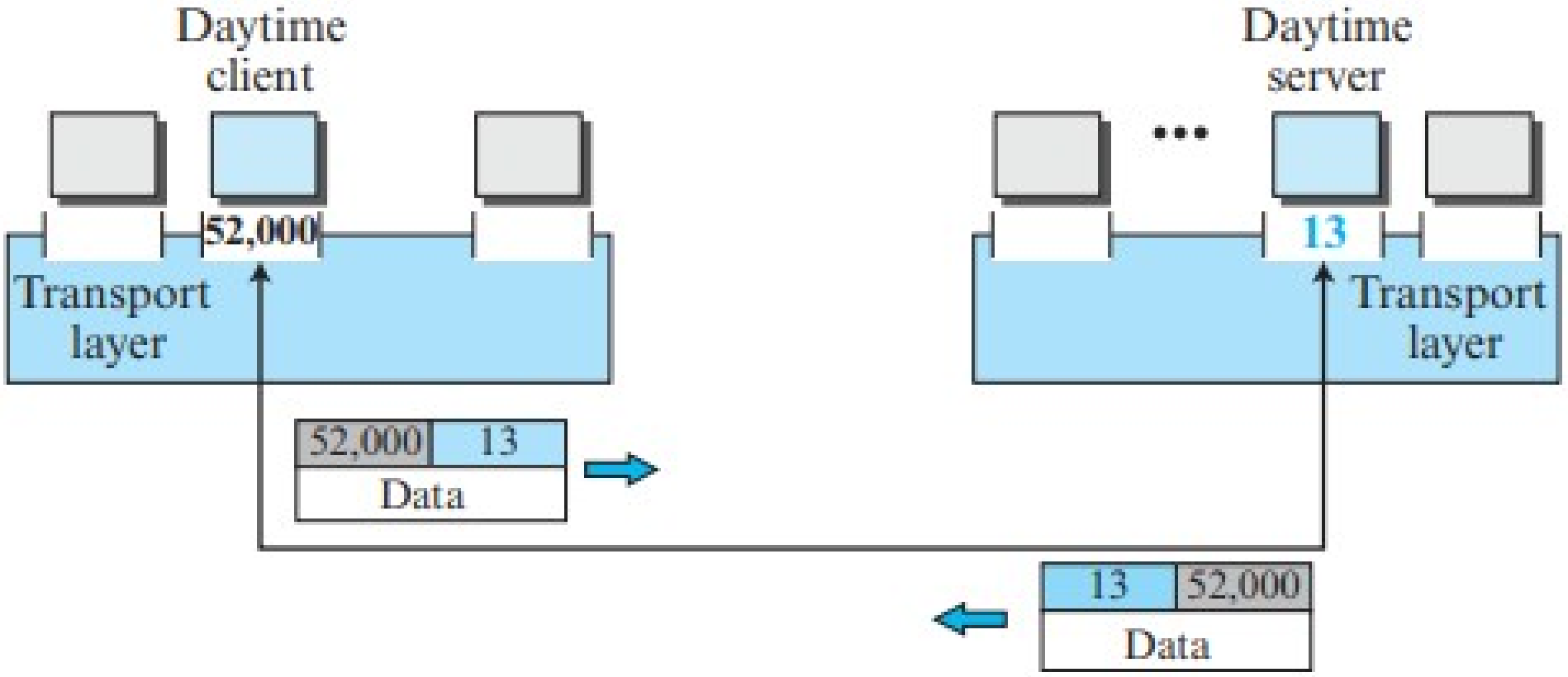
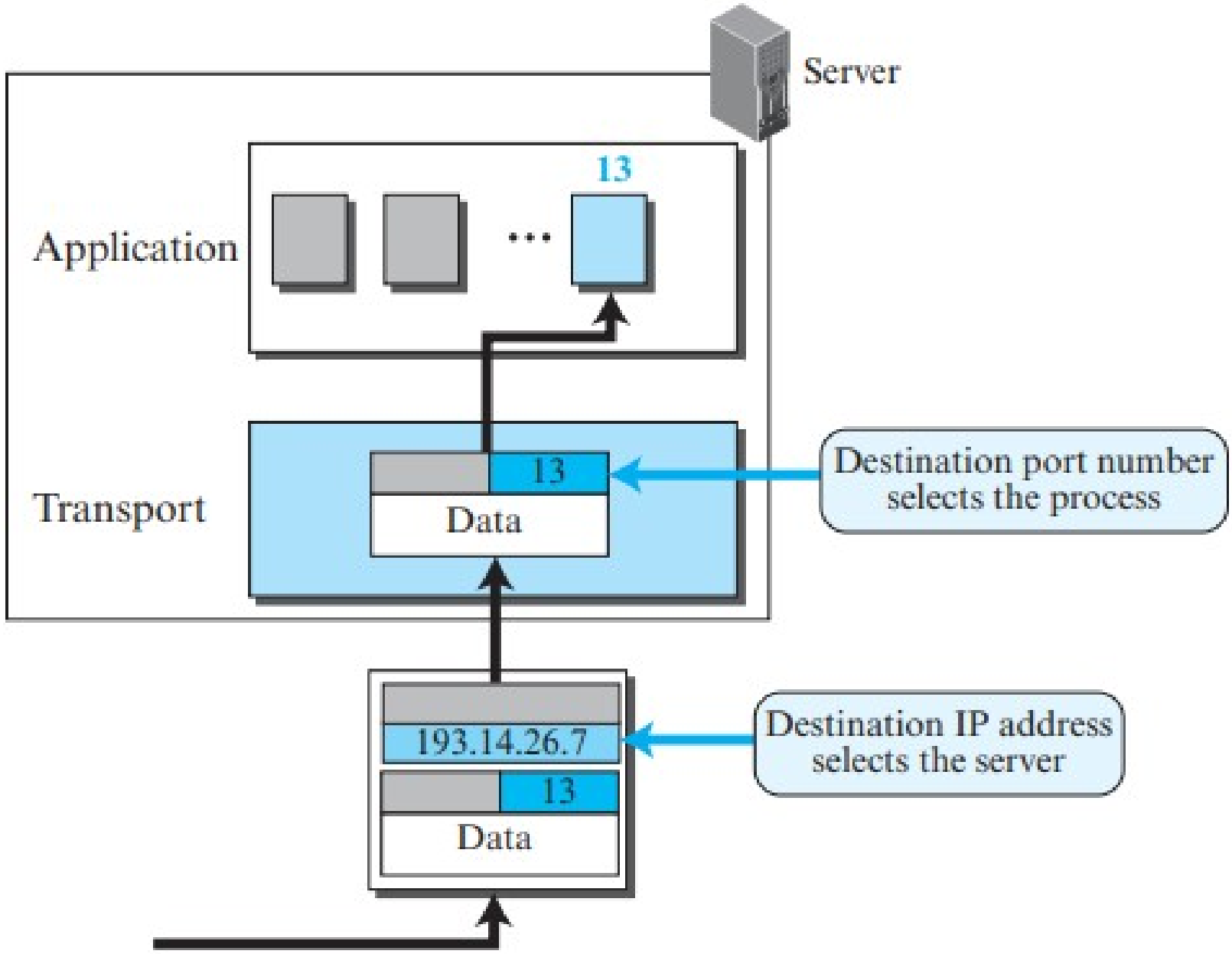
ADDRESSING: PORT NUMBERS

- Process-to-process communication often happens using the client-server approach, where one program (client) on a local computer needs services from another program (server) on a remote computer.
- Both the client and server programs have names. For example, to fetch the time from a remote machine, you need a daytime client program on your computer and a daytime server programon the remote computer.
- To ensure communication, we define the local and remote hosts using IP addresses and the processes using port numbers. Port numbers are 16-bit addresses for programs and range from 0 to 65535 in the TCP/IP protocol.
- Clients typically use **ephemeral** (short-lived) port numbers, usually greater than 1023. Servers use well-known port numbers, making it easy for clients to find them. For example, a daytime client might use a temporary port like 52,000, while the daytime server always uses the permanent port number 13. This way, clients know where to find the servers they need.

ADDRESSING: ASSIGNING PORT NUMBERS




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

ICANN Ranges

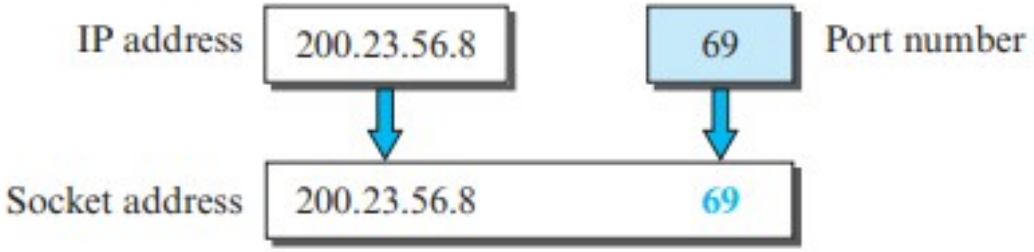
- ICANN (Internet Corporation for Assigned Names and Numbers) has divided the port numbers into three ranges -
  - **Well - Known ports:** The port numbers ranging from 0 to 1023 are called well-known port numbers and are restricted, which means that they are reserved for use by well-known application protocols such as HTTP (which uses port number 80) and SMTP(which uses port number 25).
  - **Registered ports:** The ports ranging from 1024 to 49151 are not assigned or controlled by ICANN. These can be registered for specific services or applications upon request.
  - **Dynamic or Private Ports:** The ports ranging from 49,152 to 65,535 are neither controlled nor registered. They can be used as temporary or private port numbers.

Table 4.1 Well-known port numbers used by UDP

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

Socket Address

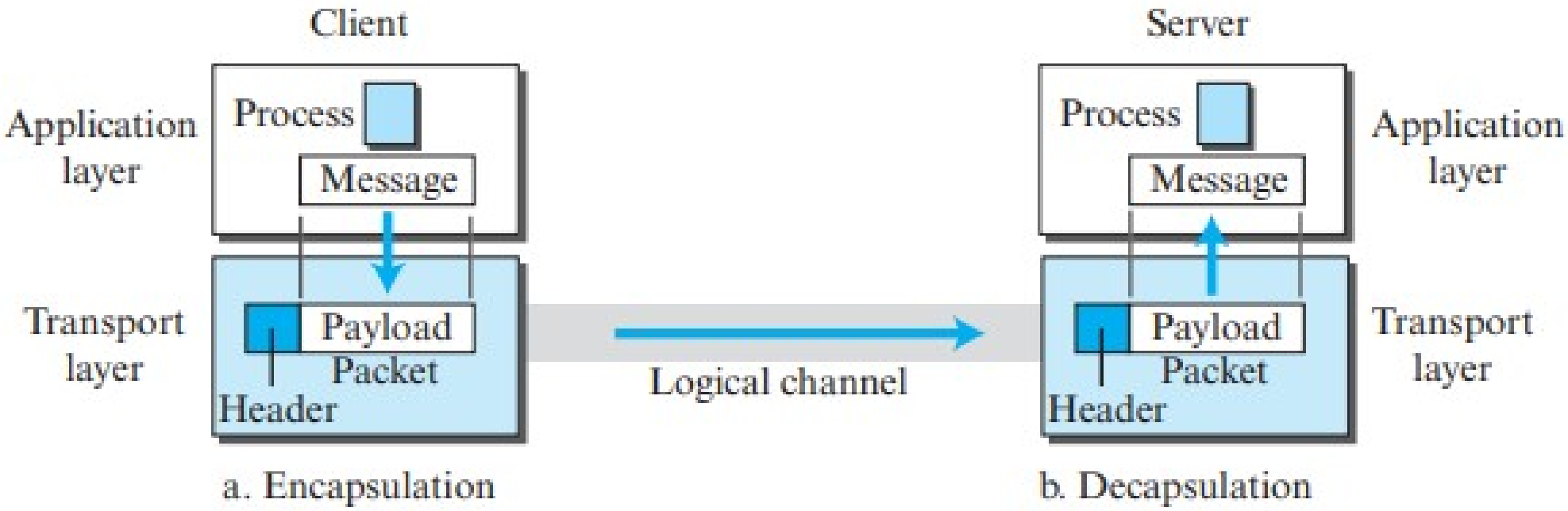
- A transport-layer protocol in the TCP suite needs both the IP address and the port number, ateach end, to make a connection. The combination of an IP address and a port number is called a socket address.
- The client socket address defines the client process uniquely just as the server socket address defines the server process uniquely.
- To use the services of the transport layer in the Internet, we need a pair of socket addresses: the client socket address and the server socket address. These four pieces of information are part of the network-layer packet header and the transport-layer packet header. The first header contains the IP addresses; the second header contains the port numbers.



Encapsulation and Decapsulation

- To send a message from one process to another, the transport-layer protocol encapsulates and decapsulates messages.
- Encapsulation happens at the sender site. When a process has a message to send, it passes the message to the transport layer along with a pair of socket addresses and some other pieces of information, which depend on the transport-layer protocol. The transport layer receives the data and adds the transport-layer header.
- Decapsulation happens at the receiver site. When the message arrives at the destination transport layer, the header is dropped and the transport layer delivers the message to the process running at the application layer. The sender socket address is passed to the process in case it needs to respond to the message received.

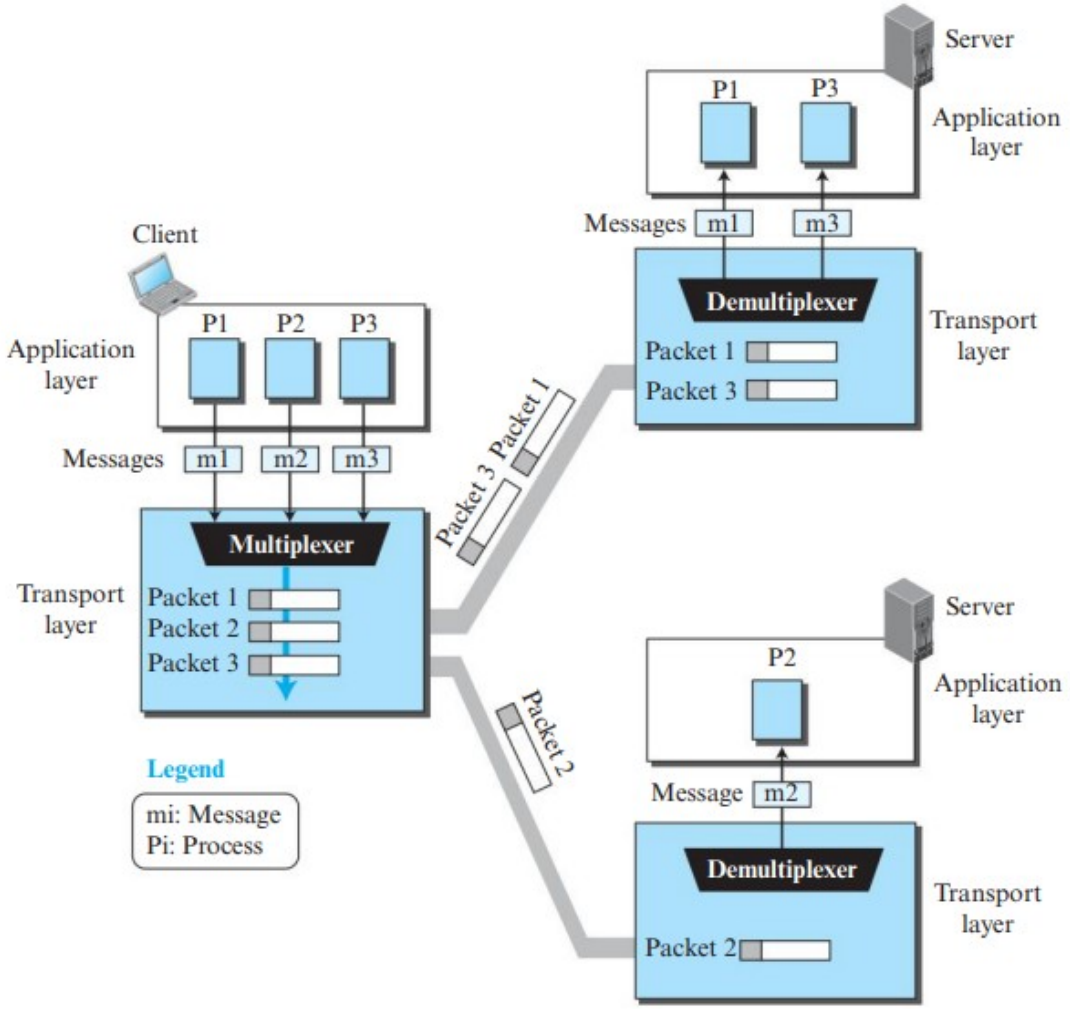
Encapsulation and Decapsulation



Multiplexing and Demultiplexing

- Multiplexing means combining multiple items into one, while demultiplexing means separating items from one source to multiple destinations.
- In the context of transport layer communication, the source transport layer performs multiplexing by taking messages from different processes and creating packets for them. These packets are sent over the network.
- At the destination, the transport layer performs demultiplexing, which means it takes incoming packets, identifies the destination processes, and delivers the messages to the right processes.

MULTIPLEXING AND DEMULTIPLEXING



Transport Layer Protocols

TCP Protocol

- Connection-oriented Protocol
- **Reliable data transfer** - Ensures that data is delivered accurately and in the correct order through error-checking mechanisms like acknowledgments, retransmissions, and timeouts.
- **Provides flow control** - Uses techniques like windowing to control the flow of data between sender and receiver, preventing one from overwhelming the other.
- **Congestion Control** - Avoids network congestion by adjusting the rate of data transmission.
- Used in Web browsing (HTTP/HTTPS), Email (SMTP), etc.

UDP Protocol

- Connection-less Protocol
- **Unreliable -** Does not guarantee the delivery of data, and there's no retransmission in case of packet loss or errors.
- **No flow or congestion control -** The sender can send data at any rate irrespective of receiver's capability to handle the data.
- Less overhead compared to TCP therefore faster in data transmission.
- Used in real-time applications like video streaming, online gaming, DNS queries, etc.

User Datagram Protocol (UDP)

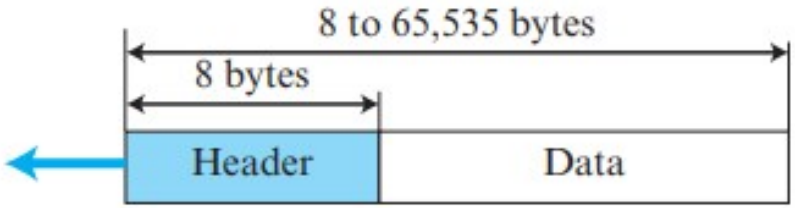
- The User Datagram Protocol (UDP) is a connectionless, unreliable transport protocol.
- It does not add anything to the services of network layer except for providing process-to-process communication while network layer only provides host-to-host communication.
- If UDP is so powerless, why would a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

WHY USE UDP?

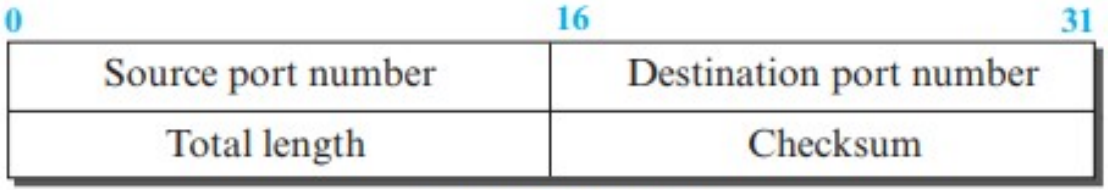
- **Finer application level control over what data is sent and when -** Under UDP, as soon as an application process passes data to UDP, UDP will package the data inside a UDP segment and immediately pass the segment to the network layer. TCP on other hand will worry about congestion control and reliable delivary adding more delays to packet transmission.
- **No connection establishment -** TCP uses a three-way handshake before it starts to transfer data. UDP just blasts away without any formal preliminaries. Thus UDP does not introduce any delay to establish a connection. This is probably the principal reason why DNS runs over UDP rather than TCP—DNS would be much slower if it ran over TCP.
- **No connection state -** . TCP maintains connection state in the end systems. This connection state includes receive and send buffers, congestion-control parameters, and sequence and acknowledgment number parameters. . UDP, on the other hand, does not maintain connection state and does not track any of these parameters. For this reason, a server devoted to a particular application can typically support many more active clients when the application runs over UDP rather than TCP.
- **Small packet header overhead -** The TCP segment has 20 bytes of header overhead in every segment, whereas UDP has only 8 bytes of overhead.

User Datagram

- UDP packets are called **User Datagrams** and have a fixed-size header of 8 bytes made of four fields each of 2 bytes (16 bits).
- The third field defines the total length of the user datagram, header plus data, in bytes.
- The 16 bits can define a total length of 0 to 65535 bytes. However, the total length needs to be less because a UDP user datagram is stored in an IP datagram with the total length of 65535 bytes.



a. UDP user datagram



b. Header format

EXERCISE

The following is the contents 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?

EXERCISE

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

CB84000D001C001C

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

Solution

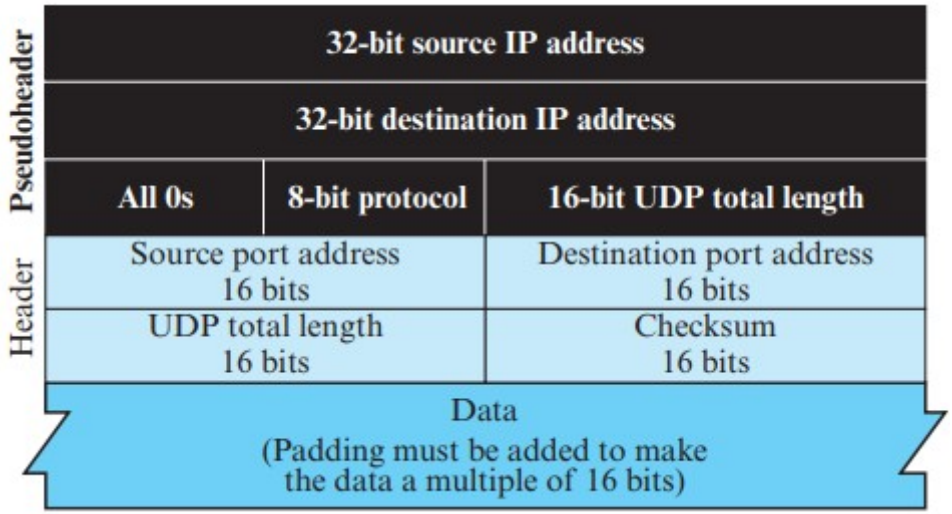
a. The source port number is the first four hexadecimal digits $(CB84)_{16}$, which means that the source port number is 52100.
b. The destination port number is the second four hexadecimal digits $(000D)_{16}$, which means that the destination port number is 13.
c. The third four hexadecimal digits $(001C)_{16}$ define the length of the whole UDP packet as 28 bytes.
d. The length of the data is the length of the whole packet minus the length of the header, or $28 - 8 = 20$ bytes.
e. Since the destination port number is 13 (well-known port), the packet is from the client to the server.
f. The client process is the Daytime (see Table 3.1).

CHECKSUM

- The UDP checksum is used to verify the integrity of the UDP header and data during transmission. It ensures that the packet has not been altered or corrupted while in transit.
- The checksum calculation involves the UDP header, the data (payload), and a pseudo-header derived from the IP header.
- The pseudo-header is a set of fields from the IP header that is added temporarily for the checksum calculation. It includes the following:
  - **Source IP Address (4 bytes)** – The IP address of the sender.
  - **Destination IP Address (4 bytes)** – The IP address of the receiver.
  - **Protocol (1 byte)** – The protocol type (set to 17 for UDP).
  - **UDP Length (2 bytes)** – The length of the UDP packet (header + data).
  - **Padding (1 byte)** – A padding byte set to 0 to ensure the pseudo-header length is a multiple of 2 bytes.

CHECKSUM

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



Checksum Calculation Steps

Step 1: Form the Checksum Calculation Input

- Combine the UDP header, UDP payload, and the pseudo-header. The checksum is calculated over all these fields.
- If the length of the data is not a multiple of 2 bytes, padding is added (a zero byte) to make it even.

Step 2: Add the 16-bit words

- The checksum is calculated by treating the data as a series of 16-bit words (2 bytes each).
- All 16-bit words (UDP header, data, and pseudo-header) are added together using one's complement addition.
- During addition, if the sum exceeds 16 bits, the overflow (carry) is added back to the sum to keep it within 16 bits.

Checksum Calculation Steps

- **Step 3: One's Complement of the Sum**
  - Once all the words have been added, the final checksum is the one's complement of the resulting sum. This means flipping all the bits in the sum.
  - The resulting 16-bit value is the UDP checksum.
- **Step 4: Insert the Checksum in the UDP Header**
  - The calculated checksum is inserted into the UDP header's checksum field.

Verification of the Checksum

- When the receiver gets the UDP packet, it performs the same checksum calculation over the received UDP header, payload, and pseudo-header.
- If the resulting checksum is all ones (0xFFFF), it means the packet is valid (i.e., no errors were detected).
- If the result is anything other than 0xFFFF, it indicates that the packet was corrupted, and the receiver discards it.
- **Special Case: Checksum field set to 0**
  - In IPv4, the UDP checksum is optional. If the checksum is not computed, the field is set to 0.
  - However, in IPv6, the checksum is mandatory, and a packet with a checksum of 0 will be discarded.

Checksum Calculation Example

- **Source IP Address**: 192.168.1.10 (in hexadecimal: 0xC0A8 010A)
- **Destination IP Address:** 192.168.1.20 (in hexadecimal: 0xC0A8 0114)
- **Protocol:** UDP (value is 17 in decimal, or 0x11 in hexadecimal)
- **UDP Length:** 16 bytes (UDP header + payload)
- **UDP Header Fields:**
  - Source Port: 12345 (0x3039 in hexadecimal)
  - Destination Port: 80 (0x0050 in hexadecimal)
  - Length: 16 (0x0010 in hexadecimal)
  - Checksum: Initially set to 0x0000 (for calculation purposes)
- **Payload (Data):** Let's assume 4 bytes of data: 0x4142 4344 ("ABCD" in ASCII)

Checksum Example

```
0xC0A8 010A (Pseudo-header: Source IP)
0xC0A8 0114 (Pseudo-header: Destination IP)
0x0011 0010 (Pseudo-header: Protocol and Length)
0x3039 0050 (UDP Header: Source and Destination Port)
0x0010 0000 (UDP Header: Length and Checksum)
0x4142 4344 (Payload: "ABCD")
```

Checksum Example

```
Source IP Address: 11000000.10101000.00000001.00001010
Dest   IP Address: 11000000.10101000.00000001.00010100
    Protocol (17): 00010001
UDP Length (16 bytes): 0000000000010000
Combined(Protocol + Length): 00000000000100010000000000010000

Source port:  0011000000111001
Dest   port:  0000000001010000
length:       0000000000010000
Checksum:     0000000000000000 (Initially)
Data:         01000001010000100100001101000100
```

```
Source IP Address: 11000000.10101000.00000001.00001010
Dest   IP Address: 11000000.10101000.00000001.00010100
    Protocol (17): 00010001
UDP Length (16 bytes): 0000000000010000
Combined(Protocol + Length): 00000000000100010000000000010000

Source port:  0011000000111001
Dest   port:  0000000001010000
length:       0000000000010000
Checksum:     0000000000000000 (Initially)
Data:         01000001010000100100001101000100



Combine all these values and create separate 16-bit words
1100000010101000 (1st word)    0000000100001010 (2nd word)
1100000010101000 (3rd word)    0000000100010100 (4th word)
0000000000010001 (5th word)    0000000000010000 (6th word)
0011000000111001 (7th word)    0000000001010000  (8th word)
0000000000010000 (9th word)    0000000000000000  (10th word)
0100000101000010 (11th word)   0100001101000100  (12th word)
```

```
Now compute the sum of each word
  1100000010101000 (1st word) + 0000000100001010 (2nd word)
= 1100000110110010

  1100000010101000 (3rd word) +   0000000100010100 (4th word)
= 1100000110111100

  0000000000010001 (5th word) +  0000000000010000 (6th word)
= 0000000000100001

  0011000000111001 (7th word) +   0000000001010000  (8th word)
= 0011000010001001

  0000000000010000 (9th word) +   0000000000000000  (10th word)
= 0000000000010000

  0100000101000010 (11th word) +   0100001101000100  (12th word)
= 1000010001110110
```

```
Now add all the intermediate sums

1100000110110010 + 1100000110111100 = 1 1000001101101110
1000001101101110 + 1 = 1000001101101111 (Adding carry)

1000001101101111 + 0000000000100001 = 1000001110010000

1000001110010000 + 0011000010001001 = 1011010000011001

1011010000011001 + 0000000000010000 = 1011010000101001

1011010000101001 + 1000010001110110 = 1 0011100010011111
0011100010011111 + 1 = 0011100010100000 (Adding carry)


Now take 1's complement of the final sum
0011100010100000 ---> 1100011101011111 (Checksum)
```

EXERCISE

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

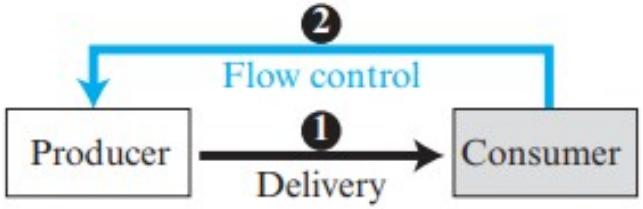
a. The value sent for the checksum field is all 0s to show that the checksum is not calculated.
b. When the sender complements the sum, the result is all 0s; the sender complements the result again before sending. The value sent for the checksum is all 1s. The second complement operation is needed to 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 the pseudoheader have nonzero values.

Flow Control

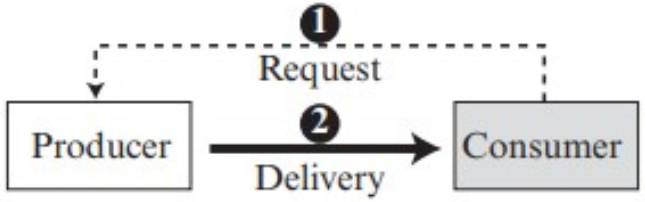
- Whenever an entity produces items and another entity consumes them, there should be a balance between production and consumption rates.
- If the items are produced faster than they can be consumed, the consumer can be overwhelmed and may need to discard some items.
- If the items are produced more slowly than they can be consumed, the consumer must wait, and the system becomes less efficient.

PUSHING AND PULLING

- Delivery of items from a producer to a consumer can occur in one of two ways: pushing or pulling.
- If the sender delivers items whenever they are produced without a prior request from the consumer—the delivery is referred to as pushing. If the producer delivers the items after the consumer has requested them, the delivery is referred to as pulling.
- When the producer pushes the items, the consumer may be overwhelmed and there is a need for flow control, in the opposite direction, to prevent discarding of the items. In other words, the consumer needs to warn the producer to stop the delivery and to inform it when it is ready again to receive the items. When the consumer pulls the items, it requests them when it is ready. In this case, there is no need for flow control.



a. Pushing

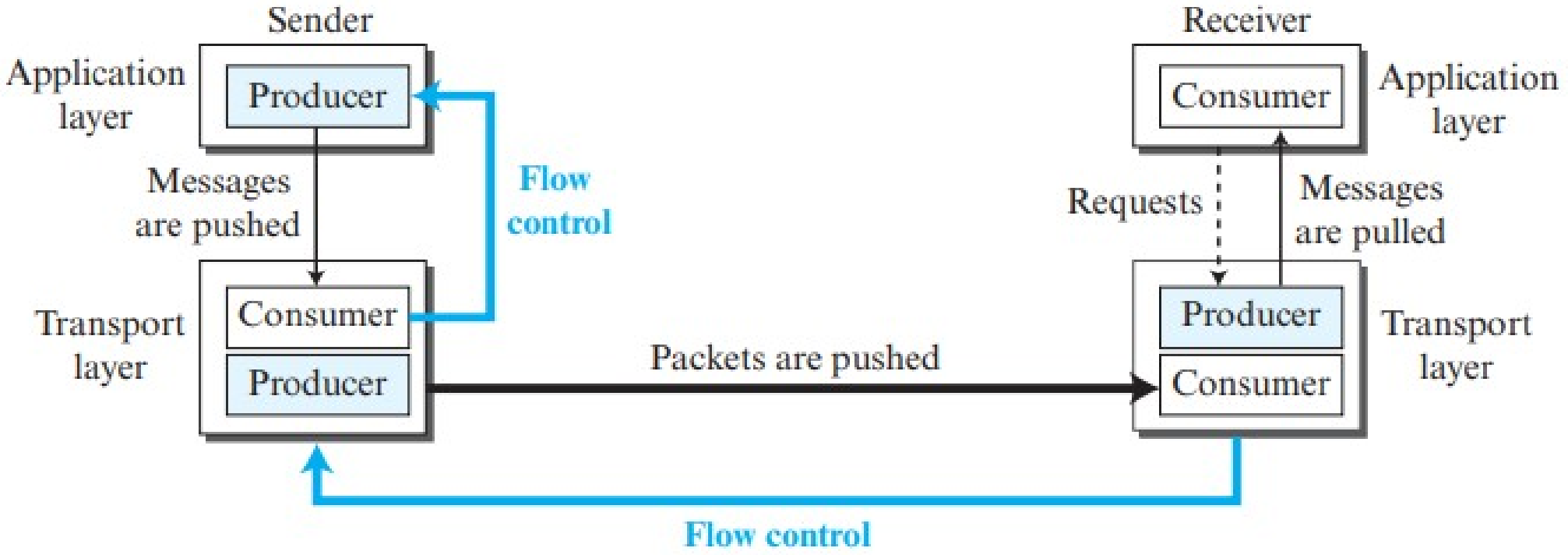


b. Pulling

Flow Control at Transport Layer

- In transport-layer communication, there are four key players: **the sender process** (which creates message chunks), **the sender transport layer** (which both consumes these message chunks and sends them), **the receiver transport layer** (which consumes packets from the sender and produces messages), and **the receiver process** (which ultimately receives these messages).
- Here's how they work together -
  - The sending process produces message chunks and sends them to the sender transport layer.
  - The sender transport layer consumes these message chunks, encapsulates them into packets, and sends them to the receiving transport layer.
  - The receiving transport layer consumes the packets from the sender, decapsulates them to extract messages, and delivers these messages to the receiving application layer. This final delivery is usually a "pull" request, meaning it waits until the application layer asks for messages.
- Because of this complex interaction, we need at least two instances of flow control: one from the sending transport layer to the sending application layer (to prevent overproduction) and another from the receiving transport layer to the sending transport layer (to maintain a balanced flow between sender and receiver).

Flow Control at Transport Layer

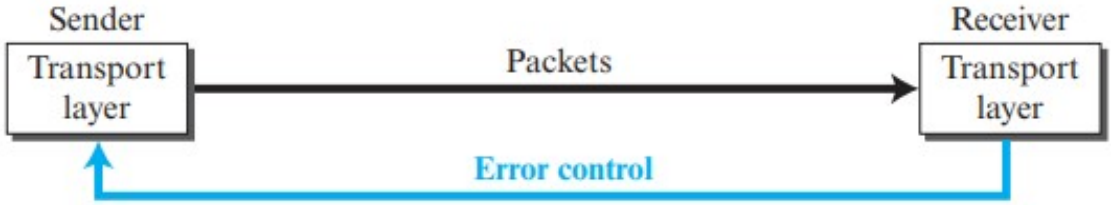


BUFFERS

- Flow control ensures that data transfer between sender and receiver happens at a manageable pace.
- One common approach to implement flow control is to use two buffers - one at the sender's transport layer and the other at the receiver's transport layer.
- **Sender Buffer:** This buffer at the sender's transport layer stores packets waiting to be sent. When it becomes full, the sender signals the application layer to stop sending message chunks. When there's space in the buffer, it tells the application layer to resume sending.
- **Receiver Buffer:** The receiver's transport layer has its buffer to store incoming packets. When it becomes full, it tells the sender's transport layer to stop sending packets. When space is available, it informs the sender's transport layer that it can resume sending.
- In essence, these buffers and signals allow the sender and receiver to coordinate and control the flow of data, preventing overload and ensuring smooth communication.

Error Control

- In the Internet, since the underlying network layer (IP) is unreliable, we need to make the transport layer reliable if the application requires reliability.
- Reliability can be achieved to add error control services to the transport layer. Error control at the transport layer is responsible for -
  - Detecting and discarding corrupted packets.
  - Keeping track of lost and discarded packets and resending them.
  - Recognizing duplicate packets and discarding them.
  - Buffering out-of-order packets until the missing packets arrive.



SEQUENCE NUMBERS

- Error control requires that the sending transport layer knows which packet is to be resent and the receiving transport layer knows which packet is a duplicate, or which packet has arrived out of order. This can be done if the packets are numbered.
- We can add a field to the transport-layer packet to hold the **sequence number** of the packet.
- When a packet is corrupted or lost, the receiving transport layer can somehow inform the sending transport layer to resend that packet using the sequence number.
- The receiving transport layer can also detect duplicate packets if two received packets have the same sequence number. The out-of-order packets can be recognized by observing gaps in the sequence numbers.

SEQUENCE NUMBERS

- Packets are numbered sequentially. However, because we need to include the sequence number of each packet in the header, we need to set a limit.
- If the header of the packet allows **m** bits for the sequence number, the sequence numbers range from 0 to $2^m - 1$.
- In case after we need more numbers, then we can wrap around the sequence. In other words, the sequence numbers are modulo 2^m.

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

ACKNOWLEDGMENT

- In simple terms, when we send data over a network, sometimes things can go wrong. To make sure our data gets to the right place, we use both positive and negative signals.
- Positive signals are like saying, "I got your data, it's all good!", This happens more often at the transport layer of the network.
- Here is how it works -
  - The receiver says, "I got your data, it's fine" by sending an acknowledgment (ACK) when it receives a data packet safely.
  - If some packets are damaged, the receiver just throws them away.
  - The sender sets a timer when it sends a packet. If it doesn't get an ACK before the timer runs out, it assumes something went wrong and sends the packet again.
  - If the receiver gets the same packet twice by mistake, it just ignores the duplicate.
  - If the packets arrive in the wrong order, the receiver can either throw them away (and the sender will think they got lost) or keep them until the missing ones arrive.

Combination of Flow Control and Error Control

- We have discussed that flow control requires the use of two buffers, one at the sender site and the other at the receiver site.
- We have also discussed that error control requires the use of sequence and acknowledgment numbers by both sides.
- These two requirements can be combined if we use two numbered buffers, one at the sender, one at the receiver.
- At the sender, when a packet is prepared to be sent, we use the number of the next free location, x, in the buffer as the sequence number of the packet.

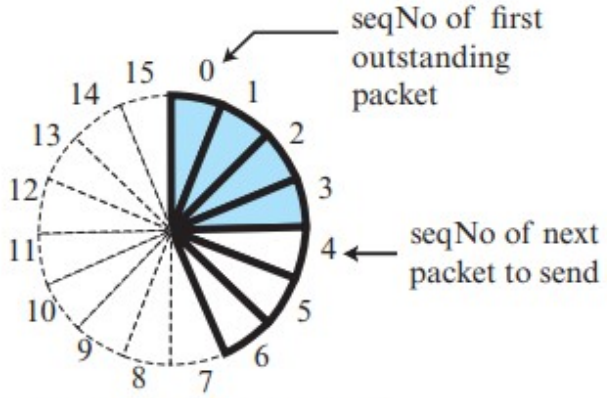
Combination of Flow and Error Control

- When the packet is sent, a copy is stored at memory location x, awaiting the acknowledgment from the other end.
- When an acknowledgment related to a sent packet arrives, the packet is purged and the memory location becomes free.
- At the receiver, when a packet with sequence number y arrives, it is stored at the memory location y until the application layer is ready to receive it. An acknowledgment can be sent to announce the arrival of packet y.

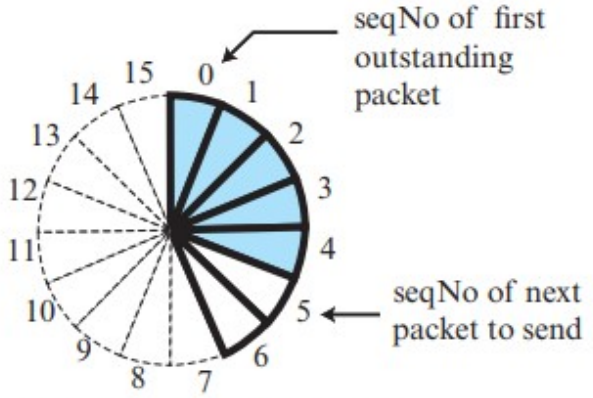
Sliding Window

- Since the sequence numbers are in modulo 2^m, a circle can represent the sequence numbers from 0 to $2^m - 1$. The buffer is represented as a set of slices, called the **sliding window**, that occupies part of the circle at any time.
- At the sender site, when a packet is sent, the corresponding slice is marked. When all the slices are marked, it means that the buffer is full and no further messages can be accepted from the application layer. When an acknowledgment arrives, the corresponding slice is unmarked.
- Note that the sliding window is just an abstraction: the actual situation uses computer variables to hold the sequence numbers of the next packet to be sent and the last packet sent.

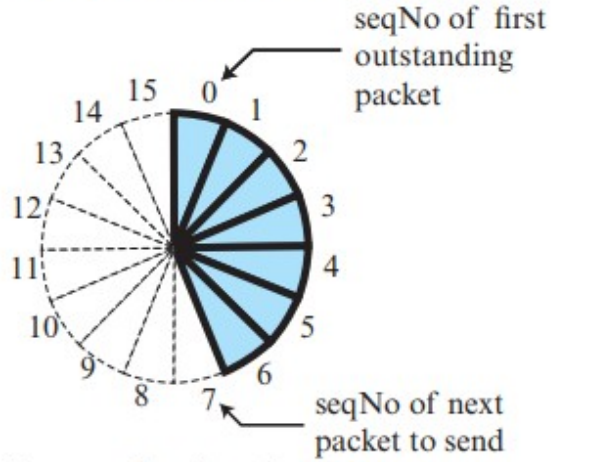
Sliding Window in Circular Format



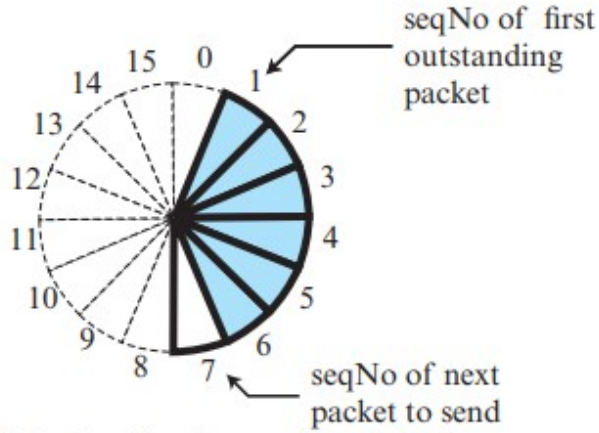
a. Four packets have been sent.



b. Five packets have been sent.



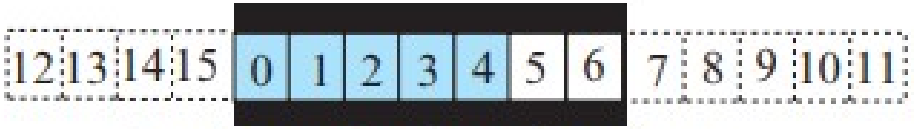
c. Seven packets have been sent; window is full.



d. Packet 0 has been acknowledged; window slides.

SLIDING WINDOW IN LINEAR FORMAT

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.

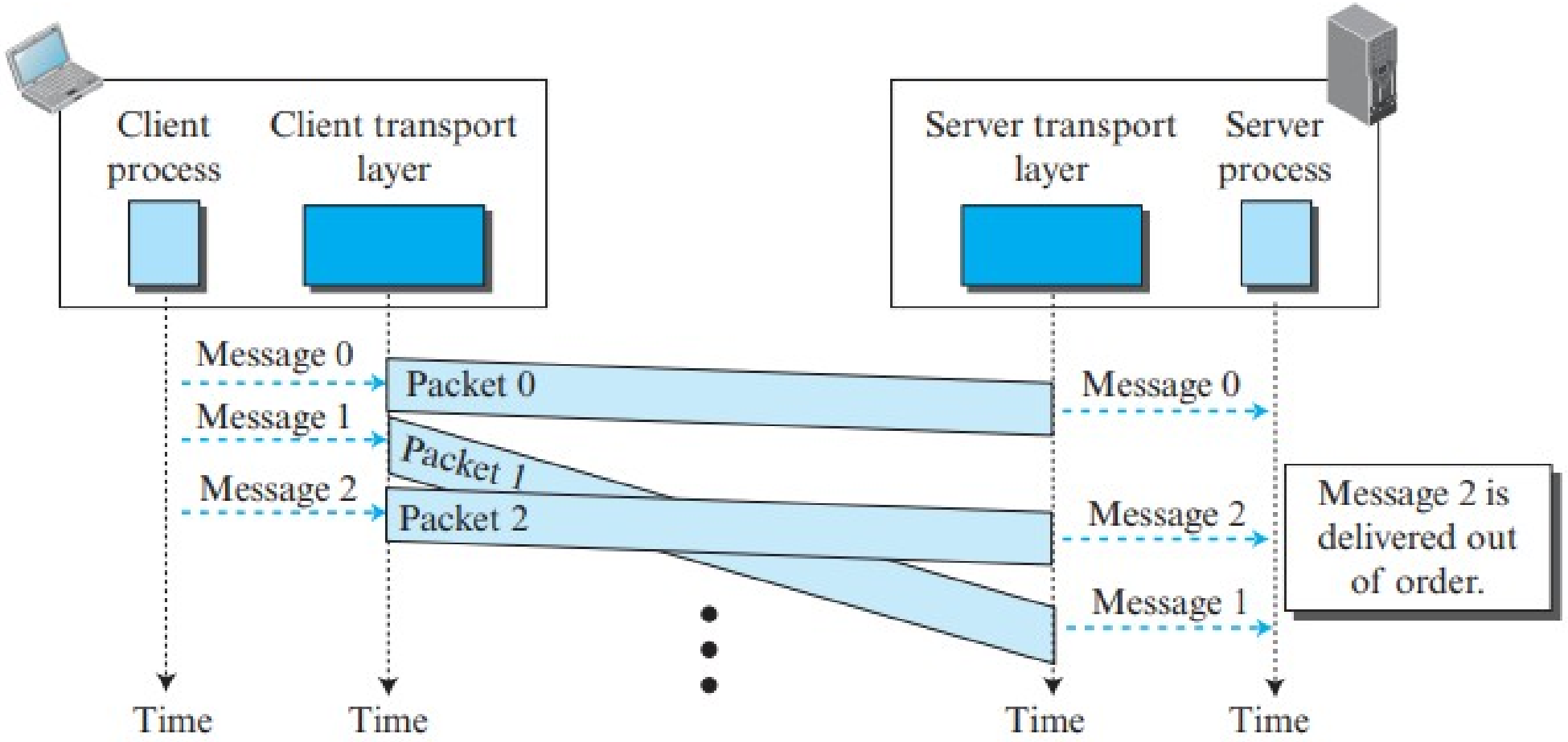
Congestion Control

- Congestion in a network, like the Internet, happens when there's more data trying to get through the network than it can handle. It's similar to a traffic jam on a road during rush hour.
- Here is why it occurs:
  - Networks have limits on how much data they can handle at once (capacity).
  - If too many data packets are sent to the network at once (load), it can't keep up.
  - Routers and switches in the network use queues to hold packets before and after processing.
  - If these queues get too full because packets are arriving faster than they can be processed, congestion occurs.
- This congestion at the network level affects the transport layer, which is higher up in the networking hierarchy. TCP (a common protocol) has its own way to deal with congestion because it can't always rely on the network layer to control it.

Connectionless Services at Transport Layer

- Connectionless service at the transport layer means independency between packets; connection-oriented means dependency between packets.
- In a connectionless service, the source process (application program) needs to divide its message into chunks of data of the size acceptable by the transport layer and deliver them to the transport layer one by one.
- The transport layer treats each chunk as a single unit without any relation between the chunks. When a chunk arrives from the application layer, the transport layer encapsulates it in a packet and sends it.

CONNECTIONLESS SERVICES AT TRANSPORT LAYER



Connectionless Services

- We know that DNS uses UDP Protocol which is connectionless. Therefore all the information should fit inside a single packet.
- What if the data doesn't fit inside a single packet?
- In the internet, most processes limits their UDP packet size to be 512 bytes.
- So if DNS response or query size is larger than 512 bytes, then DNS will be forced to use TCP rather than UDP.
- There should be consistency in DNS Zone database. To make this, DNS always transfers Zone data using TCP because TCP is reliable and make sure zone data is consistent by transferring the full zone to other DNS servers who has requested the data.
- There are other processes like DNS which can both TCP and UDP depending upon their requirements.

CONNECTIONLESS SERVICES

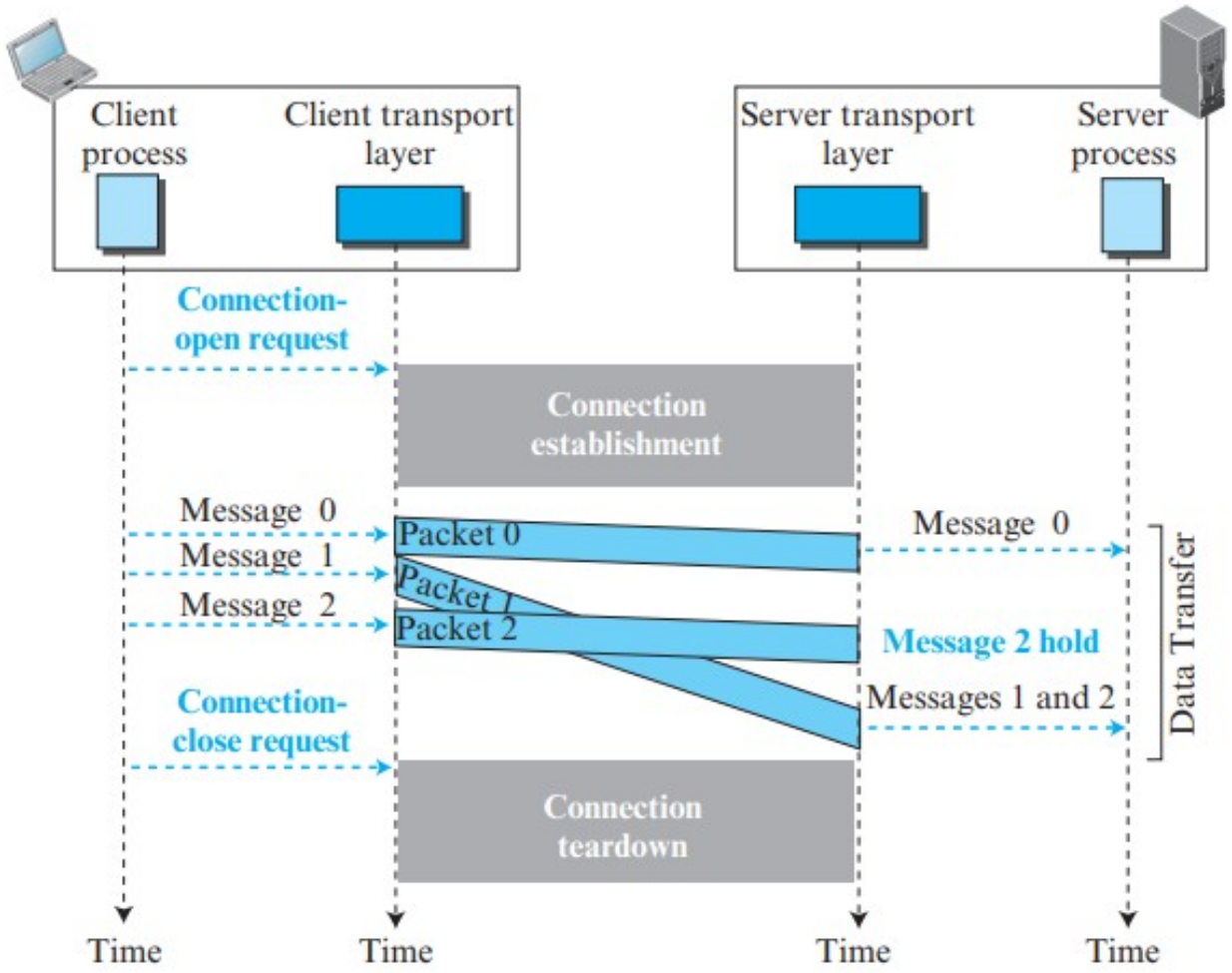
Port	*Protocol*	*UDP*	*TCP*	*Description*
7	Echo	√		Echoes back a received datagram
9	Discard	√		Discards any datagram that is received
11	Users	√	√	Active users
13	Daytime	√	√	Returns the date and the time
17	Quote	√	√	Returns a quote of the day

Port	*Protocol*	*UDP*	*TCP*	*Description*
19	Chargen	√	√	Returns a string of characters
20, 21	FTP		√	File Transfer Protocol
23	TELNET		√	Terminal Network
25	SMTP		√	Simple Mail Transfer Protocol
53	DNS	√	√	Domain Name Service
67	DHCP	√	√	Dynamic Host Configuration Protocol
69	TFTP	√		Trivial File Transfer Protocol
80	HTTP		√	Hypertext Transfer Protocol
111	RPC	√	√	Remote Procedure Call
123	NTP	√	√	Network Time Protocol
161, 162	SNMP		√	Simple Network Management Protocol

Connection Oriented Services at Transport Layer

- In a connection-oriented service, the client and the server first need to establish a logical connection between themselves.
- The data exchange can only happen after the connection establishment. After data exchange, the connection needs to be torn down.

CONNECTION ORIENTED SERICES



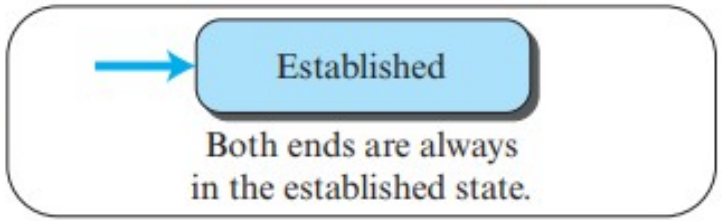
FINITE STATE MACHINE

- The behavior of a transport-layer protocol, both when it provides a connectionless and when it provides a connection-oriented protocol, can be better shown as a **finite state machine (FSM)**.
- Using this tool, each transport layer (sender or receiver) is taught as a machine with a finite number of states. The machine is always in one of the states until an event occurs.

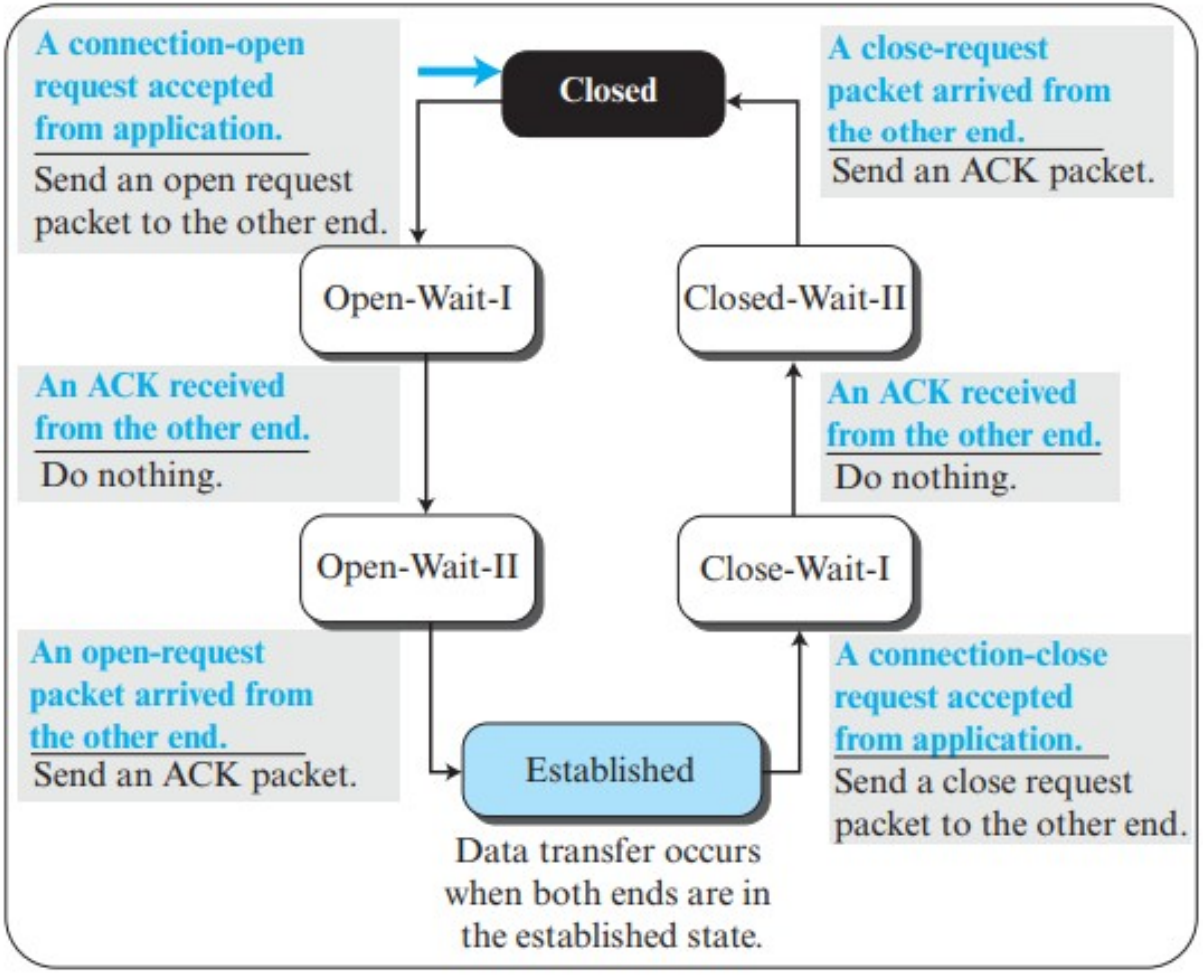
FINITE STATE MACHINE

Note:
The colored arrow shows the starting state.

FSM for connectionless transport layer

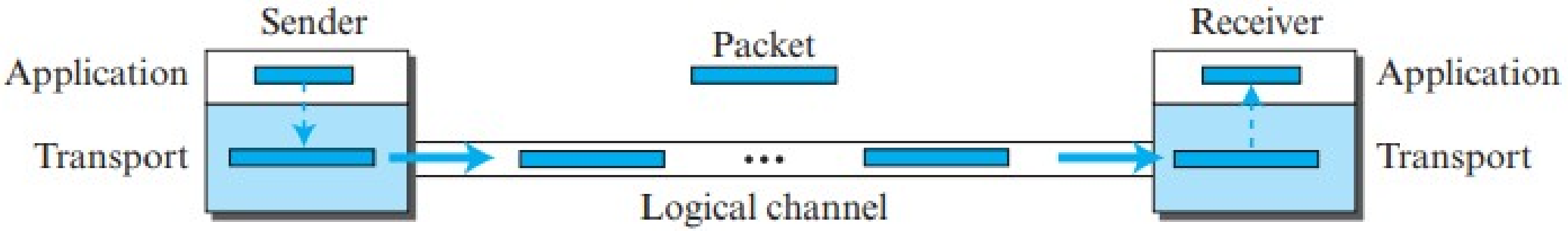


FSM for connection-oriented transport layer

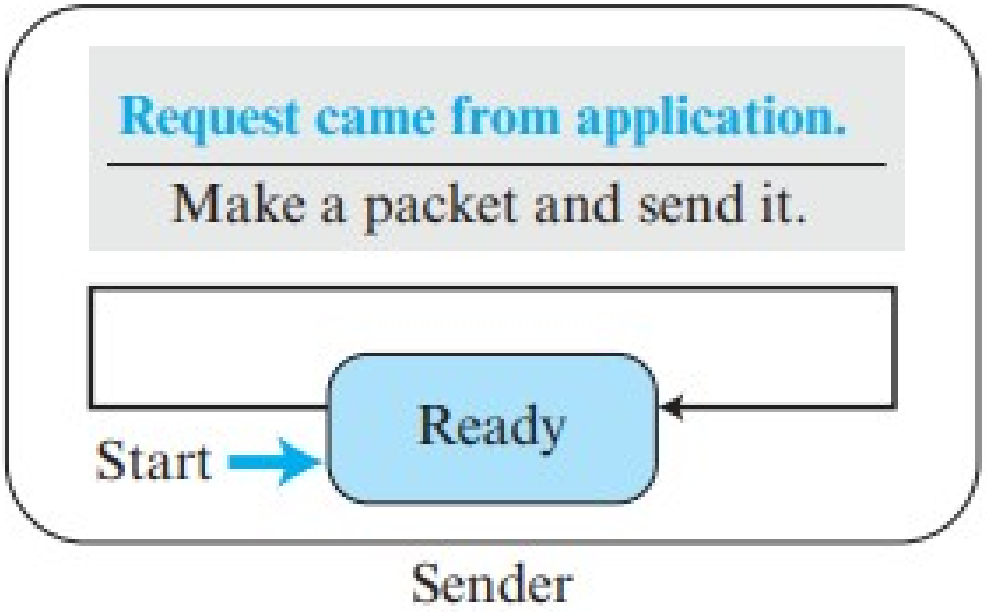


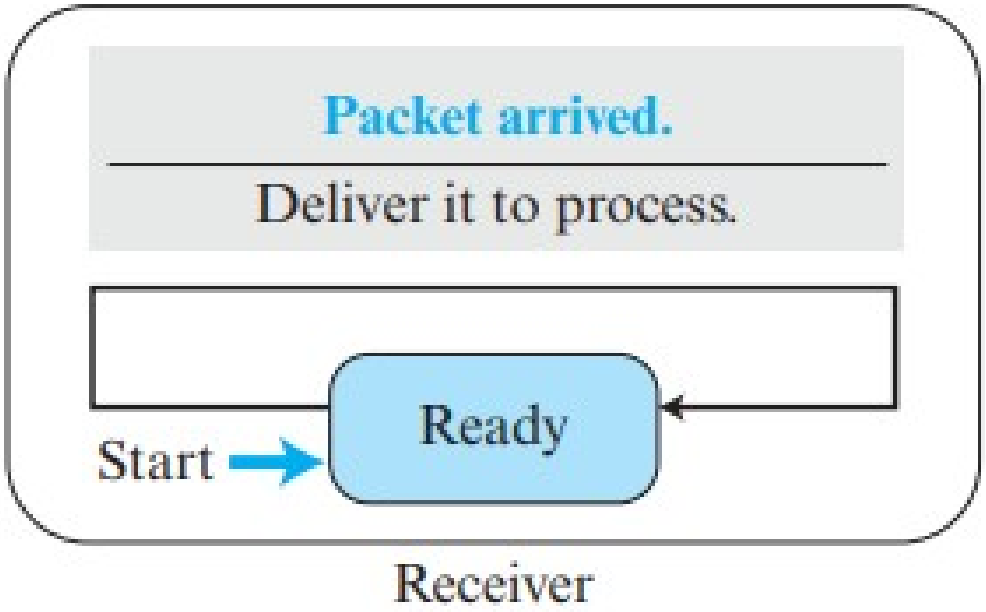
TRANSPORT-LAYER PROTOCOLS: SIMPLE PROTOCOL

- Let's start our discussion on transport layer protocols with a simple protocol which is a connectionless protocol and does not provide flow and error control.
- Simple protocol is a unidirectional protocol (simplex) in which data packets move in one direction only.



SIMPLE PROTOCOL: FSMS



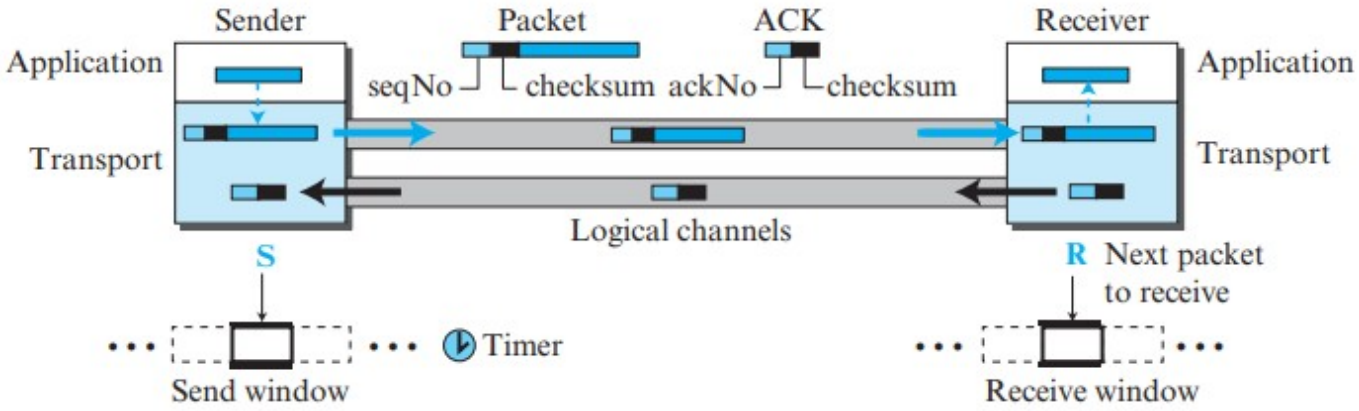


TRANSPORT LAYER PROTOCOLS: STOP-AND-WAIT PROTOCOL

- **Stop-and-Wait protocol** is a connection oriented protocol which uses both flow and error control.
- Both the sender and the receiver use a sliding window of size 1.
- The sender sends one packet at a time and waits for an acknowledgment before sending the next one.
- To detect corrupted packets, we need to add a checksum to each data packet.
- When a packet arrives at the receiver site, it is checked. If its checksum is incorrect, the packet is corrupted and silently discarded. The silence of the receiver is a signal for the sender that a packet was either corrupted or lost.

STOP-AND-WAIT PROTOCOL

- Every time the sender sends a packet, it starts a timer. If an acknowledgment arrives before the timer expires, the timer is stopped and the sender sends the next packet (if it has one to send).
- If the timer expires, the sender resends the previous packet, assuming that the packet was either lost or corrupted. This means that the sender needs to keep a copy of the packet until its acknowledgment arrives.



SEQUENCE NUMBERS

- To prevent duplicate packets, the protocol uses sequence numbers and acknowledgment numbers.
- A field is added to the packet header to hold the sequence number of that packet. One important consideration is the range of the sequence numbers. Since we want to minimize the packet size, we look for the smallest range that provides unambiguous communication.
- Assume we have used x as a sequence number; we only need to use x + 1 after that. There is no need for x + 2. To show this, assume that the sender has sent the packet with sequence number x. Three things can happen.

Sequence Numbers

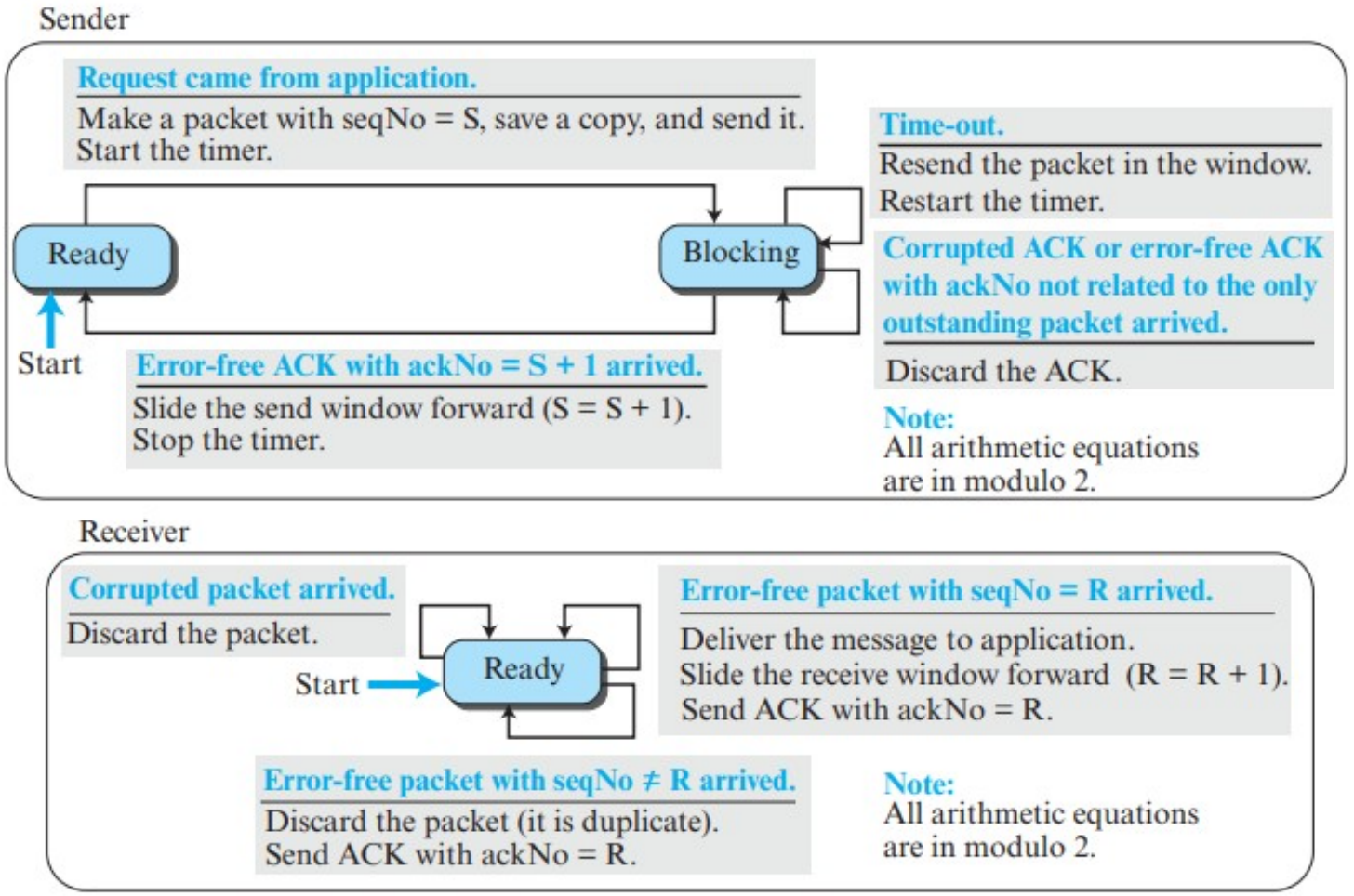
1. The packet arrives safe and sound at the receiver site; the receiver sends an acknowledgment. The acknowledgment arrives at the sender site, causing the sender to send the next packet numbered x + 1.
2. The packet is corrupted or never arrives at the receiver site; the sender resends the packet (numbered x) after the time-out. The receiver returns an acknowledgment.
3. The packet arrives safe and sound at the receiver site; the receiver sends an acknowledgment, but the acknowledgment is corrupted or lost. The sender resends the packet (numbered x) after the time-out. Note that the packet here is a duplicate. The receiver can recognize this fact because it expects packet x + 1 but packet x was received.

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

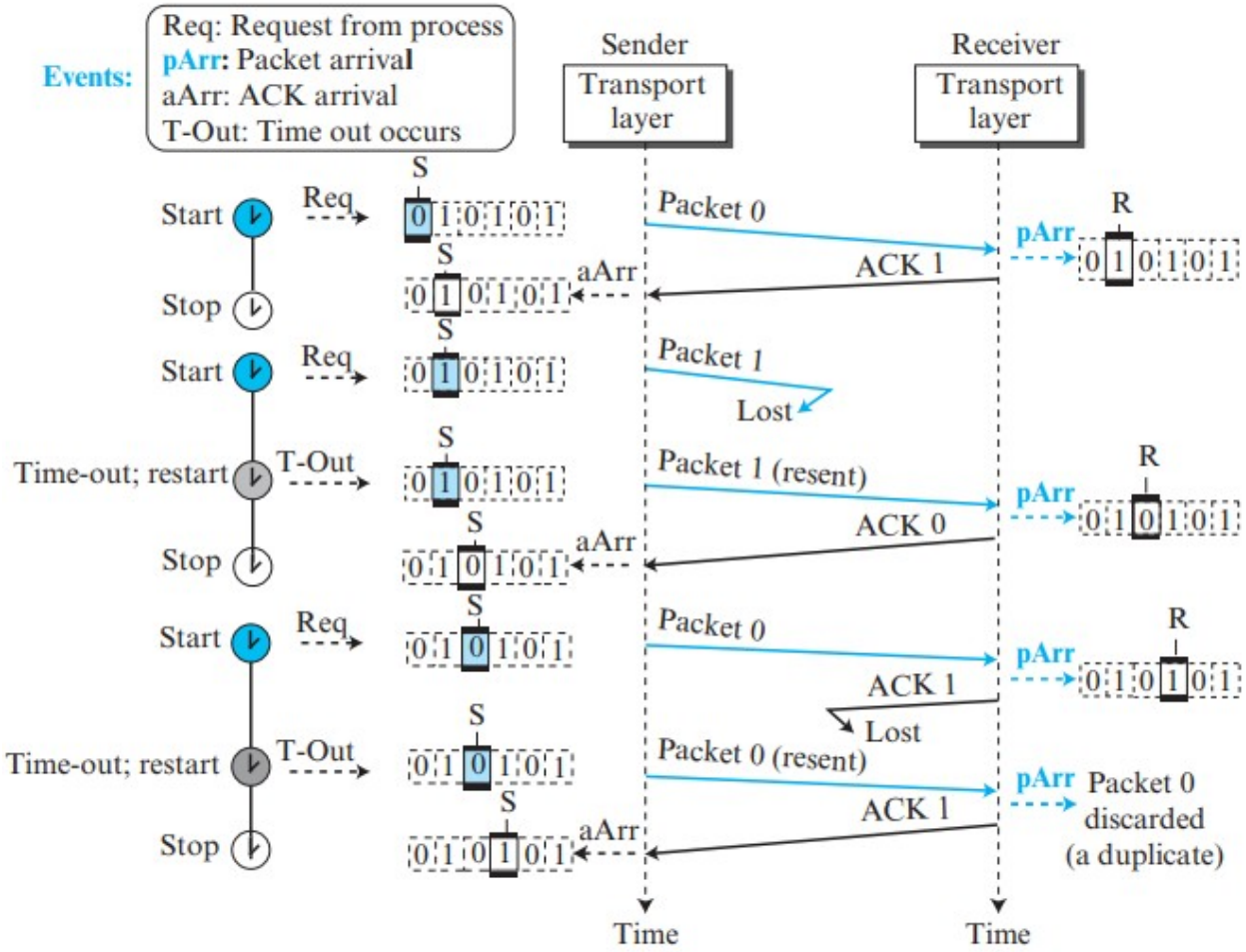
Acknowledgement Numbers

- Since the sequence numbers must be suitable for both data packets and acknowledgments, we use this convention: *The acknowledgment numbers always announce the sequence number of the next packet expected by the receiver.*
- For example, if packet 0 has arrived safe and sound, the receiver sends an ACK with acknowledgment 1 (meaning packet 1 is expected next). If packet 1 has arrived safe and sound, the receiver sends an ACK with acknowledgment 0 (meaning packet 0 is expected).
- The sender has a control variable, which we call S (sender), that points to the only slot in the send window. The receiver has a control variable, which we call R (receiver), that points to the only slot in the receive window.

STOP-AND-WAIT PROTOCOL: FSMS



STOP-AND-WAIT PROTOCOL: FLOW DIAGRAM



EXAMPLE

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

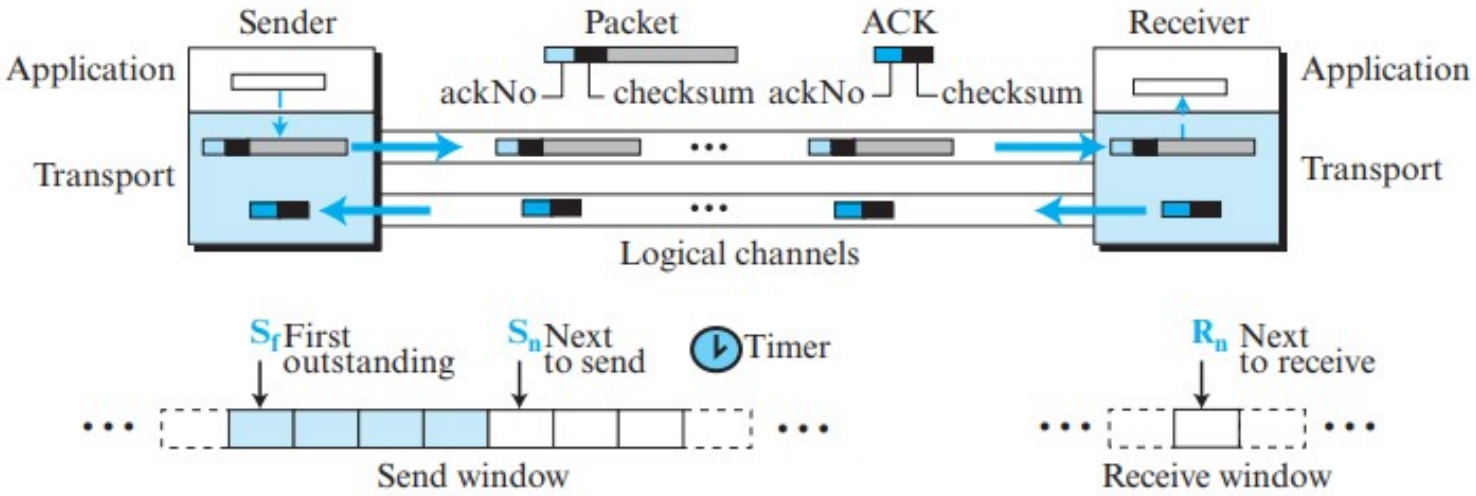
Solution

The bandwidth-delay product is $(1 \times 10^6) \times (20 \times 10^{-3}) = 20{,}000$ bits. The system can send 20,000 bits during the time it takes for the data to go from the sender to the receiver and the acknowledgment to come back. However, the system sends only 1,000 bits. We can say that the link utilization is only 1,000/20,000, or 5 percent. For this reason, in a link with a high bandwidth or long delay, the use of Stop-and-Wait wastes the capacity of the link.

What will be the utilization percentage of the link if we have a protocol that can send up to 15 packets before stopping and worrying about the acknowledgments?

TRANSPORT LAYER PROTOCOL: GO-BACK-*N* PROCOTOL (GBN)

- To improve the efficiency of transmission (to fill the pipe), multiple packets must be in transition while the sender is waiting for acknowledgment.
- The key to Go-back-N is that sender can send several packets before receiving acknowledgments, but the receiver can only buffer one packet.
- Sender keeps a copy of the sent packets until the acknowledgements arrive.

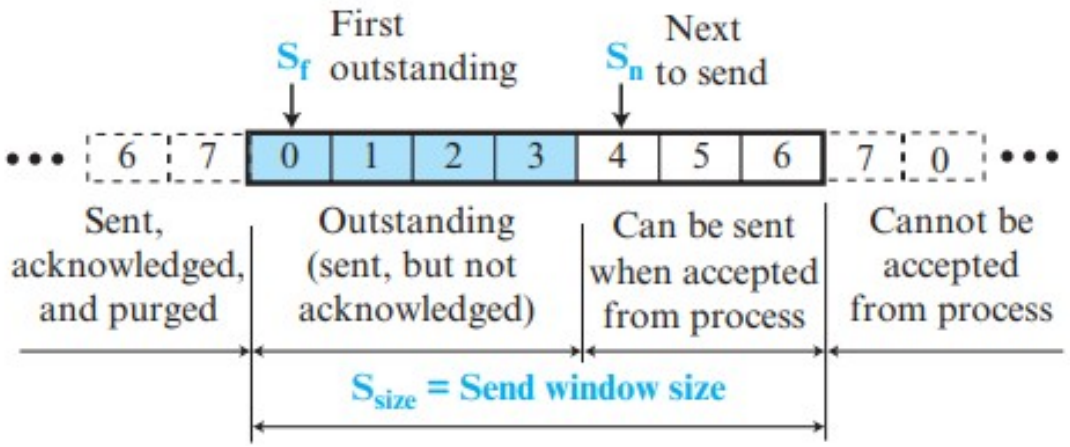


GO-BACK-N PROTOCOL

- **Sequence Numbers -**
  - The sequence numbers are modulo 2^m, where m is the size of the sequence number field in bits.
- **Acknowledgement Numbers -**
  - An acknowledgment number in this protocol is cumulative and defines the sequence number of the next packet expected.
  - For example, if the acknowledgment number (ackNo) is 7, it means all packets with sequence number up to 6 have arrived, safe and sound, and the receiver is expecting the packet with sequence number 7.

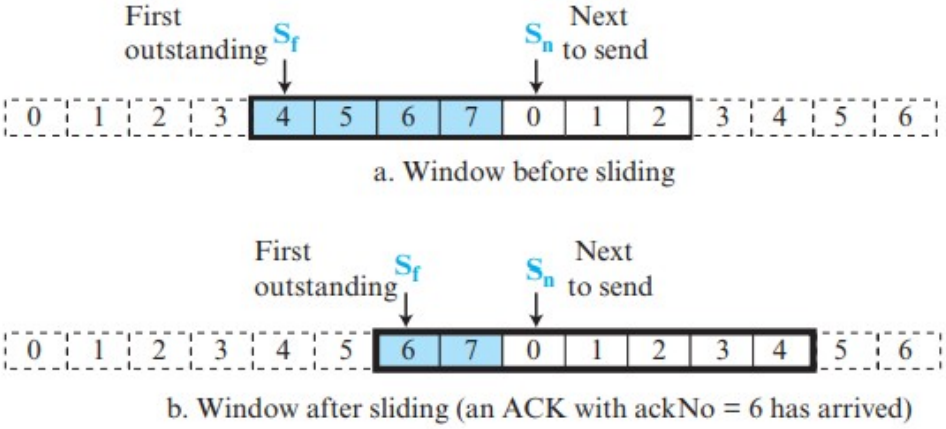
SEND WINDOW

- The send window is an imaginary box covering the sequence numbers of the data packets that can be in transit or can be sent.
- In each window position, some of these sequence numbers define the packets that have been sent; others define those that can be sent.



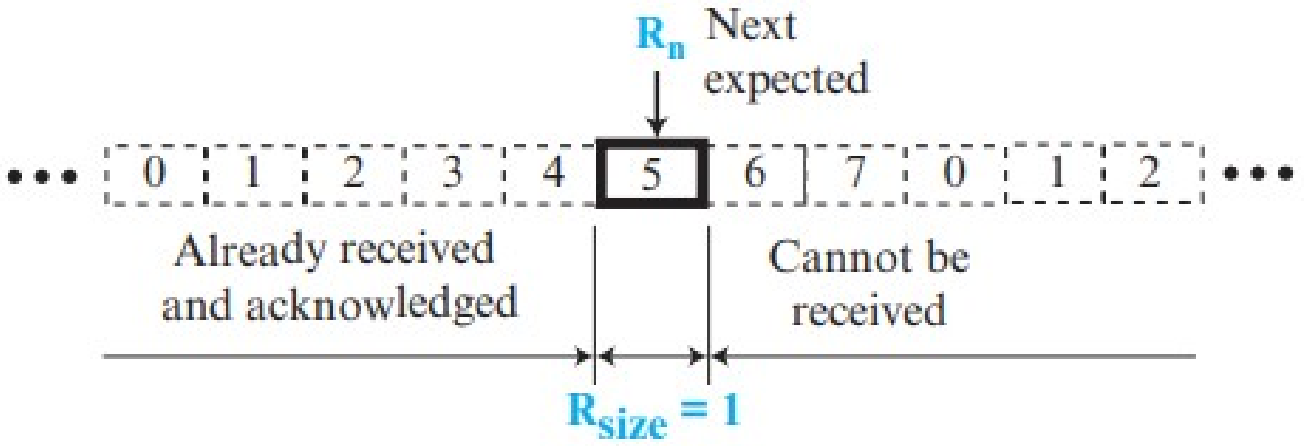
SEND WINDOW

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



Receive Window

- The receive window makes sure that the correct data packets are received and that the correct acknowledgments are sent. In Go-back-N, the size of the receive window is always 1.
- The receiver is always looking for the arrival of a specific packet. Any packet arriving out of order is discarded and needs to be resent.

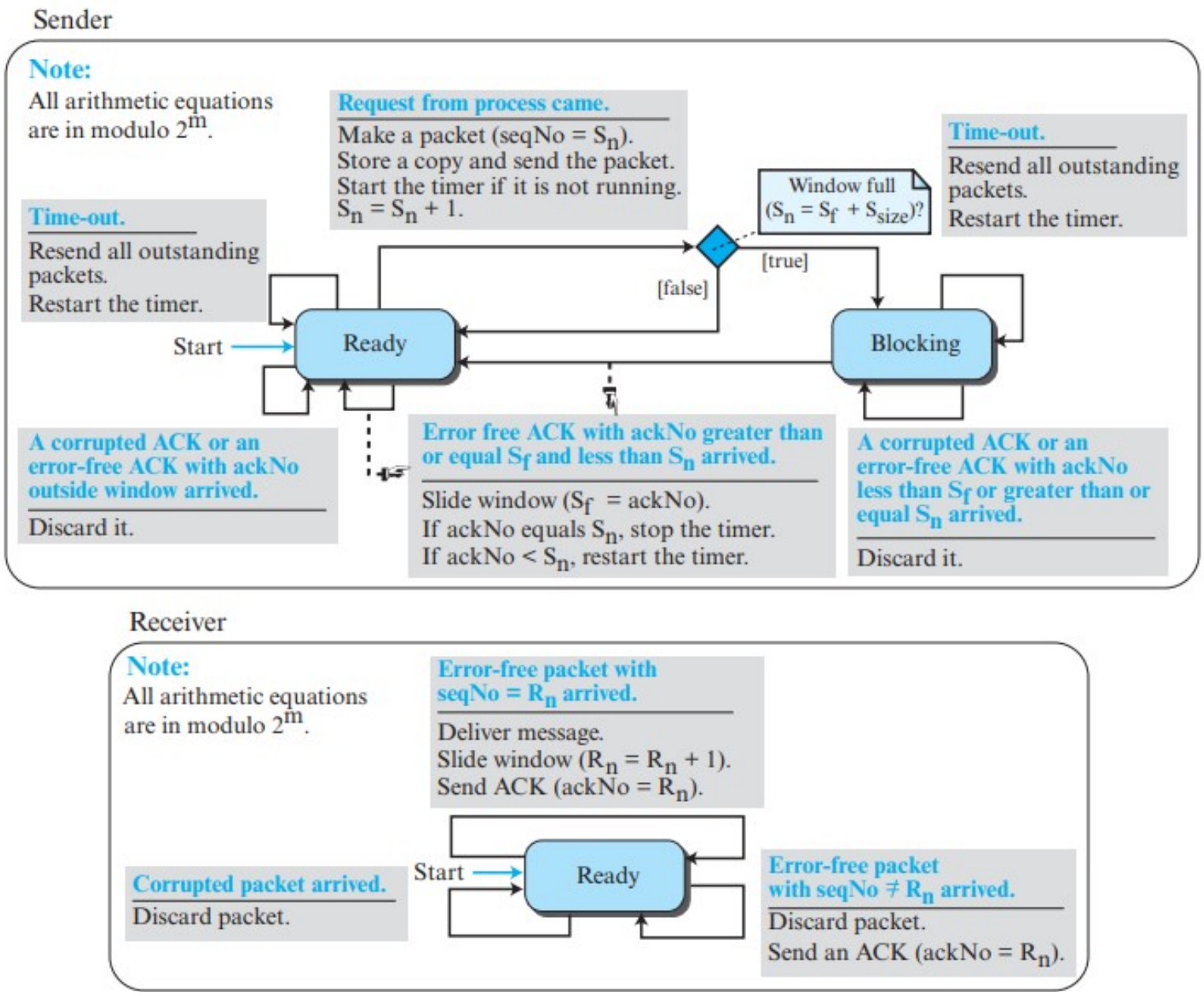


GO-BACK-N PROTOCOL

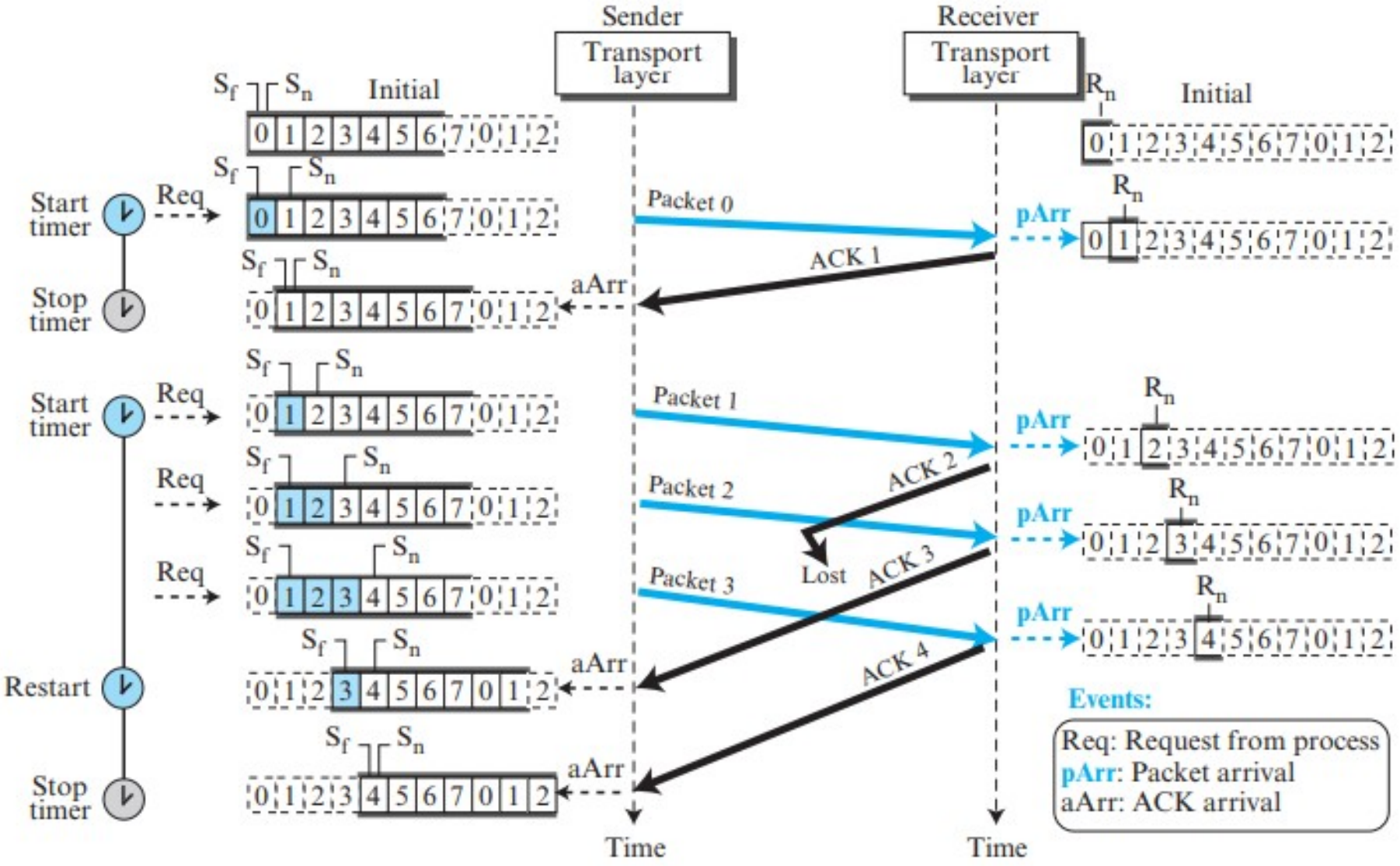
Timers -

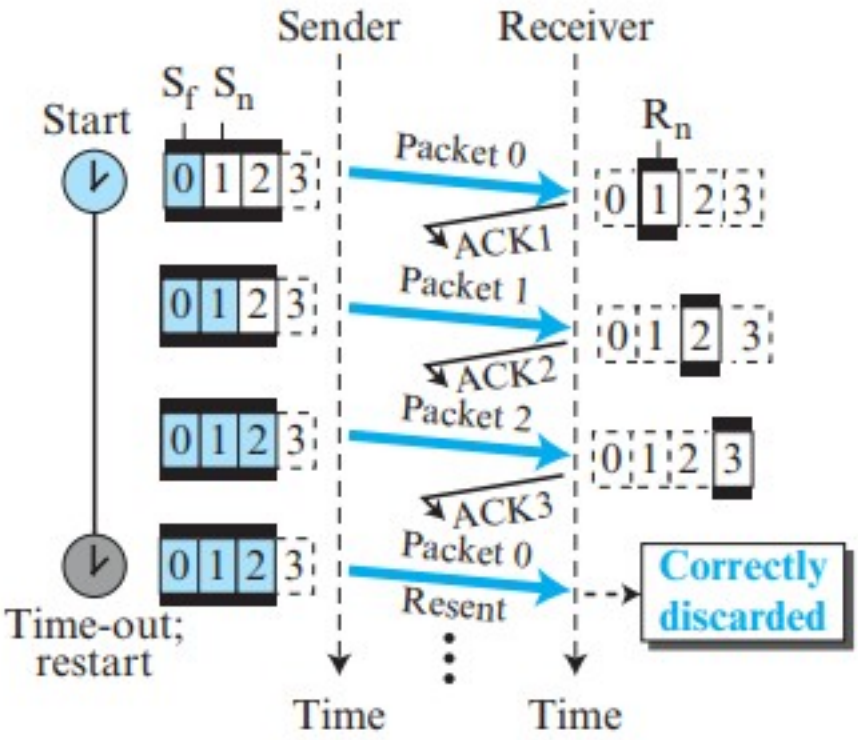
- Although there can be a timer for each packet that is sent, in our protocol we use only one. The reason is that the timer for the first outstanding packet always expires first. We resend all outstanding packets when this timer expires.
- For example, suppose the sender has already sent packet 6 ($S_n = 7$), but the only timer expires. If $S_f = 3$, this means that packets 3, 4, 5, and 6 have not been acknowledged; the sender goes back and resends packets 3, 4, 5, and 6.
- That is why the protocol is called Go-Back-N. On a time-out, the machine goes back N locations and resends all packets.

GO-BACK-N PROTOCOL: FSMS

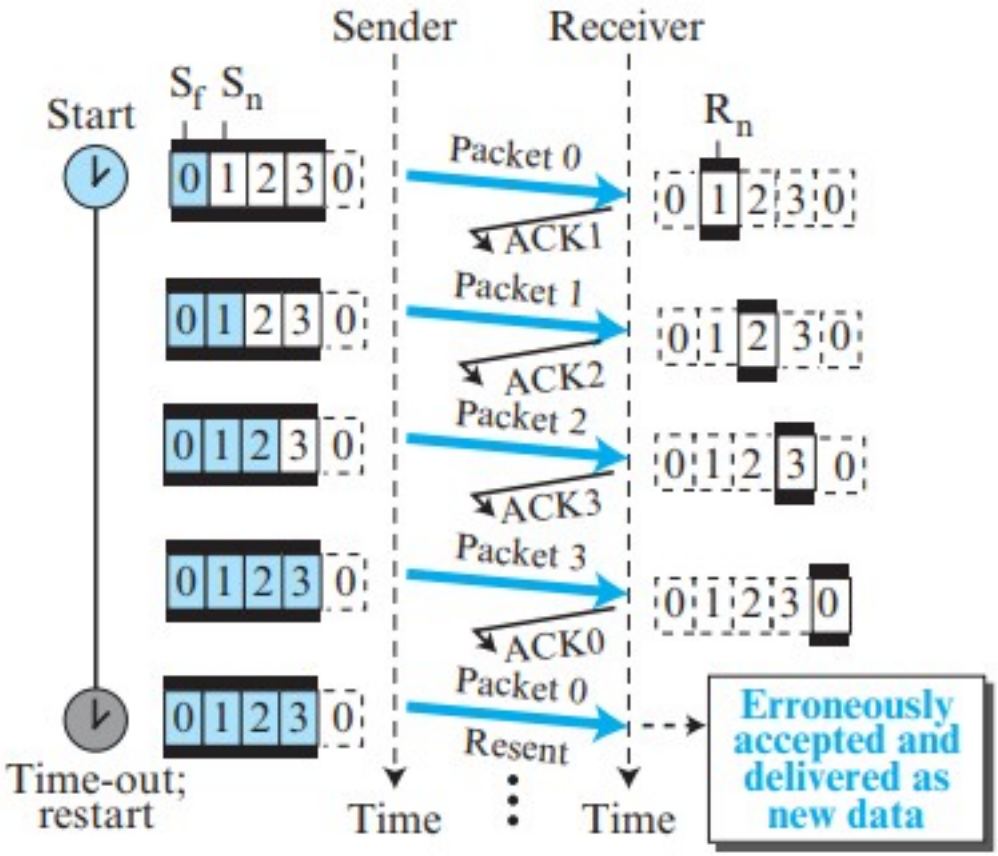


EXAMPLE


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



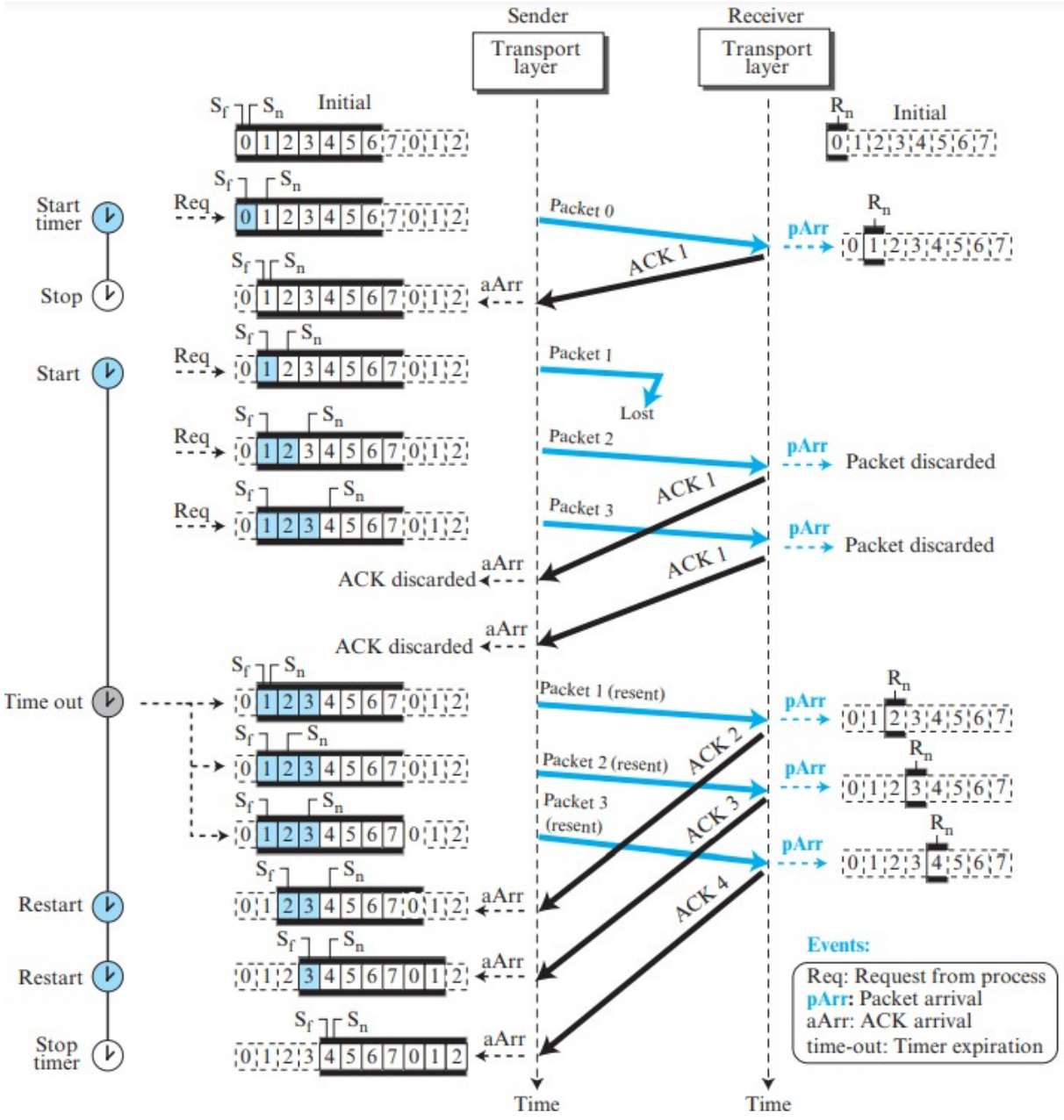
a. Send window of size < 2^m



b. Send window of size = 2^m

In the Go-Back-*N* protocol, the size of the send window must be less than 2^m; the size of the receive window is always 1.

Example

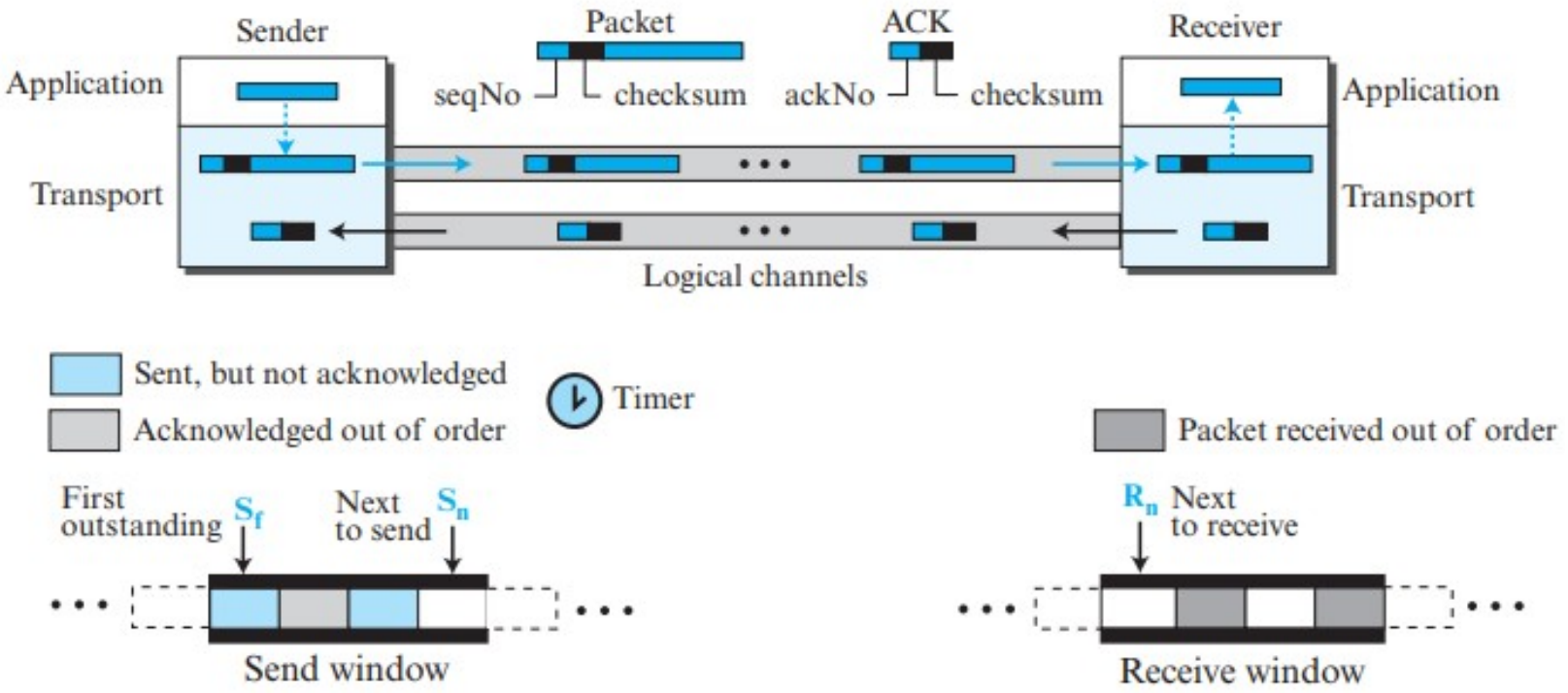

Sender
Receiver
Transport layer
Transport layer
Initial
Start timer
Stop
Start
Time out
Restart
Restart
Stop timer
Req
Packet 0
ACK 1
pArr
aArr
Packet 1
Lost
Packet 2
Packet discarded
Packet 3
ACK discarded
Packet 1 (resent)
Packet 2 (resent)
Packet 3 (resent)
ACK 2
ACK 3
ACK 4
Events:
Req: Request from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

GO-BACK-N PROTOCOL: PROBLEMS

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

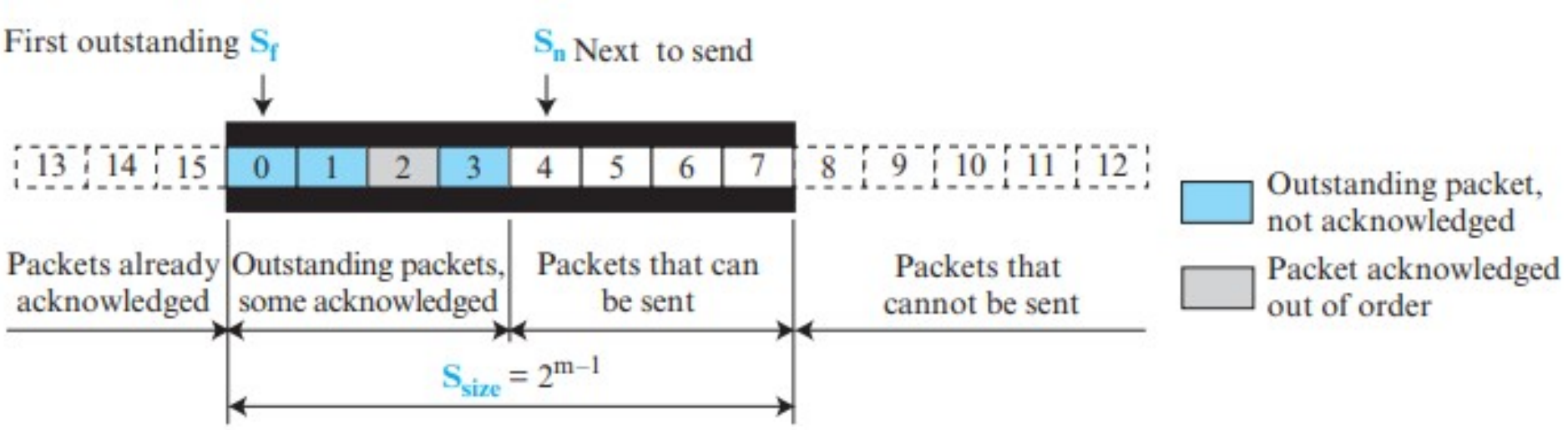
TRANSPORT LAYER PROTOCOLS: SELECTIVE-REPEAT PROTOCOL

- As the name suggests, **Selective-Repeat (SR) Protocol** resends only selective packets, those that are actually lost.



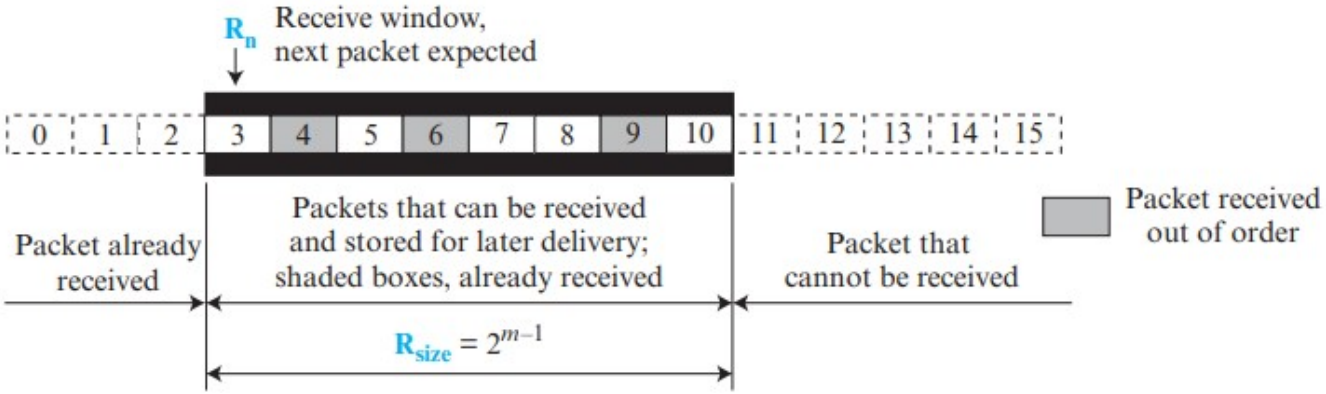
SELECTIVE REPEAT

- The maximum size of send window in Selective-Repeat is 2^{m-1} and receive window size is same as send window.



SELECTIVE-REPEAT

- The Selective-Repeat protocol allows as many packets as the size of the receive window to arrive out of order and be kept until there is a set of consecutive packets to be delivered to the application layer.
- Because the sizes of the send window and receive window are the same, all the packets in the send packet can arrive out of order and be stored until they can be delivered.



SELECTIVE-REPEAT

- **Timer -**
  - Theoretically, Selective-Repeat uses one timer for each outstanding packet. When a timer expires, only the corresponding packet is resent.
  - However, most transport layer protocols that implement SR use only one single timer.
- **Acknowledgments -**
  - In SR, an ackNo defines the sequence number of one single packet that is received safe and sound; there is no feedback for any other.

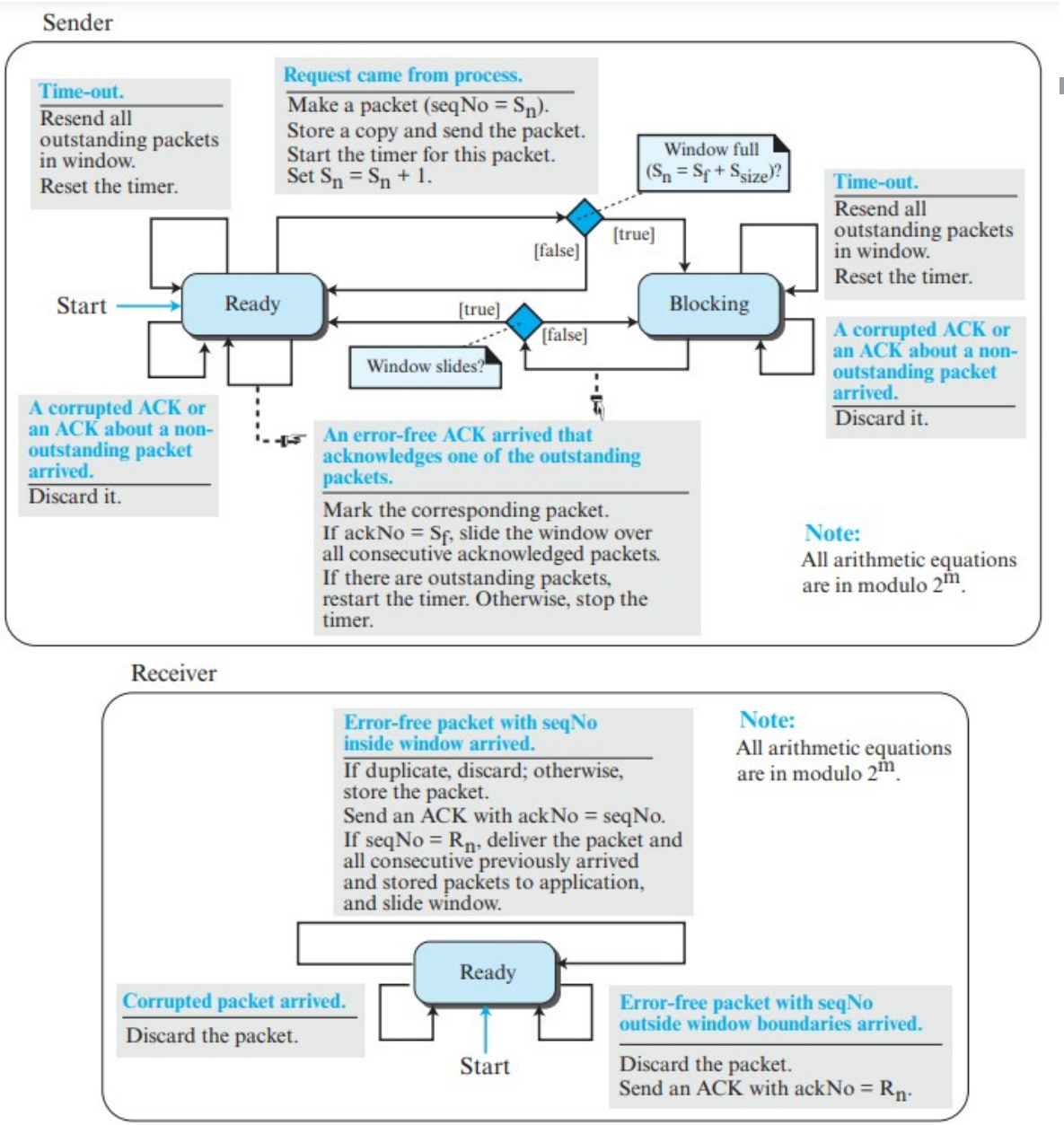
EXAMPLE

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

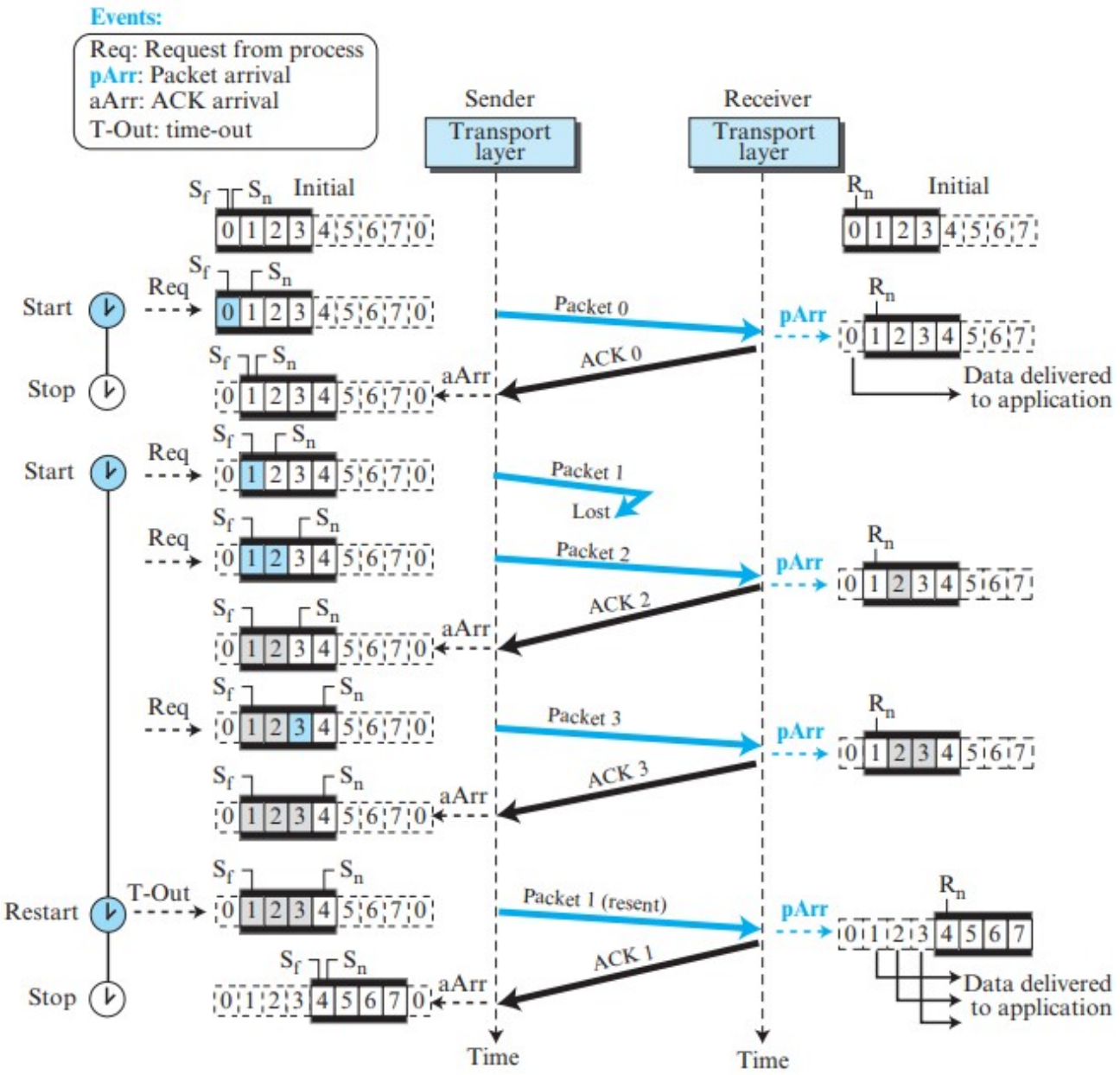
Solution

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

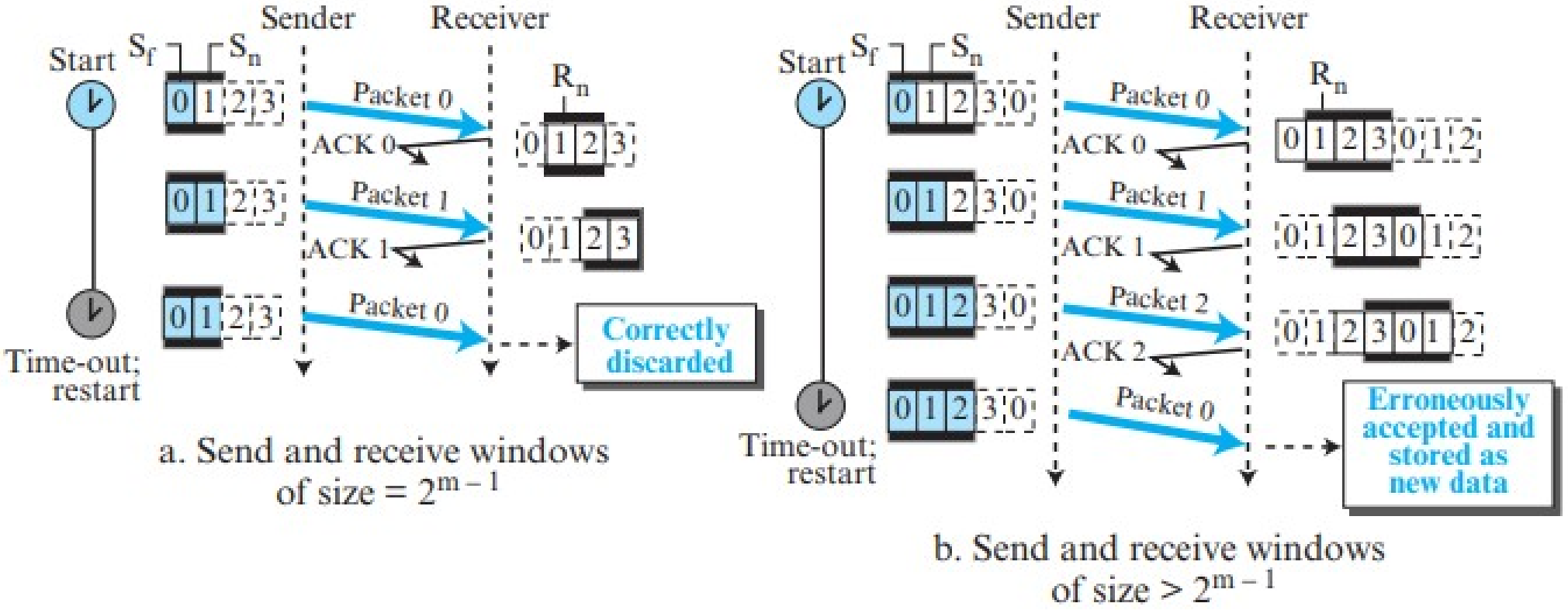
SELECTIVE-REPEAT: FSMS



EXAMPLE


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

SELECTIVE-REPEAT: WINDOW SIZE

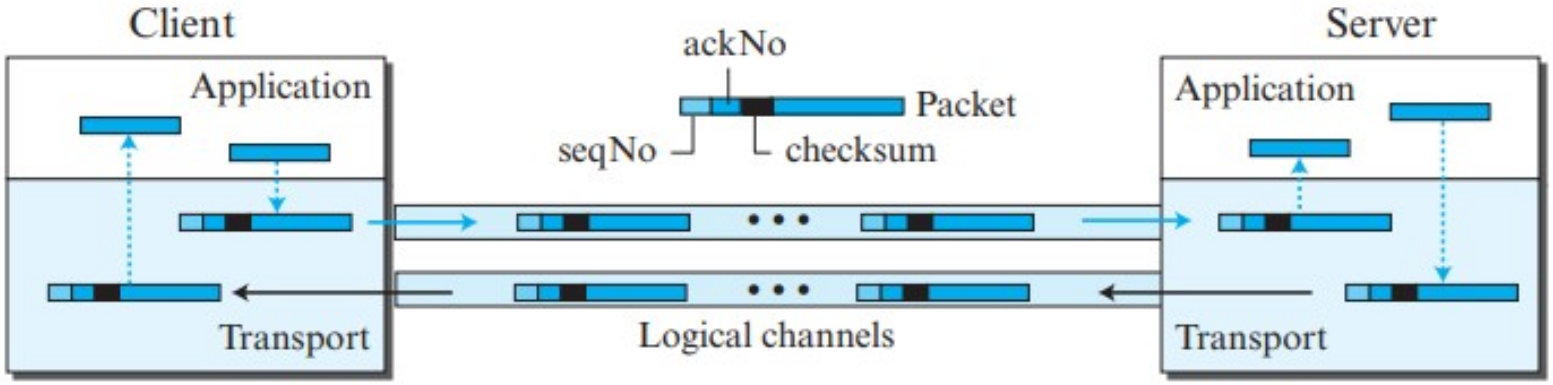


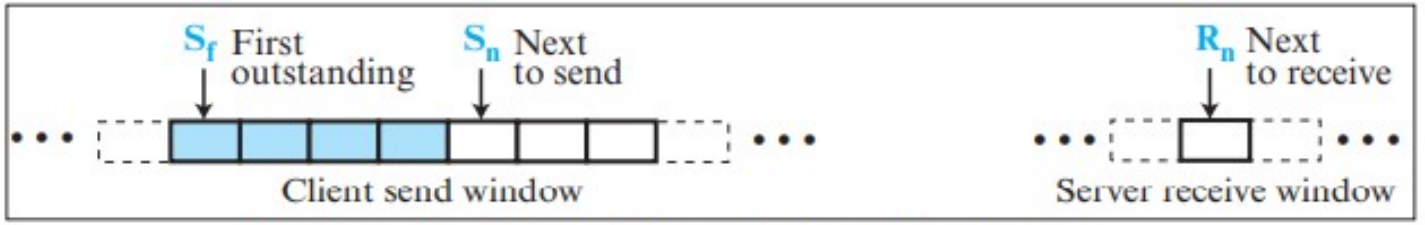
In Selective-Repeat, the size of the sender and receiver window can be at most one-half of 2^m.

BIDIRECTIONAL PROTOCOLS: PIGGYBACKING

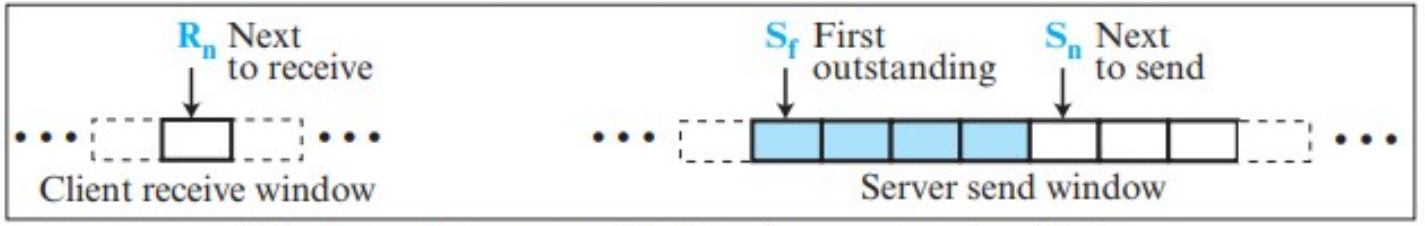
- The protocols we have discussed so far are all unidirectional or simplex protocols in which data packets flow in only one direction and acknowledgements travel in the other direction.
- In real life, data packets are normally flowing in both directions: from client to server and from server to client. 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; when a packet is carrying data from B to A, it can also carry acknowledgment feedback about the arrived packets from A.

PIGGYBACKING IN GO-BACK-N





Windows for communication from client to server



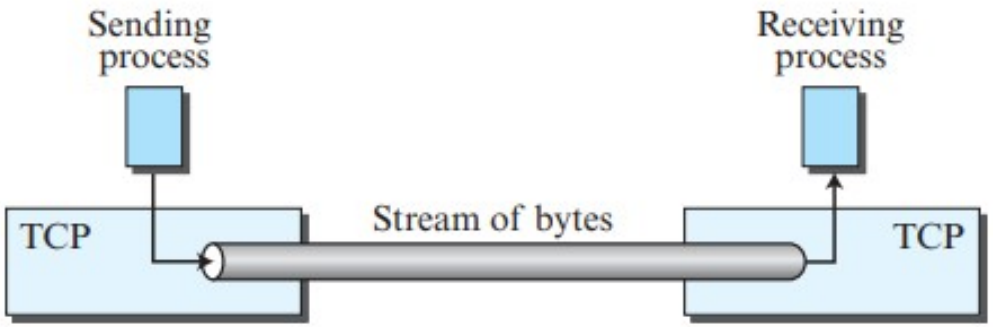
Windows for communication from server to client

TCP/IP Transport Layer Protocol: TCP

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

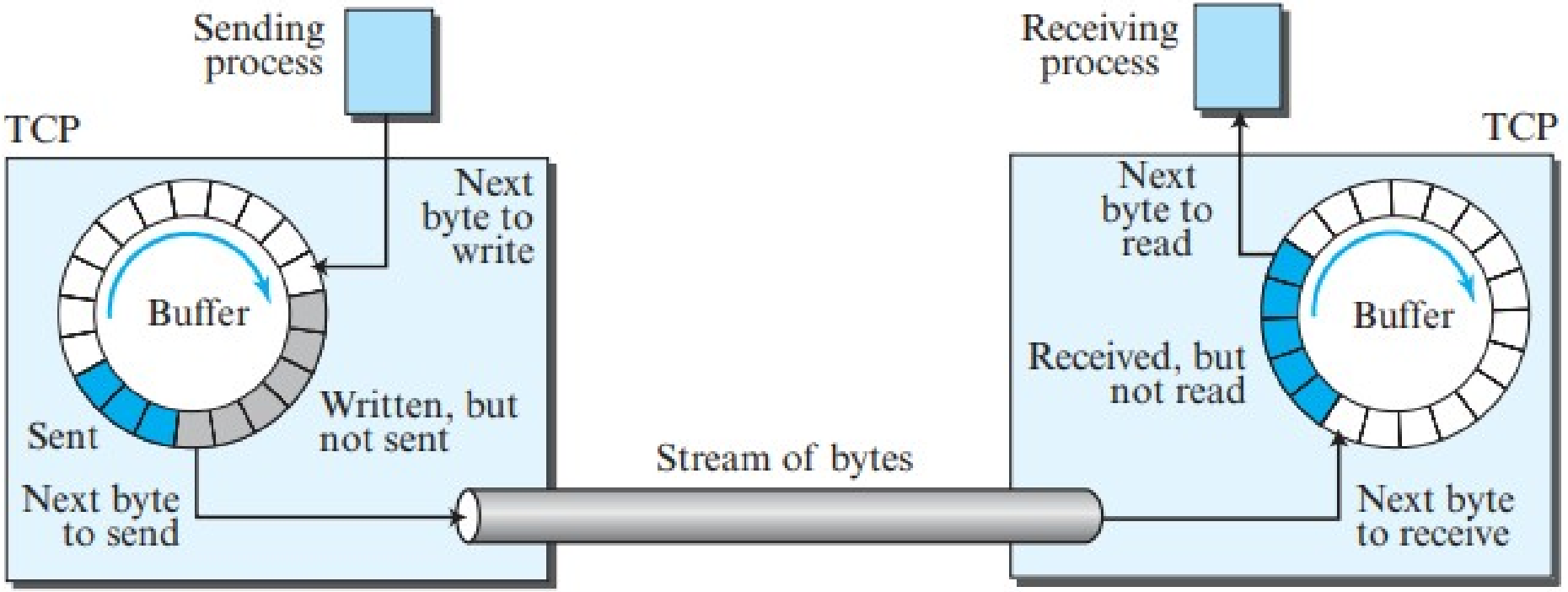
SERVICES PROVIDED BY TCP

- Process-to-Process Communication
- **Stream Delivery Service**
  - In UDP, a process sends messages with predefined boundaries to UDP for delivery. Each user datagram is treated independently by UDP and other layers of the network.
  - TCP allows the sending process to deliver data as a stream of bytes. In TCP consecutive packets might carry the same message.



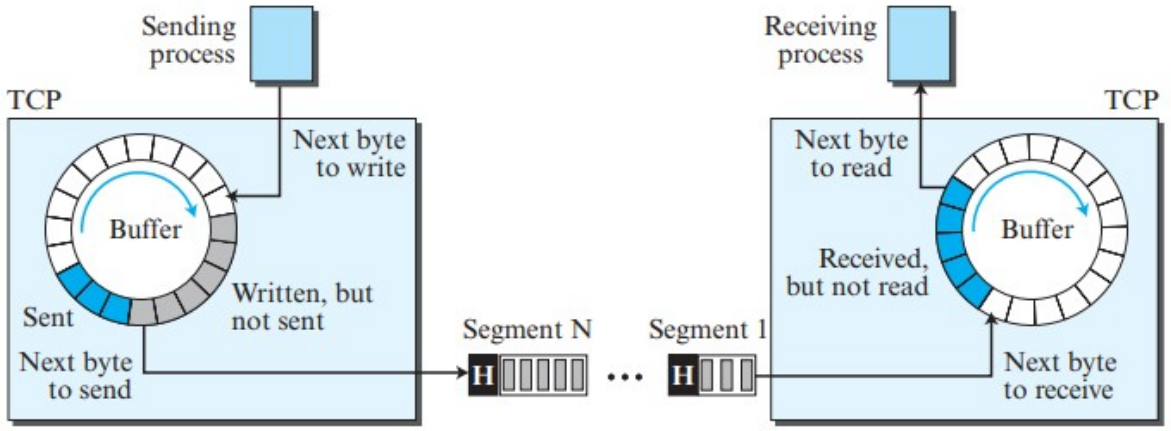
TCP Services: Sending and Receiving Buffers

- The colored area holds bytes that have been sent but not yet acknowledged. The TCP sender keeps these bytes in the buffer until it receives an acknowledgment. The shaded area contains bytes to be sent by the sending TCP.



TCP SERVICES: SEGMENTS

- The network layer, as a service provider for TCP, needs to send data in packets, not as a stream of bytes.
- At the transport layer, TCP groups a number of bytes together into a packet called a **segment**.
- TCP adds a header to each segment (for control purposes) and delivers the segment to the network layer for transmission.



TCP Services

- Full-Duplex Communication
  - TCP offers full-duplex service, where data can flow in both directions at the same time. Each TCP endpoint then has its own sending and receiving buffer, and segments move in both directions.
- Multiplexing and Demultiplexing
- Connection-Oriented Service
- Reliable Service

TCP Features: Numbering System

- Although the TCP software keeps track of the segments being transmitted or received, there is no field for a segment number value in the segment header. Instead, there are two fields, called the **sequence number** and the **acknowledgment number**. These two fields refer to **a byte number** and not a segment number.
- **Byte Number**
  - TCP numbers all data bytes (octets) that are transmitted in a connection. Numbering is independent in each direction.
  - When TCP receives bytes of data from a process, TCP stores them in the sending buffer and numbers them.
  - The numbering does not necessarily start from 0. Instead, TCP chooses an arbitrary number between 0 and $2^{32} - 1$ for the number of the first byte.
  - For example, if the arbitrary number happens to be 1,057 and the total data to be sent is 6,000 bytes, the bytes are numbered from 1,057 to 7,056.

TCP Features: Numbering System

Sequence Number

- After the bytes have been numbered, TCP assigns a sequence number to each segment that is being sent. The sequence number, in each direction, is defined as follows -
  - The sequence number of the first segment is the ISN (Initial Sequence Number), which is a random number.
  - The sequence number of any other segment is the sequence number of the previous segment plus the number of bytes carried by the previous segment.
- When a segment carries a combination of data and control information (piggybacking), it uses a sequence number. If a segment does not carry user data, it does not logically define a sequence number. The field is there, but the value is not valid.
- However, some segments, when carrying only control information, need a sequence number to allow an acknowledgment 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.

TCP Features: Numbering System

Acknowledgment Number

- Communication in TCP is full duplex; when a connection is established, both parties can send and receive data at the same time.
- The sequence number in each direction shows the number of the first byte carried by the segment.
- Each party also uses an acknowledgment number to confirm the bytes it has received. However, the acknowledgment number defines the number of the next byte that the party expects to receive.
- The term **cumulative** here means that if a party uses 5,643 as an acknowledgment number, it has received all bytes from the beginning up to 5,642.

EXAMPLE

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 1,000 bytes?

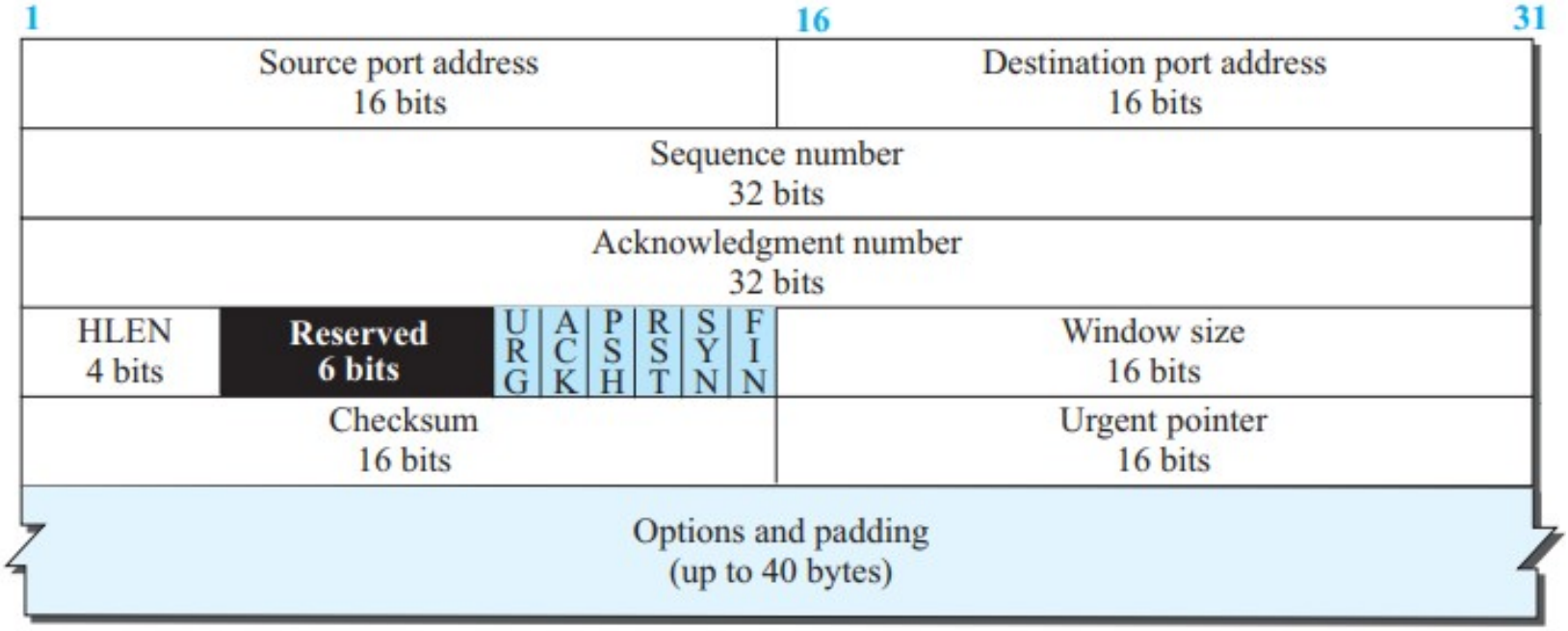
Solution

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

FORMAT OF TCP SEGMENT

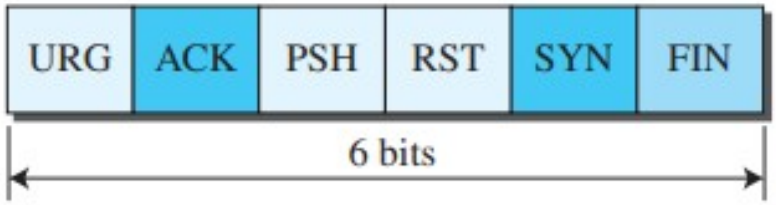
a. Segment



b. Header

FORMAT OF TCP SEGMENT

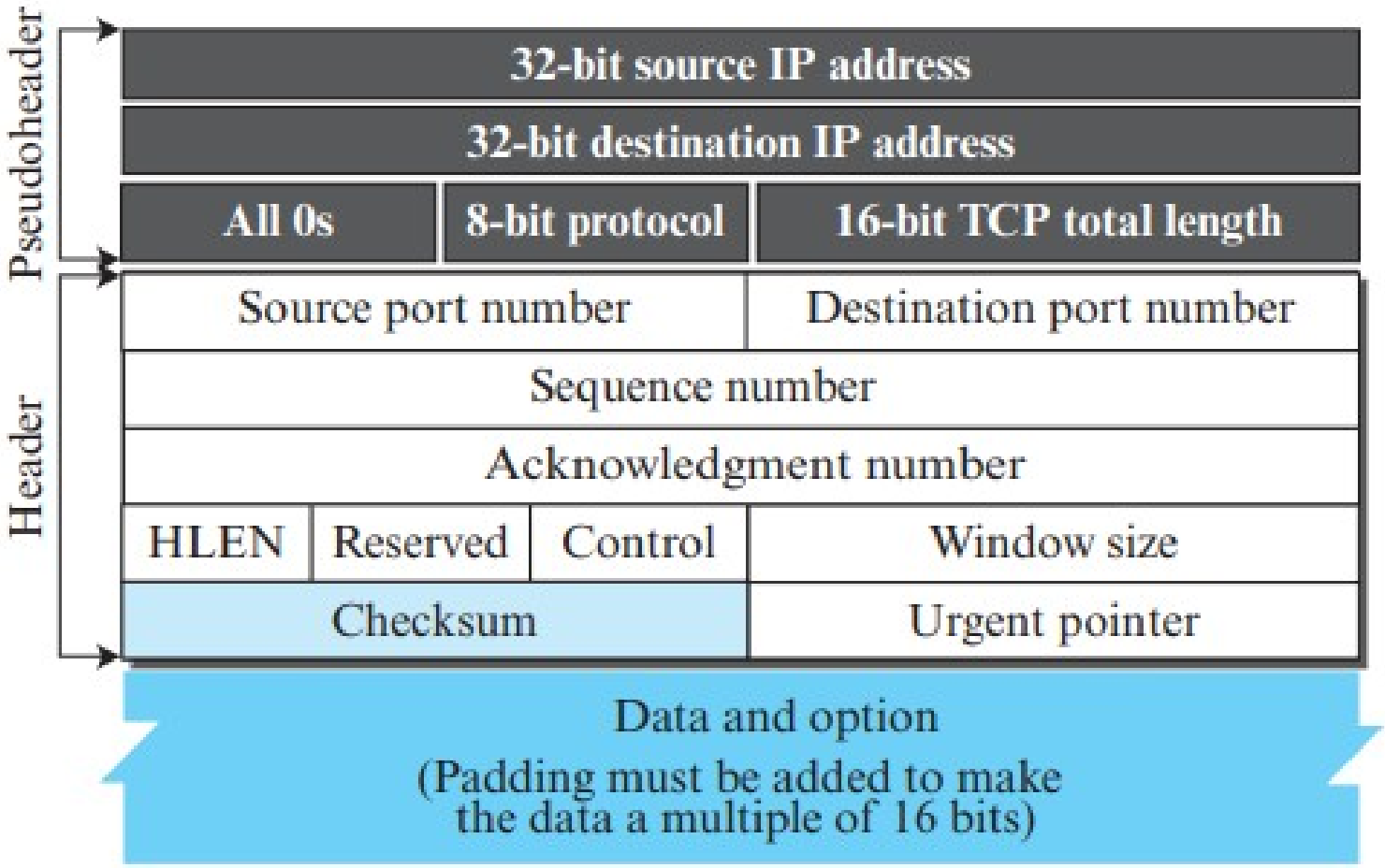
- **Header length (HLEN)** - This 4-bit field indicates the number of *4-byte words in the TCP header*. The length of the header can be between 20 and 60 bytes. Therefore, the value of this field is always between 5 (5 × 4 = 20) and 15 (15 × 4 = 60).
- **Control -**
  - This field defines 6 different control flags. One or more of these flags can be set at a time.
  - These bits enable flow control, connection establishment and termination, connection abortion and the mode of data transfer in TCP.



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

FORMAT OF TCP SEGMENT

- **Window Size**
  - This field defines the window size of the sending TCP in bytes. Note that the length of this field is 16 bits, which means that the maximum size of the window is 65,535 bytes.
  - This value is normally referred to as the receiving window and is determined by the receiver. The sender must obey the dictation of the receiver in this case.
- **Checksum**
  - This 16-bit field contains the checksum.
  - The calculation of checksum is the same as UDP.
  - In UDP checksum was optional but in TCP checksum is mandatory.



The use of the checksum in TCP is mandatory.

FORMAT OF TCP SEGMENT

- **Urgent Pointer**
  - This 16-bit field, which is valid only if the urgent flag is set, is used when the segment contains urgent data.
  - The Urgent Pointer is used when the TCP segment contains urgent data that should be processed immediately at the destination, bypassing some of the usual processing queues.
  - It defines a value that must be added to the sequence number to obtain the number of the last urgent byte in the data section of the segment i.e. *Sequence Number + Urgent Pointer* defines the last byte number which is urgent data.
- **Options**
  - There can be up to 40 bytes of optional information in the TCP header.

EXERCISE

Given a TCP header with a header length field value of 0111, calculate the actual length of the TCP header in bytes. How many bits optional data is added with the header in this case?

ANS: 0111 X 4 = 7 X 4= 28

28-20=8 bytes=64 bits

EXERCISE

In a network using the Go-Back-N protocol with m = 3 and the sending window of size 7, the values of variables are $\mathbf{S_f} = 62$, $\mathbf{S_n} = 66$, and $\mathbf{R_n} = 64$. Assume that the network does not duplicate or reorder the packets. What are the sequence numbers of data packets in transit?

ANS: Given Rn =64, this indicates the receiver is expecting a packet with sequence number 64.

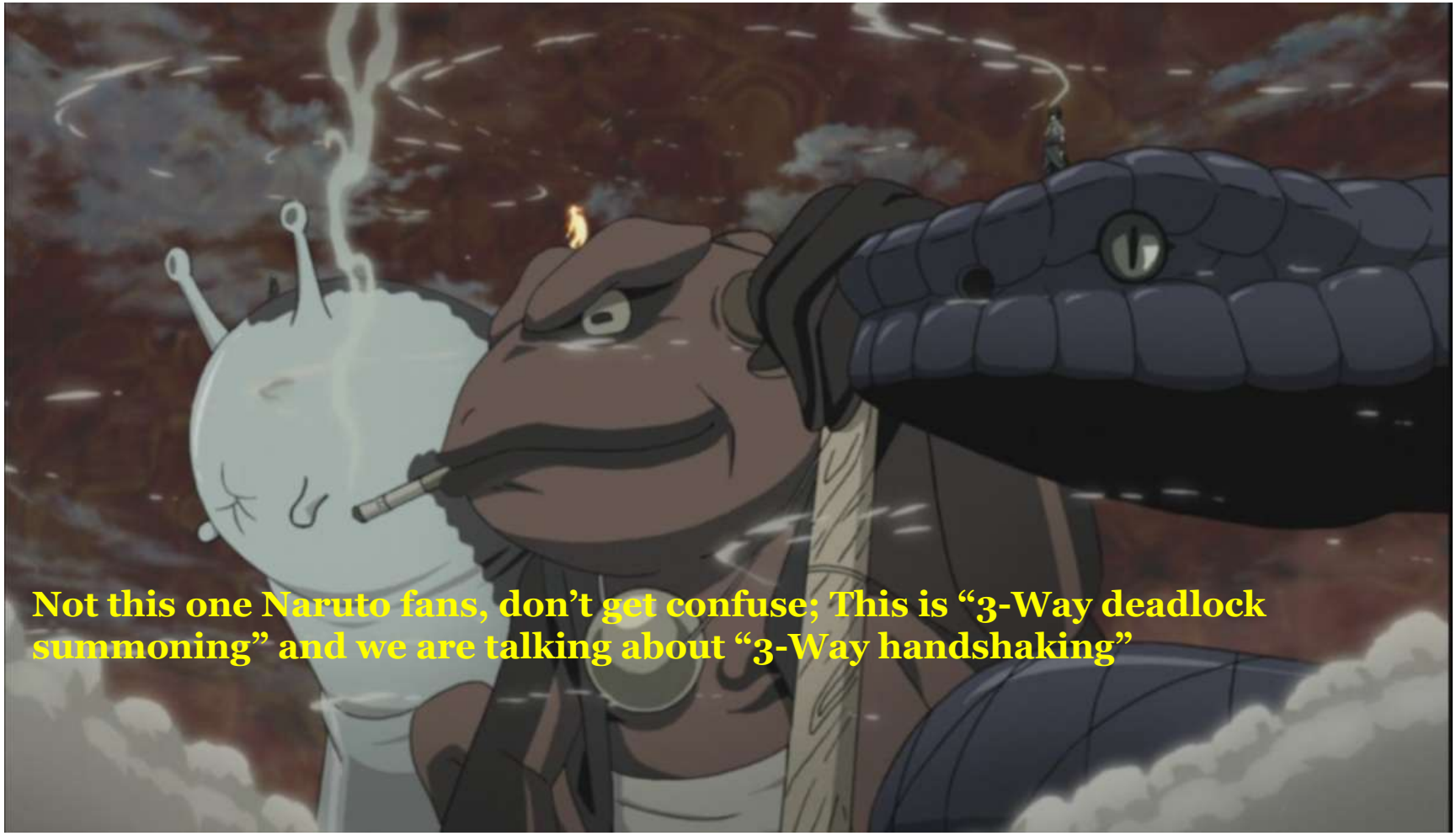
Given that Sf =62 and Sn=66, this indicates the packet from 62 to 65 are transmitting and are waiting for acknowledgment as Rn =64, the packets with sequence number 62 and 63 are already received by receiver.

The packet number 64 and 65 are in transit from Sender to receiver.

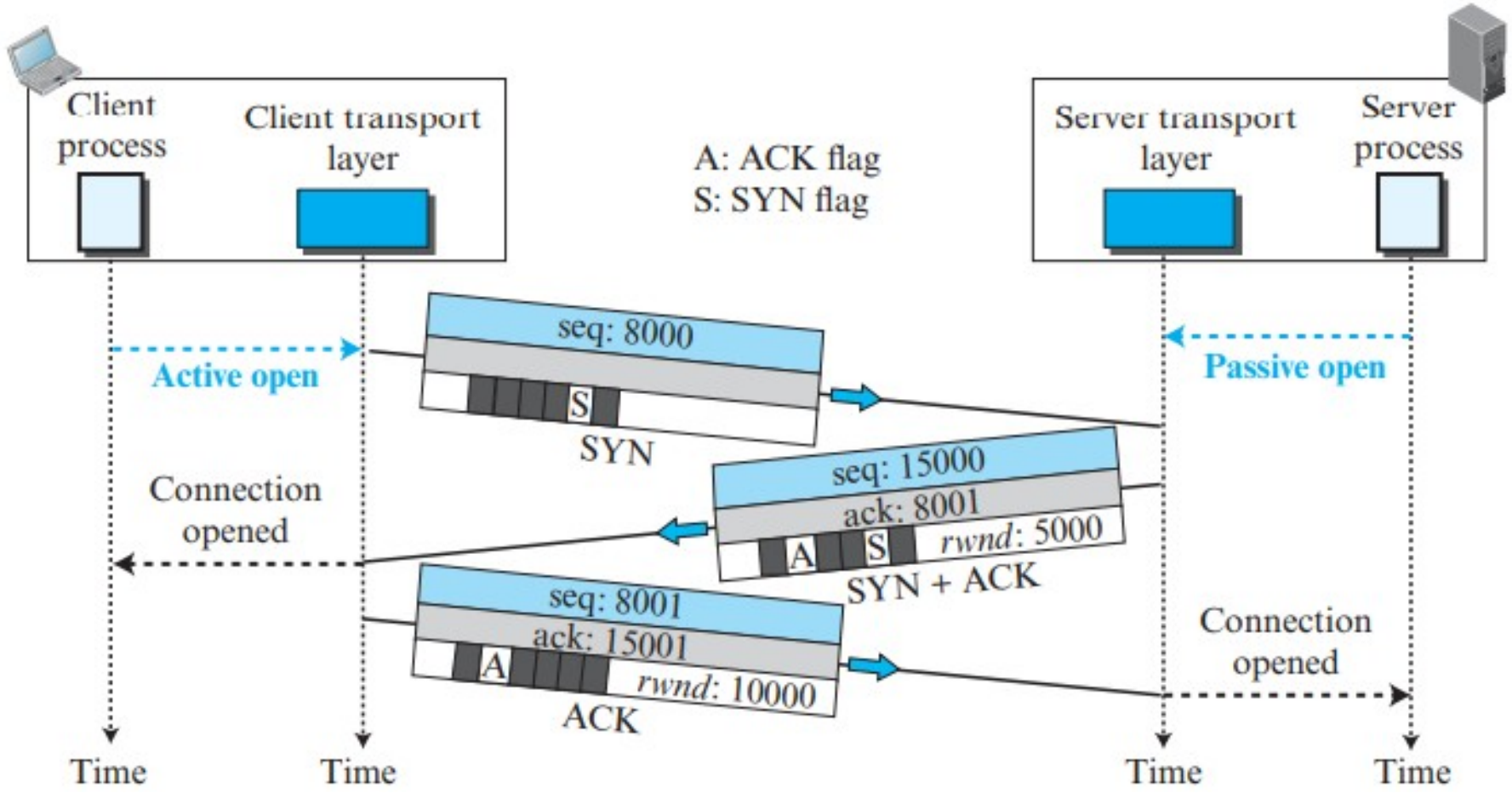
TCP Connection

- In TCP, connection-oriented transmission requires three phases: connection establishment, data transfer, and connection termination.
- **Connection Establishment**
  - The connection establishment process in TCP is called **three-way handshaking.**
  - Let's discuss the steps of connection establishment between a client and a server.
  - The process starts with the server. The server program tells its TCP that it is ready to accept a connection. This request is called a *passive open*. Although the server TCP is ready to accept a connection from any machine in the world, it cannot make the connection itself.
  - The client program issues a request for an *active open*. A client that wishes to connect to an open server tells its TCP to connect to a particular server.

3-WAY HANDSHAKING

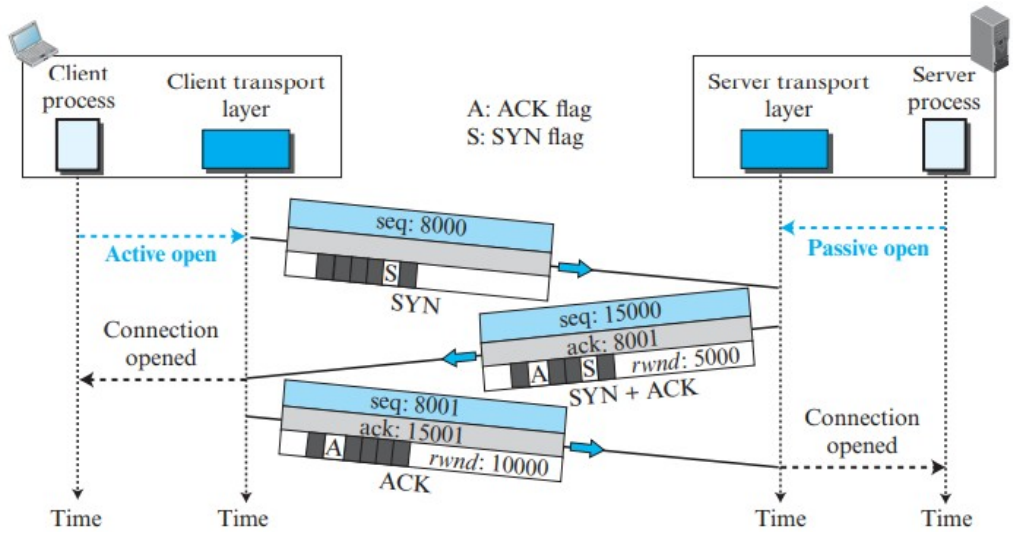


TCP Connection Establishment: 3-way Handshaking



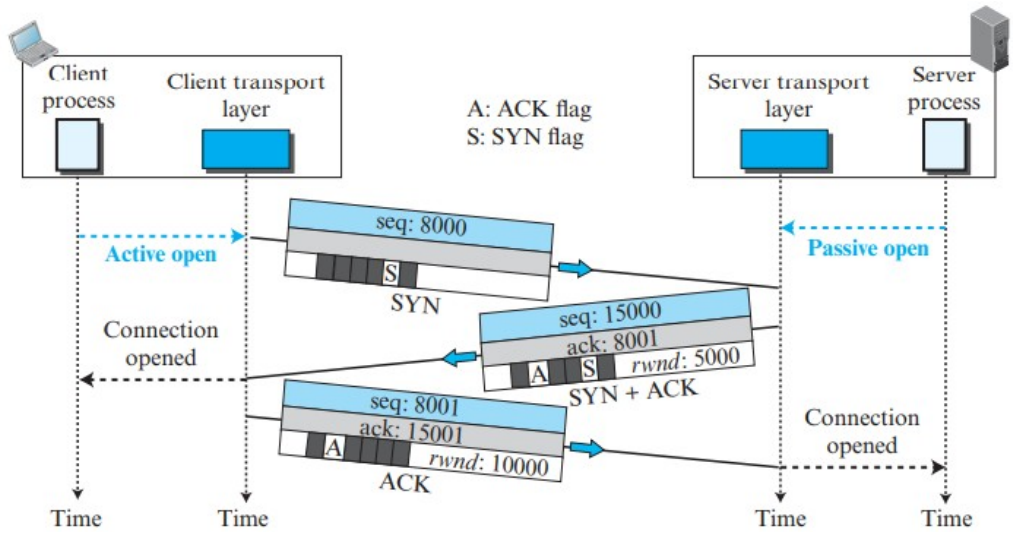
TCP Connection Establishment: 3-way Handshaking (Step 1)

- The client sends the first segment, a SYN segment, in which only the SYN flag is set. This segment is for synchronization of sequence numbers. The client in our example chooses a random number, 8000, as the first sequence number and sends this number to the server. This sequence number is called the **initial sequence number (ISN)**.
- Note that this segment does not contain an acknowledgment number. It does not define the window size either; a window size definition makes sense only when a segment includes an acknowledgment.
- *Note that the SYN segment is a control segment and carries no data. However, it consumes one sequence number because it needs to be acknowledged.*



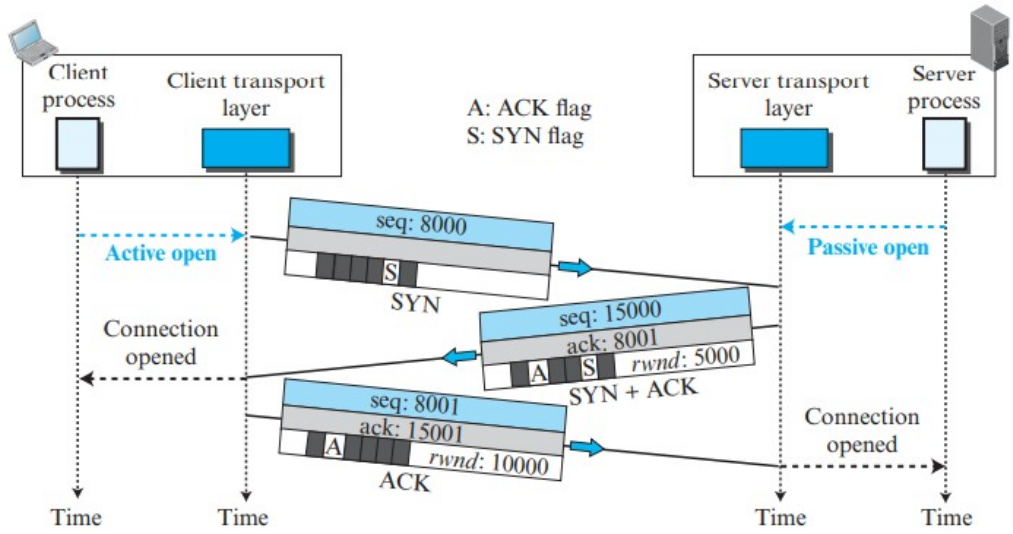
TCP Connection Establishment: 3-way Handshaking (Step 2)

- The server sends the second segment, a SYN + ACK segment with two flag bits set as: SYN and ACK. This segment has a dual purpose.
- First, it is a SYN segment for communication in the other direction. The server uses this segment to initialize a sequence number for numbering the bytes sent from the server to the client.
- The server also acknowledges the receipt of the SYN segment from the client by setting the ACK flag and displaying the next sequence number it expects to receive from the client. Because it contains an acknowledgment, it also needs to define the receive window size, ***rwnd*** (to be used by the client), as we will see later when we will discuss about the flow control in TCP.
- Since this segment is playing the role of a SYN segment, it needs to be acknowledged. It, therefore, consumes one sequence number.



TCP Connection Establishment: 3-way Handshaking (Step 3)

- The client sends the third segment. This is just an ACK segment. It acknowledges the receipt of the second segment with the ACK flag and acknowledgment number field.
- Note that the ACK segment does not consume any sequence numbers if it does not carry data.
- Some implementations allow this third segment in the connection phase to carry the first chunk of data from the client. In this case, the segment consumes as many sequence numbers as the number of data bytes.



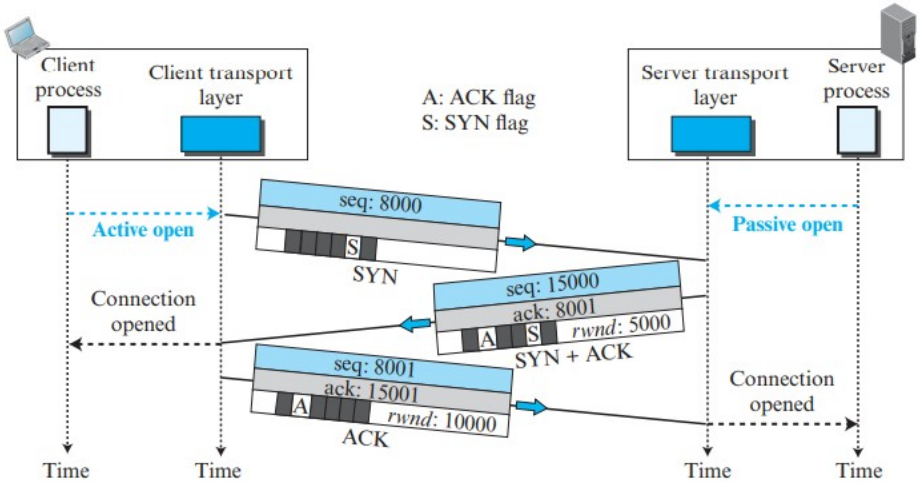
SYN Flooding Attack

- The connection establishment procedure in TCP is susceptible to a serious security problem called the **SYN flooding attack**.
- This happens when a malicious attacker sends a large number of SYN segments to a server, pretending that each of them is coming from a different client by faking the source IP addresses in the datagrams.
- The server, assuming that the clients are issuing an active open, allocates the necessary resources, such as creating communication tables and setting timers.
- The TCP server then sends the SYN +ACK segments to the fake clients, which are lost. During this time, however, a lot of resources are occupied without being used.
- If, during this short time, the number of SYN segments is large, the server eventually runs out of resources and may crash.
- This SYN flooding attack belongs to a type of security attack known as a ***denial-of-service attack (DoS Attack)***, in which an attacker monopolizes a system with so many service requests that the system collapses and denies service to every request.

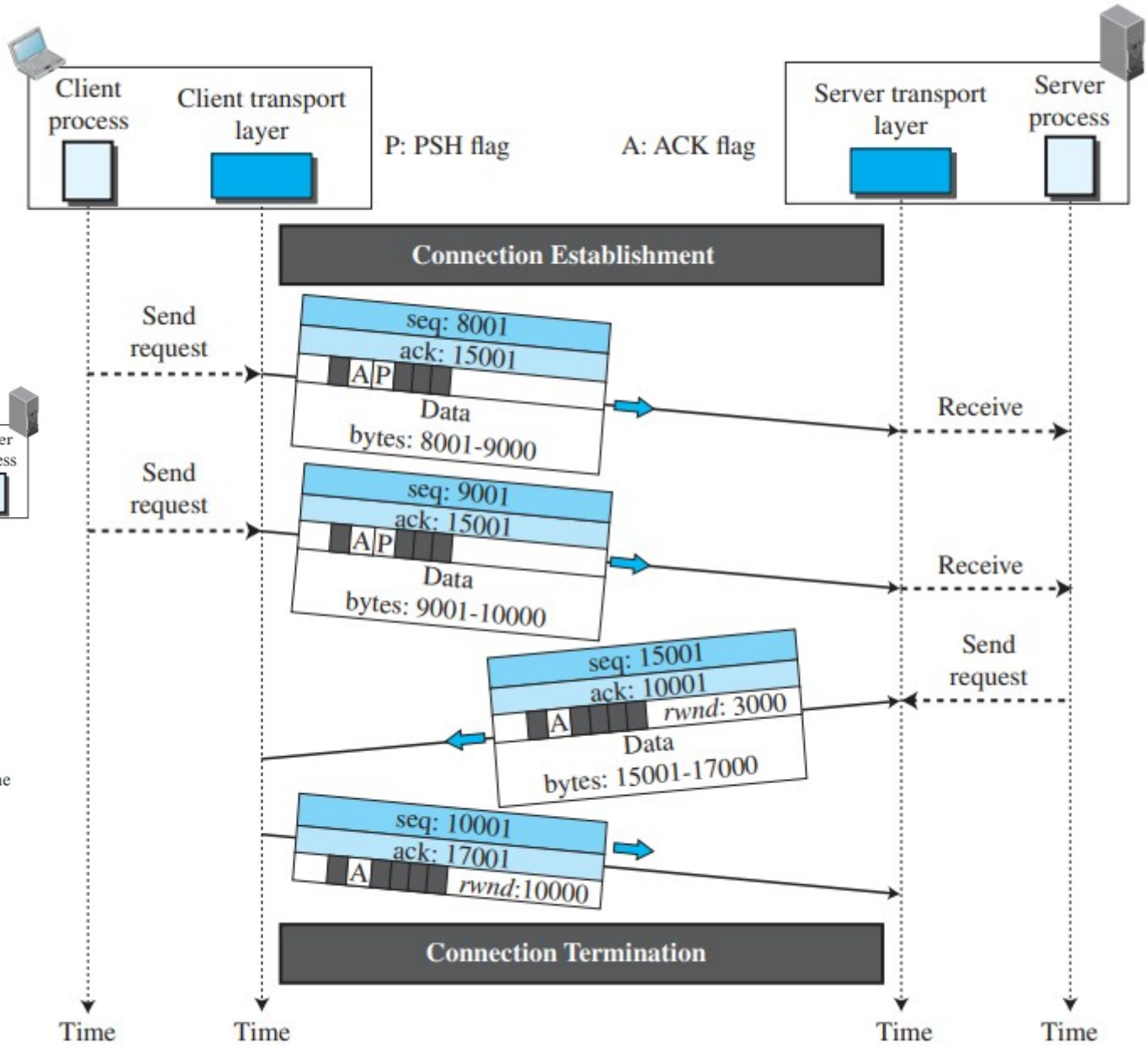
SYN Flooding Attack Prevention

- Some implementations of TCP have strategies to alleviate the effect of a SYN attack.
- Some have imposed a limit of connection requests during a specified period of time. Others try to filter out datagrams coming from unwanted source addresses.
- One recent strategy is to postpone resource allocation until the server can verify that the connection request is coming from a valid IP address, by using what is called a **cookie**.

TCP CONNECTION: DATA TRANSFER



TCP Connection Establishment



TCP Connection: Data Transfer

Pushing Data

- The application program at the sender can request a push operation. This means that the sending TCP must not wait for the window to be filled. It must create a segment of corresponding byte and send it immediately.
- The sender can enable **PSH Flag** in TCP header of the segment. This flag informs receiver that this data should be delivered to application layer as soon as possible.

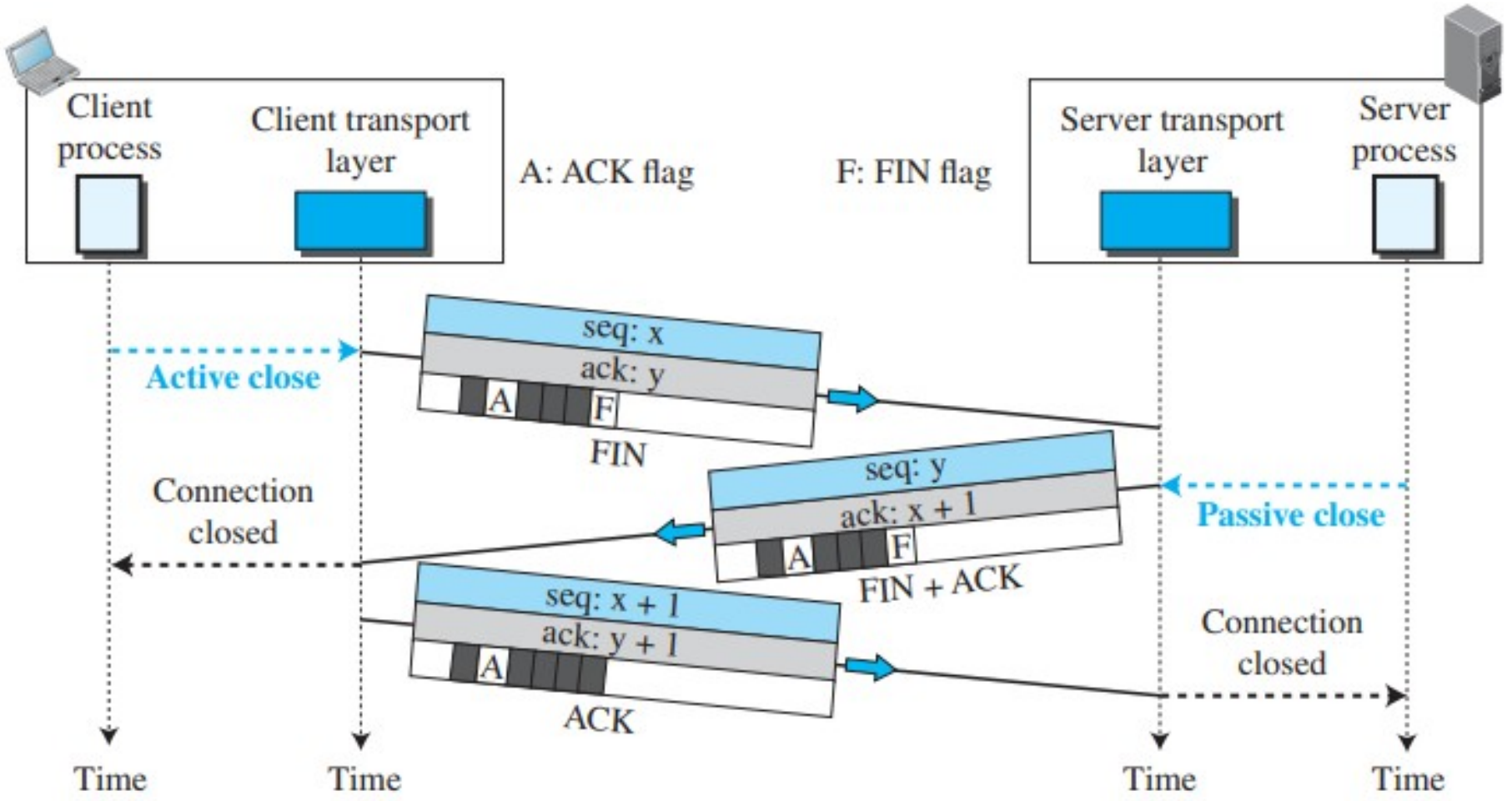
Urgent Data

- As we know that TCP provides numbers to each byte and bytes are delivered to application layer in the order. However on some occasions sender might wanna send *urgent bytes*.
- The sending application program tells the sending TCP that the piece of data is urgent. The sending TCP creates a segment and inserts the urgent data at the beginning of the segment. The rest of the segment can contain normal data from the buffer. The urgent pointer field in the header defines the end of the urgent data and the start of normal data.
- When the receiving TCP receives a segment with the **URG** bit set, it extracts the urgent data from the segment, using the value of the urgent pointer, and delivers them, out of order, to the receiving application program.

TCP Connection: Connection Termination

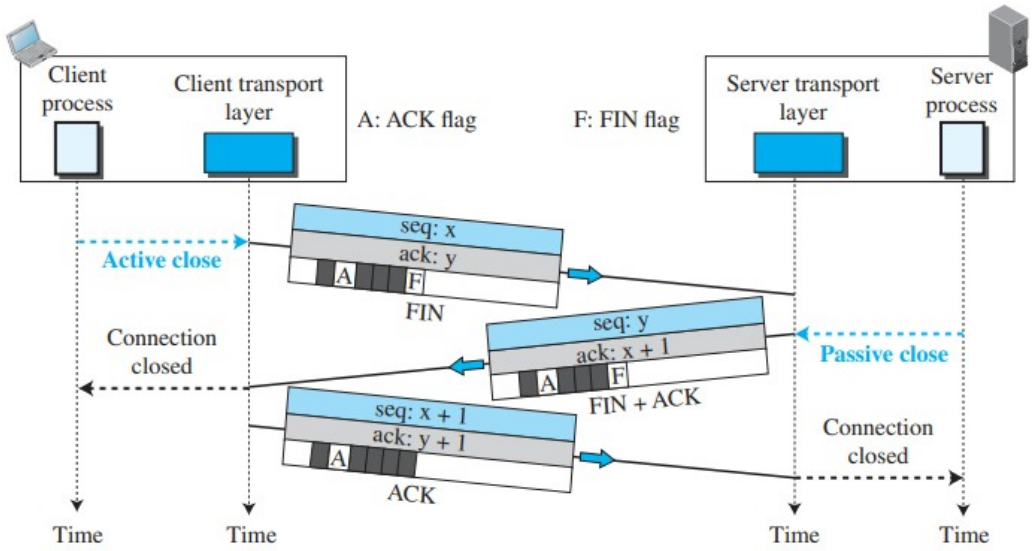
- Either of the two parties involved in exchanging data (client or server) can close the connection, although it is usually initiated by the client.
- Most implementations today allow two options for connection termination -
  - three-way handshaking
  - four-way handshaking with a half-close option.

CONNECTION TERMINATION: 3-WAY HANDSHAKING



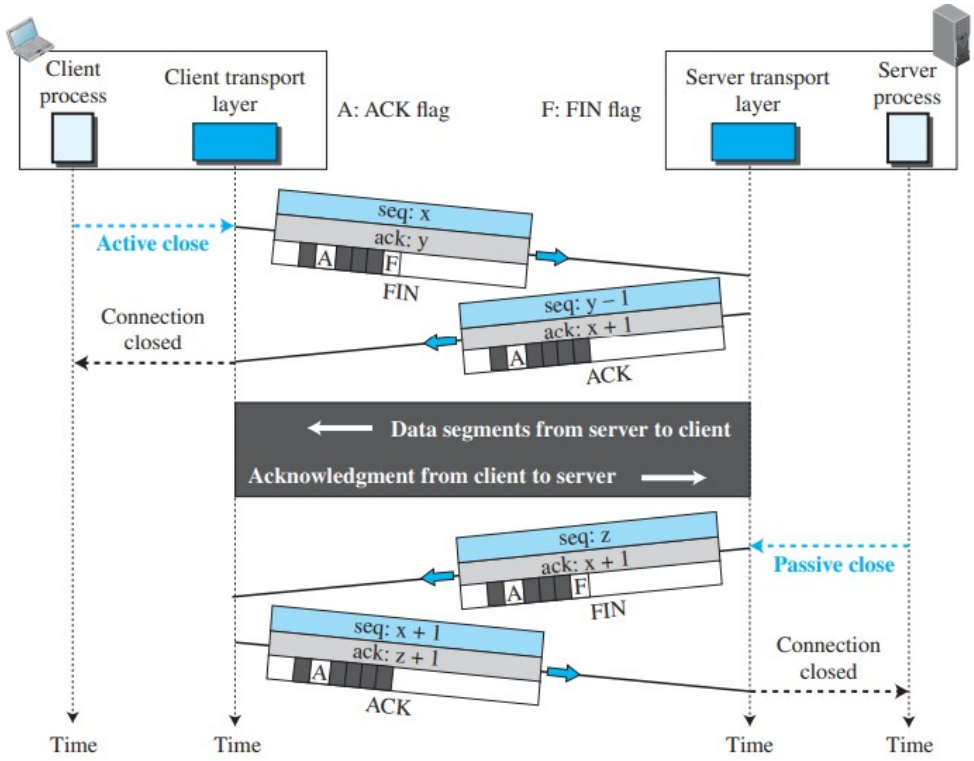
Connection Termination: 3-way Handshaking

- In this situation, the client TCP, after receiving a close command from the client process, sends the first segment, a FIN segment in which the FIN flag is set. Note that a FIN segment can include the last chunk of data sent by the client or it can be just a control segment as shown in the figure. If it is only a control segment, it consumes only one sequence number because it needs to be acknowledged.
- The server TCP, after receiving the FIN segment, informs its process of the situation and sends the second segment, a FIN+ACK segment, to confirm the receipt of the FIN segment from the client and at the same time to announce the closing of the connection in the other direction. This segment can also contain the last chunk of data from the server. If it does not carry data, it consumes only one sequence number because it needs to be acknowledged.
- The client TCP sends the last segment, an ACK segment, to confirm the receipt of the FIN segment from the TCP server. This segment contains the acknowledgment number, which is one plus the sequence number received in the FIN segment from the server. This segment cannot carry data and consumes no sequence numbers.

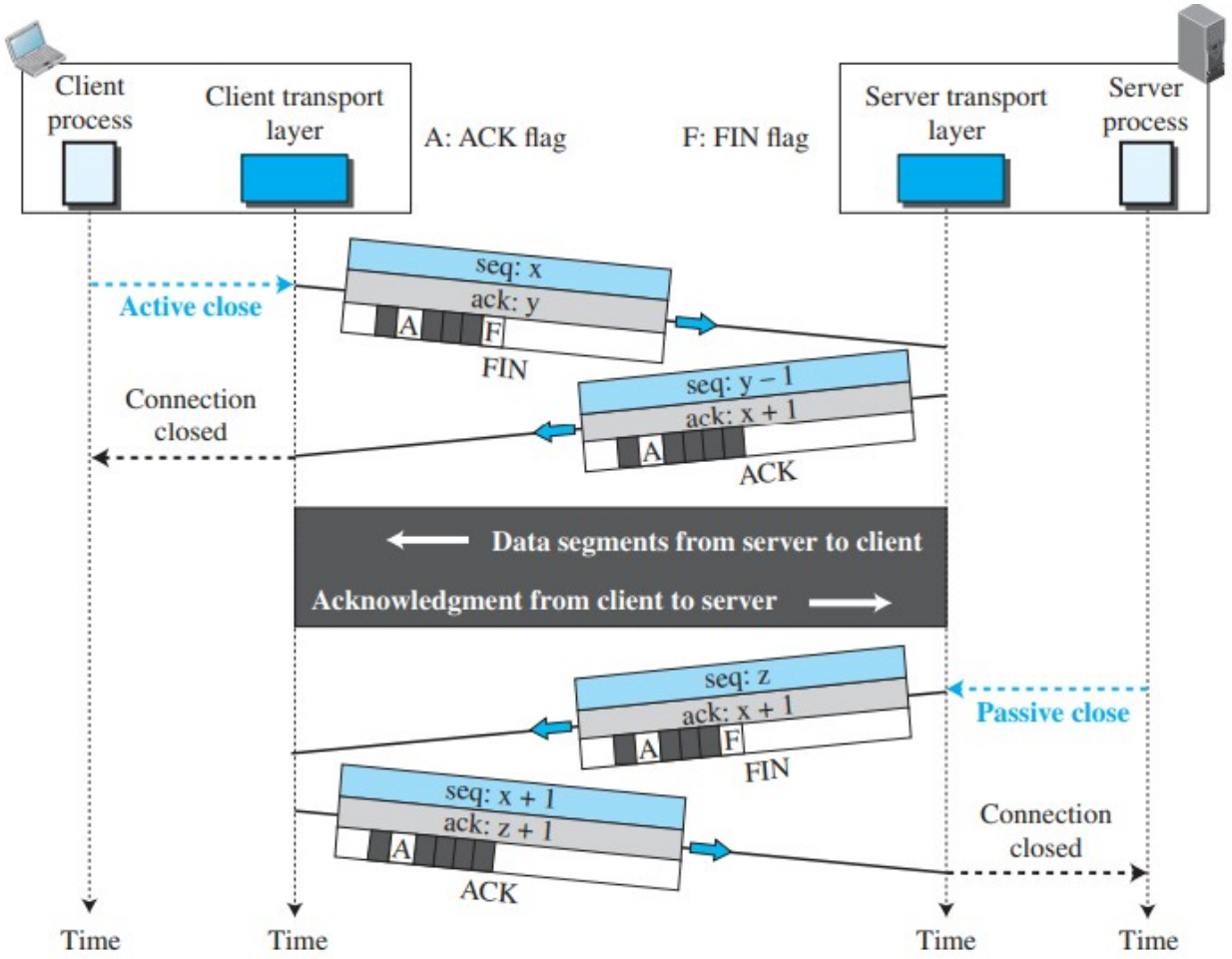


Connection Termination: Half-Close

- 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 sends data to the server to be sorted, the server needs to receive all the data before sorting can start. This means the client, after sending all data, can close the connection in the client-to-server direction. However, the server-to-client direction must remain open to return the sorted data.
- The client half-closes the connection by sending a FIN segment. The server accepts the half-close by sending the ACK segment. The server, however, can still send data. When the server has sent all of the processed data, it sends a FIN segment, which is acknowledged by an ACK from the client.

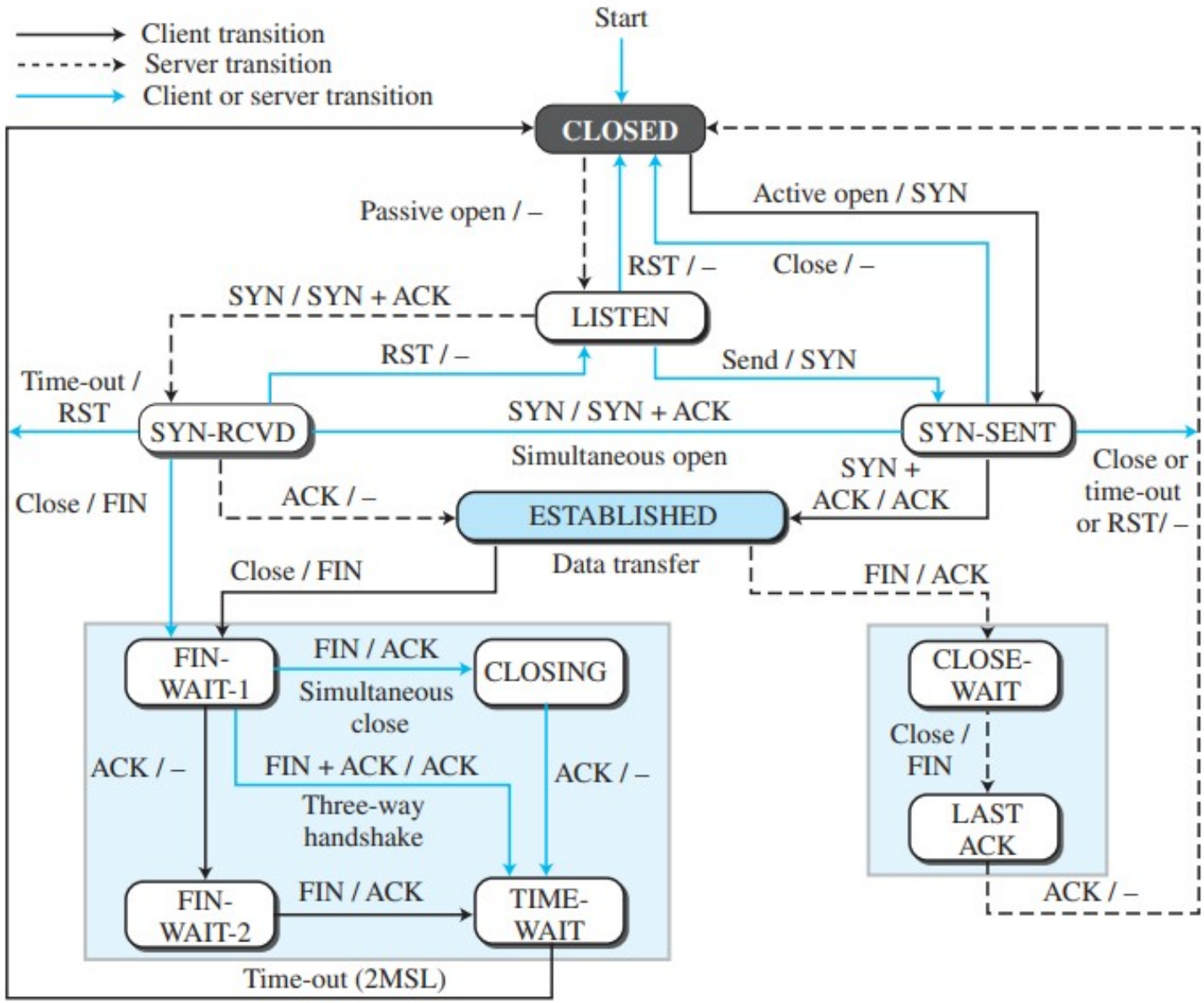


CONNECTION TERMINATION: HALF-CLOSE



State Transition Diagram

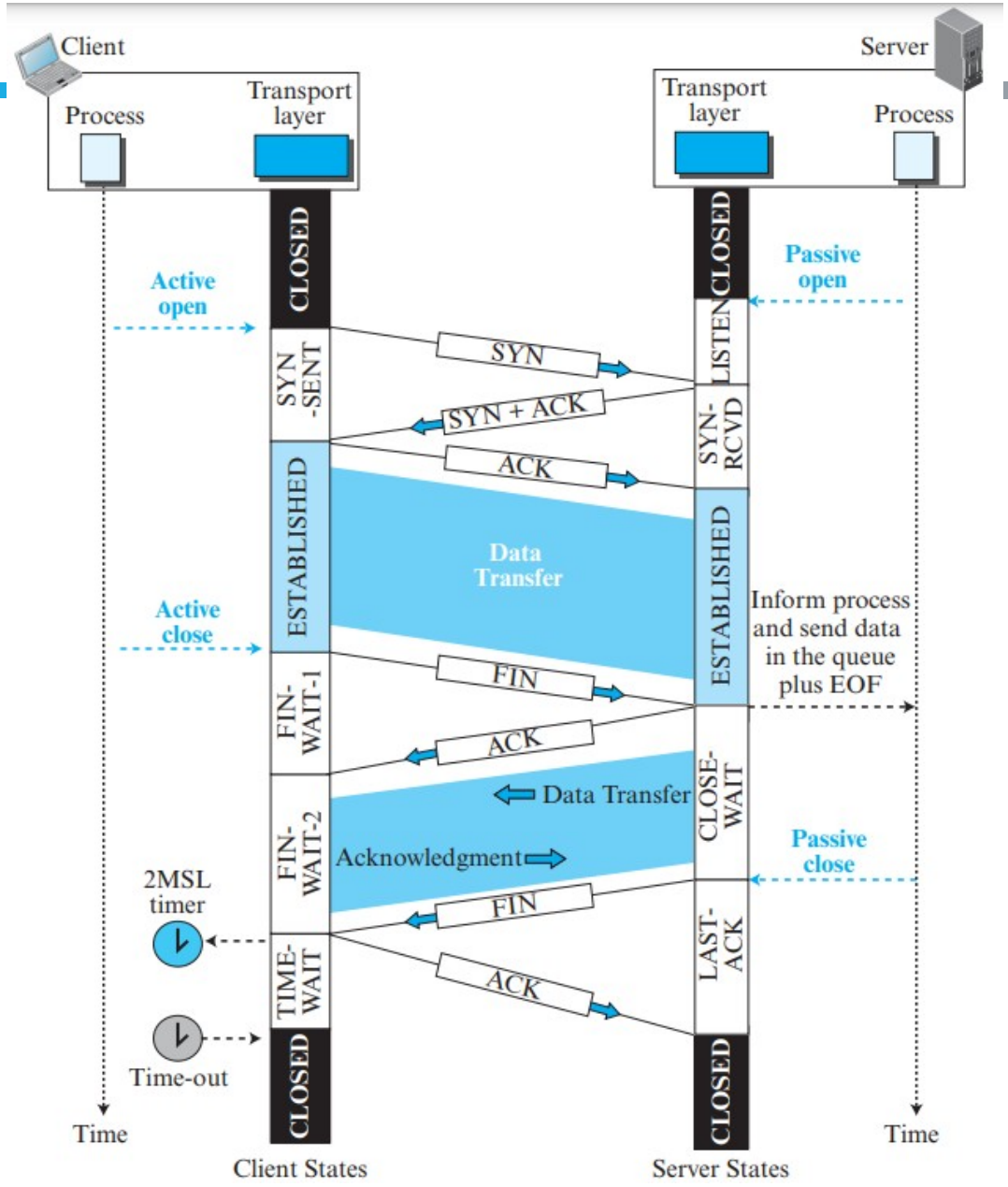
- States are represented using rounded-corner rectangles.
- Each transition has two strings written above the line separated by a slash.
- First string represents the input received by the TCP in that state.
- Second string represents the output sent by the TCP. If no output is sent then it's shown by - symbol.
- MSL (Maximum Segment Lifetime) is the maximum amount of time that a TCP segment can exist in the network before being discarded.



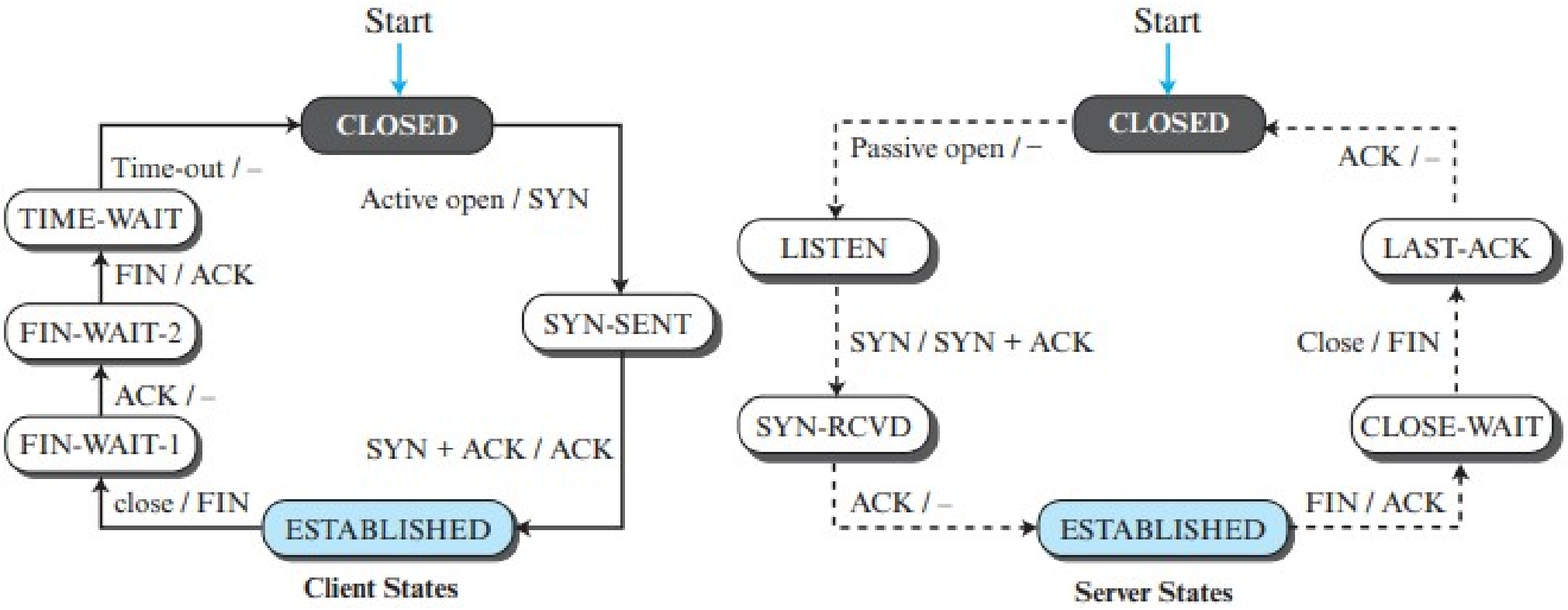
STATES FOR TCP

State	*Description*
CLOSED	No connection exists
LISTEN	Passive open received; waiting for SYN
SYN-SENT	SYN sent; waiting for ACK
SYN-RCVD	SYN+ACK sent; waiting for ACK
ESTABLISHED	Connection established; data transfer in progress
FIN-WAIT-1	First FIN sent; waiting for ACK
FIN-WAIT-2	ACK to first FIN received; waiting for second FIN
CLOSE-WAIT	First FIN received, ACK sent; waiting for application to close
TIME-WAIT	Second FIN received, ACK sent; waiting for 2MSL time-out
LAST-ACK	Second FIN sent; waiting for ACK
CLOSING	Both sides decided to close simultaneously

TIME-LINE DIAGRAM FOR A COMMON SCENARIO



HALF-CLOSE SCENARIO



WINDOWS IN TCP

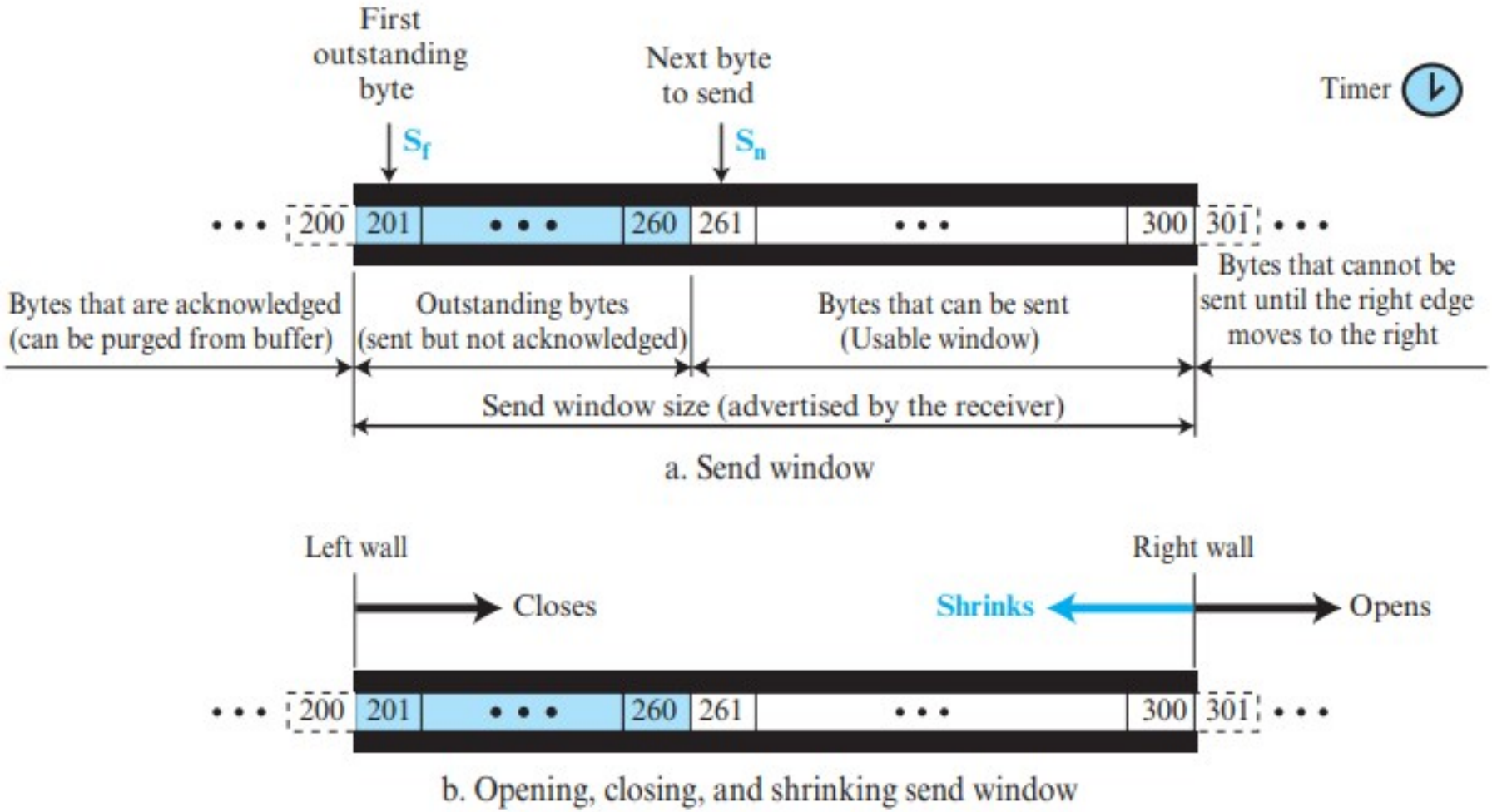
- TCP uses two windows (send window and receive window) for each direction of data transfer, which means four windows for a bidirectional communication.
- To make the discussion simple, we make an unrealistic assumption that communication is only unidirectional (say from client to server); the bidirectional communication can be inferred using two unidirectional communications with piggybacking.

WINDOWS IN TCP

Send Window

- The window size is 100 bytes but can be changed depending receiver's flow control capacity and network's congestion control capacity.
- The send window in TCP is same as send window in selective-repeat protocol with few differences -
  - The window in SR stores packets but in case TCP window stores bytes.
  - As we have discussed that theoretical Selective-Repeat protocol may use a separate timer for each packet sent but the TCP only uses one timer.

Send Window in TCP

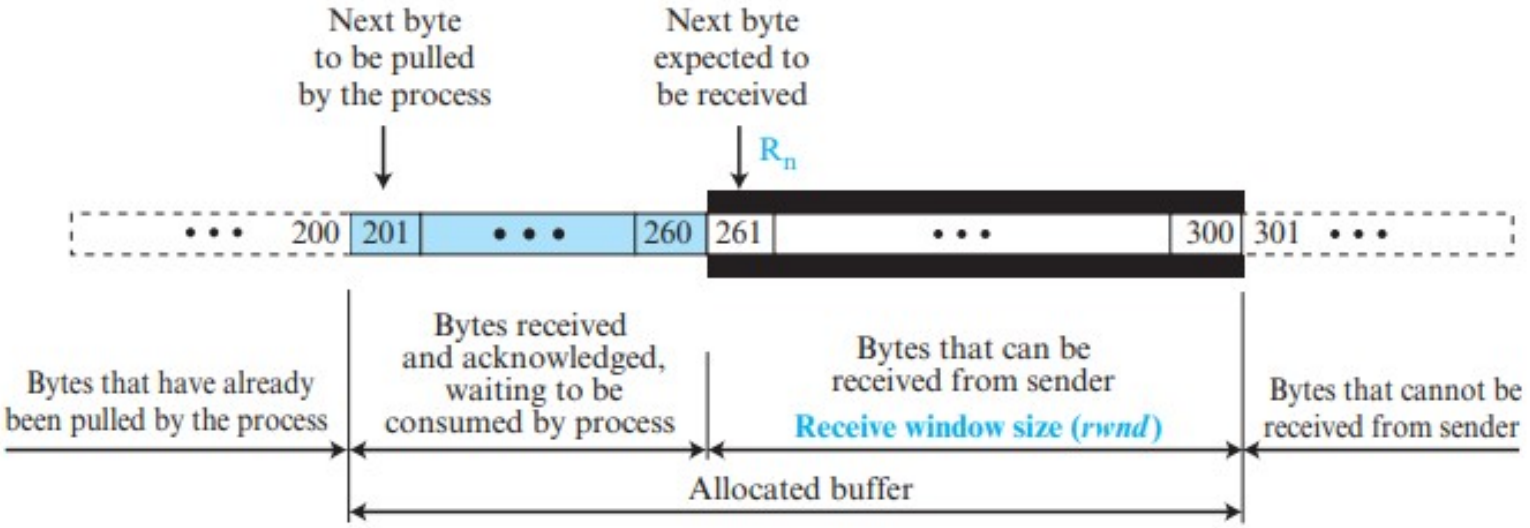


WINDOWS IN TCP

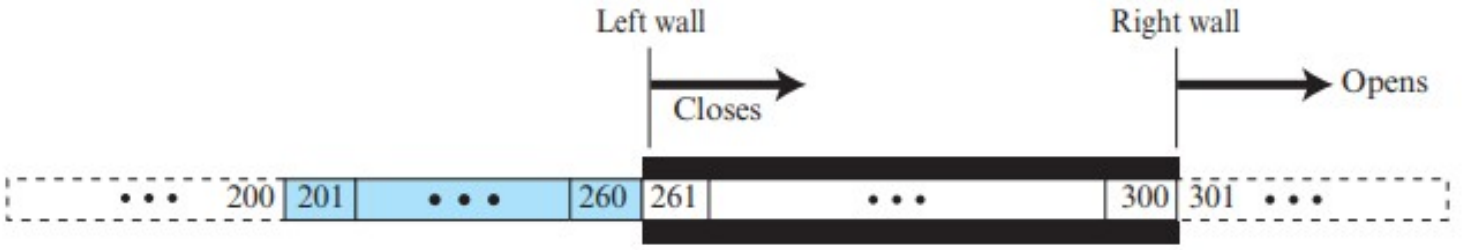
Receive Window

- In TCP, Receive window size is 100 bytes same as the send window. Receive window also stores bytes instead of packets.
- Few differences from receive window in selective-repeat -
  - The first difference is that TCP allows the receiving process to pull data at its own pace. This means that part of the allocated buffer at the receiver may be occupied by bytes that have been received and acknowledged, but are waiting to be pulled by the receiving process. The receive window size is then always smaller or equal to the buffer size. The receive window size determines the number of bytes that the receive window can accept from the sender before being overwhelmed (flow control).
  - The second difference is the way acknowledgments are used in the TCP protocol. Remember that an acknowledgement in SR is selective, defining the uncorrupted packets that have been received. The major acknowledgment mechanism in TCP is a cumulative acknowledgment announcing the next expected byte to receive (in this way TCP looks like GBN).

Receive Window in TCP



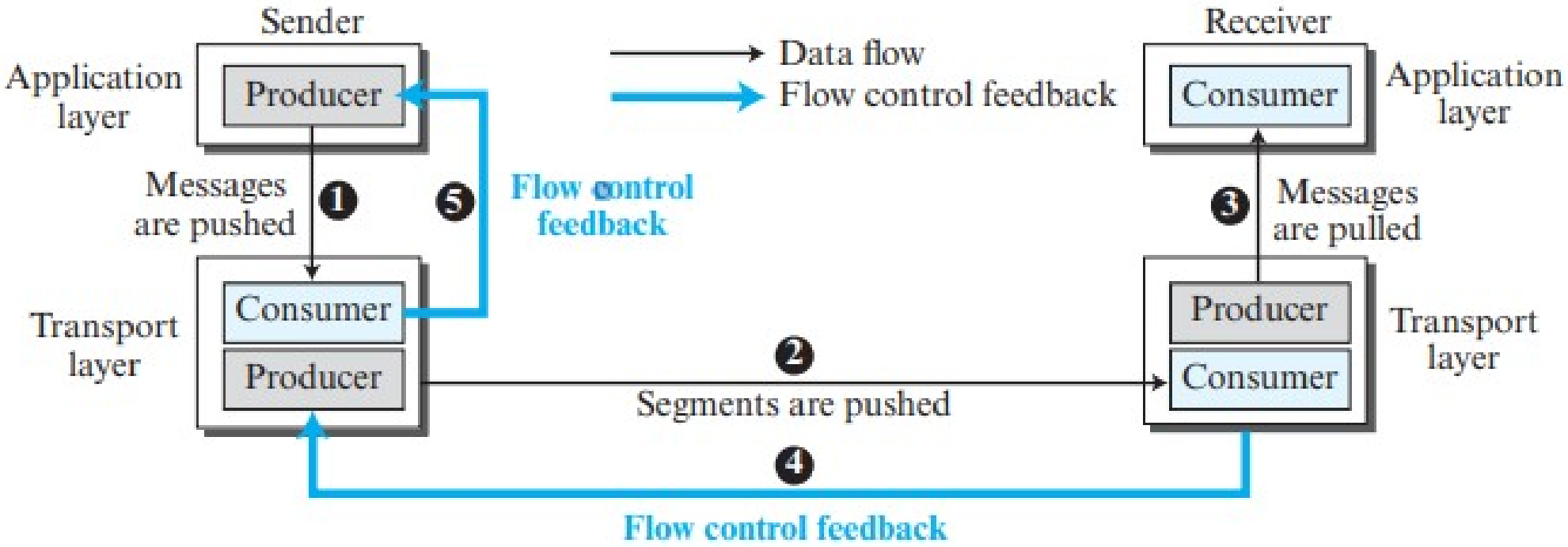
a. Receive window and allocated buffer



b. Opening and closing of receive window

rwnd = buffer size - number of bytes waiting to be pulled by receiver process

FLOW CONTROL IN TCP

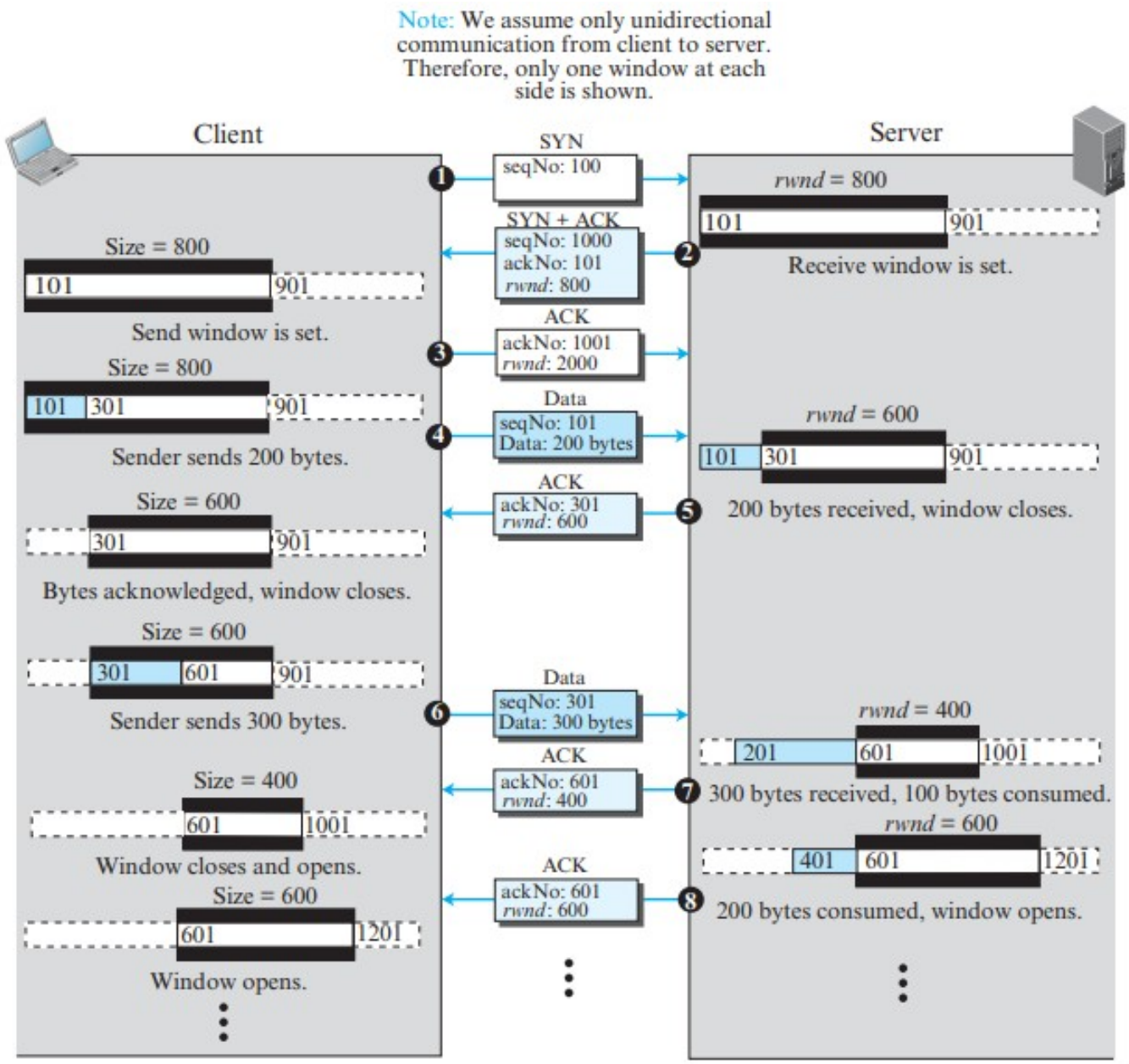


Flow control feedback from the sending TCP to the sending process (path 5) is achieved through simple rejection of data by the sending TCP when its window is full. This means that our discussion of flow control concentrates on the feedback sent from the receiving TCP to the sending TCP (path 4).

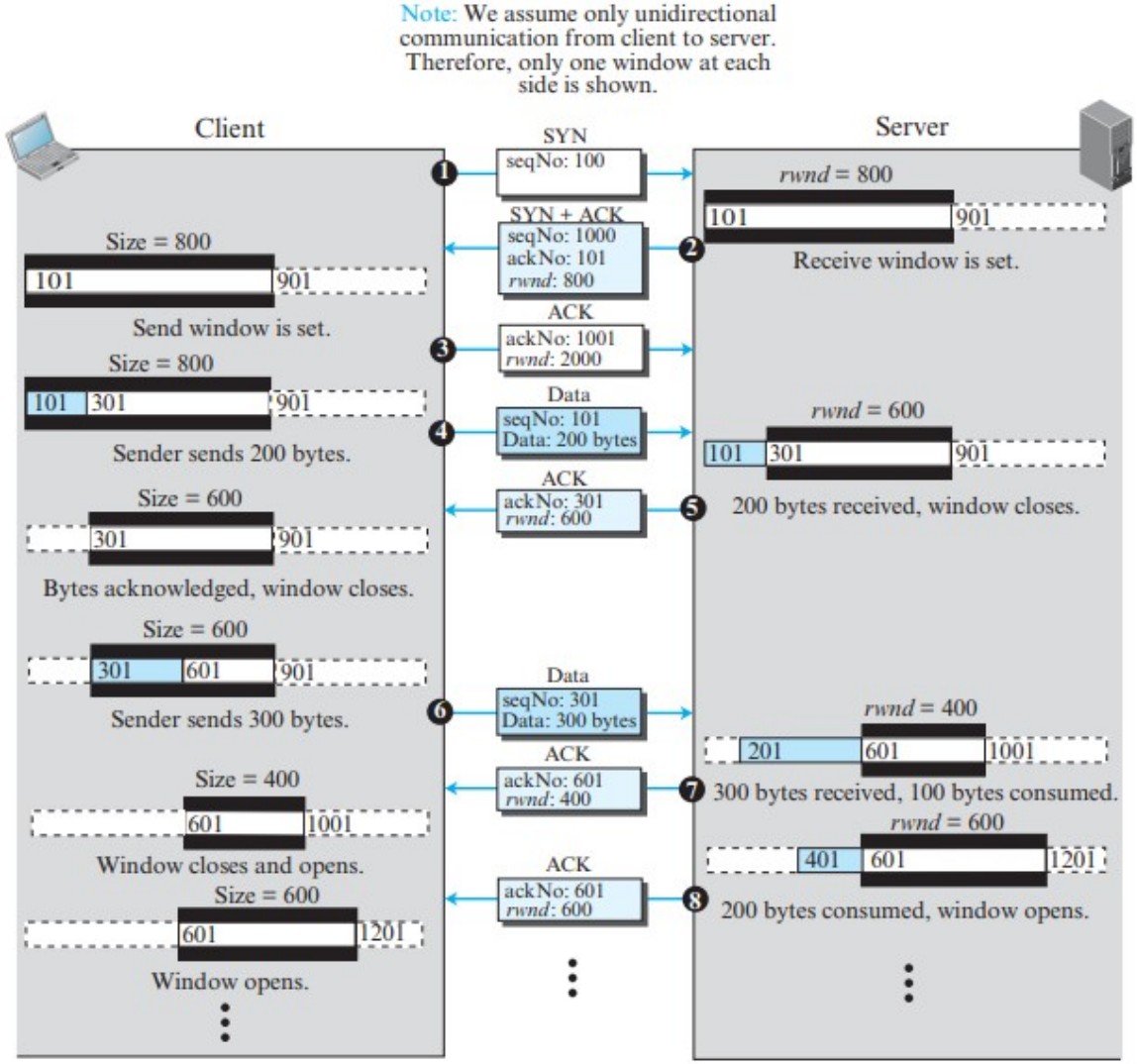
Opening and Closing of Windows

- 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; it opens (moves its right wall to the right) when more bytes are pulled by the process.
- 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 acknowledgment 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 (**new ackNo + new rwnd > last ackNo + last rwnd**).

Simple Scenario

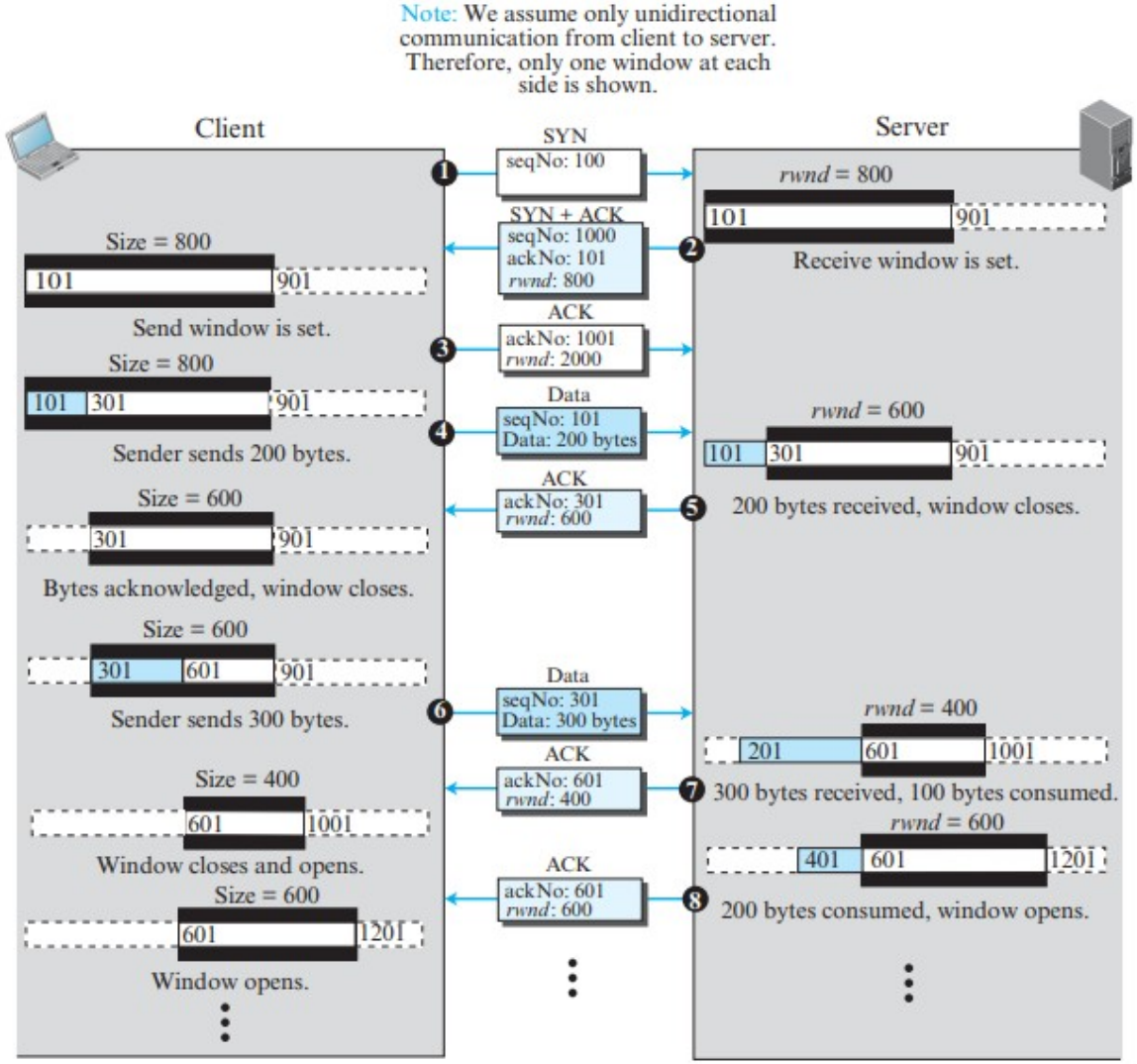


Simple Scenario



1. The first segment is from the client to the server (a SYN segment) to request connection. The client announces its initial seqNo = 100. When this segment arrives at the server, it allocates a buffer size of 800 (an assumption) and sets its window to cover the whole buffer (rwnd = 800). Note that the number of the next byte to arrive is 101.
2. The second segment is from the server to the client. This is an ACK + SYN segment. The segment uses ackNo = 101 to show that it expects to receive bytes starting from 101. It also announces that the client can set a buffer size of 800 bytes.
3. The third segment is the ACK segment from the client to the server. Note that the client has defined a rwnd of size 2000, but we do not use this value in our figure because the communication is only in one direction.

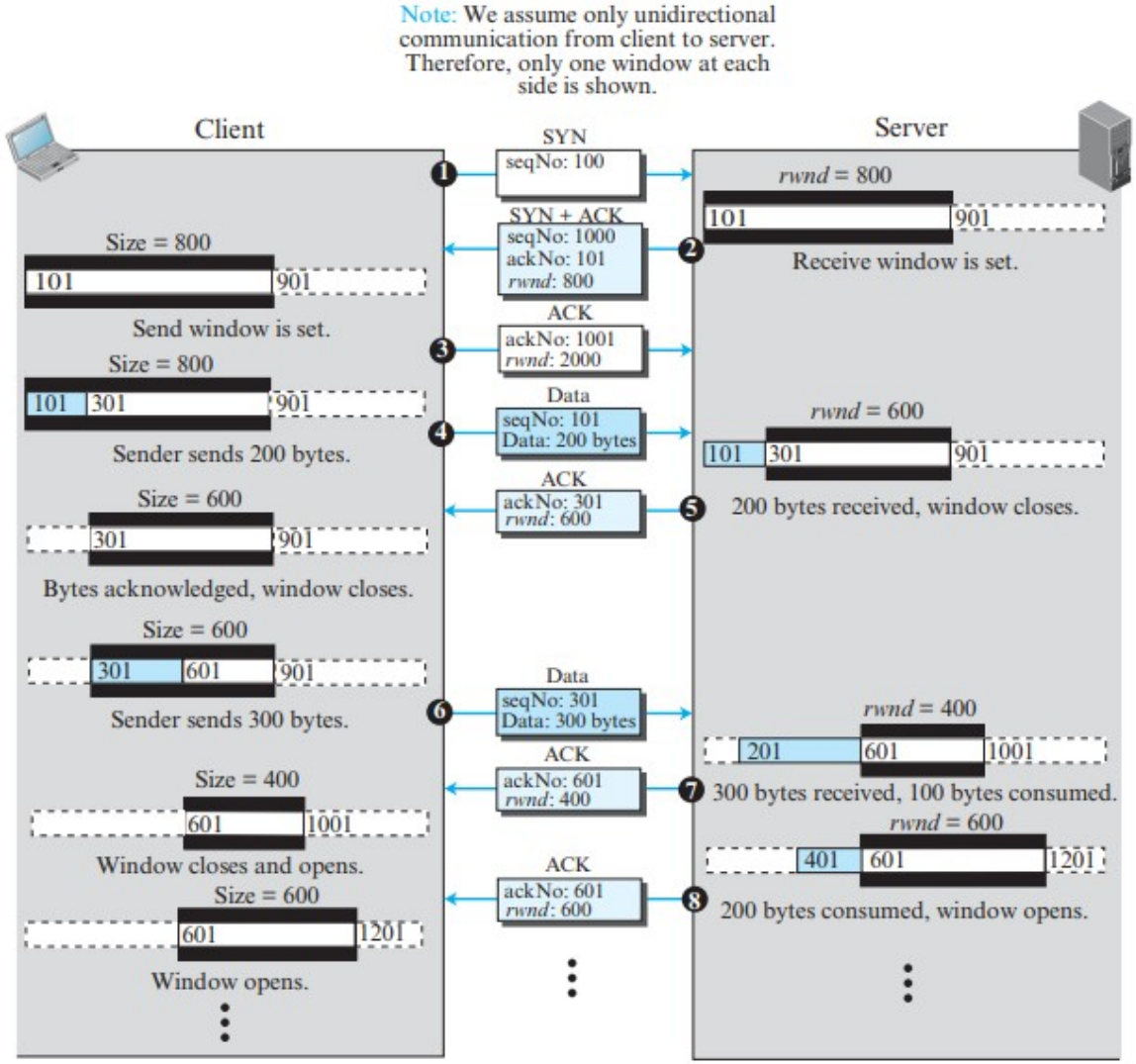
Simple Scenario



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

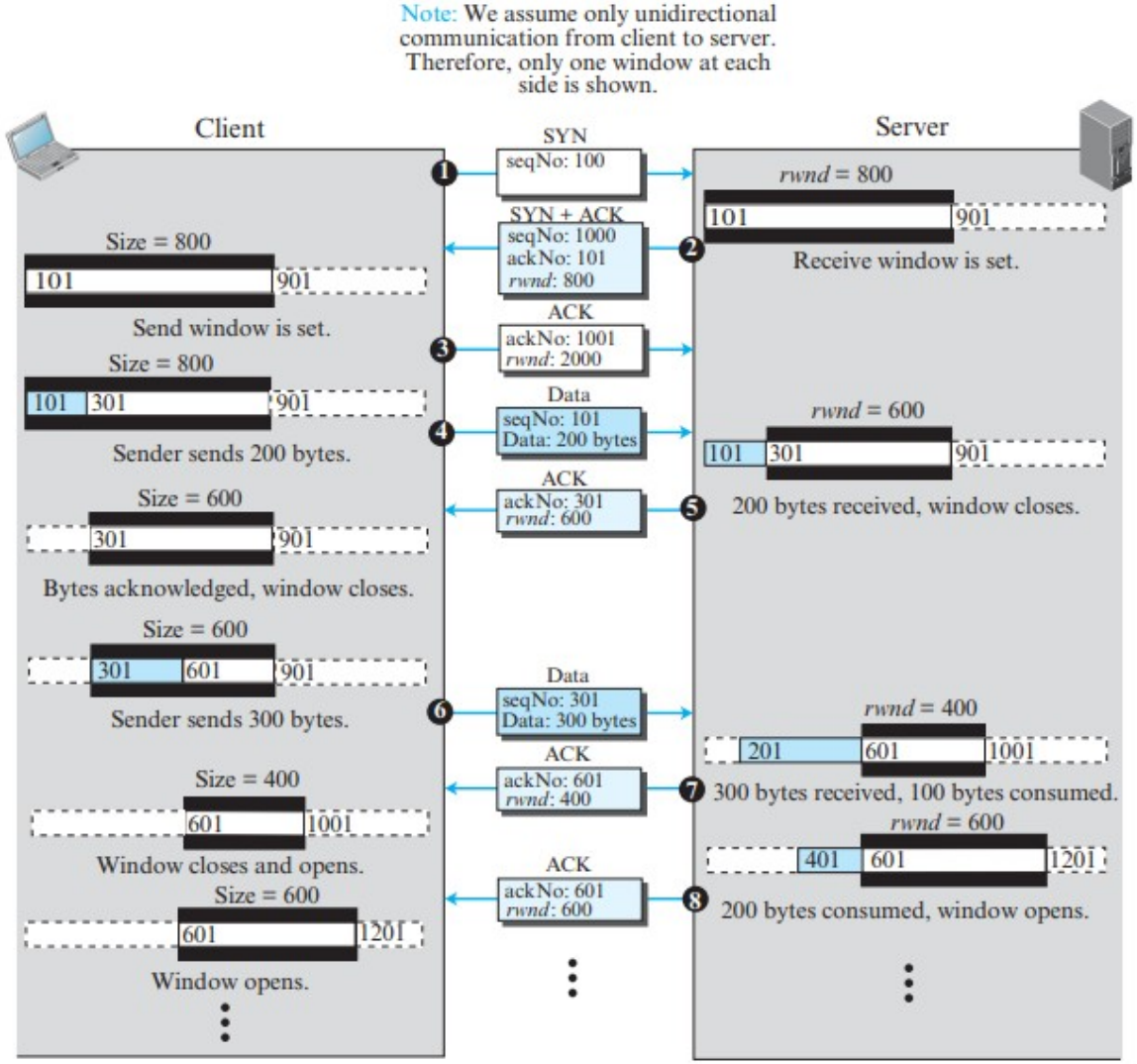
5. The fifth segment is the feedback from the server to the client. The server acknowledges bytes up to and including 300 (expecting to receive byte 301). The segment also carries the size of the receive window after decrease (600). The client, after receiving this segment, purges the acknowledged bytes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

Simple Scenario



6. Segment 6 is sent by the client after its process pushes 300 more bytes. The segment defines seqNo as 301 and contains 300 bytes. When this segment arrives at the server, the server stores them, but it has to reduce its window size. After its process has pulled 100 bytes of data, the window closes from the left for the amount of 300 bytes, but opens from the right for the amount of 100 bytes. The result is that the size is only reduced 200 bytes. The receiver window size is now 400 bytes.
7. In segment 7, the server acknowledges the receipt of data, and announces that its window size is 400. When this segment arrives at the client, the client has no choice but to reduce its window again and set the window size to the value of rwnd = 400 advertised by the server. The send window closes from the left by 300 bytes, and opens from the right by 100 bytes.

Simple Scenario



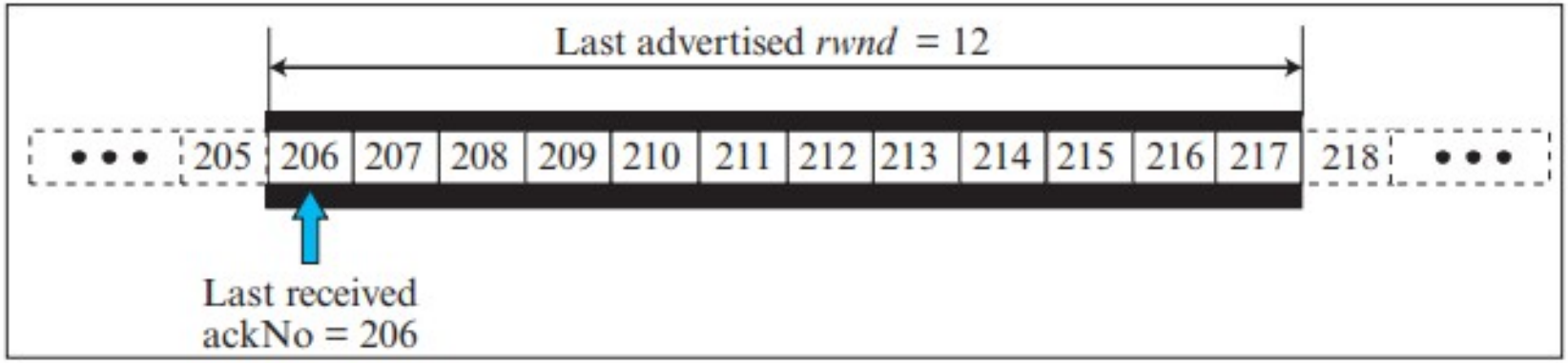
8. Segment 8 is also from the server after its process has pulled another 200 bytes. Its window size increases. The new rwnd value is now 600. The segment informs the client that the server still expects byte 601, but the server window size has expanded to 600. We need to mention that the sending of this segment depends on the policy imposed by the implementation. Some implementations 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.

SHRINKING OF WINDOWS

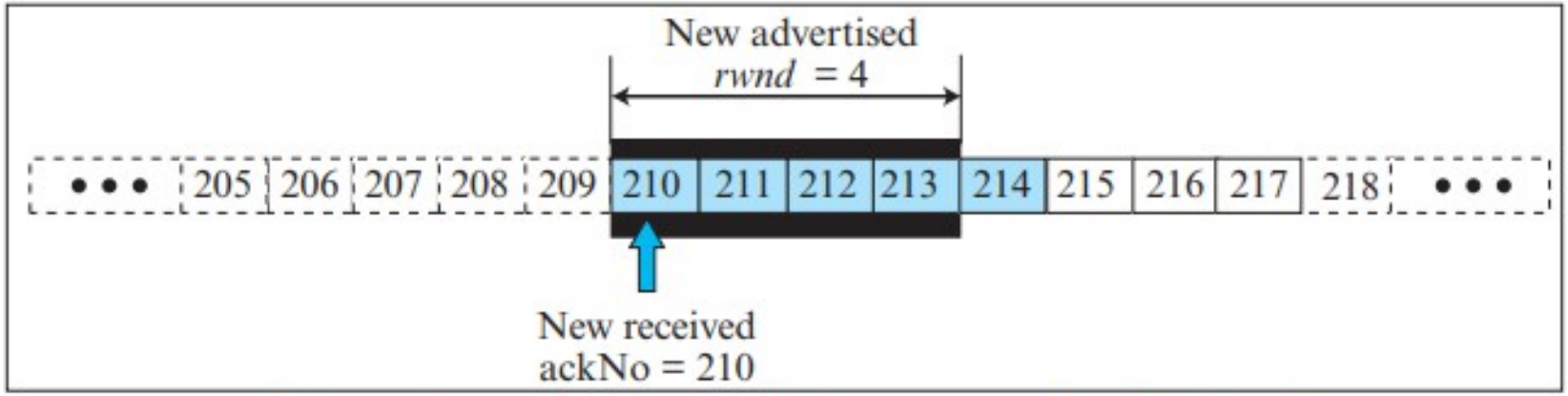
- As we said before, the receive window cannot shrink. The send window, on the other hand, can shrink if the receiver defines a value for rwnd that results in shrinking the window.
- However, some implementations do not allow shrinking of the send window. The limitation does not allow the right wall of the send window to move to the left.
- In other words, the receiver needs to keep the following relationship between the last and new acknowledgment and the last and new rwnd values to prevent shrinking of the send window.

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

SHRINKING OF WINDOWS



a. The window after the last advertisement



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

Window Shutdown

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

Silly Window Syndrome

- A serious problem can arise in the sliding window operation when either the sending application program creates data slowly or the receiving application program consumes data slowly, or both.
- Any of these situations results in the sending of data in very small segments, which reduces the efficiency of the operation. This problem is called the **Silly Window Syndrome.**

Silly Window Syndrome: Created by the Sender

- The sending TCP may create a silly window syndrome if it is serving an application program that creates data slowly, for example, 1 byte at a time.
- The application program writes 1 byte at a time into the buffer of the sending TCP. If the sending TCP does not have any specific instructions, it may create segments containing 1 byte of data.
- The solution is to prevent the sending TCP from sending the data byte by byte. The sending TCP must be forced to wait and collect data to send in a larger block.
- But how long should sending TCP wait?

Silly Window Syndrome: Created by the Sender

- **Nagle's Algorithm -**
  1. The sending TCP sends the first piece of data it receives from the sending application program even if it is only 1 byte.
  2. After sending the first segment, the sending TCP accumulates data in the output buffer and waits until either the receiving TCP sends an acknowledgment or until enough data has accumulated to fill a maximum-size segment. At this time, the sending TCP can send the segment.
  3. Step 2 is repeated for the rest of the transmission. Segment 3 is sent immediately if an acknowledgment is received for segment 2, or if enough data have accumulated to fill a maximum-size segment.
- The elegance of Nagle's algorithm is in its simplicity and in the fact that 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).

Silly Window Syndrome: Created by the Receiver

- The receiving TCP may create a silly window syndrome if it is serving an application program that consumes data slowly, for example, 1 byte at a time.
- Suppose that the sending application program creates data in blocks of 1 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 advertisement 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. Again we have an efficiency problem and the silly window syndrome.

Silly Window Syndrome: Created by the Receiver

- Two solutions have been proposed to prevent the silly window syndrome created by an application program that consumes data more slowly than they arrive.
- The first solution **(Clark's solution)** is to send an acknowledgment as soon as the data arrive, but to announce a window size of zero until either there is enough space to accommodate a segment of maximum size or until at least half of the receive buffer is empty.
- The second solution is to delay sending the acknowledgment. This means that when a segment arrives, it is not acknowledged immediately. The receiver waits until there is a decent amount of space in its incoming buffer before acknowledging the arrived segments.

Error Control in TCP

- TCP is a reliable transport-layer protocol. This means that an application program that delivers a stream of data to TCP relies on TCP to deliver the entire stream to the application program on the other end in order, without error, and without any part lost or duplicated.
- Error control includes mechanisms for detecting and resending corrupted segments, resending lost segments, storing out-oforder segments until missing segments arrive, and detecting and discarding duplicated segments.
- Error control in TCP is achieved through the use of three simple tools: checksum, acknowledgment, and time-out.

ACKNOWLEDGMENT

- TCP uses acknowledgments to confirm the receipt of data segments. Control segments like SYN that carry no data, but consume a sequence number, are also acknowledged.
- ACK segments are never acknowledged.
- TCP uses two types of acknowledgments -
  - Cummulative Acknowledgment
  - Selective Acknowledgment

CUMMULATIVE 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. This is sometimes 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.

SELECTIVE ACKNOWLEDGMENT (SACK)

- Newer implementations of TCP are adding another type of acknowdgment called *selective acknowledgment* or SACK.
- A SACK does not replace an ACK, but reports additional information to the sender.
- A SACK reports a block of bytes that is out of order, and also a block of bytes that is duplicated, i.e., received more than once.
- However, since there is no provision in the TCP header for adding this type of information, SACK is implemented as an option at the end of the TCP header.

GENERATING ACKNOWLEDGMENTS

- When does a receiver generate acknowledgments?
  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.
  2. 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 segments.

Generating Acknowledgments

- When does a receiver generate acknowledgments?

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 of Segments

Retransmission after RTO

- The sending TCP maintains one **retranmission time-out (RTO)** for each connection. When there is time out, TCP resends the segment with the smallest sequence number and restarts the timer.

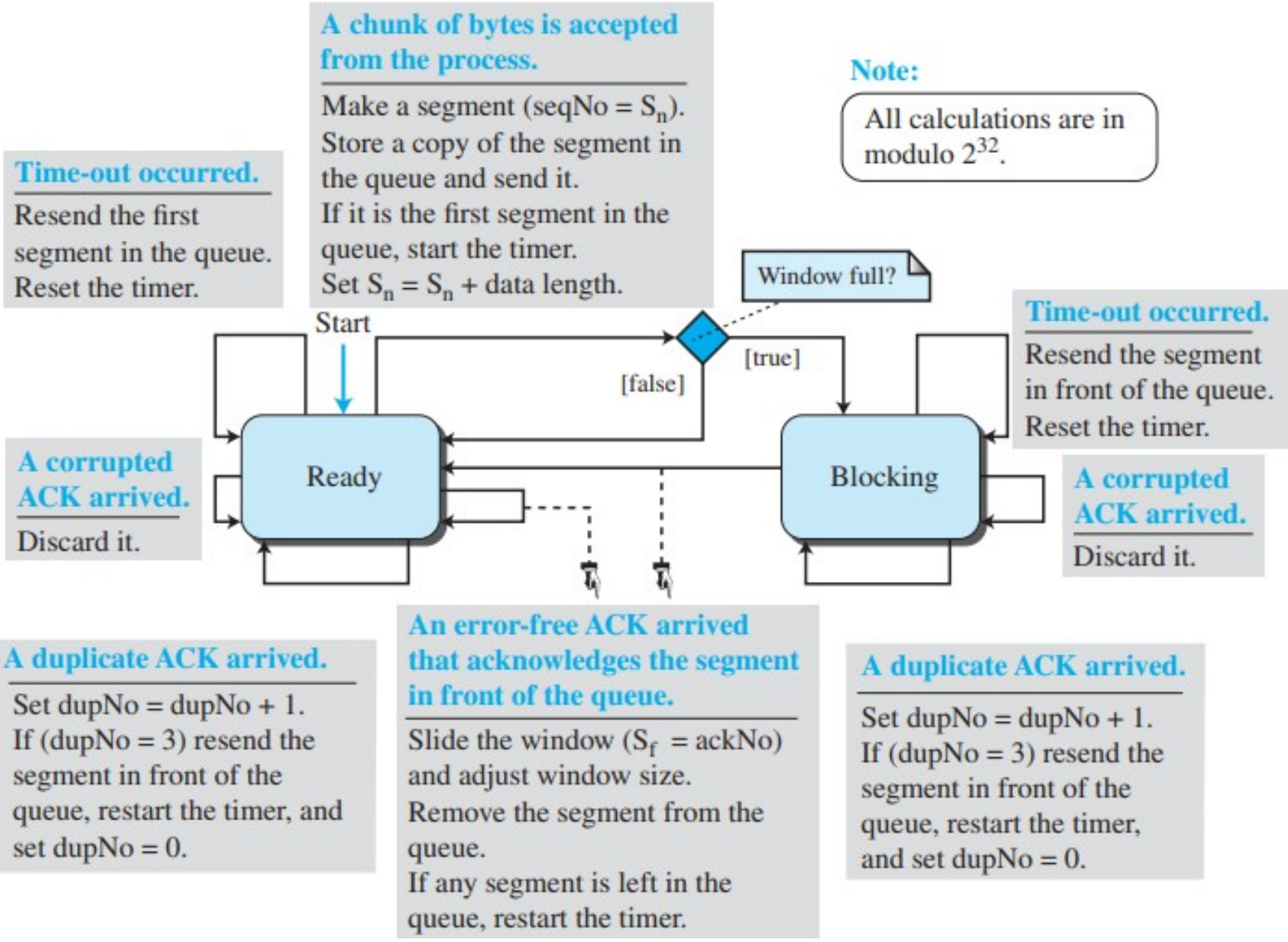
Retransmission after Three Duplicate ACK Segments

- To expedite service throughout the Internet by allowing senders to retransmit without waiting for a time out, most implementations today follow the three duplicate ACKs rule and retransmit the missing segment immediately. This feature is called fast retransmission.
- In this version, if three duplicate acknowledgments (i.e., an original ACK plus three exactly identical copies) arrive for a segment, the next segment is retransmitted without waiting for the time-out.

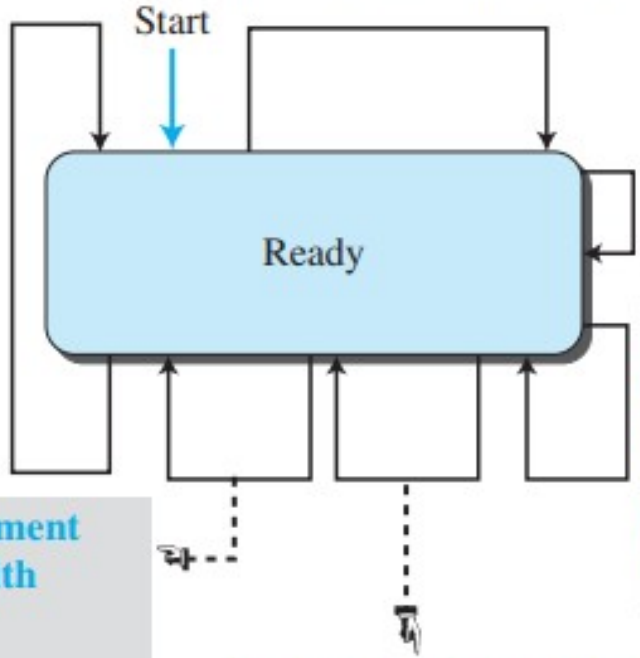
FSMs for Data Transfer in TCP

- Data transfer in TCP is close to the Selective-Repeat protocol with a slight similarity to GBN.
- Since TCP accepts out-of-order segments, TCP can be thought of as behaving more like the SR protocol, but since the original acknowledgments are cumulative, it looks like GBN.
- However, if the TCP implementation uses SACKs, then TCP is closest to SR.

Sender-Side FSM



Receiver-Side FSM



Note:

All calculations are in modulo 2^{32}.

An expected error-free segment arrived.

Buffer the message.
$R_n = R_n$ + data length.
If the ACK-delaying timer is running, stop the timer and send a cumulative ACK. Otherwise, start the ACK-delaying timer.

A request for delivery of k bytes of data from process came.

Deliver the data.
Slide the window and adjust window size.

ACK-delaying timer expired.

Send the delayed ACK.

An error-free, but out-of order segment arrived.

Store the segment if not duplicate.
Send an ACK with ackNo equal to the sequence number of expected segment (duplicate ACK).

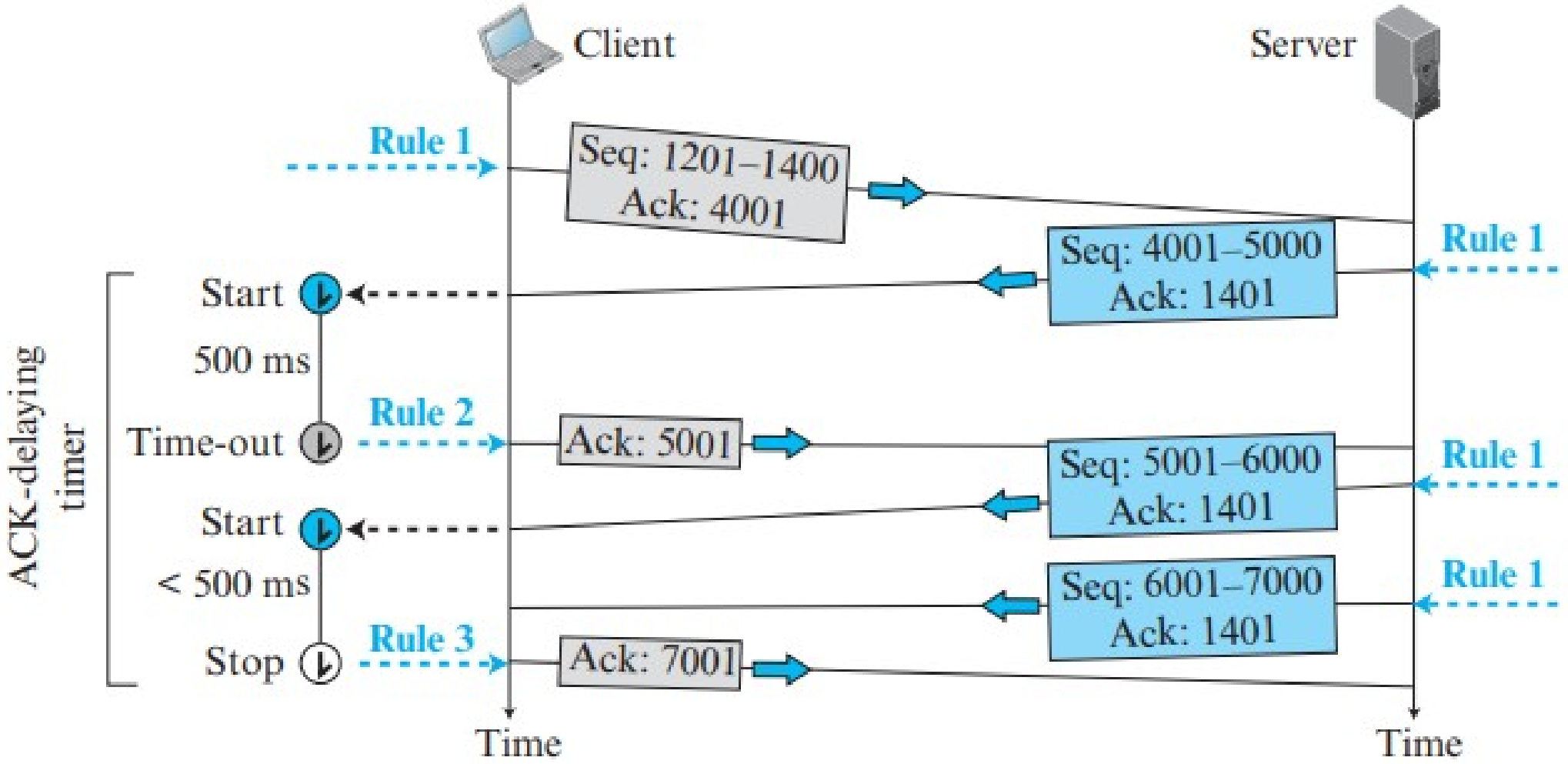
An error-free duplicate segment or an error-free segment with sequence number outside window arrived.

Discard the segment.
Send an ACK with ackNo equal to the sequence number of expected segment (duplicate ACK).

A corrupted segment arrived.

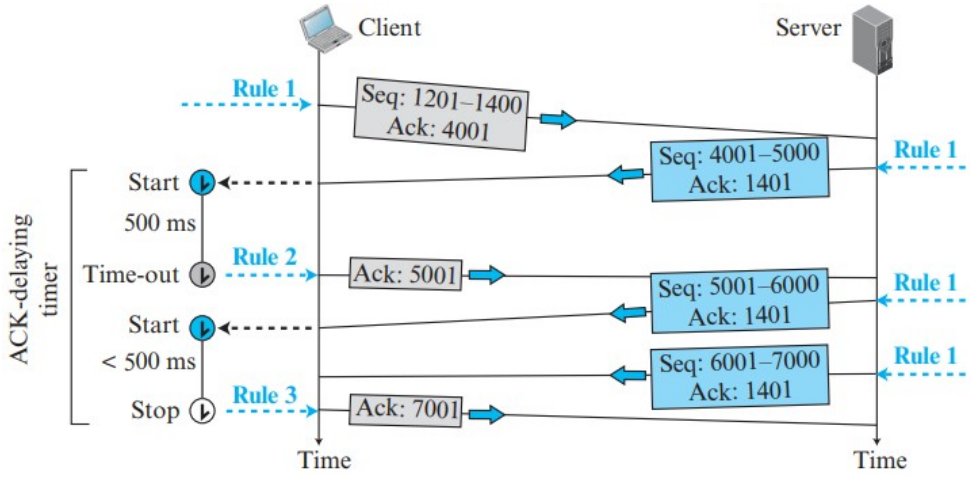
Discard the segment.

Normal Operation Scenario

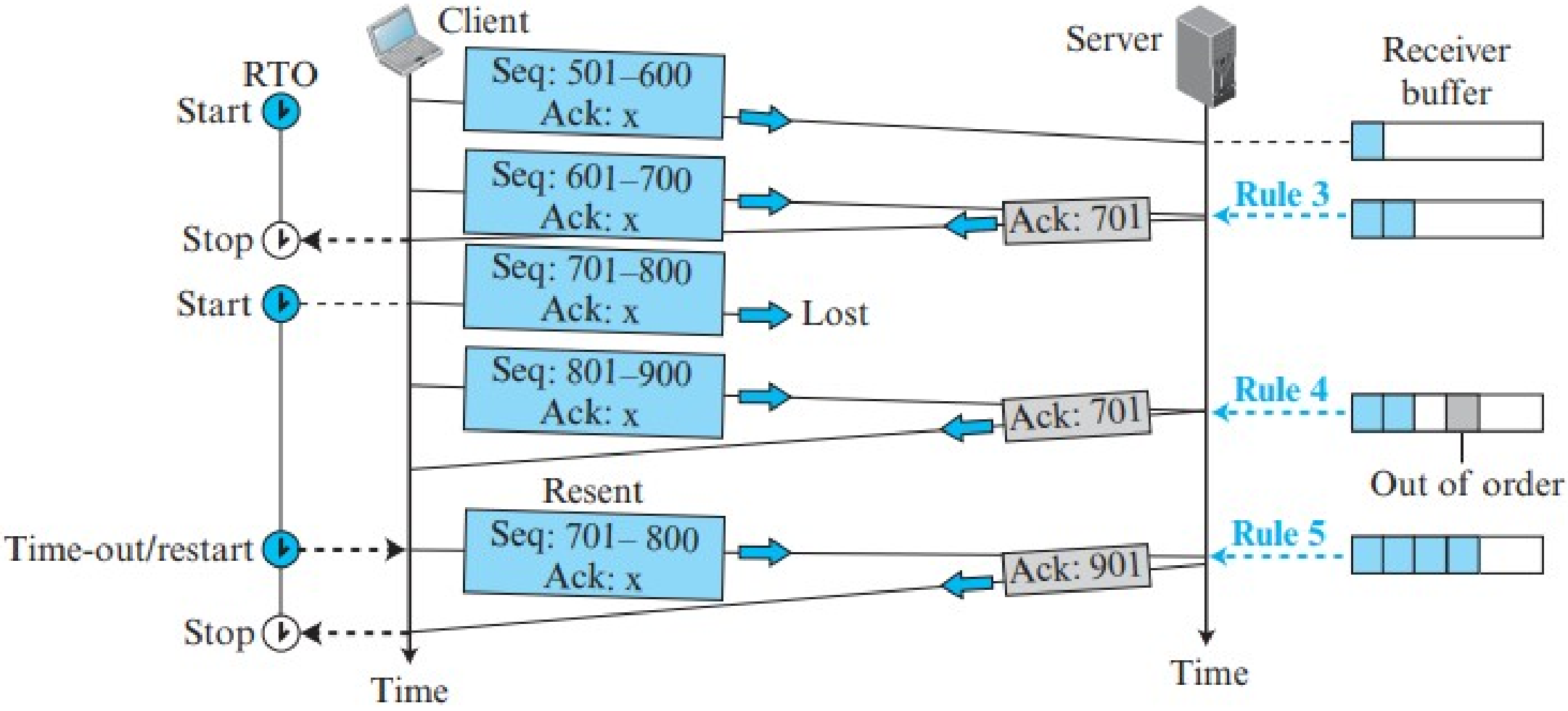


Normal Operation Scenario

- The client TCP sends one segment; the server TCP sends three. The figure shows which rule applies to each acknowledgment. At the server site, only rule 1 applies. There are data to be sent, so the segment displays the next byte expected.
- When the client receives the first segment from the server, it does not have any more data to send; it needs to send only an ACK segment. However, according to rule 2, the acknowledgment needs to be delayed for 500 ms to see if any more segments arrive. When the ACK-delaying timer matures, it triggers an acknowledgment. This is because the client has no knowledge if other segments are coming; it cannot delay the acknowledgment forever.
- When the next segment arrives, another ACK-delaying timer is set. However, before it matures, the third segment arrives. The arrival of the third segment triggers another acknowledgment based on rule 3. We have not shown the RTO timer because no segment is lost or delayed. We just assume that the RTO timer performs its duty.

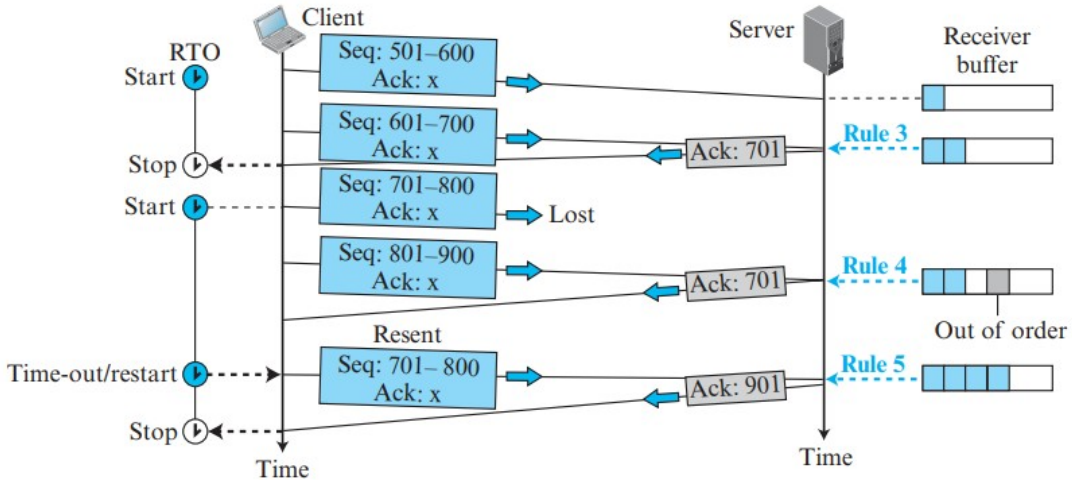


Lost Segment Scenario

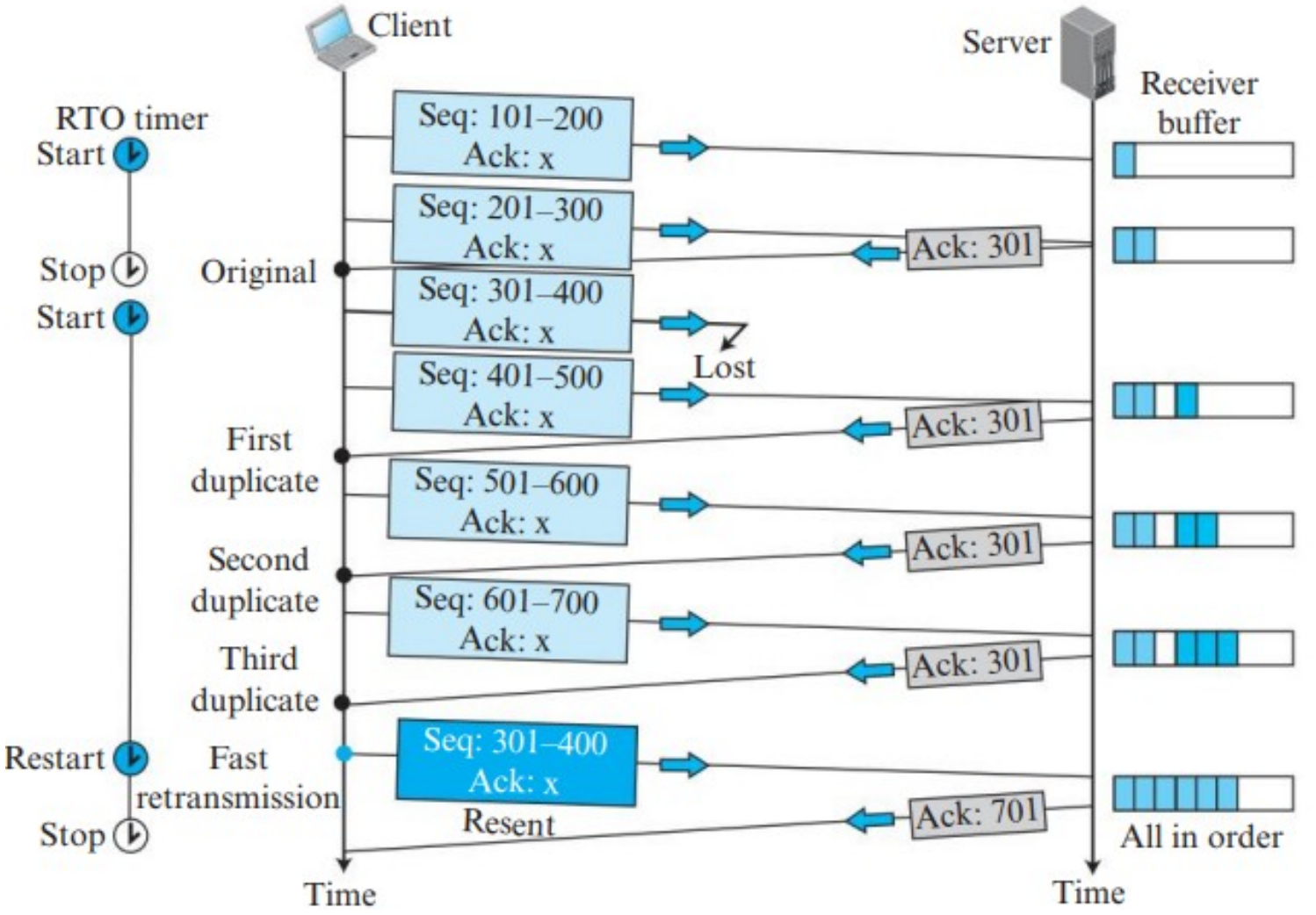


LOST SEGMENT SCENARIO

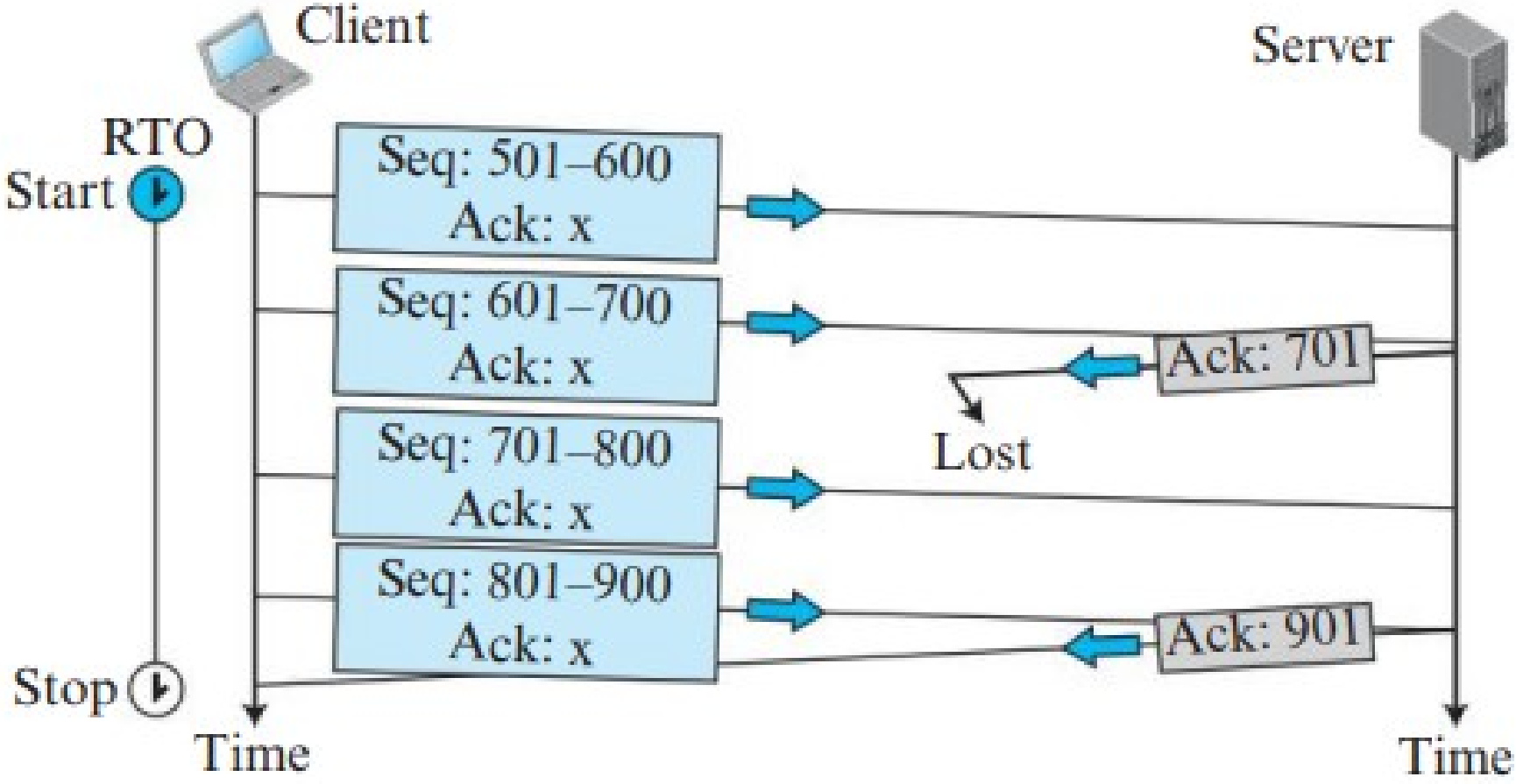
- In this scenario, we show what happens when a segment is lost or corrupted. A lost or corrupted segment is treated the same way by the receiver. A lost segment is discarded somewhere in the network; a corrupted segment is discarded by the receiver itself. Both are considered lost.
- We are assuming that data transfer is unidirectional: one site is sending, the other receiving. In our scenario, the sender sends segments 1 and 2, which are acknowledged immediately by an ACK (rule 3).
- Segment 3, however, is lost. The receiver receives segment 4, which is out of order. The receiver stores the data in the segment in its buffer but leaves a gap to indicate that there is no continuity in the data. The receiver immediately sends an acknowledgment to the sender displaying the next byte it expects (rule 4). Note that the receiver stores bytes 801 to 900, but never delivers these bytes to the application until the gap is filled.



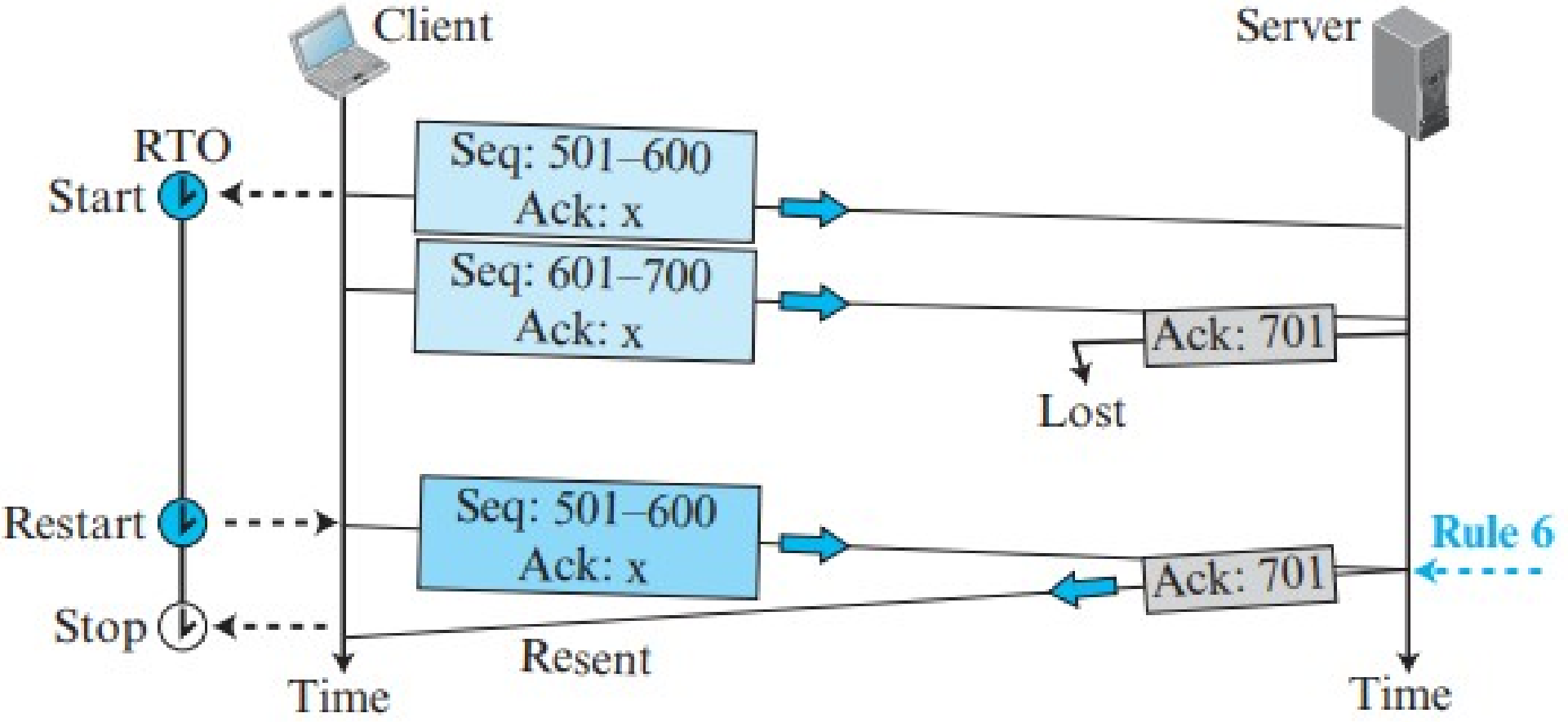
Fast Retransmission Scenario



Automatically Corrected Lost ACK



Lost ACK Corrected by Resending a Segment



Deadlock Created by Lost Acknowledgment

- There is one situation in which loss of an acknowledgment may result in system deadlock. This is the case in which a receiver sends an acknowledgment with rwnd set to 0 and requests that the sender shut down its window temporarily.
- After a while, the receiver wants to remove the restriction; however, if it has no data to send, it sends an ACK segment and removes the restriction with a nonzero value for rwnd.
- A problem arises if this acknowledgment is lost. The sender is waiting for an acknowledgment that announces the nonzero rwnd. The receiver thinks that the sender has received this and is waiting for data.
- This situation is called a deadlock; each end is waiting for a response from the other end and nothing is happening. A retransmission timer is not set.

TCP Congestion Control

- When we discussed flow control in TCP, we mentioned that the ***size of the send window is controlled by the receiver using the value of rwnd***, which is advertised in each segment traveling in the opposite direction. The use of this strategy guarantees that the receive window is never overflowed with the received bytes (no congestion at end systems).
- This, however, does not mean that the intermediate buffers, buffers in the routers, do not become congested.
- TCP needs to worry about congestion in the middle because many segments lost may seriously affect the error control. More segment loss means resending the same segments again, resulting in worsening the congestion, and finally the collapse of the communication.
- TCP is an end-to-end protocol that uses the services of IP Protocol of Network layer. The congestion in the router is in the IP territory and should be taken care by IP. But we will discuss in the next unit that IP does not provide any congestion control mechanism.

CONGESTION WINDOW

- Since IP does not handle congestion at router, TCP itself has to take the responsibility and send the data segments at appropriate rate so that network does not become congested while utilizing the network bandwidth efficiently.
- *TCP needs to define policies that accelerate the data transmission when there is no congestion and decelerate the transmission when congestion is detected.*
- To control the number of segments to transmit, TCP uses another variable called a **congestion window, *cwnd*,** whose size is controlled by the congestion situation in the network.
- The ***cwnd*** variable and the ***rwnd*** variable together define the size of the send window in TCP. The first is related to the congestion in the middle (network); the second is related to the congestion at the end.

Actual send window size = minimum (rwnd, cwnd)

Congestion Detection

- How does TCP sender detect the possible existence of congestion in the network?
- The TCP sender uses the occurrence of two events as signs of congestion in the network: *time-out* and *receiving three duplicate ACKs*.
- **Time-Out Event**
  - If the sender doesn't get an acknowledgment (ACK) for a segment within a certain time (timeout), it assumes that segment got lost due to congestion.
  - This is a sign of strong congestion. It means the network is struggling and TCP needs to be careful.

Congestion Detection

- **Three Duplicate ACKs Event**
  - If the sender receives three duplicate ACKs (four ACKs with the same acknowledgment number), it means that one segment is missing, but the receiver got the other three segments which were sent later by sender.
  - This could indicate a less severe congestion.
  - This could indicate that network is either slightly congested or has recovered from the severe congestion.
- Earlier versions of TCP, called *Taho TCP*, treated both events similarly. But the newer version of TCP, called *Reno TCP*, treats these two signs differently based on their severity.
- A very interesting point in TCP congestion is that the TCP sender uses only one feedback from the other end to detect congestion: ACKs. The lack of regular, timely receipt of ACKs, which results in a time-out, is the sign of a strong congestion; the receiving of three duplicate ACKs is the sign of a weak congestion in the network.

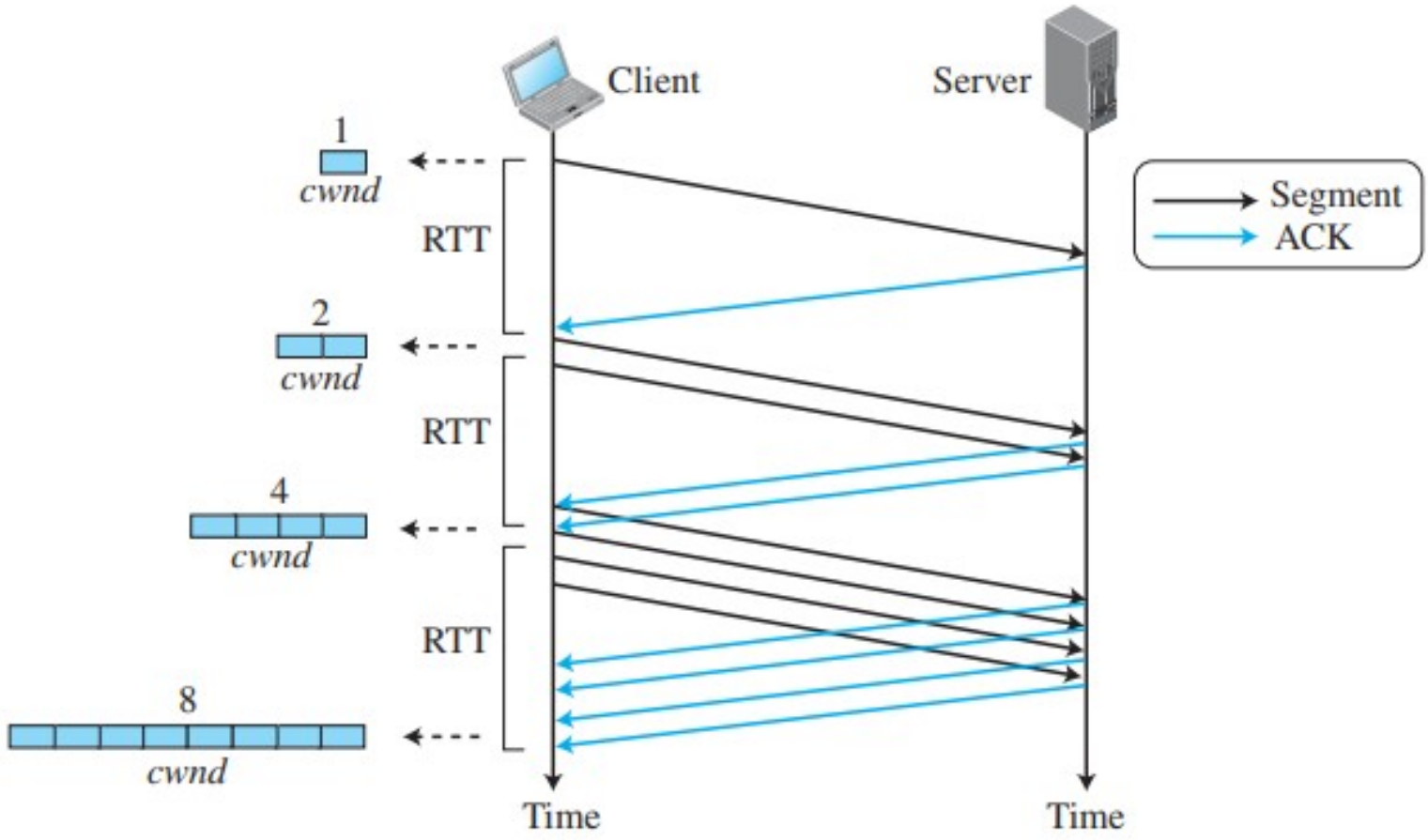
Congestion Policies

- TCP's general policy for handling congestion is based on three algorithms -
  - slow start
  - congestion avoidance
  - fast recovery.

SLOW START: EXPONENTIAL INCREASE

- The slow-start algorithm is based on the idea that the size of the congestion window (cwnd) starts with one *maximum segment size (MSS)*, but it increases one MSS each time one acknowledgment arrives.
- The Maximum Segment Size (MSS) is a parameter of the *Options* field of the TCP header that specifies the largest amount of data (in bytes) that can be sent in a single TCP Segment. It only counts data not headers.
- The MSS is decided during the connection establishment phase in TCP and can't be changed later till the end of connection.
- **Starting Slowly:**
  - At begining, you can only send one segment (a chunk of data).
  - Each time you get an acknowledgment (ACK) that your sent data was received, you can send double the amount next time.
  - So, it starts slow but grows pretty fast.

SLOW START: EXPONENTIAL INCREASE



Slow Start: Exponential Increase

- **Window Size Calculation:**
  - The congestion window (cwnd) is like the number of segments you can send.
  - If an ACK arrives, cwnd increases by 1.
  - So, if you get more ACKs, you can send more data.
- **Exponential Growth:**
  - The cwnd grows exponentially with each round-trip time (RTT).
  - It's a bit like a very eager approach—growing really quickly.
- **Stopping Slow Start:**
  - It can’t go on foreever, though. There is a limit called slow-start threshold (ssthresh).
  - When the window size reaches this threshold, slow start stops amd a new phase begins.

Slow Start: Exponential Increase

- **Consideration for Delayed Acknowledgements:**
  - If acknowledgments are delayed, the growth is a bit slower.
  - For every ACK, cwnd increases by only 1, even if multiple segments are acknowledged together.
  - So, the growth is still fast but not as fast as in ideal conditions.

If an ACK arrives, cwnd = cwnd + 1.

SLOW START: EXPONENTIAL INCREASE

- **In the slow-start algorithm, the size of the congestion window increases exponentially until it reaches a threshold.**

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

Congestion Avoidance: Additive Increase

After the initial slow start, TCP has another trick called "congestion avoidance" to prevent things from getting out of control.

1. Congestion Avoidance:
- Instead of growing really fast (exponentially), TCP wants to be more careful.
- It uses an algorithm called congestion avoidance to increase the congestion window (cwnd) more steadily.

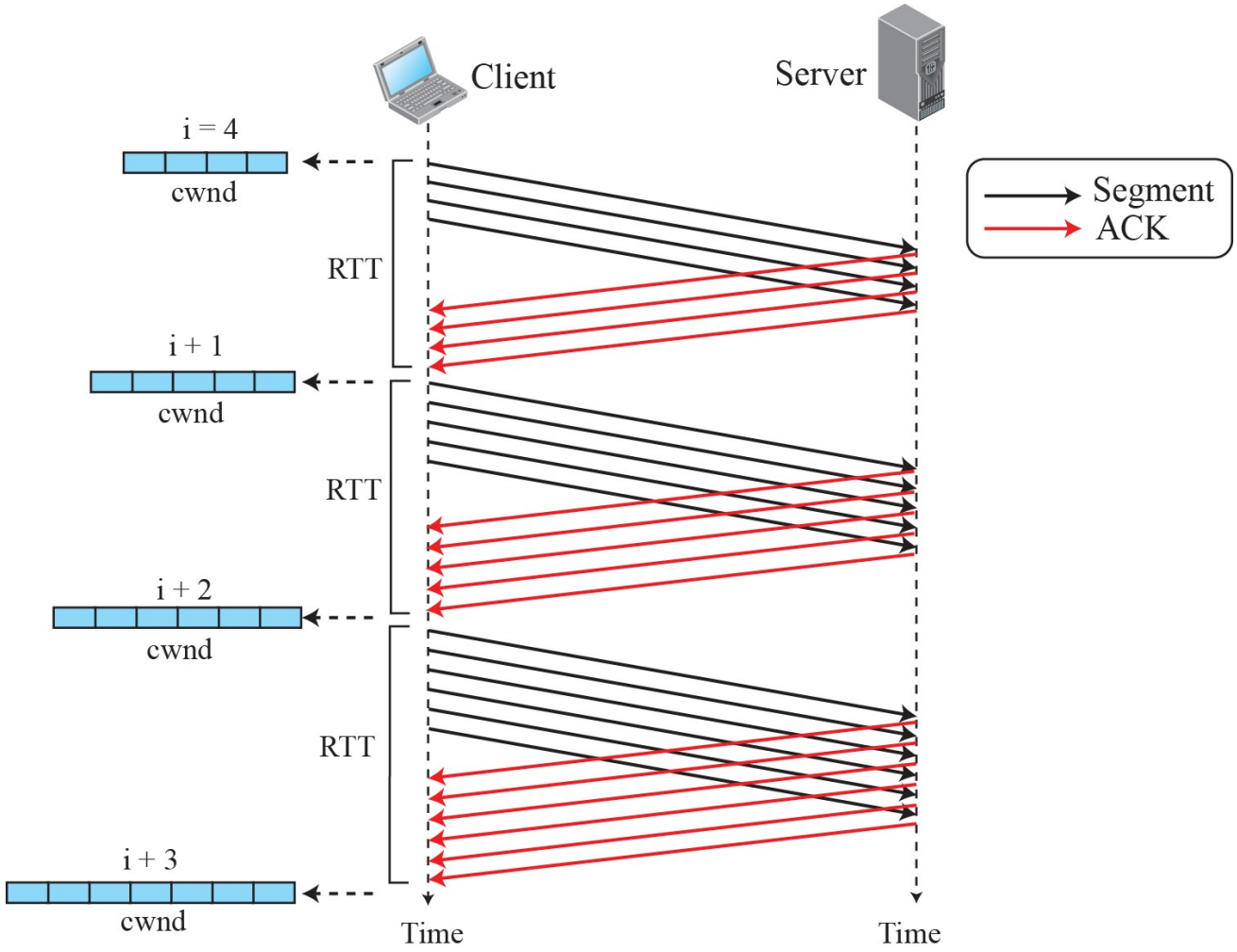
2. How It Works:
- After the slow start, when cwnd reaches a certain point (slow-start threshold), it switches to the congestion avoidance phase.
- **In this phase, each time a whole "window" of segments is acknowledged, cwnd increases by one.**
- A window is the number of segments sent during one round-trip time (RTT).

Congestion Avoidance: Additive Increase

3. Example:
- Let's say the sender starts with cwnd = 4, meaning it can send four segments initially.
- After getting four acknowledgments, it can send one more segment, and the window becomes 5.
- The process continues, increasing the window each time the whole window is acknowledged.

4. Math Behind It:
- The size of the window increases by a fraction of the maximum segment size (MSS), specifically 1/cwnd.
- **This means that all segments in the previous window must be acknowledged to increase the window by one MSS.**

5. Growth Rate:
- If you look at the size of cwnd in terms of round-trip times (RTTs), **the growth is more steady and linear compared to the earlier slow start.**


Client
Server
Segment
ACK
i = 4
cwnd
RTT
i + 1
cwnd
RTT
i + 2
cwnd
RTT
i + 3
cwnd
Time
Time

Congestion Avoidance: Additive Increase

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.

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

FAST RECOVERY

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

If a duplicate ACK arrives, $cwnd = cwnd + (1 / cwnd)$.