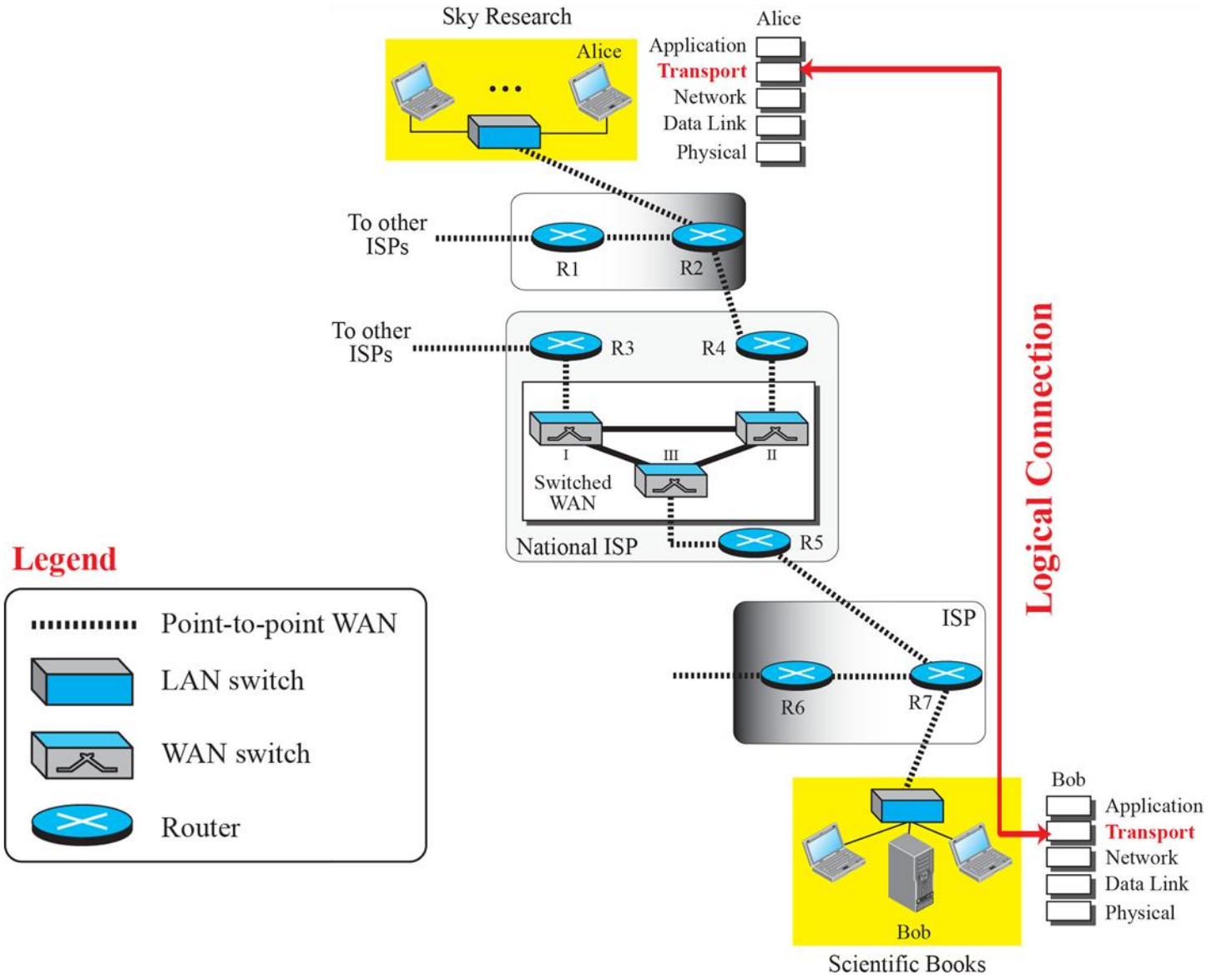
Transport layer

Introduction

The transport layer provides **process-to-process** communication between two application layers. Communication is provided using a logical connection, which means that the two application layers assume that there is an imaginary direct connection through which they can send and receive messages.

Figure 3.1: Logical connection at the transport layer



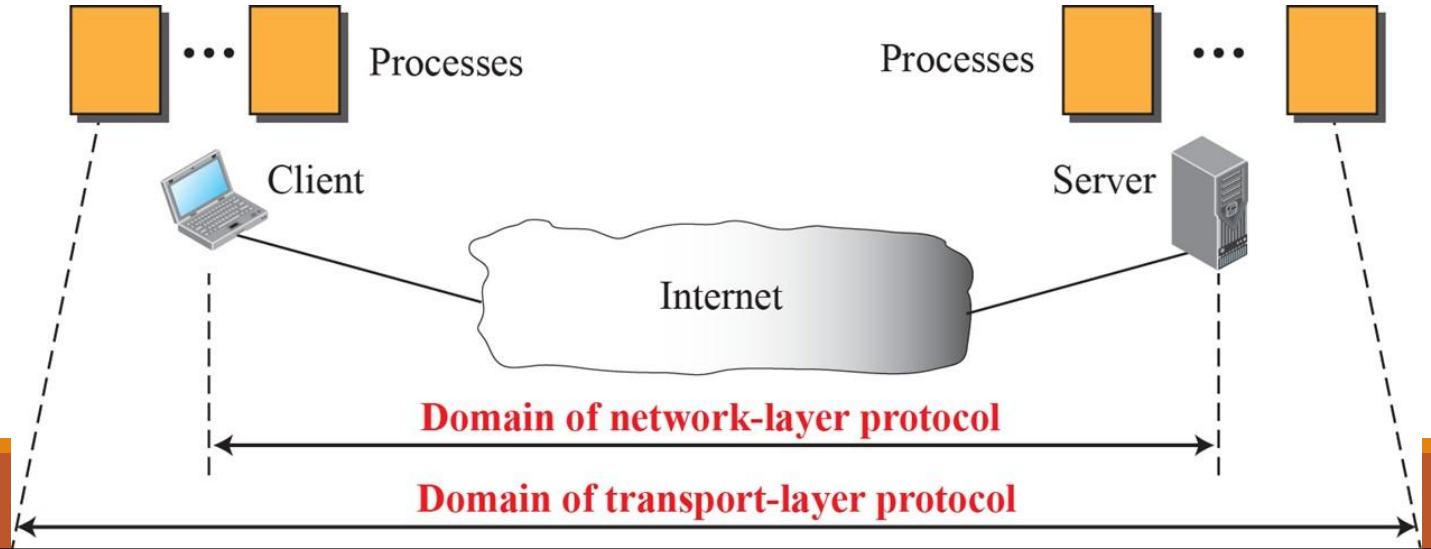
Transport-layer services

As we discussed in last unit, the transport layer is located between the network layer and the application layer. The transport layer is responsible for providing services to the application layer; it receives services from the network layer. In this section, we discuss the services that can be provided by the transport layer; in the next section, we discuss several transport-layer protocols.

Process-to-Process Communication

The main job of a transport-layer protocol is to enable communication between programs (called processes) running on different devices. Imagine it as making sure the right messages reach the right programs.

The network layer handles communication between whole devices (like computers), ensuring messages get to the correct device. But it doesn't concern itself with the specific programs on those devices. This is where the transport layer comes in. It takes care of getting the message to the right program running on the destination device. So, transport layer bridges the gap between devices (network layer) and programs (application layer) in a network.



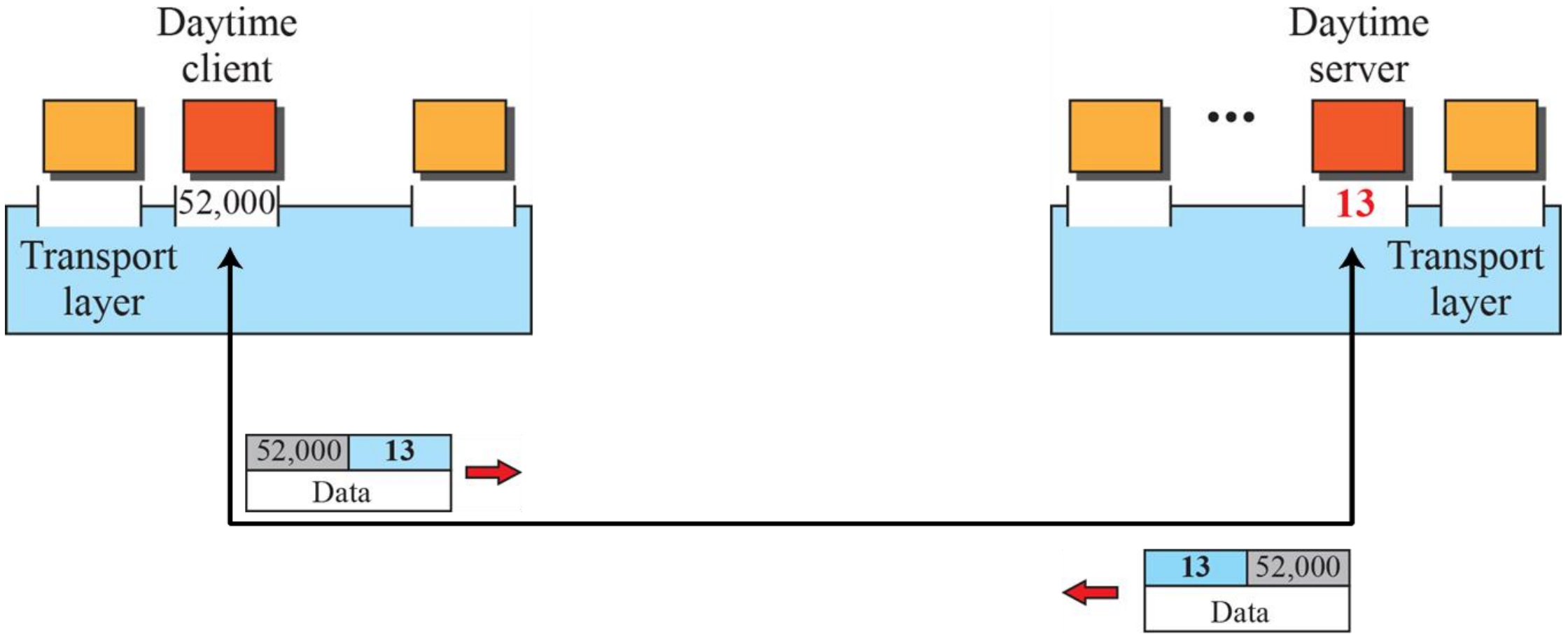
Addressing: Port Numbers

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 program on 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 like addresses for programs and range from 0 to 65,535 in the TCP/IP protocol.

Clients typically use ephemeral (short-lived) port numbers, usually greater than 1,023. 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.


Daytime client
52,000
Transport layer
Daytime server
13
Transport layer
52,000
13
Data
13
52,000
Data

ICANN Ranges

1. Well-known ports. The ports ranging from 0 to 1,023 are assigned and controlled by ICANN. These are the well-known ports.
2. Registered ports. The ports ranging from 1,024 to 49,151 are not assigned or controlled by ICANN. They can only be registered with ICANN to prevent duplication.
3. Dynamic 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.

Example

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

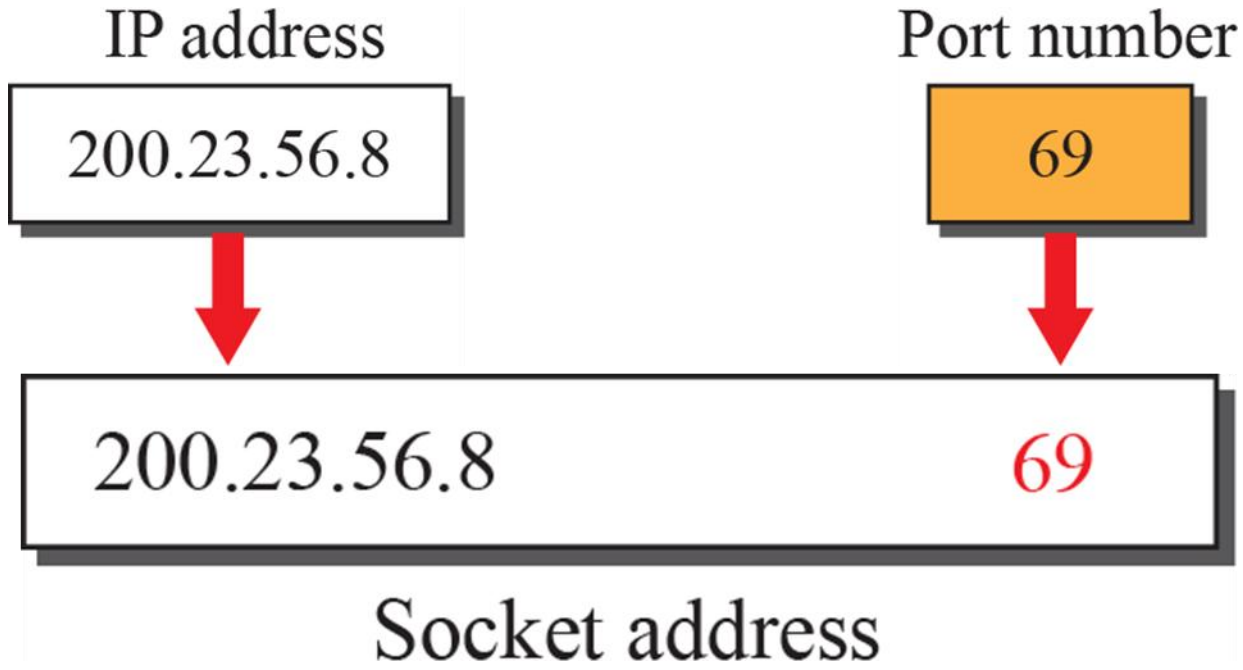
```
$grep     tftp/etc/services
tftp 69/tcp
tftp 69/udp
```

SNMP (see Chapter 9) uses two port numbers (161 and 162), each for a different purpose.

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

Socket Addresses

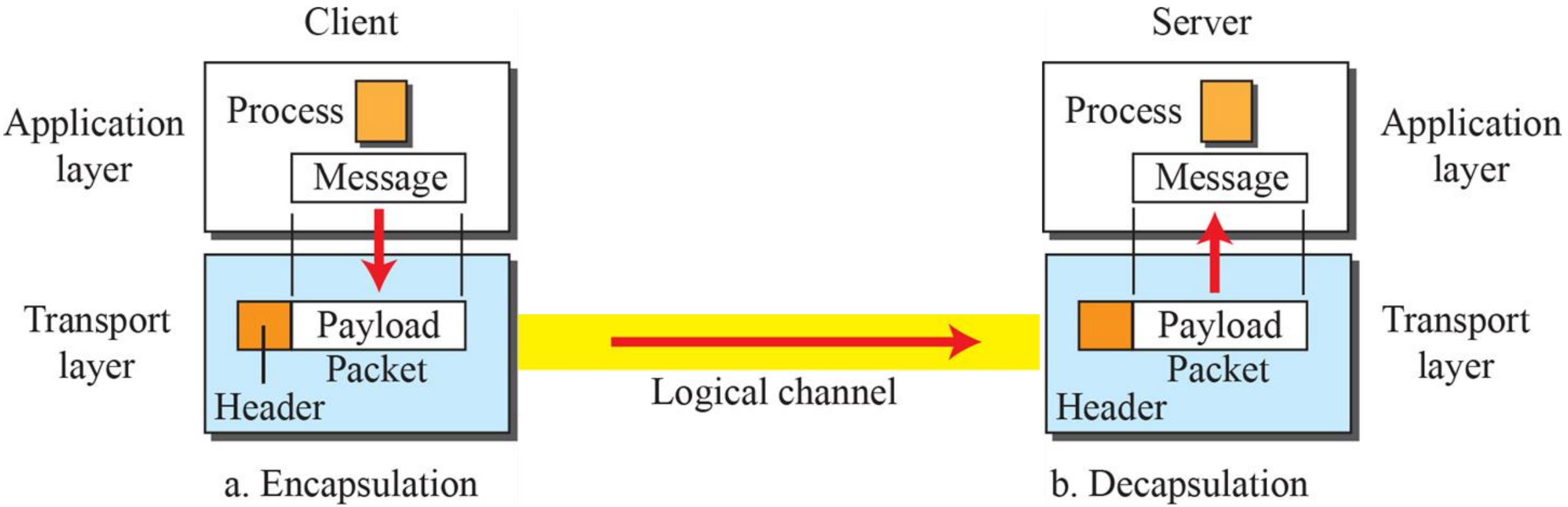
A transport-layer protocol in the TCP suite needs both the IP address and the port number, at each end, to make a connection. The combination of an IP address and a port number is called a socket address.



Encapsulation and Decapsulation

To send a message from one program to another, the transport-layer protocol adds some information to the message at the sender's end (encapsulation). This includes details like socket addresses. Then, it sends this as a packet over the network.

When the packet reaches its destination, the receiver's transport layer removes this added information (decapsulation) and delivers the message to the right program at the receiving end. The sender's address is also passed along in case a response is needed. This ensures that messages get to the correct programs.


Client
Server
Application layer
Process
Message
Transport layer
Payload
Packet
Header
Logical channel
Application layer
Process
Message
Transport layer
Payload
Packet
Header
a. Encapsulation
b. Decapsulation

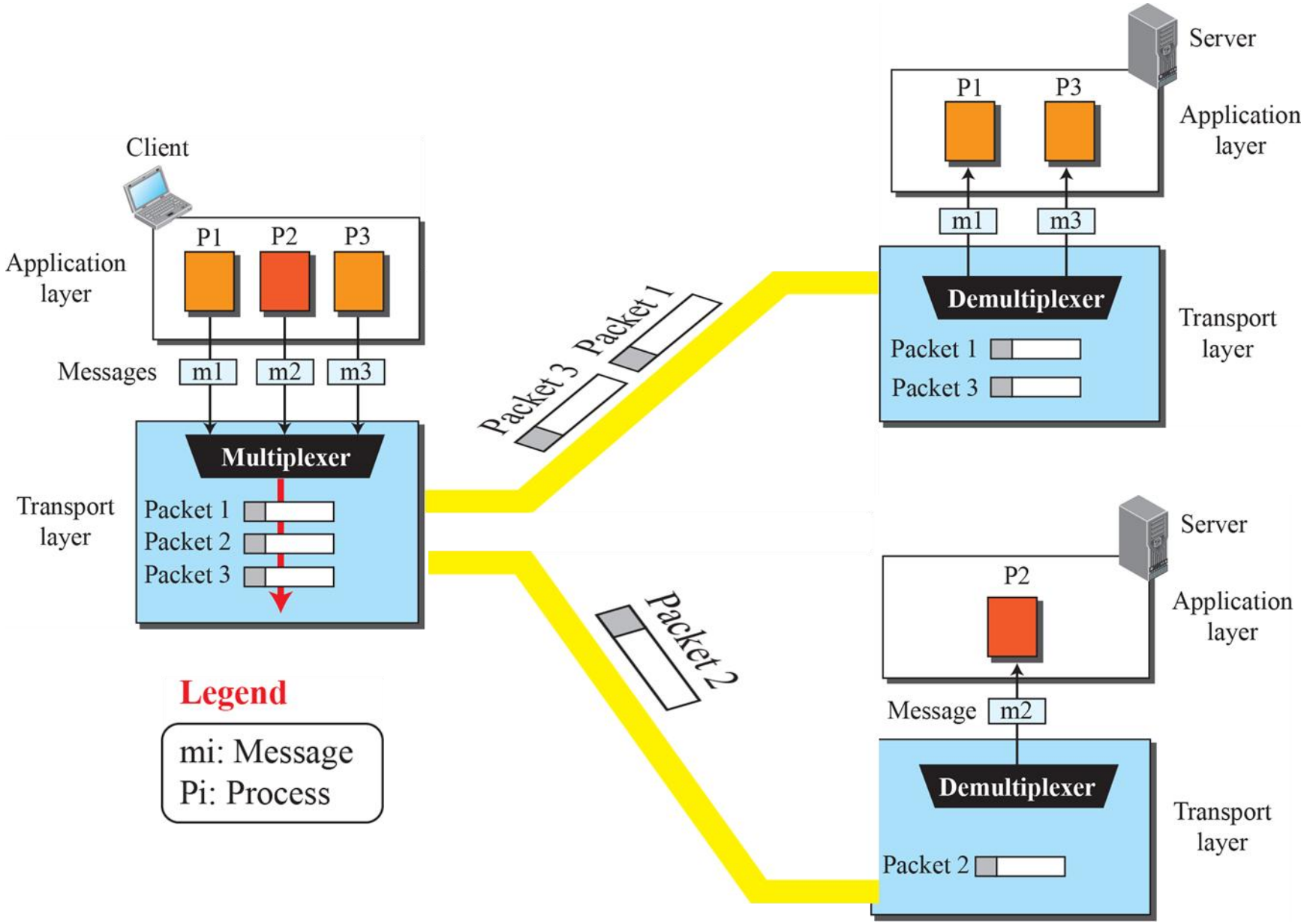
Multiplexing and Demultiplexing

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.

For example, imagine a client with three processes (P1, P2, and P3) that need to communicate with two servers. The client's transport layer combines their messages into packets and sends them. At the server end, the transport layer separates these messages and directs them to the appropriate server processes. Even when there's only one message, demultiplexing ensures it reaches the correct destination process.


Client
Application layer
P1
P2
P3
Messages
m1
m2
m3
Multiplexer
Transport layer
Packet 1
Packet 2
Packet 3
Packet 3
Packet 1
Packet 2
Server
P1
P3
Application layer
m1
m3
Demultiplexer
Packet 1
Packet 3
Transport layer
Server
P2
Application layer
Message
m2
Demultiplexer
Packet 2
Transport layer
Legend
mi: Message
Pi: Process

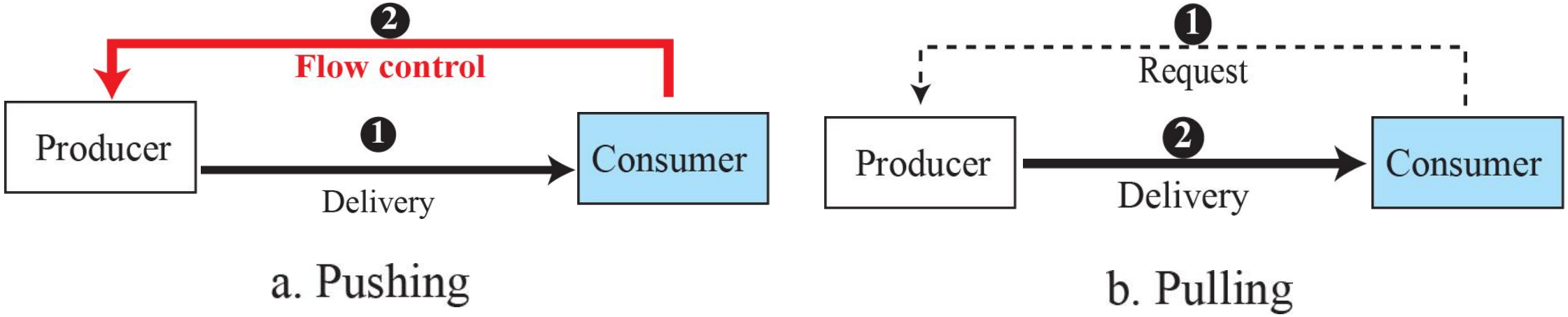
Flow Control

Whenever an entity produces items and another entity consumes them, **there should be a balance between production and consumption rates.** If the items are produced faster than they can be consumed, the consumer can be overwhelmed and may need to discard some items. If the items are produced more slowly than they can be consumed, the consumer must wait, and the system becomes less efficient. Flow control is related to the first issue. We need to prevent losing the data items at the consumer site.

Pushing or Pulling

Delivery of items from a sender to a receiver can happen in two ways: pushing and pulling.

- Pushing: The sender delivers items as soon as they are produced, without waiting for a request from the receiver. This can sometimes overwhelm the receiver, so there may be a need for the receiver to tell the sender to stop temporarily to prevent overload. This is called flow control.
- Pulling: The receiver requests items from the sender when it's ready to receive them. In this case, there's no need for flow control because the receiver only gets items when it wants them.



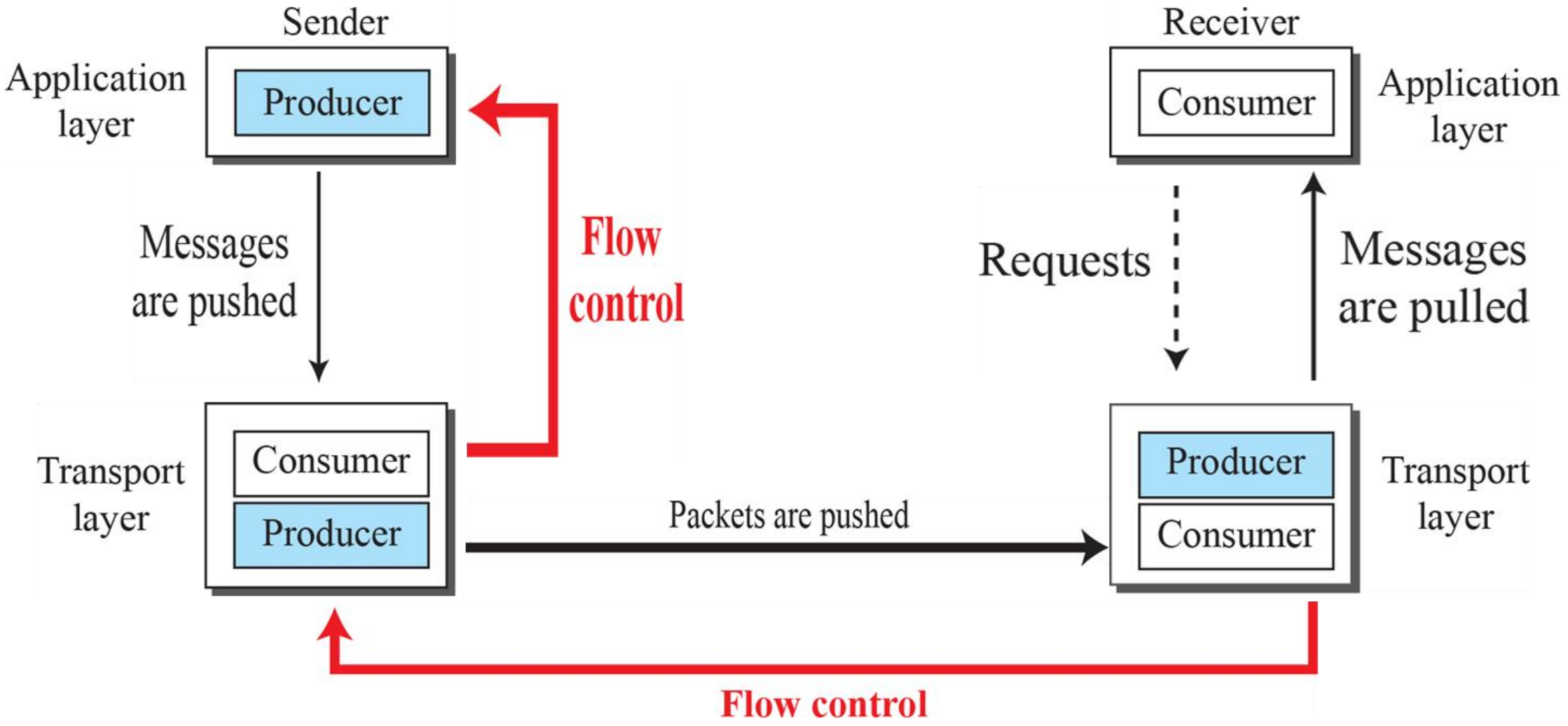
Flow Control at Transport Layer

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).


Sender
Receiver
Application layer
Producer
Consumer
Application layer
Messages are pushed
Flow control
Requests
Messages are pulled
Transport layer
Consumer
Producer
Packets are pushed
Producer
Consumer
Transport layer
Flow control

Buffers

Flow control ensures that data transfer between sender and receiver happens at a manageable pace. One common approach is to use two buffers—one at the sender's transport layer and the other at the receiver's transport layer.

Here's how it works:

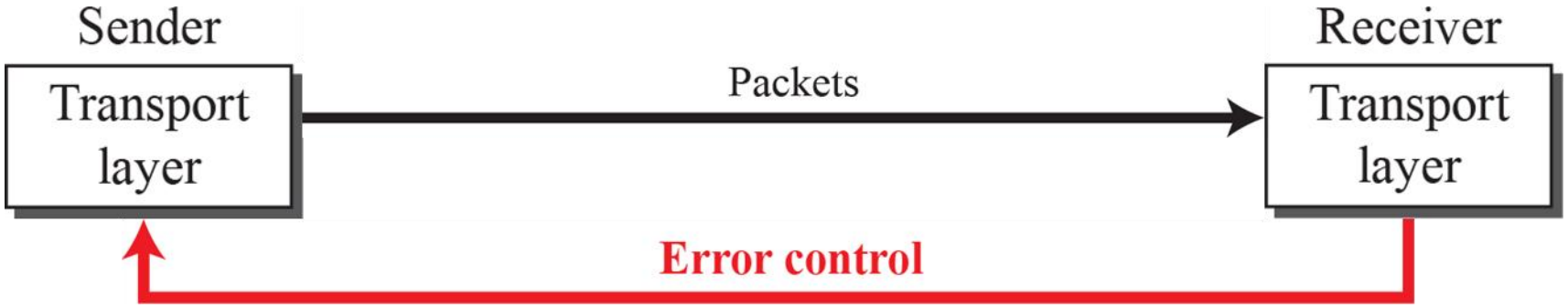
- Sender Buffer: This buffer at the sender's transport layer stores packets waiting to be sent. When it becomes full, the sender signals the application layer to stop sending message chunks. When there's space in the buffer, it tells the application layer to resume sending.
- Receiver Buffer: The receiver's 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 beachieved to add error control services to the transport layer. Error control at the transport layer is responsible for

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



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.

Packets are numbered sequentially. However, because we need to include the sequence number of each packet in the header, we need to set a limit. If the header of the packet allows m bits for the sequence number, the sequence numbers range from 0 to 2^(m) − 1. For example, if m is 4, the only sequence numbers are 0 through 15, inclusive.

For error control, the sequence numbers are modulo 2m, where m is the size of the sequence number field in bits.

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's how it works:

1. The receiver says, "I got your data, it's fine" by sending an acknowledgment (ACK) when it receives a bunch of data packets safely.
2. If some packets are damaged, the receiver just throws them away.
3. 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.
4. If the receiver gets the same packet twice by mistake, it just ignores the duplicate.
5. 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.

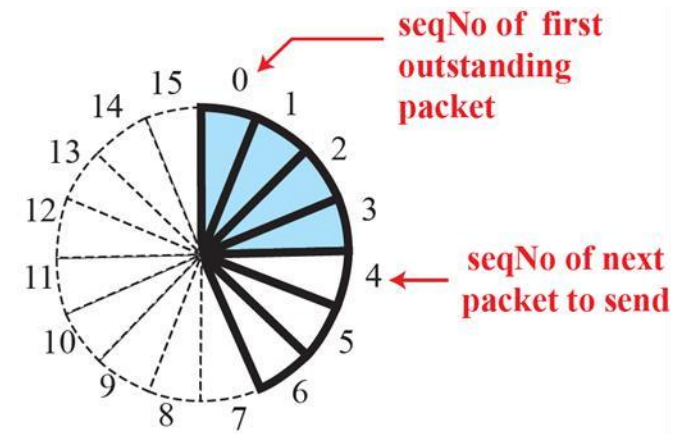
Congestion Control

congestion in a network, like the Internet, happens when there's more data trying to go through the network than it can handle. It's similar to a traffic jam on a freeway during rush hour.

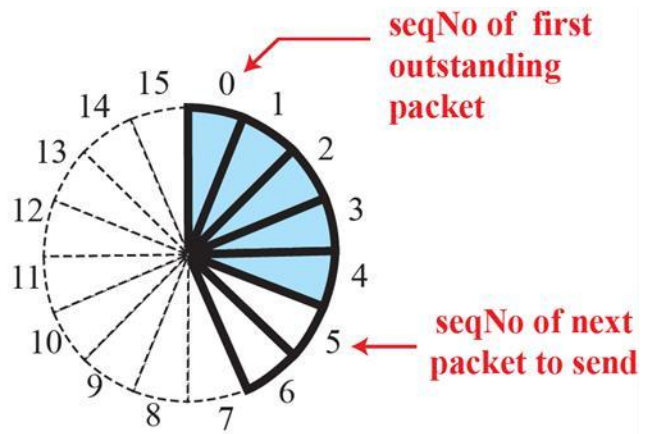
Here's why it occurs:

1. Networks have limits on how much data they can handle at once (capacity).
2. If too many data packets are sent to the network at once (load), it can't keep up.
3. Routers and switches in the network use queues to hold packets before and after processing.
4. If these queues get too full because packets are arriving faster than they can be processed, congestion occurs.

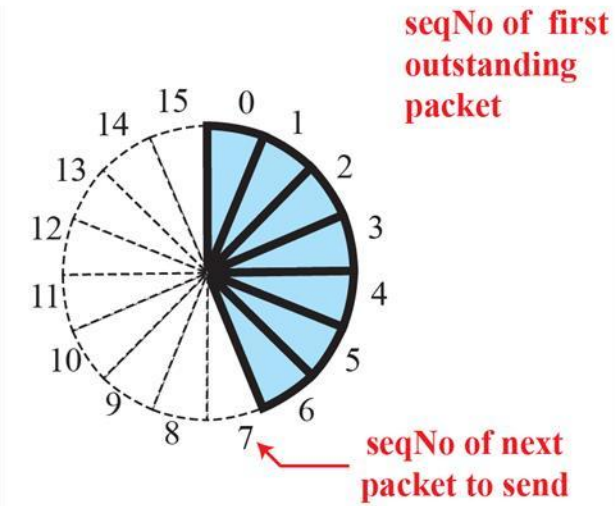
This congestion at the network level affects the transport layer, which is higher up in the networking hierarchy. TCP (a common protocol) has its own way to deal with congestion because it can't always rely on the network layer to control it.



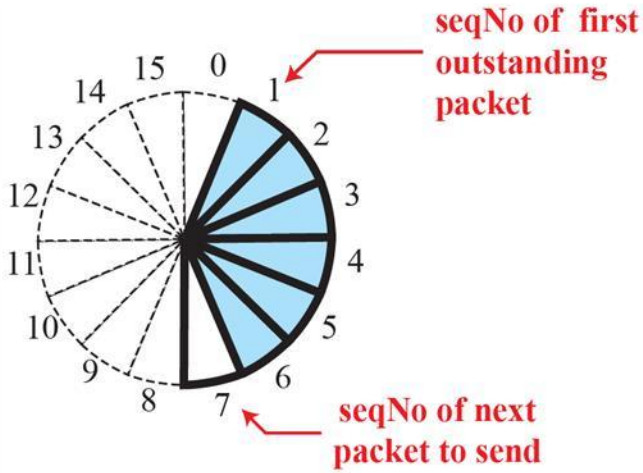
a. Four packets have been sent.



b. Five packets have been sent.

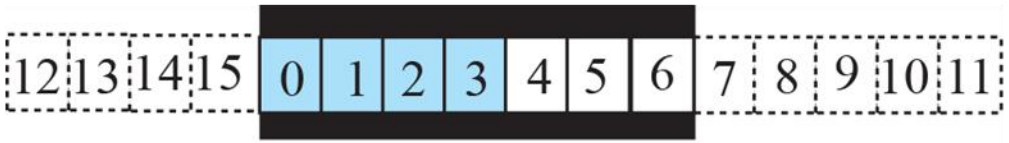


c. Seven packets have been sent;
window is full.

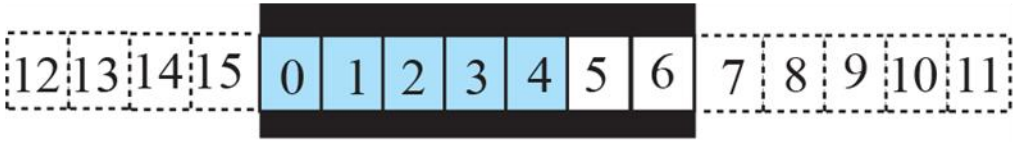


d. Packet 0 has been acknowledged;
window slides.

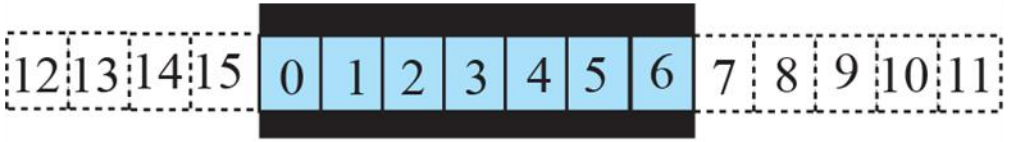
Figure 3.13: ***Sliding window in linear format***



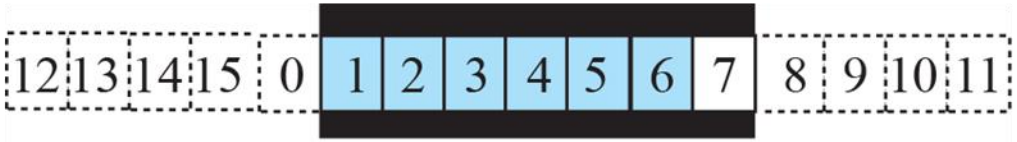
a. Four packets have been sent.



b. Five packets have been sent.



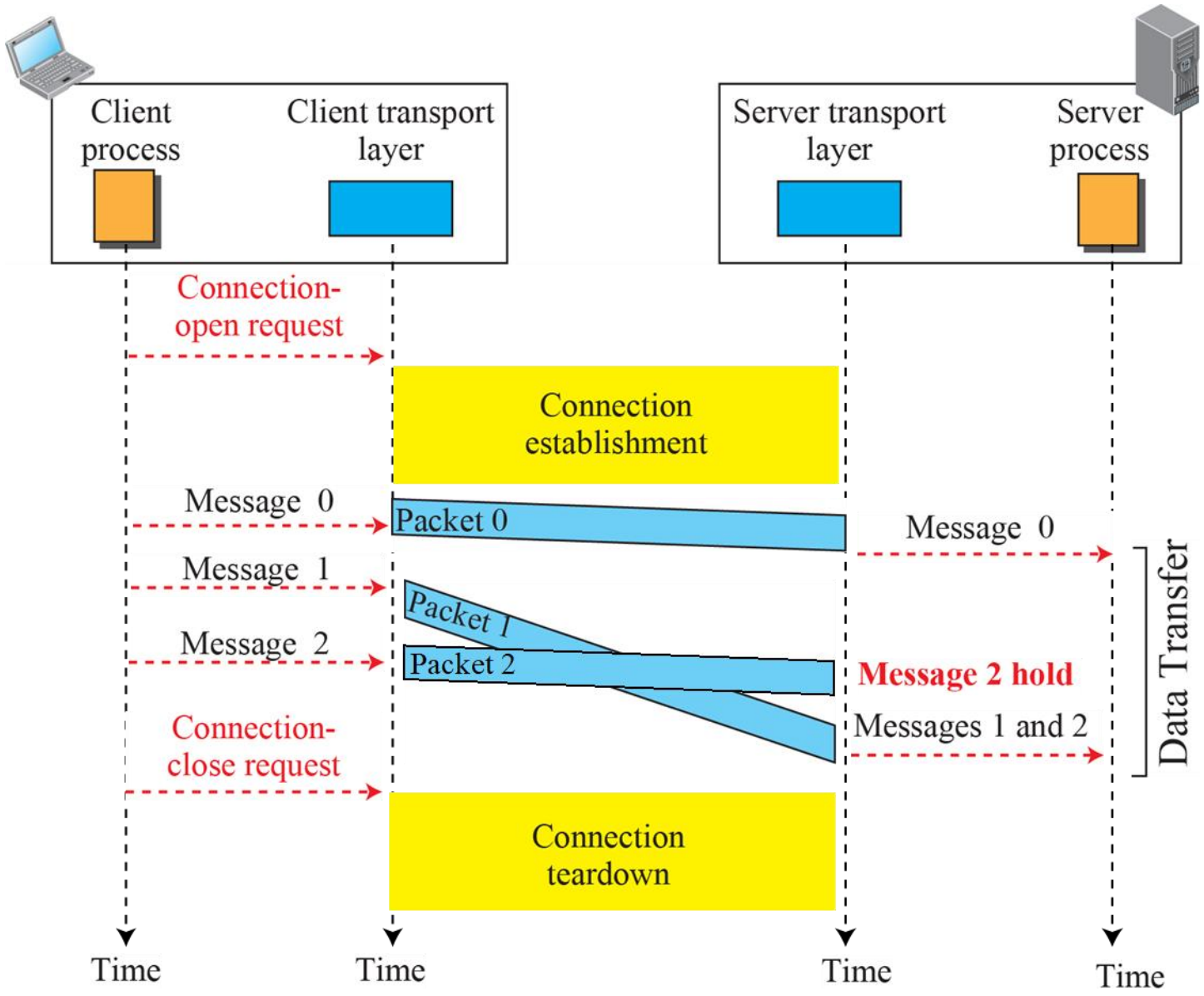
c. Seven packets have been sent;
window is full.



d. Packet 0 has been acknowledged;
window slides.

Connectionless and Connection-Oriented Services

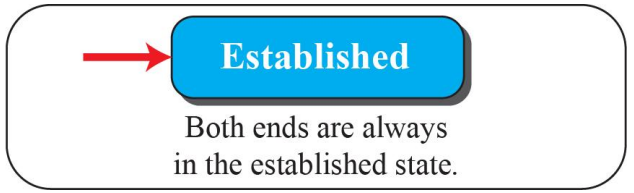
A transport-layer protocol, like a network-layer protocol, can provide two types of services: connectionless and connection-oriented. The nature of these services at the transport layer, however, is different from the ones at the network layer. **At the network layer, a connectionless service may mean different paths for different datagrams belonging to the same message**. At the transport layer, we are not concerned about the physical paths of packets (we assume a logical connection between two transport layers). **Connectionless service at the transport layer means independency between packets; connection-oriented means dependency**


Client process
Client transport layer
Server transport layer
Server process
Connection-open request
Connection establishment
Message 0
Packet 0
Message 0
Message 1
Packet 1
Message 2
Packet 2
Message 2 hold
Messages 1 and 2
Connection-close request
Connection teardown
Data Transfer
Time
Time
Time
Time

Finite State Machine

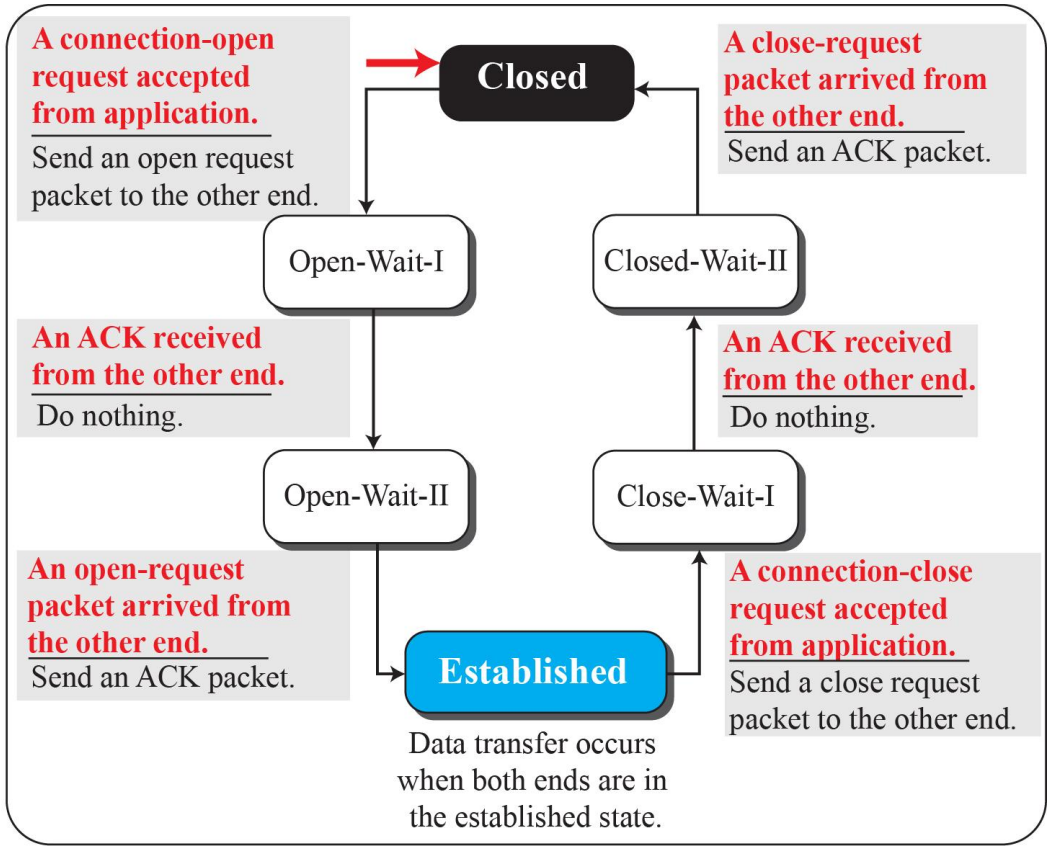
The behavior of a transport-layer protocol, both when it provides a connectionless and when it provides a connection-oriented protocol, can be better shown as a finite state machine (FSM).

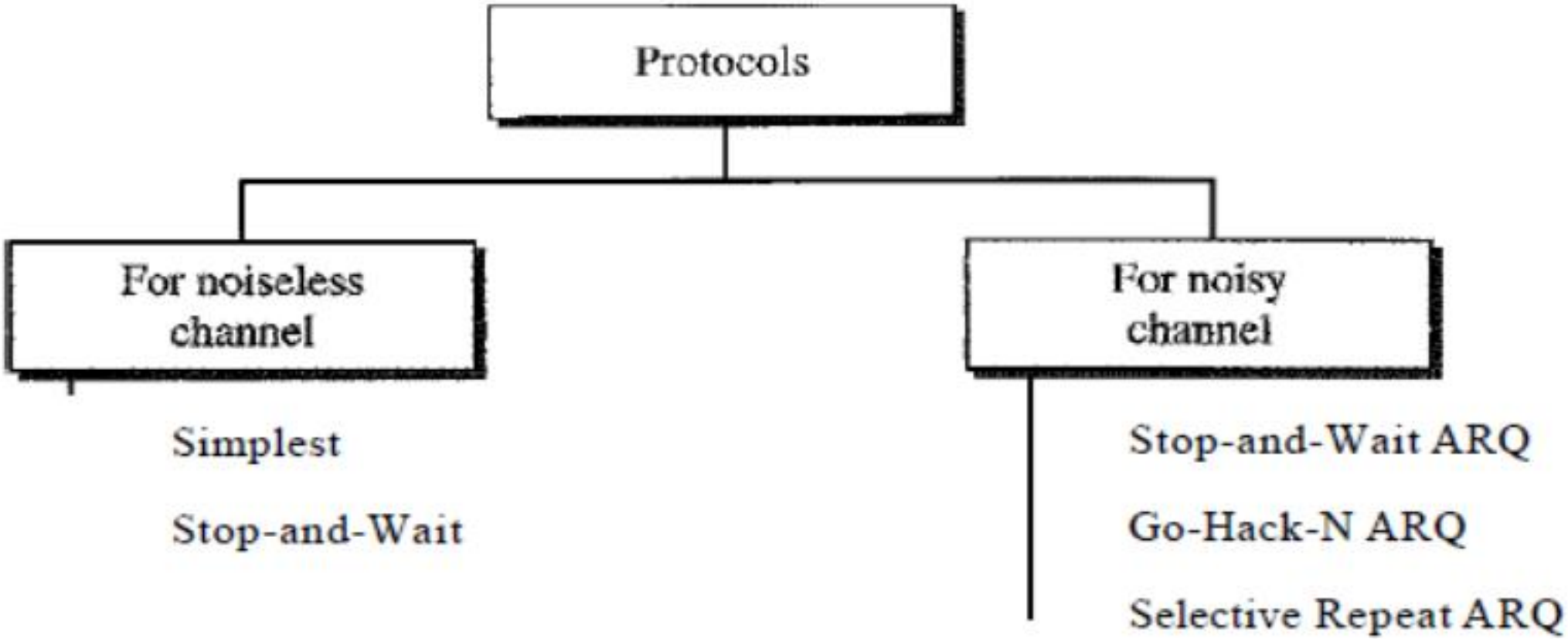
FSM for connectionless transport layer



Note:
The colored arrow shows the starting state.

FSM for connection-oriented transport layer





Flow Control Mechanism/Protocols

Since the simplest protocol is unidirectional, there is no acknowledgment (ACK). Also, as there is no data loss in the transmission, there is no need for data re-transmission.

The following assumptions have been made for developing the simplest protocol:

The transmission channel is completely noiseless (a channel in which no frames are lost, corrupted, or duplicated).

There is no error and flow control mechanism.

The buffer space for storing the frames at the sender's end and the receiver's end is infinite.

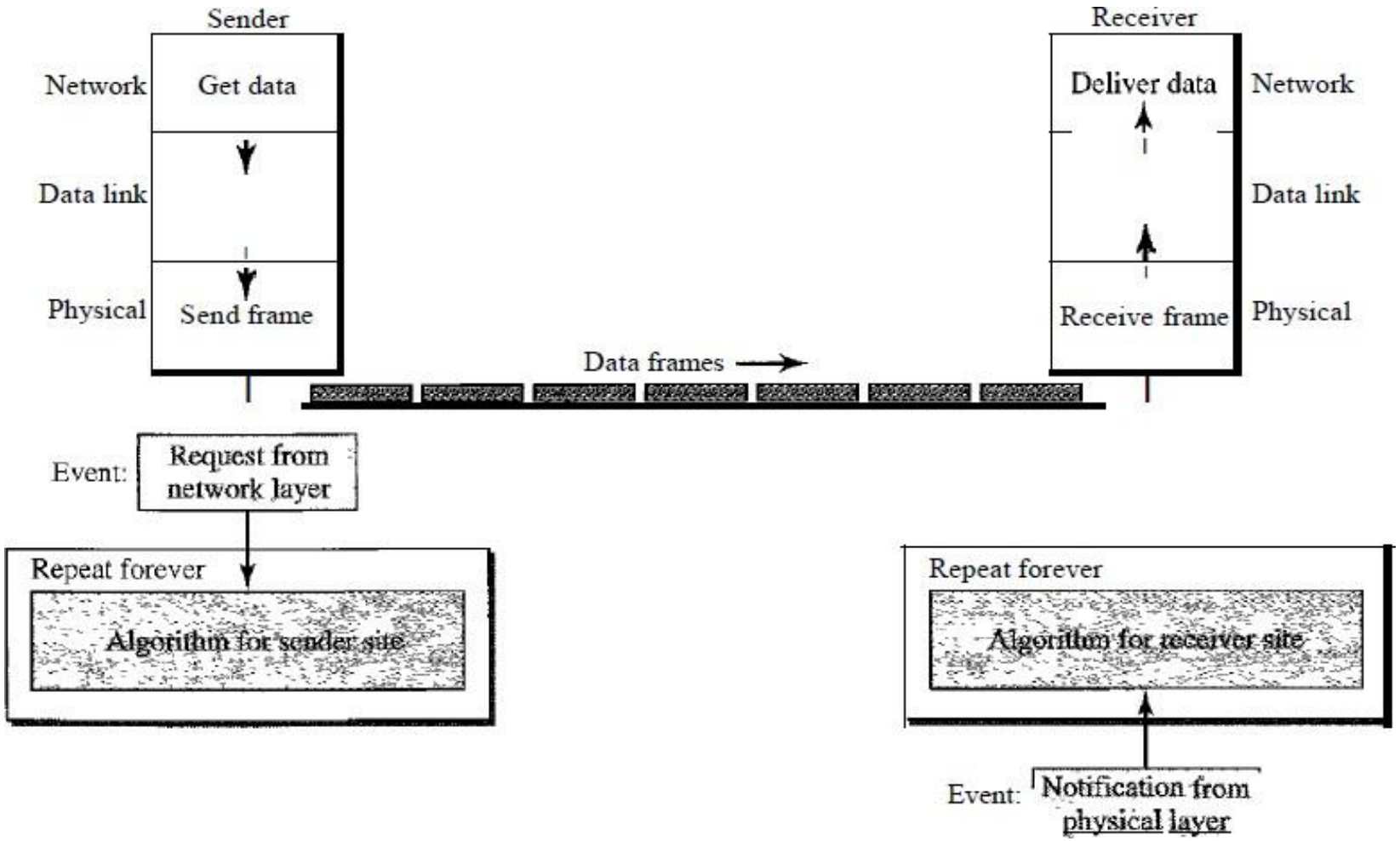
Important Points related to the simplest protocol:

The processing time of the simplest protocol is very short. Hence it can be neglected.

The sender and the receiver are always ready to send and receive data.

The sender sends a sequence of data frames without thinking about the receiver.

There is no data loss hence no ACK or NACK.



The design of the simplest protocol with no flow or error control

Stop–and–Wait Protocol

- ★ Stop – and – Wait protocol is data link layer protocol for transmission of frames over noiseless channels.
- ★ It provides unidirectional data transmission with flow control facilities but without error control facilities.
- ★ The idea of stop-and-wait protocol is straightforward.
- ★ After transmitting one frame, the sender waits for an acknowledgement before transmitting the next frame.

Primitives of Stop–and–Wait Protocol

Sender side

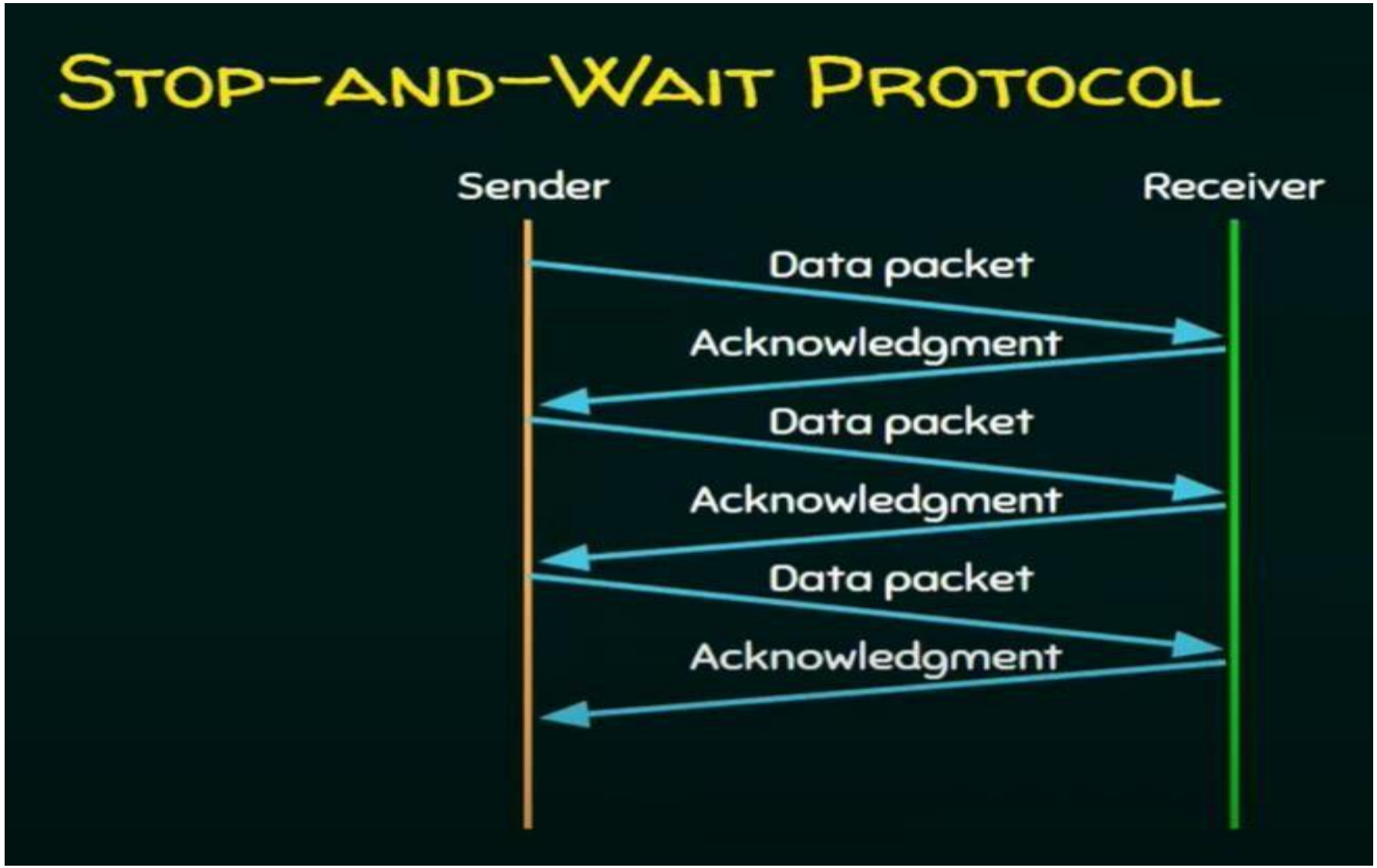
Rule 1 : Send one data packet at a time.

Rule 2 : Send the next packet only after receiving ACK for the previous.

Receiver side

Rule 1 : Receive and consume data packet.

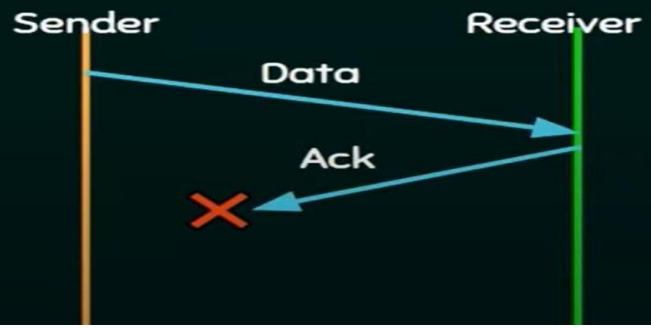
Rule 2 : After consuming packet, ACK need to be sent (Flow Control).


STOP-AND-WAIT PROTOCOL
Sender
Receiver
Data packet
Acknowledgment
Data packet
Acknowledgment
Data packet
Acknowledgment

Problems of Stop-and-Wait Protocol

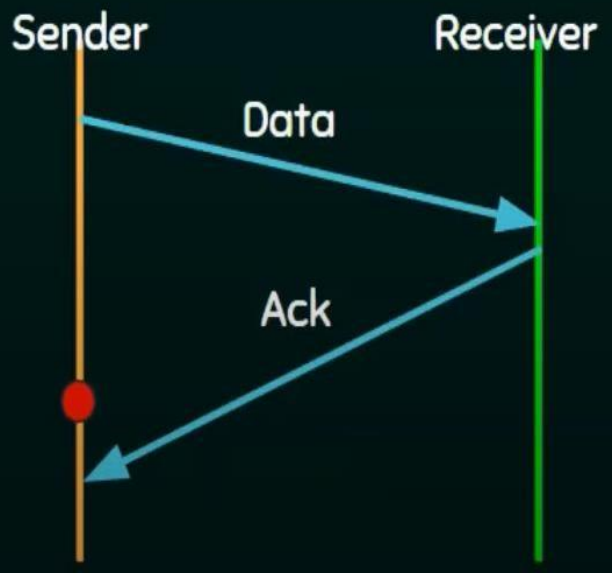
1. Problems due to lost data.
   - ★ Sender waits for ack for an infinite amount of time.
   - ★ Receiver waits for data an infinite amount of time.

2. Problems due to lost ACK.
   - ★ Sender waits for an infinite amount of time for ack.



3. Problems due to delayed ACK/data.

★ After timeout on sender side, a delayed ack might be wrongly considered as ack of some other data packet.



Analysis-

Now, let us analyze in depth how the transmission is actually carried out-

---

Sender puts the data packet on the transmission link.

Data packet propagates towards the receiver's end.

Data packet reaches the receiver and waits in its buffer.

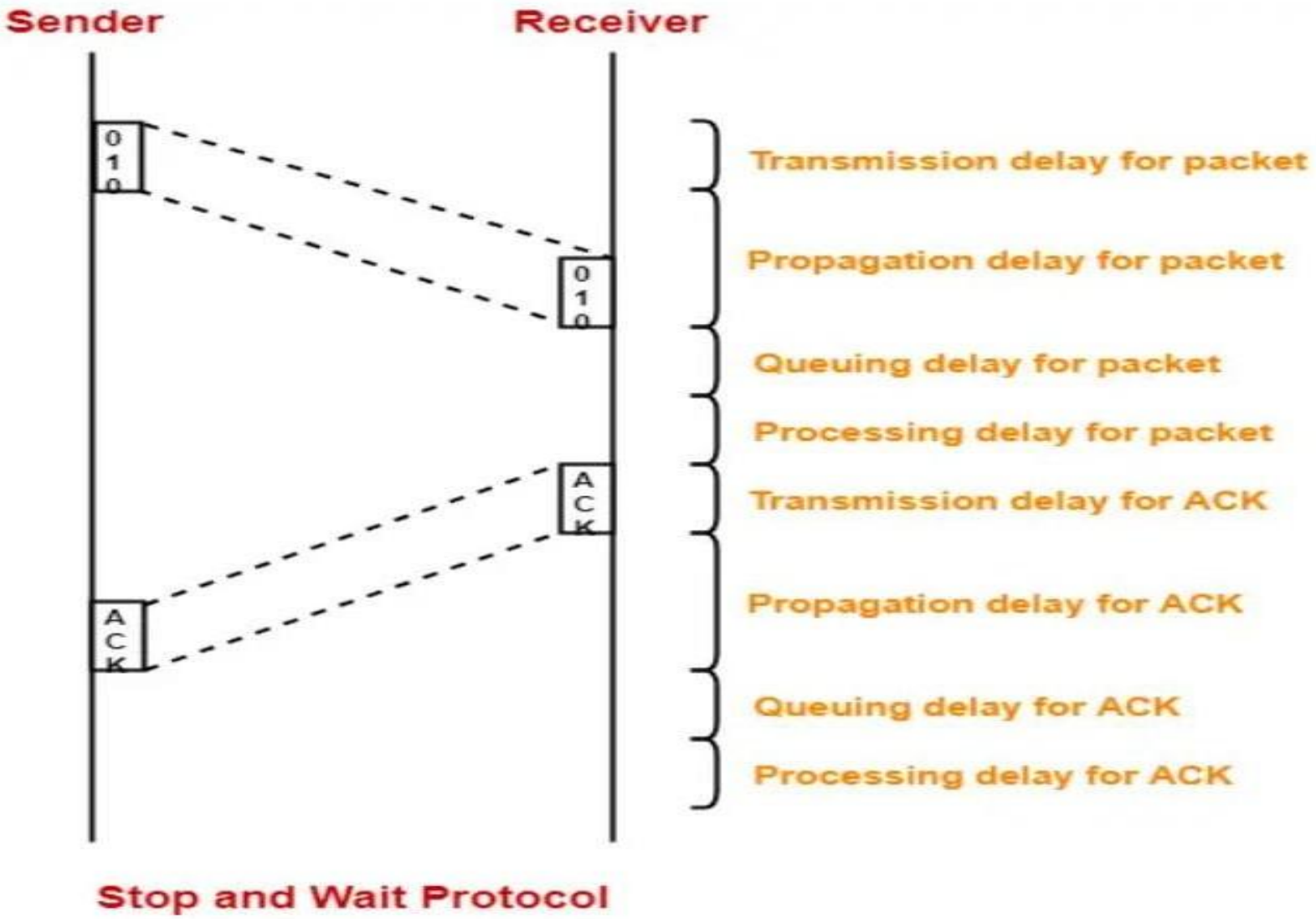
Receiver processes the data packet.

Receiver puts the acknowledgement on the transmission link.

Acknowledgement propagates towards the sender's end.

Acknowledgement reaches the sender and waits in its buffer.

Sender processes the acknowledgement.



Stop and Wait Protocol

Total time taken in sending one data packet:

= (Transmission delay + Propagation delay + Queuing delay + Processing delay)$_{packet}$ + (Transmission delay + Propagation delay + Queuing delay + Processing delay)$_{ACK}$

Assume-

Queuing delay and processing delay to be zero at both sender and receiver side.

Transmission time for the acknowledgement to be zero since it's size is very small.

Under the above assumptions:

Total time taken in sending one data packet:

= (Transmission delay + Propagation delay)$_{packet}$ + (Propagation delay)$_{ACK}$

We know,

Propagation delay depends on the distance and speed.

So, it would be same for both data packet and acknowledgement.

Therefore, Total time taken in sending one data packet

$= (\text{Transmission delay})_{packet} + 2 \times \text{Propagation delay}$

Efficiency of any flow control protocol is given by-

Efficiency (η) = Useful Time / Total Time

where-

Useful time = Transmission delay of data packet = $(\text{Transmission delay})_{packet}$

Useless time = Time for which sender is forced to wait and do nothing = 2 x Propagation delay

Total time = Useful time + Useless time

$$\text{Efficiency } (\eta) = \frac{(\text{Transmission delay})_{\text{packet}}}{(\text{Transmission delay})_{\text{packet}} + 2 \times \text{Propagation delay}}$$

OR

$$\text{Efficiency } (\eta) = \frac{T_t}{T_t + 2T_p}$$

OR

$$\text{Efficiency } (\eta) = \frac{1}{1 + 2\left(\frac{T_p}{T_t}\right)}$$

OR

$$\text{Efficiency } (\eta) = \frac{1}{1 + 2a}, \text{ where } a = \left(\frac{T_p}{T_t}\right)$$

Throughput: Number of bits that can be sent through the channel per second is called as its throughput.

Throughput = Efficiency * Bandwidth

Round Trip Time (RTT) = 2 * Propagation delay

Advantages-

The advantages of stop and wait protocol are-

It is very simple to implement.

The incoming packet from receiver is always an acknowledgement.

Limitations-

Point-01:

It is extremely inefficient because-

It makes the transmission process extremely slow.

It does not use the bandwidth entirely as each single packet and acknowledgement uses the entire time to traverse the link.

Important Notes-

Note-01:

Efficiency may also be referred by the following names-

- Line Utilization
- Link Utilization
- Sender Utilization
- Utilization of Sender

Note-02:

Throughput may also be referred by the following names-

- Bandwidth Utilization
- Effective Bandwidth
- Maximum data rate possible
- Maximum achievable throughput

Note-03:

Stop and Wait protocol performs better for LANs than WANs.

This is because-

Efficiency of the protocol is inversely proportional to the distance between sender and receiver.

So, the protocol performs better where the distance between sender and receiver is less.

The distance is less in LANs as compared to WANs.

Q. On a wireless link, the probability of packet error is 0.2. A stop and wait protocol is used to transfer data across the link. The channel condition is assumed to be independent from transmission to transmission. What is the average number of transmission attempts required to transfer 100 packets?

Probability of packet error = 0.2

We have to transfer 100 packets

When we transfer 100 packets, number of packets in which error will occur = 0.2 x 100 = 20.

Then, these 20 packets will have to be retransmitted.

When we retransmit 20 packets, number of packets in which error will occur = 0.2 x 20 = 4.

Then, these 4 packets will have to be retransmitted.

When we retransmit 4 packets, number of packets in which error will occur = 0.2 x 4 = 0.8 ≅ 1.

Then, this 1 packet will have to be retransmitted.

From here, average number of transmission attempts required = 100 + 20 + 4 + 1 = 125.

Q. If the bandwidth of the line is 1.5 Mbps, RTT is 45 msec and packet size is 1 KB, then find the link utilization in stop and wait.

Calculating Transmission Delay-

Transmission delay (T_t)

= Packet size / Bandwidth

= 1 KB / 1.5 Mbps

= (2^{10} x 8 bits) / (1.5 x 10^6 bits per sec)

= 5.461 msec

Calculating Propagation Delay-

Propagation delay (T_p)

= Round Trip Time / 2

= 45 msec / 2

= 22.5 msec

Calculating Value Of ‘a’-

$a = T_p / T_t$

$a = 22.5 \text{ msec} / 5.461 \text{ msec}$

$a = 4.12$

Calculating Link Utilization-

Link Utilization or Efficiency (η)

$= 1 / 1+2a$

$= 1 / (1 + 2 \times 4.12)$

$= 1 / 9.24$

$= 0.108$

$= 10.8\ \%$

Q. A channel has a bit rate of 4 Kbps and one way propagation delay of 20 msec. The channel uses stop and wait protocol. The transmission time of the acknowledgement frame is negligible. To get a channel efficiency of at least 50%, what would be the minimum frame size?

Given-

Bandwidth = 4 Kbps

Propagation delay (T_p) = 20 msec

Efficiency >= 50%

Let the required frame size = L bits.

Transmission delay (T_t)

= Packet size / Bandwidth

= L bits / 4 Kbps

$a = T_p / T_t$

a = 20 msec / (L bits / 4 Kbps)

a = (20 msec x 4 Kbps) / L bits

For efficiency to be at least 50%, we must have-

1 / 1+2a >= ½ **a <= ½**

Substituting the value of 'a', we get-

(20 msec x 4 Kbps) / L bits <= 1/2

L bits >= (20 msec x 4 Kbps) x 2

L bits >= (20 x 10^{-3} sec x 4 x 10^3 bits per sec) x 2

L bits >= 20 x 4 bits x 2

L >= 160

Please visit the link: https://www.gatevidyalay.com/stop-and-wait-protocol-practice-problems/ for more problems on Stop-and-Wait protocol.

Stop-and-Wait Automatic Repeat Request (ARQ)

In stop and wait protocol,

Sender sends one data packet and then waits for its acknowledgement.

Sender sends the next packet only after it receives the acknowledgement for the previous packet.

The main problem faced by the Stop and Wait protocol is the occurrence of deadlock due to-

Loss of data packet

Loss of acknowledgement

Stop and Wait ARQ is an improved and modified version of Stop and Wait protocol.

Stop and Wait ARQ assumes-

The communication channel is noisy.

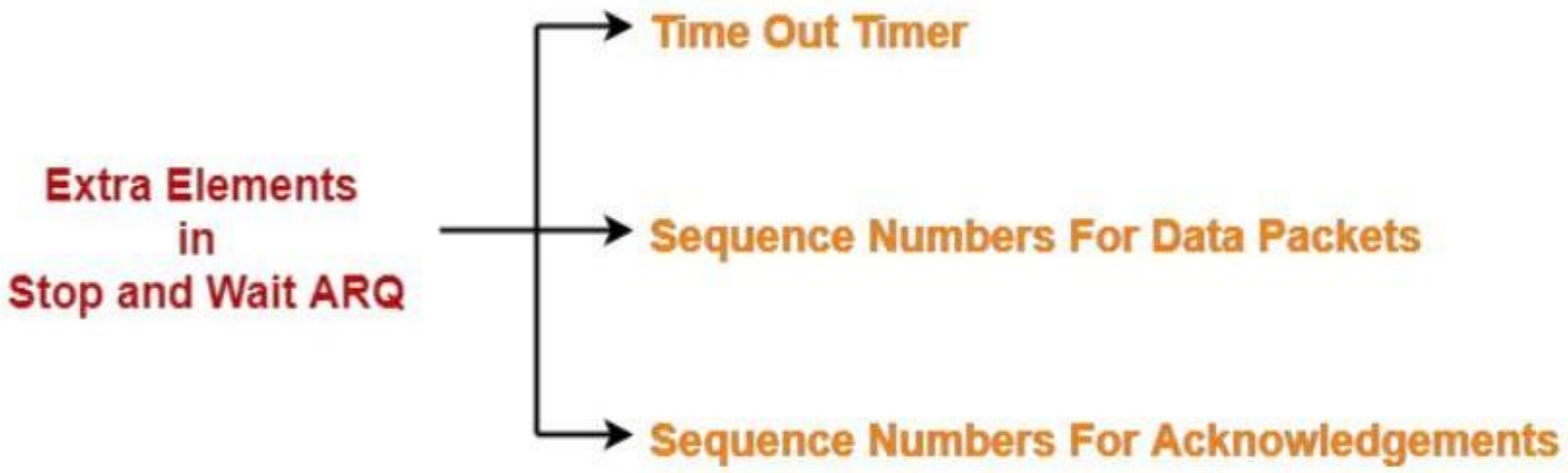
Errors may get introduced in the data during the transmission.

Working-

Stop and wait ARQ works similar to stop and wait protocol.

It provides a solution to all the limitations of stop and wait protocol.

Stop and wait ARQ includes the following three extra elements:



Thus, we can say-

Stop and Wait ARQ

= Stop and Wait Protocol + Time Out Timer + Sequence Numbers for Data Packets and Acknowledgements

In stop and wait ARQ, Minimum number of **sequence numbers** required:

= Sender Window Size + Receiver Window Size

= 1 + 1

= 2

Thus,

Minimum number of sequence numbers required in Stop and Wait ARQ = 2.

The two sequence numbers used are 0 and 1.

How Stop and Wait ARQ Solves All Problems?

1. Problem of Lost Data Packet-

Time out timer helps to solve the problem of lost data packet.

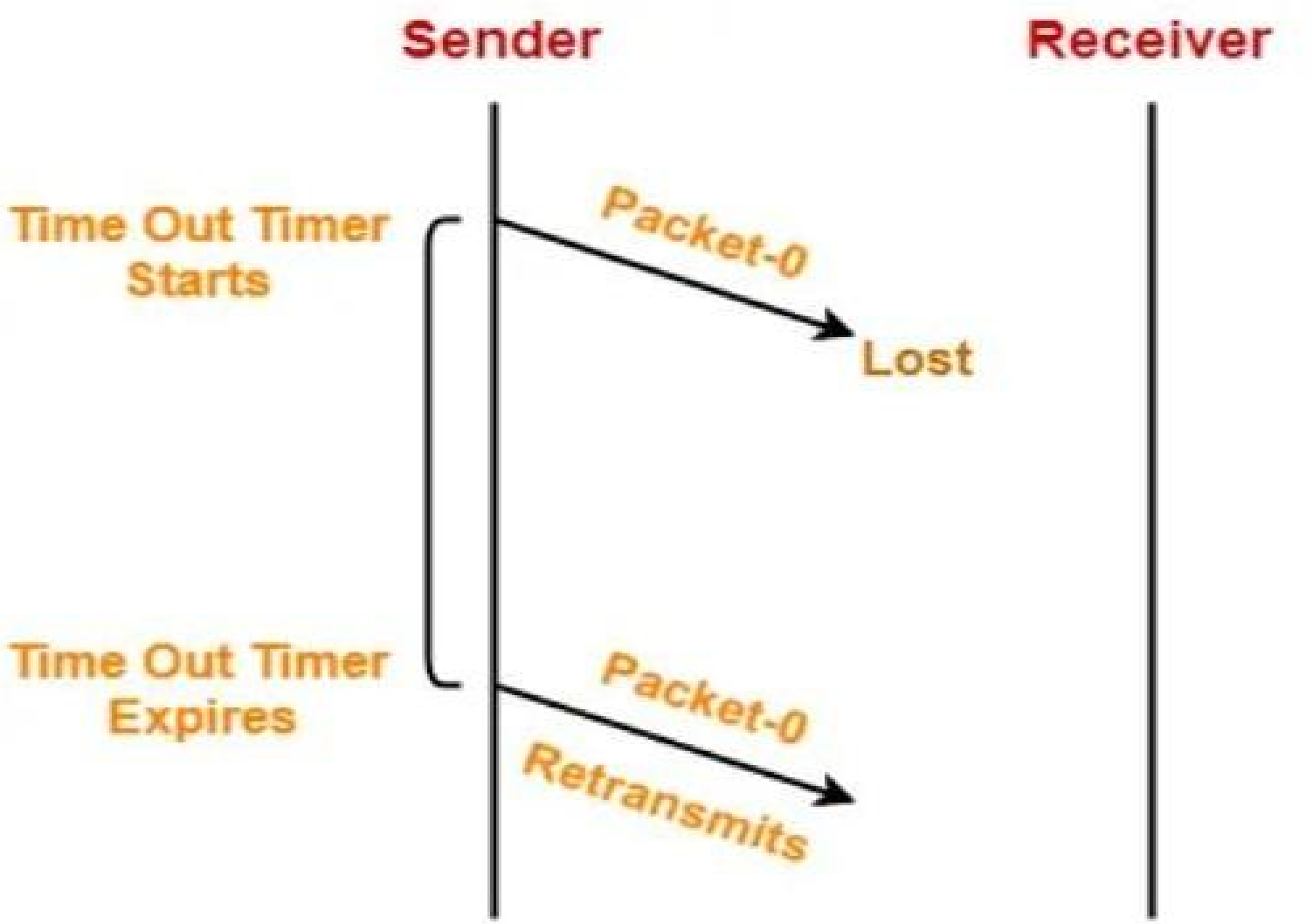
After sending a data packet to the receiver, sender starts the time out timer.

If the data packet gets acknowledged before the timer expires, sender stops the time out timer.

If the timer goes off before receiving the acknowledgement, sender retransmits the same data packet.

After retransmission, sender resets the timer.

This prevents the occurrence of deadlock.


Sender
Receiver
Time Out Timer Starts
Packet-0
Lost
Time Out Timer Expires
Packet-0
Retransmits

2. Problem of Lost Acknowledgement-

Sequence number on data packets help to solve the problem of delayed acknowledgement.

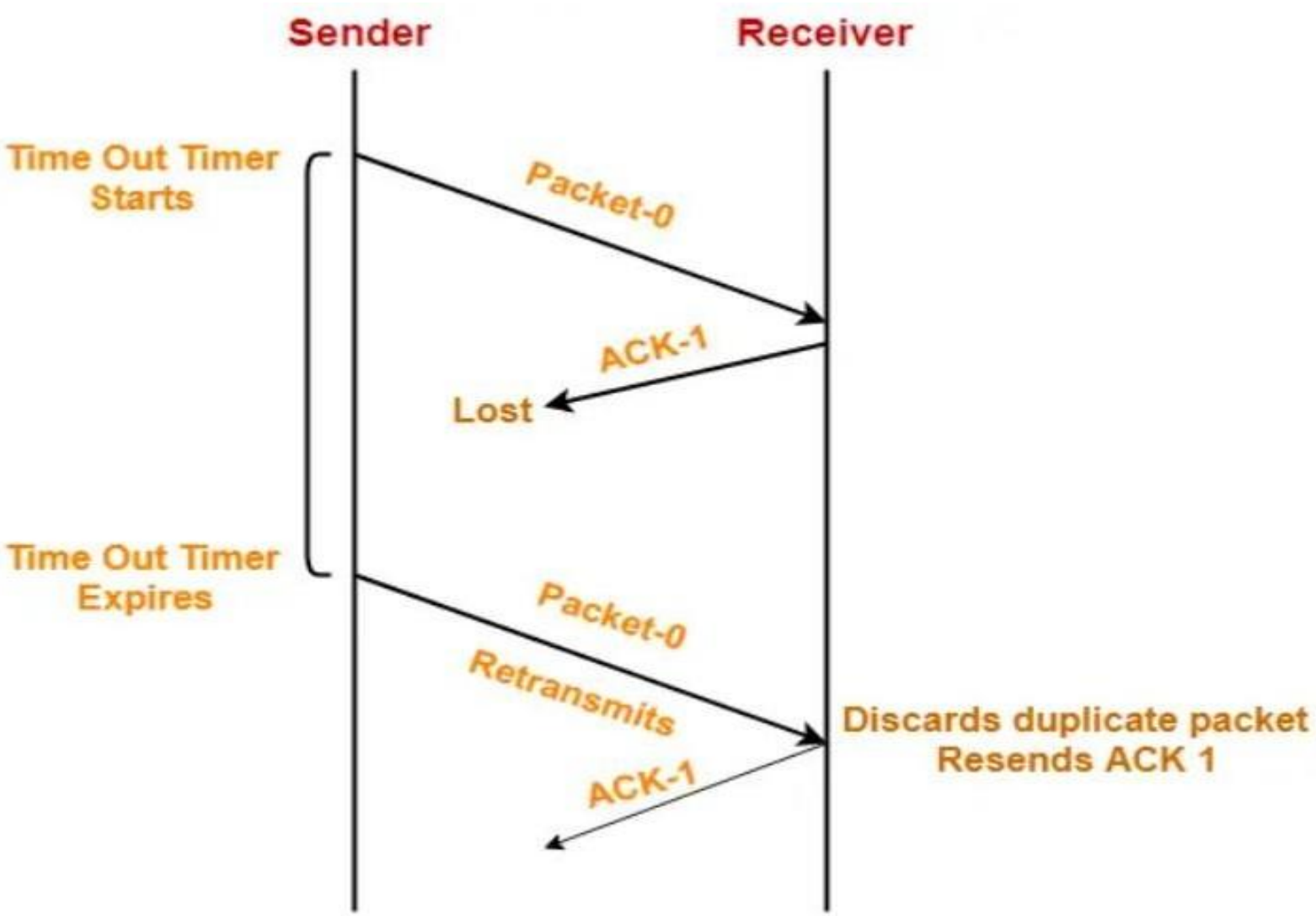
Consider the acknowledgement sent by the receiver gets lost.

Then, sender retransmits the same data packet after its timer goes off.

This prevents the occurrence of deadlock.

The sequence number on the data packet helps the receiver to identify the duplicate data packet.

Receiver discards the duplicate packet and re-sends the same acknowledgement.


Sender
Receiver
Time Out Timer Starts
Packet-0
ACK-1
Lost
Time Out Timer Expires
Packet-0
Retransmits
Discards duplicate packet
Resends ACK 1
ACK-1

Role of Sequence Number on Data Packets:

Sender sends a data packet with sequence number-0 to the receiver.

Receiver receives the data packet correctly.

Receiver now expects data packet with sequence number-1.

Receiver sends the acknowledgement ACK-1.

Acknowledgement ACK-1 sent by the receiver gets lost on the way.

Sender receives no acknowledgement and time out occurs.

Sender retransmits the same data packet with sequence number-0.

This will be a duplicate packet for the receiver.

Receiver receives the data packet and discovers it is the duplicate packet.

It expects the data packet with sequence number-1 but receiving the data packet with sequence number-0.

It discards the duplicate data packet and re-sends acknowledgement ACK-1.

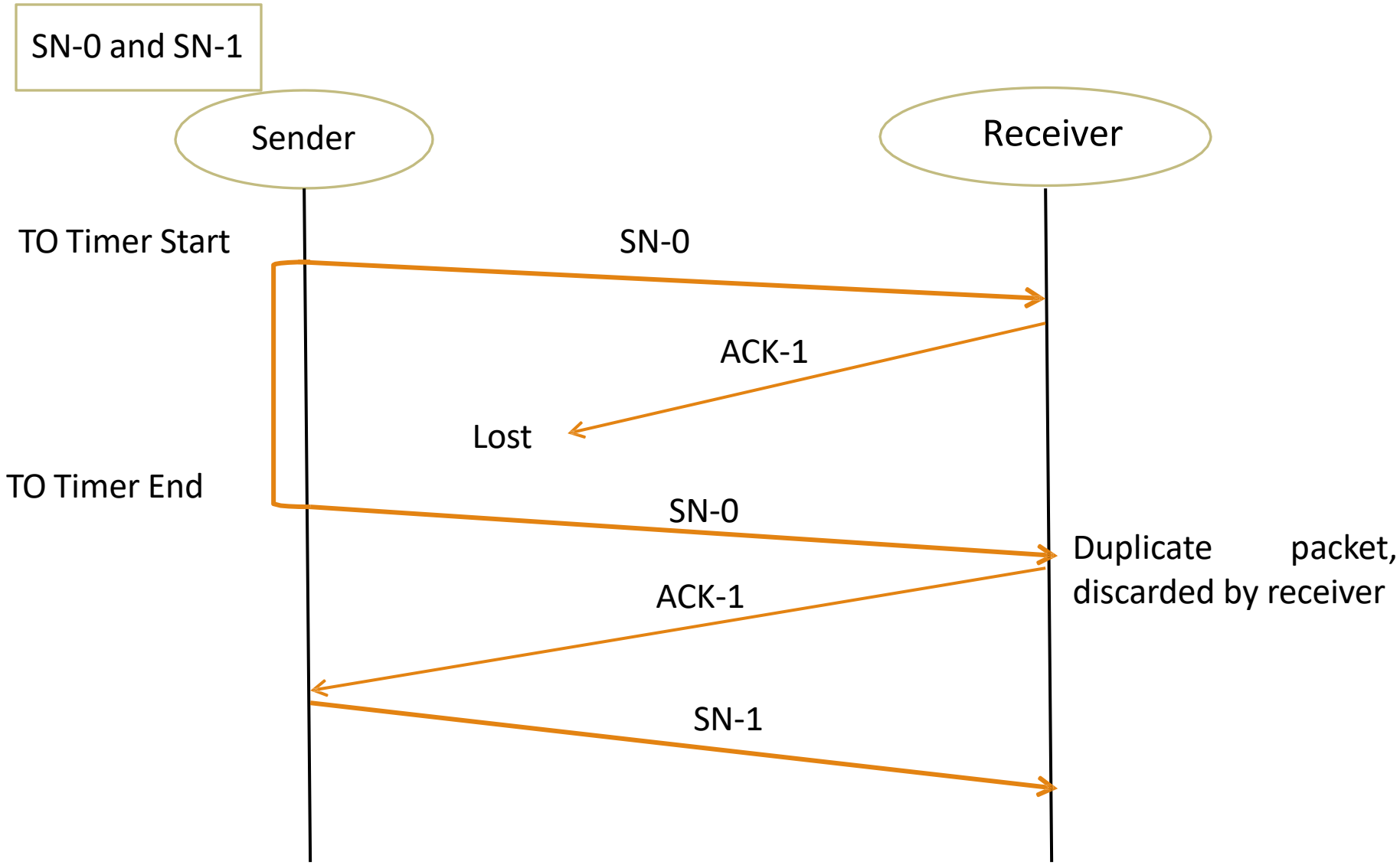
ACK-1 requests the sender to send a data packet with sequence number-1.

This avoids the inconsistency of data.

Conclusion-

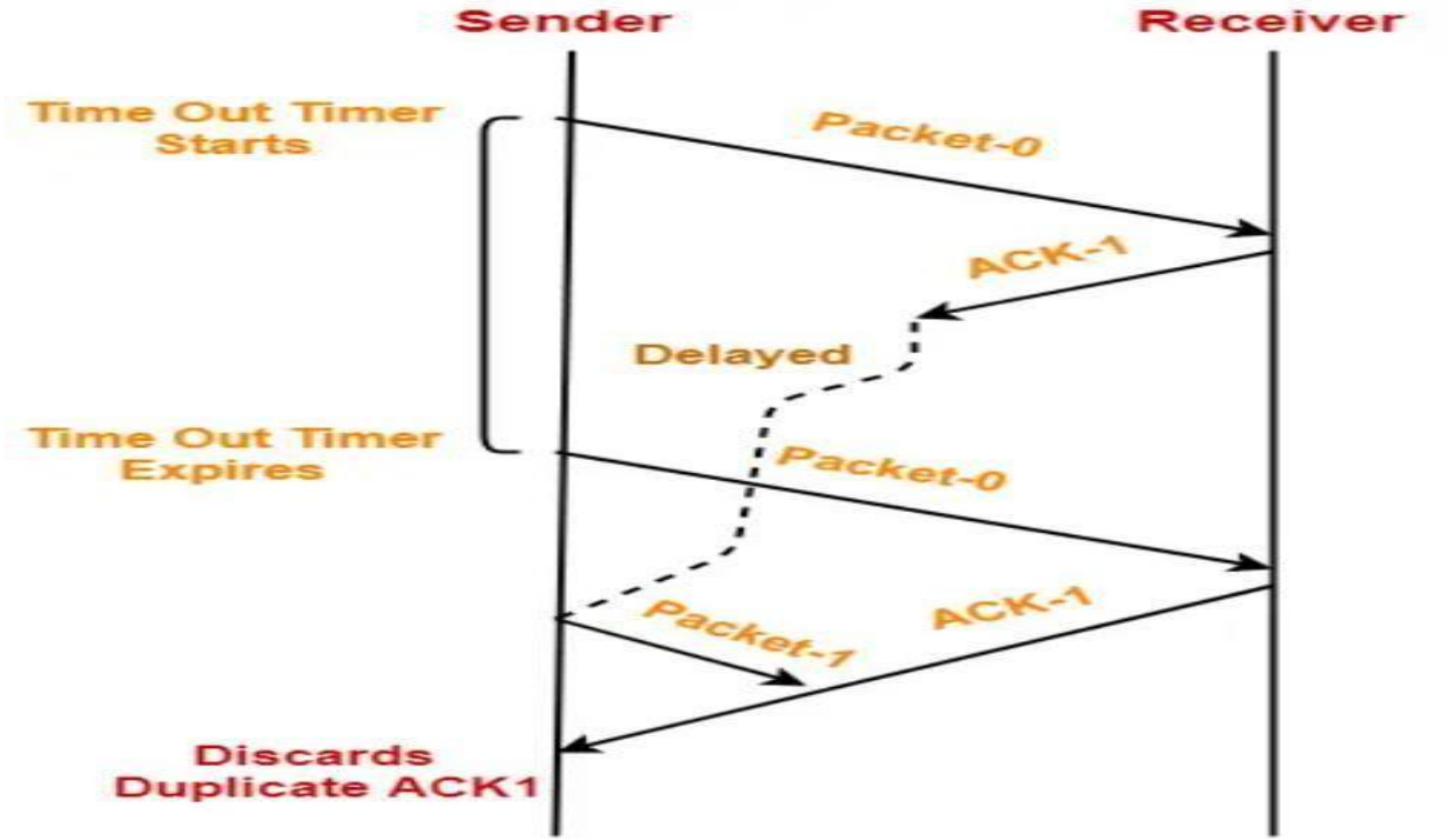
Had the sequence numbers not been allotted to the data packets, receiver would have accepted the duplicate data packet thinking of it as the new data packet.

This is how sequence numbers allotted to the data packets prove to be useful for identifying the duplicate data packets and discarding them.


SN-0 and SN-1
Sender
Receiver
TO Timer Start
SN-0
ACK-1
Lost
TO Timer End
SN-0
Duplicate packet, discarded by receiver
ACK-1
SN-1

3. Problem of Delayed Acknowledgement-

Sequence number on acknowledgements help to solve the problem of delayed acknowledgement.



Role of Sequence Number on Acknowledgements:

Sender sends a data packet with sequence number-0 to the receiver.

Receiver receives the data packet correctly.

Receiver now expects data packet with sequence number-1.

Receiver sends the acknowledgement ACK-1.

Acknowledgement ACK-1 sent by the receiver gets delayed in reaching the sender.

Sender receives no acknowledgement and time out occurs.

Sender retransmits the same data packet with sequence number-0.

This will be a duplicate packet for the receiver.

Receiver receives the data packet and discovers it is the duplicate packet.

It expects the data packet with sequence number-1 but receiving the data packet with sequence number-0.

It discards the duplicate data packet and re-sends acknowledgement ACK-1.

ACK-1 requests the sender to send a data packet with sequence number-1.

Two acknowledgements ACK1 reaches the sender.

When first acknowledgement ACK1 reaches the sender, sender sends the next data packet with sequence number 1.

When second acknowledgement ACK1 reaches the sender, sender rejects the duplicate acknowledgement.

This is because it has already sent the data packet with sequence number-1 and now sender expects the acknowledgement with sequence number 0 from the receiver.

Conclusion-

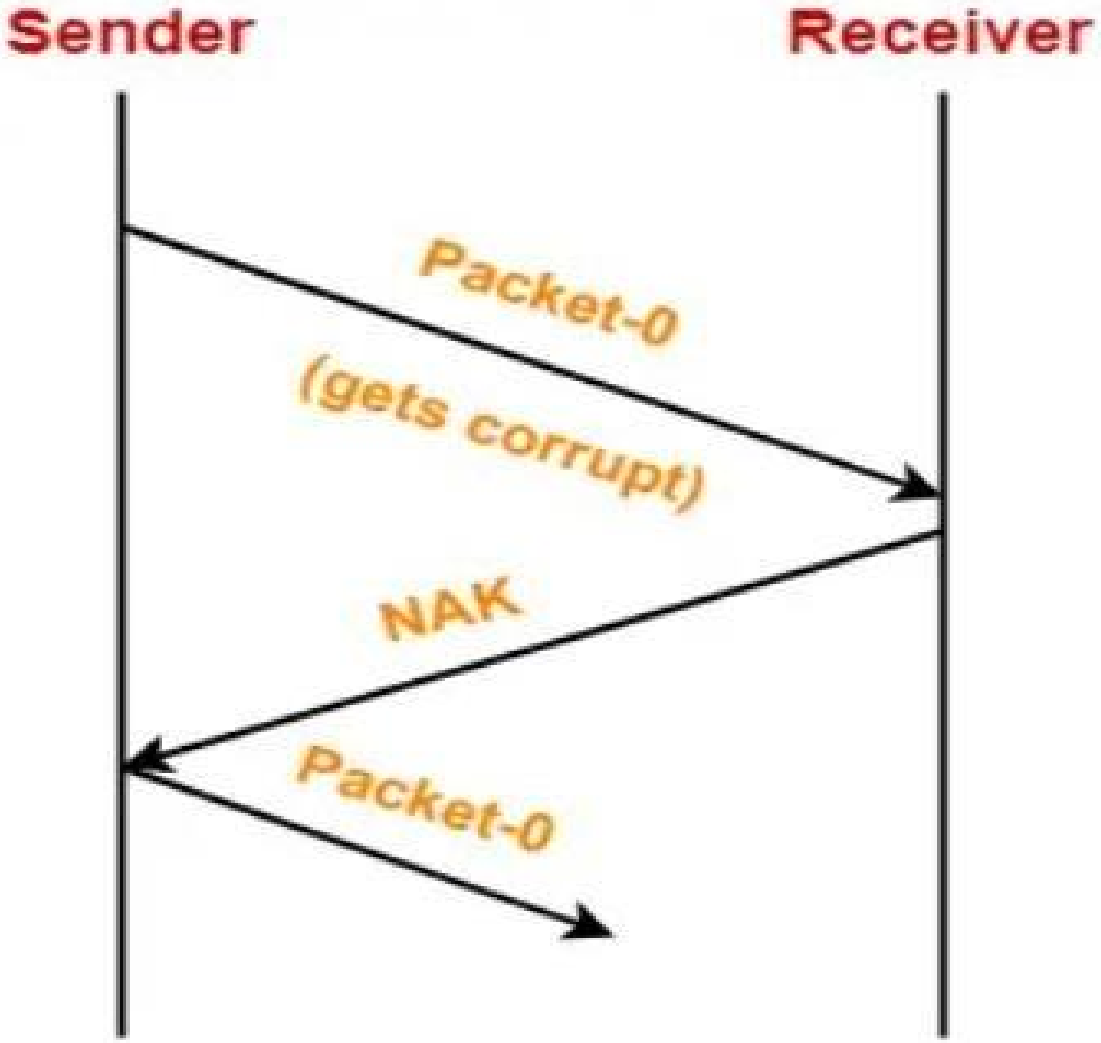
Had the sequence numbers not been allotted to the acknowledgements, sender would have accepted the duplicate acknowledgement thinking of it as the new acknowledgement for the latest data packet sent by it.

This is how sequence numbers allotted to the acknowledgements prove to be useful for identifying duplicate acknowledgements and discarding them.

4. Problem of Damaged Packet-

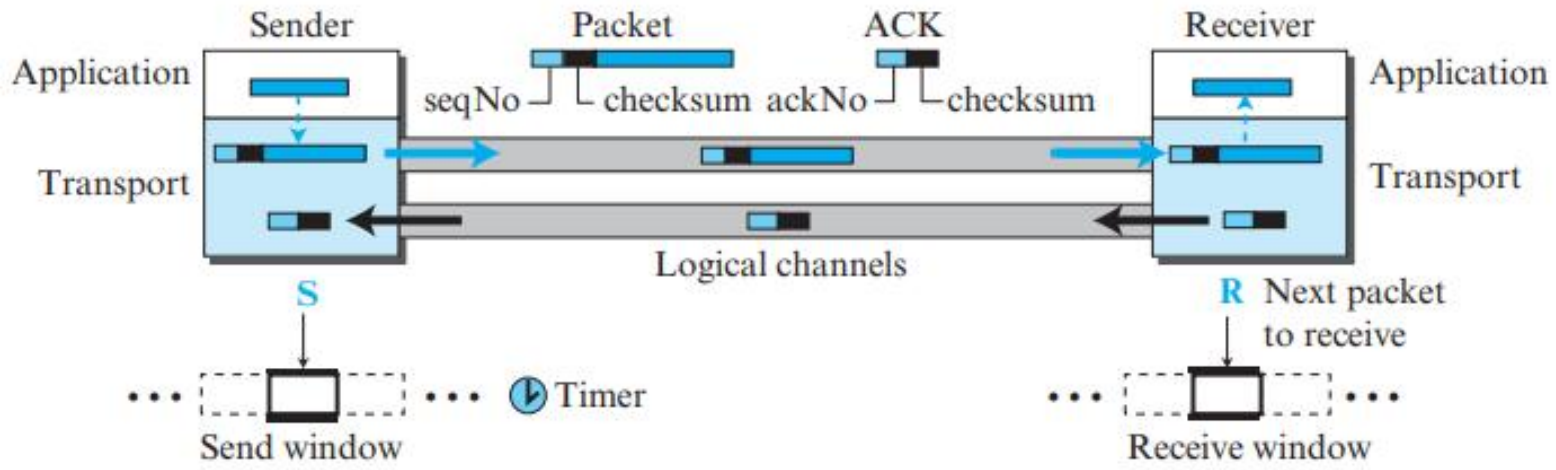
If receiver receives a corrupted data packet from the sender, it sends a negative acknowledgement (NAK) to the sender.

NAK requests the sender to send the data packet again.


Sender
Receiver
Packet-0
(gets corrupt)
NAK
Packet-0

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.

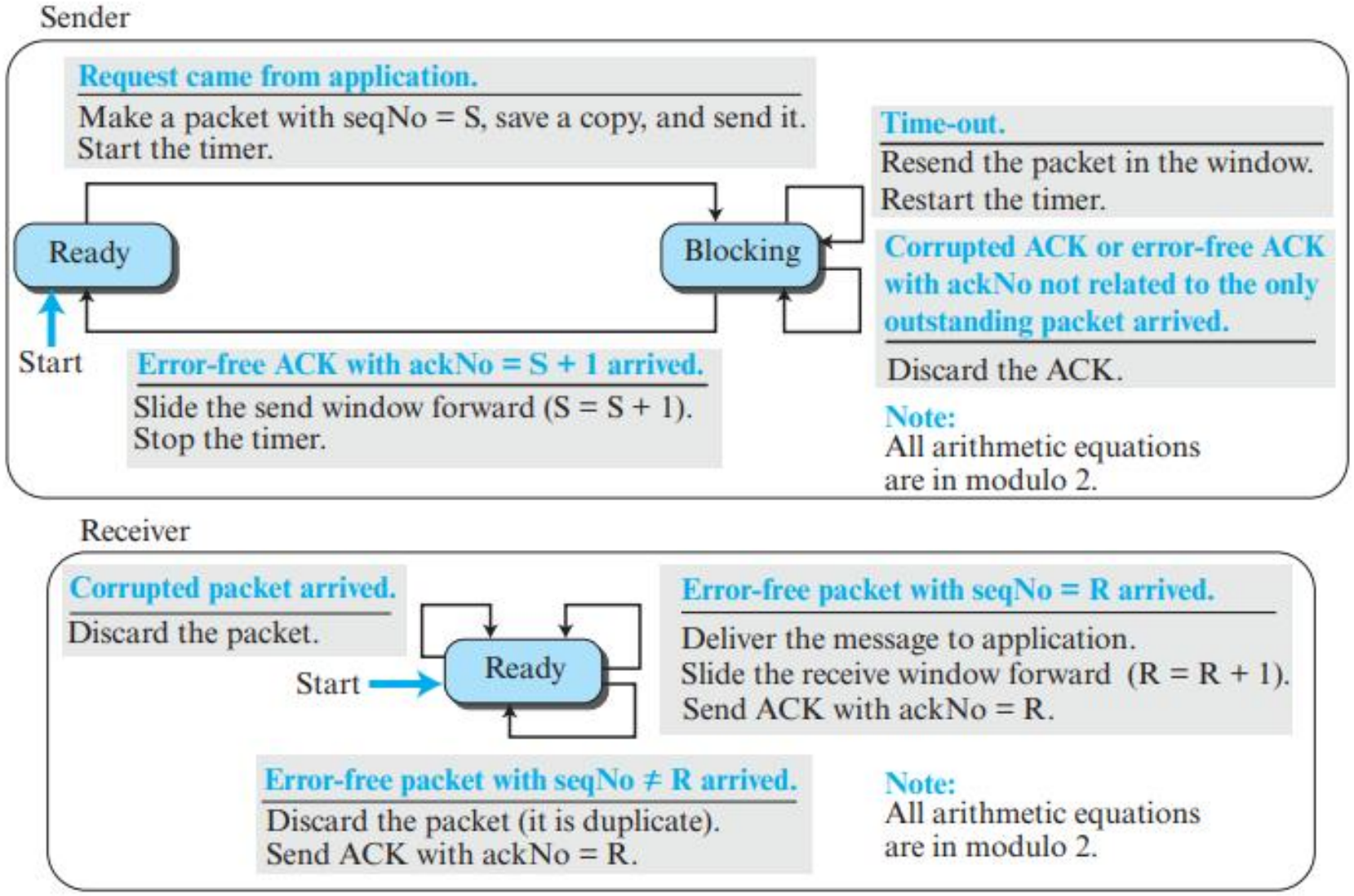
➢This means that the sequence is 0, 1, 0, 1, 0, and so on. This is referred to as modulo 2 arithmetic.

Acknowledgement Numbers

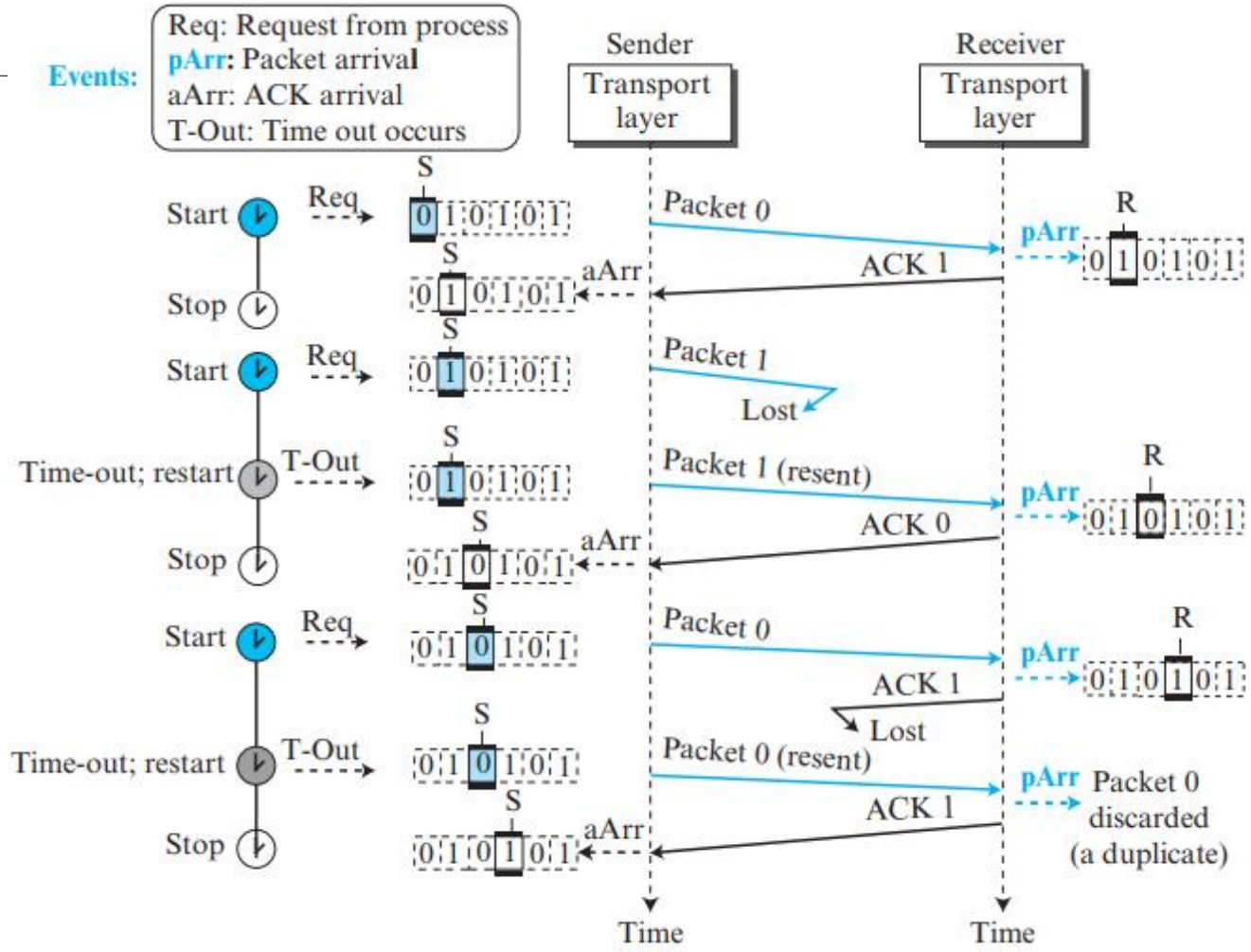
Since the sequence numbers must be suitable for both data packets and acknowledgments, we use this convention: *The acknowledgment numbers always announce the sequence number of the next packet expected by the receiver.* For example, if packet 0 has arrived safe and sound, the receiver sends an ACK with 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?

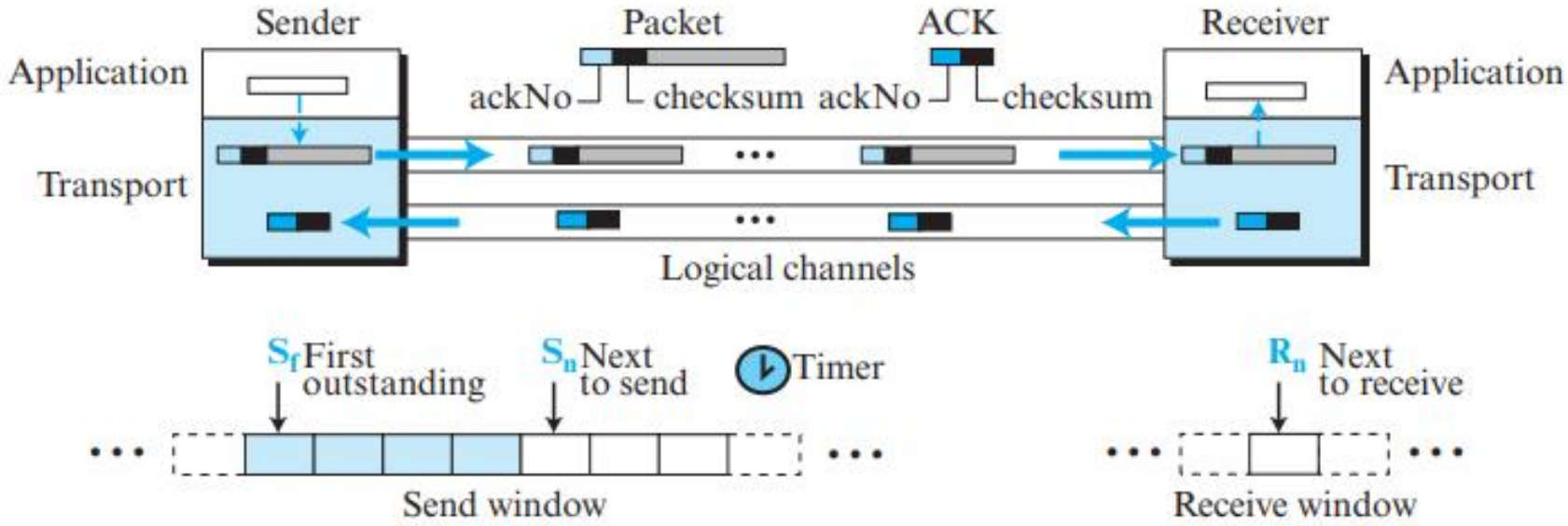
Solution

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.

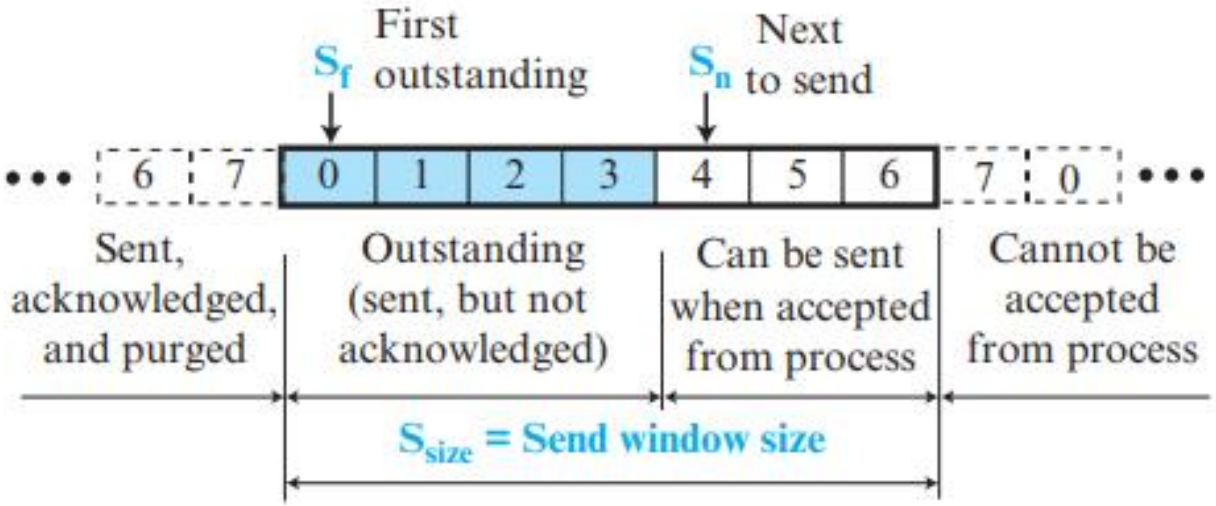
Acknowledgement Numbers -

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.

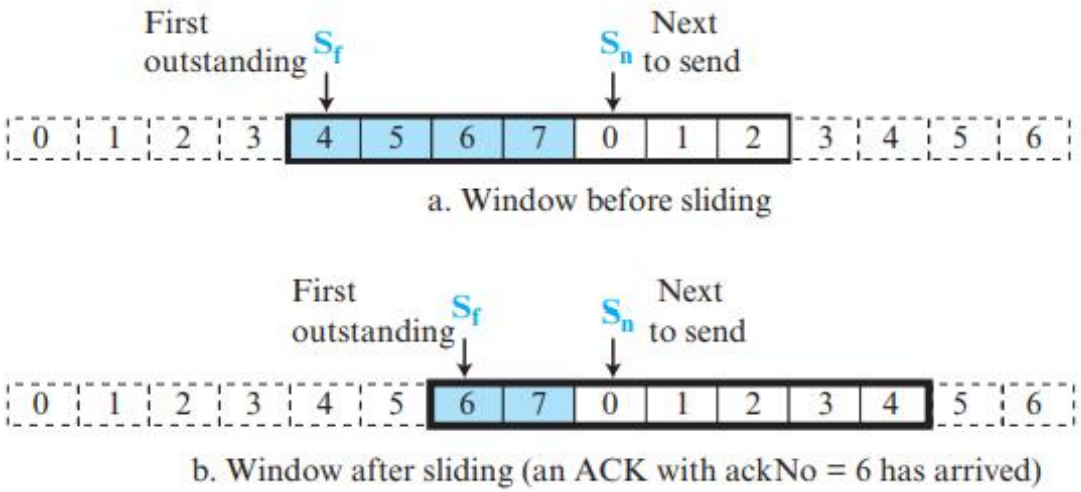
Send Window

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.



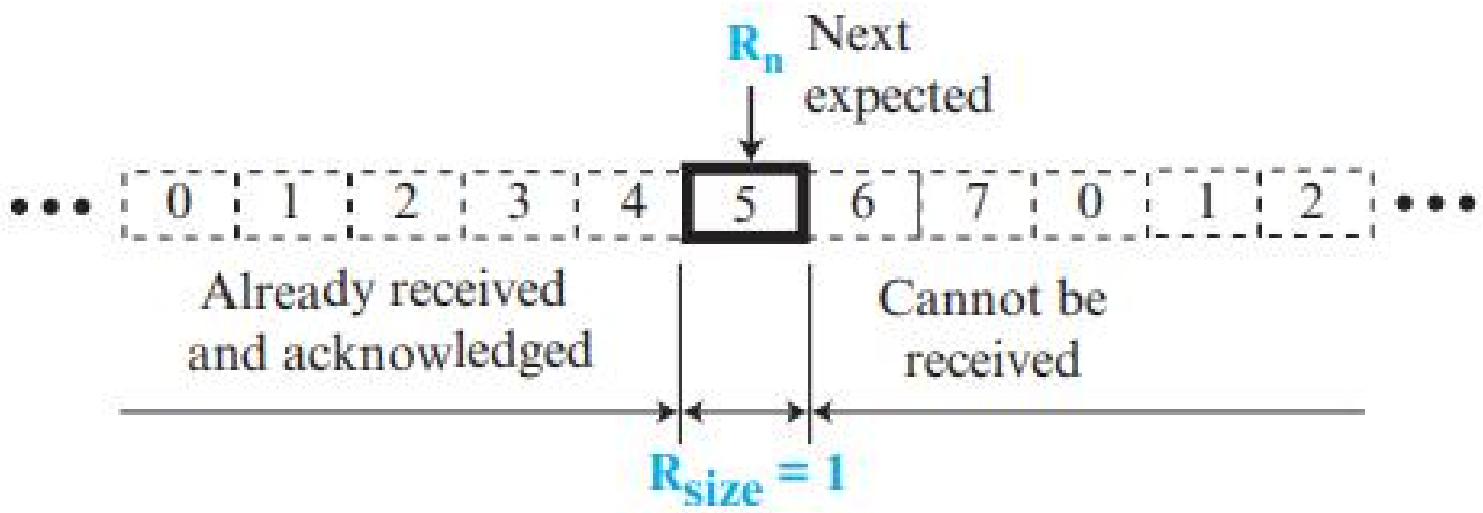
Send Window

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



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

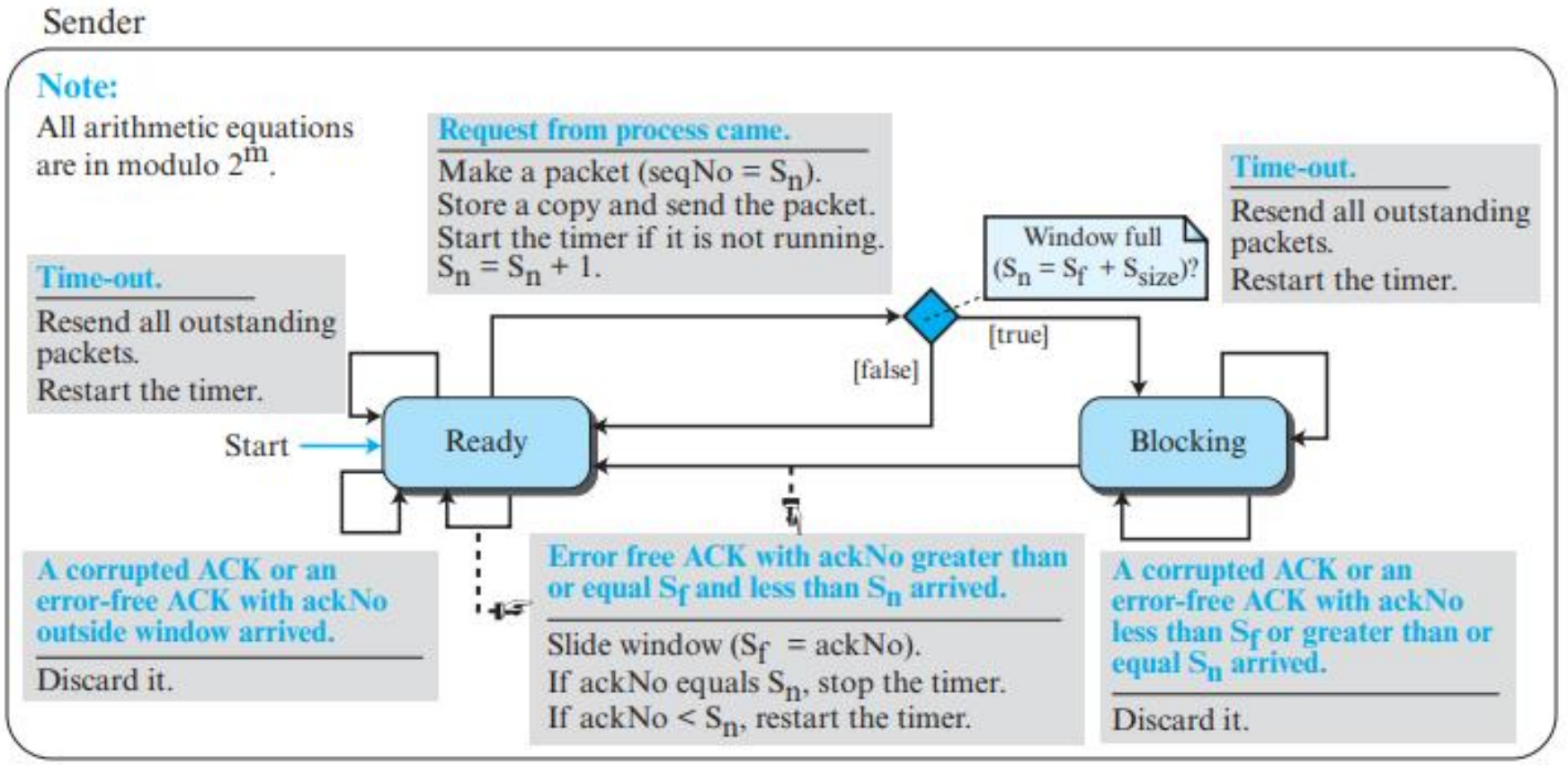
Timers -

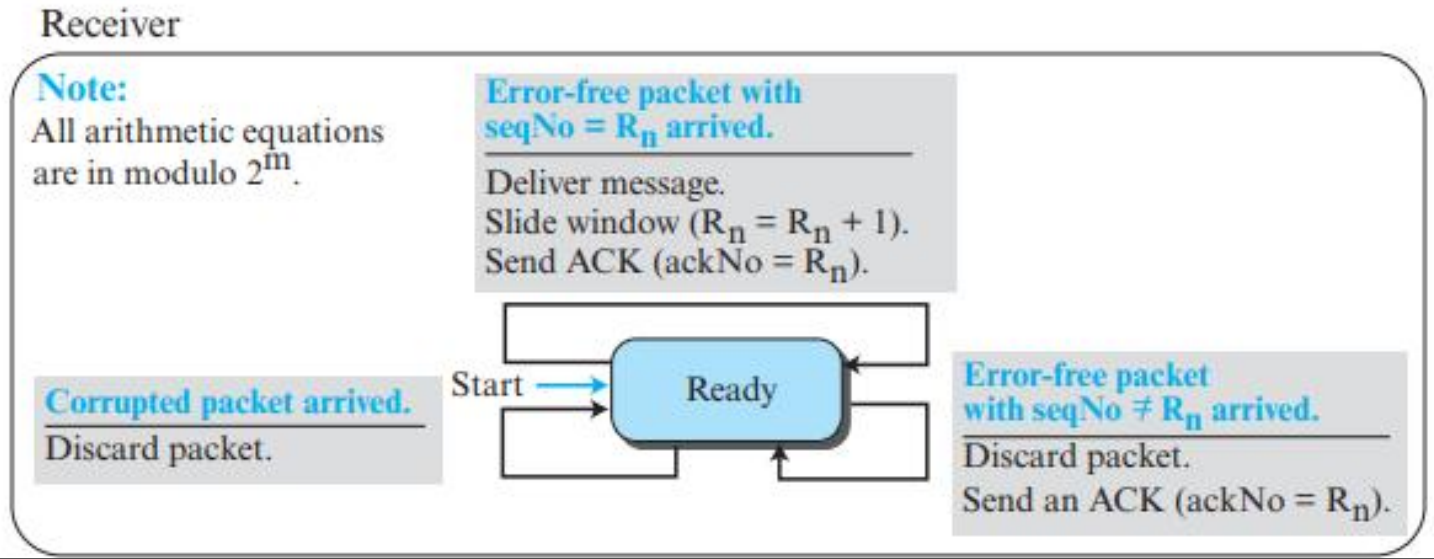
Although there can be a timer for each packet that is sent, in our protocol we use only one. The reason is that the timer for the first outstanding packet always expires first. We resend all outstanding packets when this timer expires.

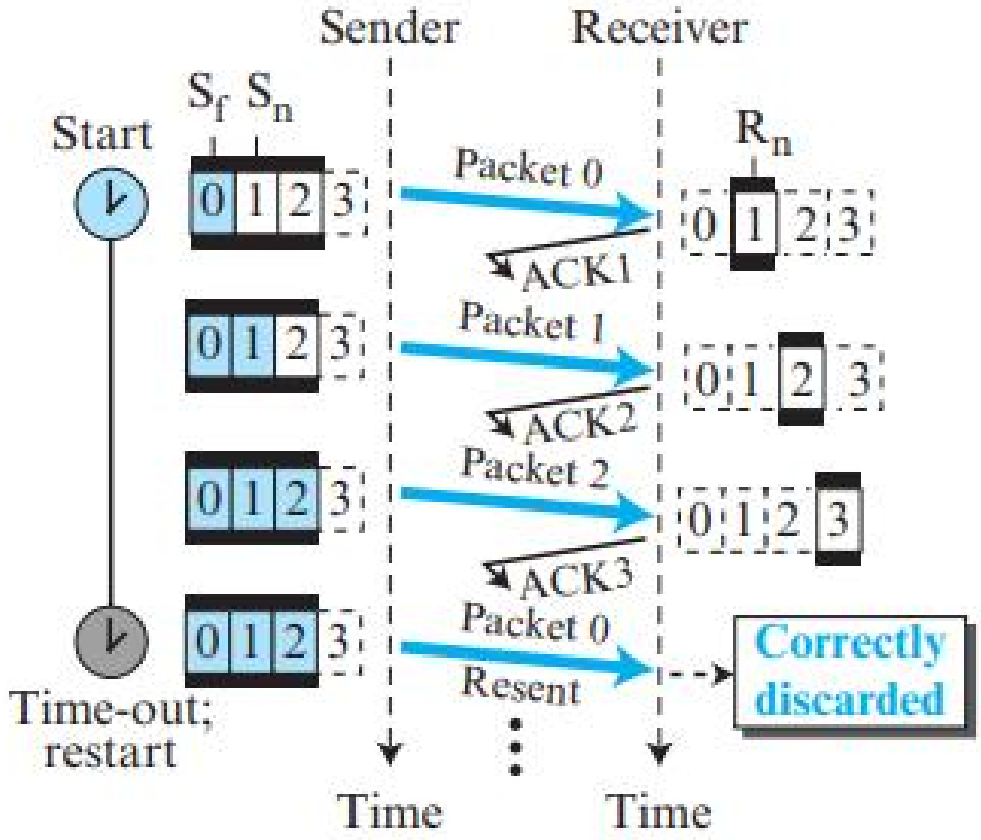
For example, suppose the sender has already sent packet 6 ($S_n = 7$), but the only timer expires. If $S_f = 3$, this means that packets 3, 4, 5, and 6 have not been acknowledged; the sender goes back and resends packets 3, 4, 5, and 6.

That is why the protocol is called Go-Back-N. On a time-out, the machine goes back N locations and resends all packets.

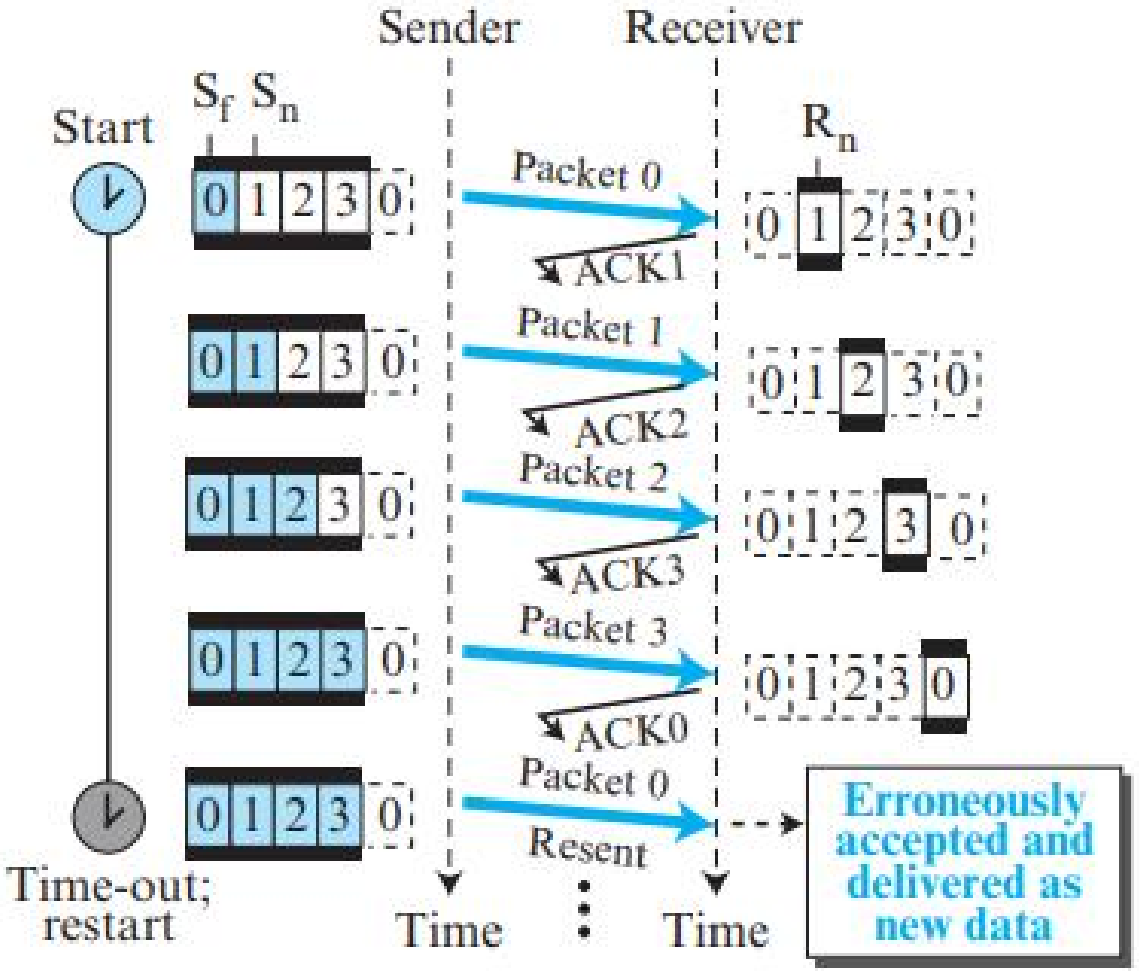
Go-Back-N Protocol: FSMs







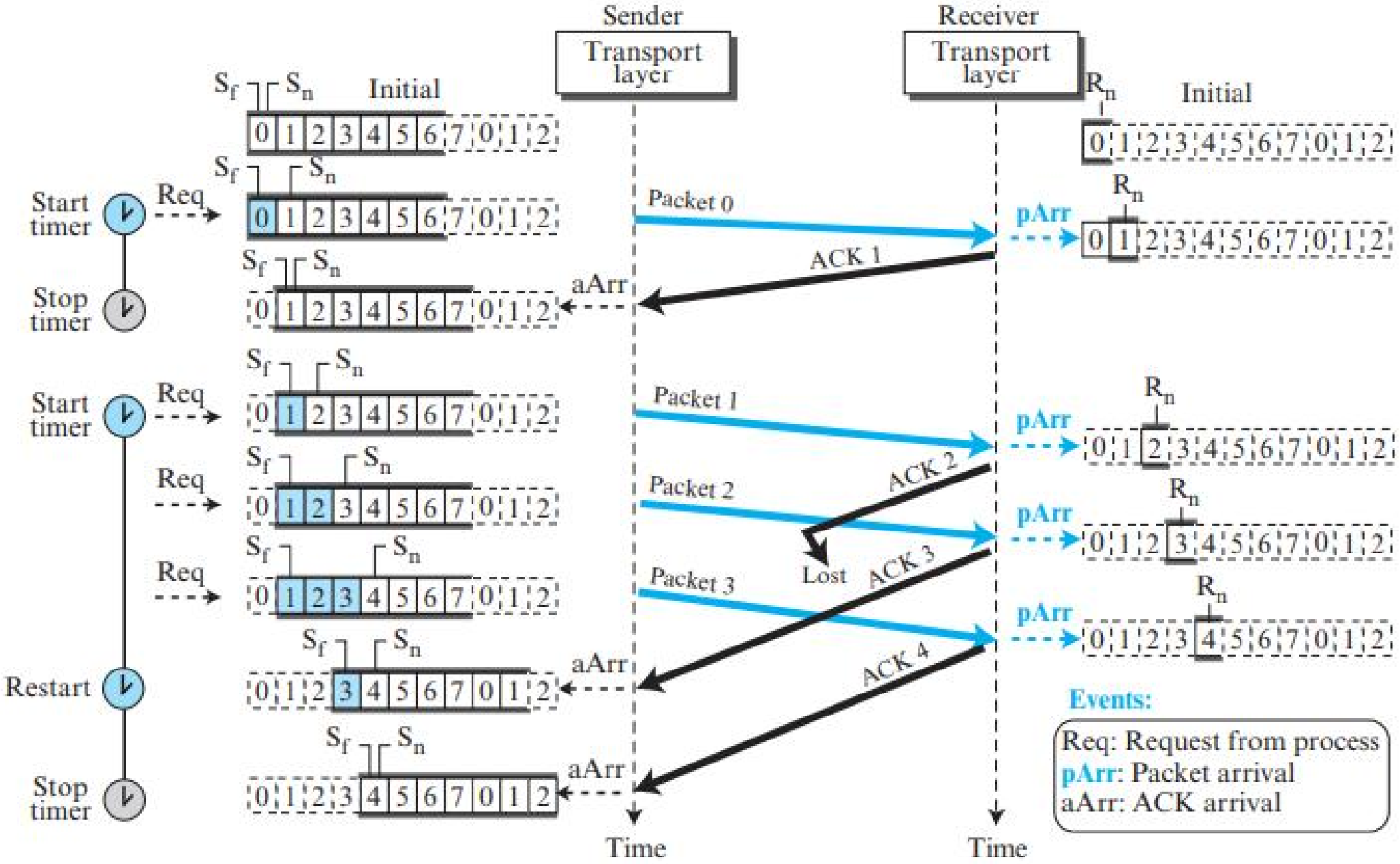
a. Send window of size < 2^m



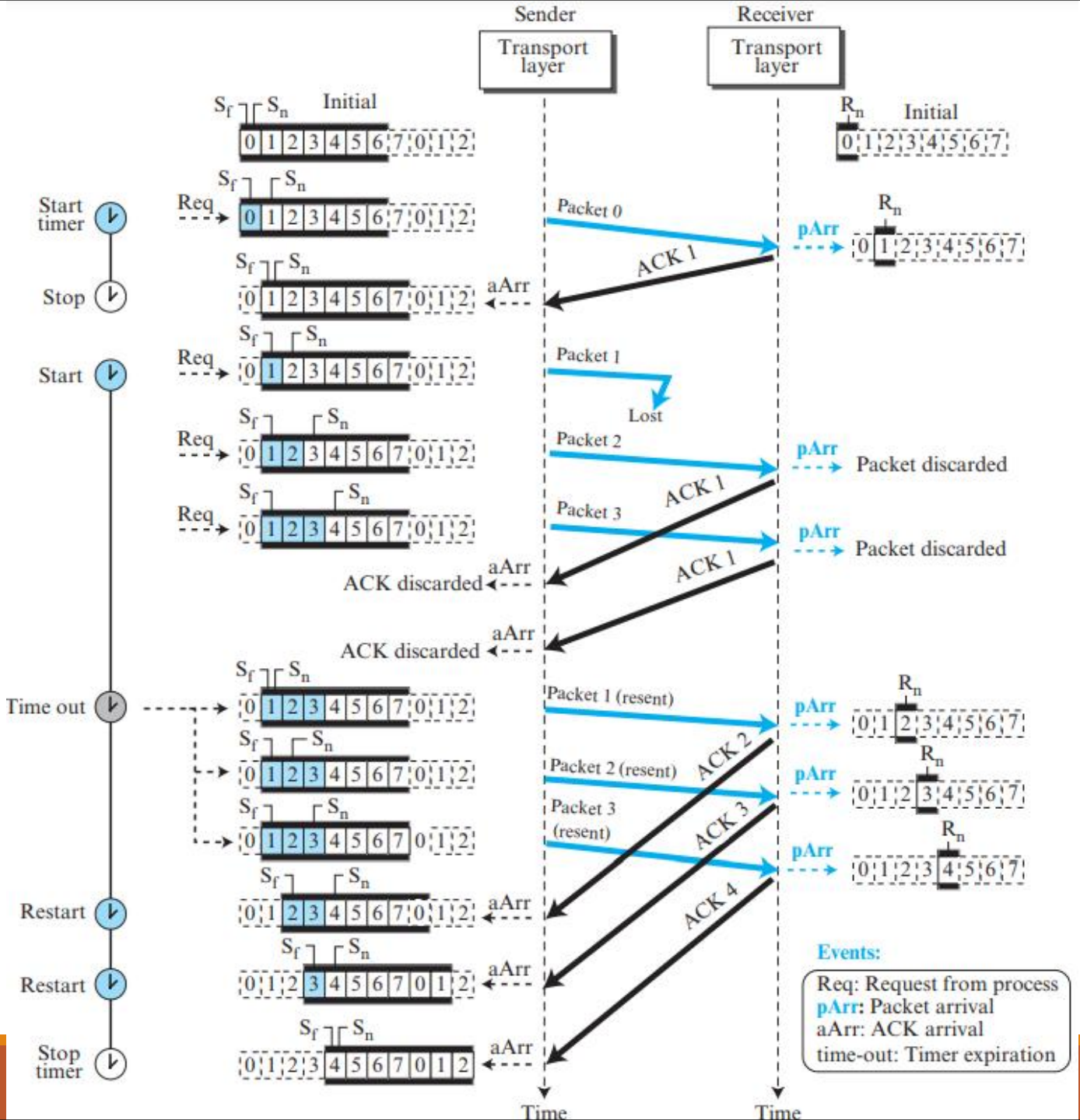
b. Send window of size = 2^m

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

Example


Sender
Transport layer
Receiver
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)
ACK 2
Packet 2 (resent)
ACK 3
Packet 3 (resent)
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.

Q. A 20 Kbps satellite link has a propagation delay of 400 ms. The transmitter employs the "go back n ARQ" scheme with n set to 10.

Assuming that each frame is 100 bytes long, what is the maximum data rate possible?

Given-

Bandwidth = 20 Kbps

Propagation delay (T_p) = 400 ms

Frame size = 100 bytes

Go back N is used where N = 10

Transmission delay (T_t)

= Frame size / Bandwidth

= 100 bytes / 20 Kbps

= (100 x 8 bits) / (20 x 10^3 bits per sec)

= 0.04 sec

= 40 msec

Calculating Value Of 'a'-

a = Tp / Tt

a = 400 msec / 40 msec

a = 10

Calculating Efficiency-

Efficiency (η)

= N / (1+2a)

= 10 / (1 + 2 x 10)

= 10 / 21

= 0.476

= 47.6 %

Calculating Maximum Data Rate Possible-

Maximum data rate possible

or Throughput

= Efficiency x Bandwidth

= 0.476 x 20 Kbps

= 9.52 Kbps

≅ 10 Kbps

Q. Station A needs to send a message consisting of 9 packets to station B using a sliding window (window size 3) and go back n error control strategy. All packets are ready and immediately available for transmission. If every 5th packet that A transmits gets lost (but no ACKs from B ever get lost), then what is the number of packets that A will transmit for sending the message to B?

Given-

Total number of packets to be sent = 9

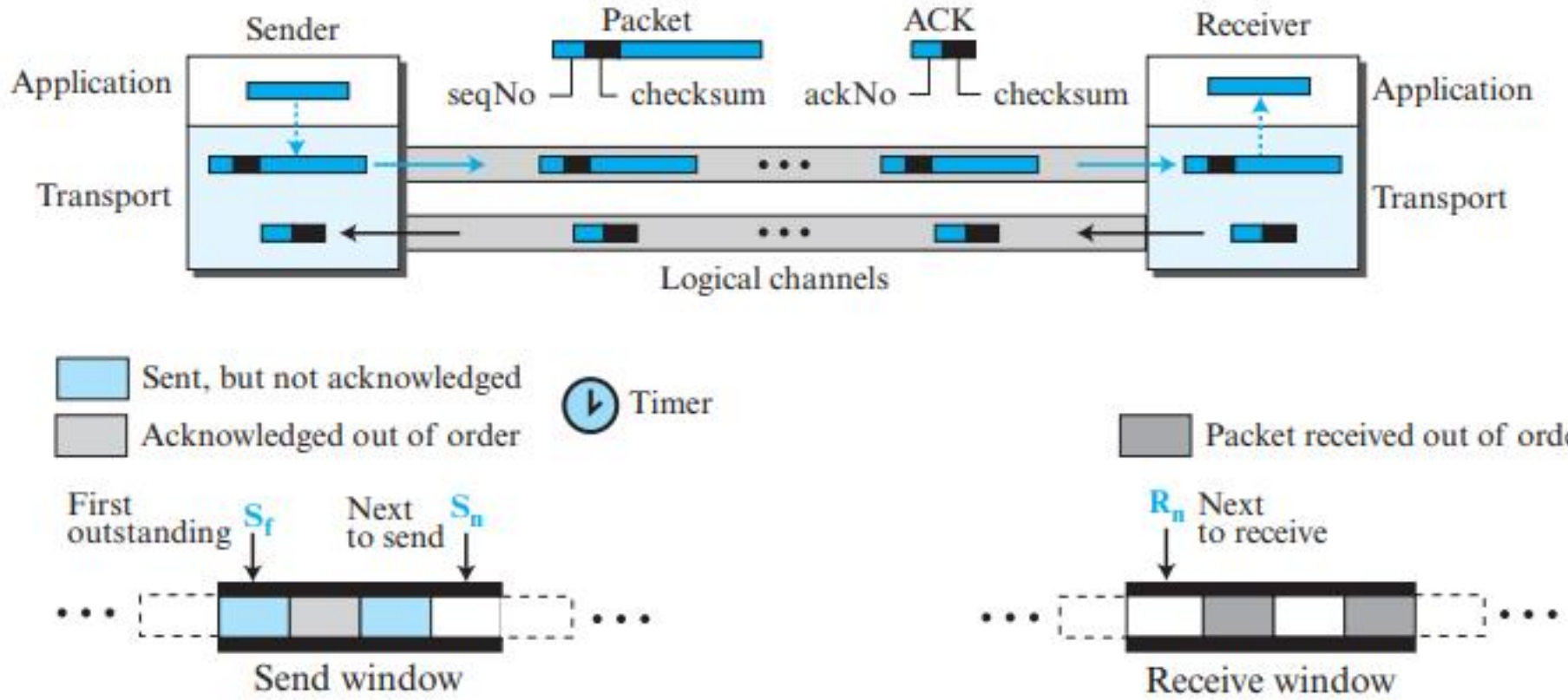
Go back N is used where N = 3

Every 5th packet gets lost

NOTE: For more practice problems on Go-Back-N, visit the link: https://www.gatevidyalay.com/go-back-n-practice-problems/#google_vignette

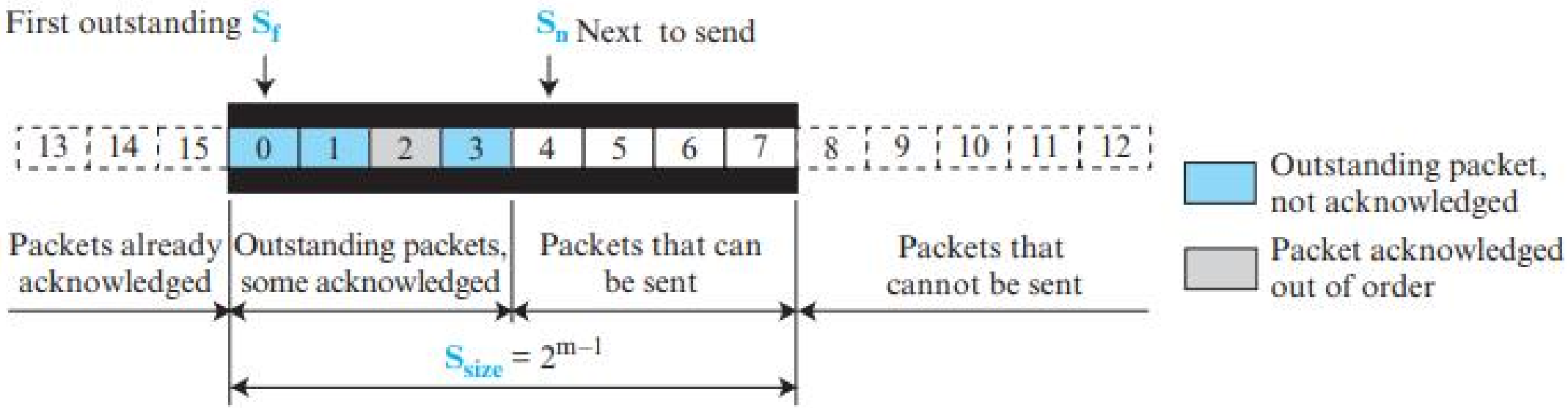
Transport Layer Protocols: Selective-Repeat Protocol

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



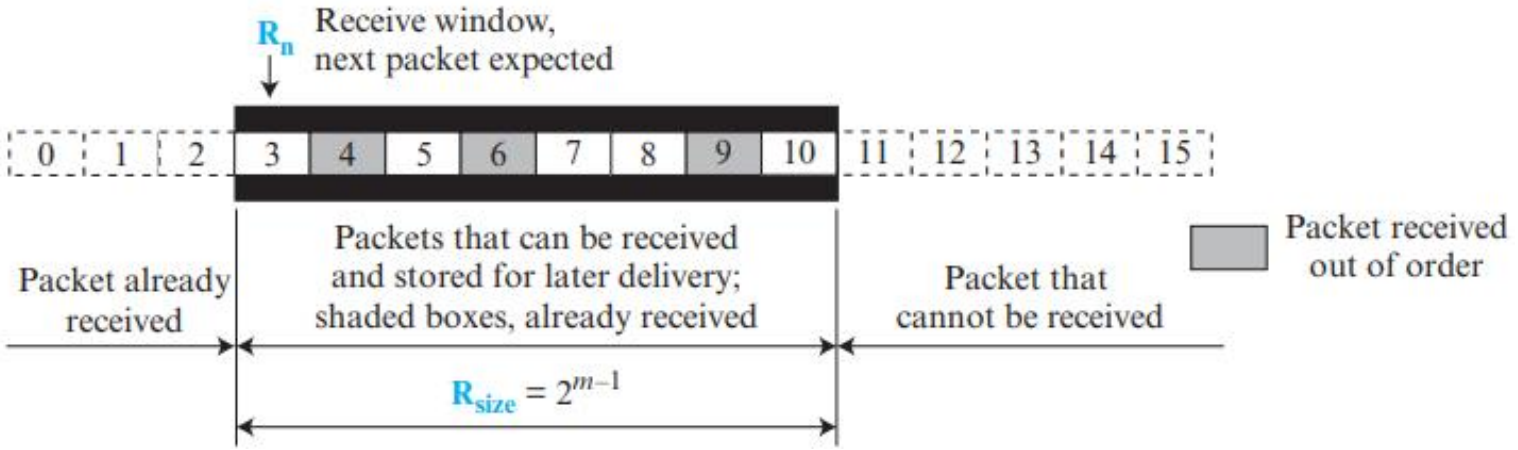
Selective Repeat

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

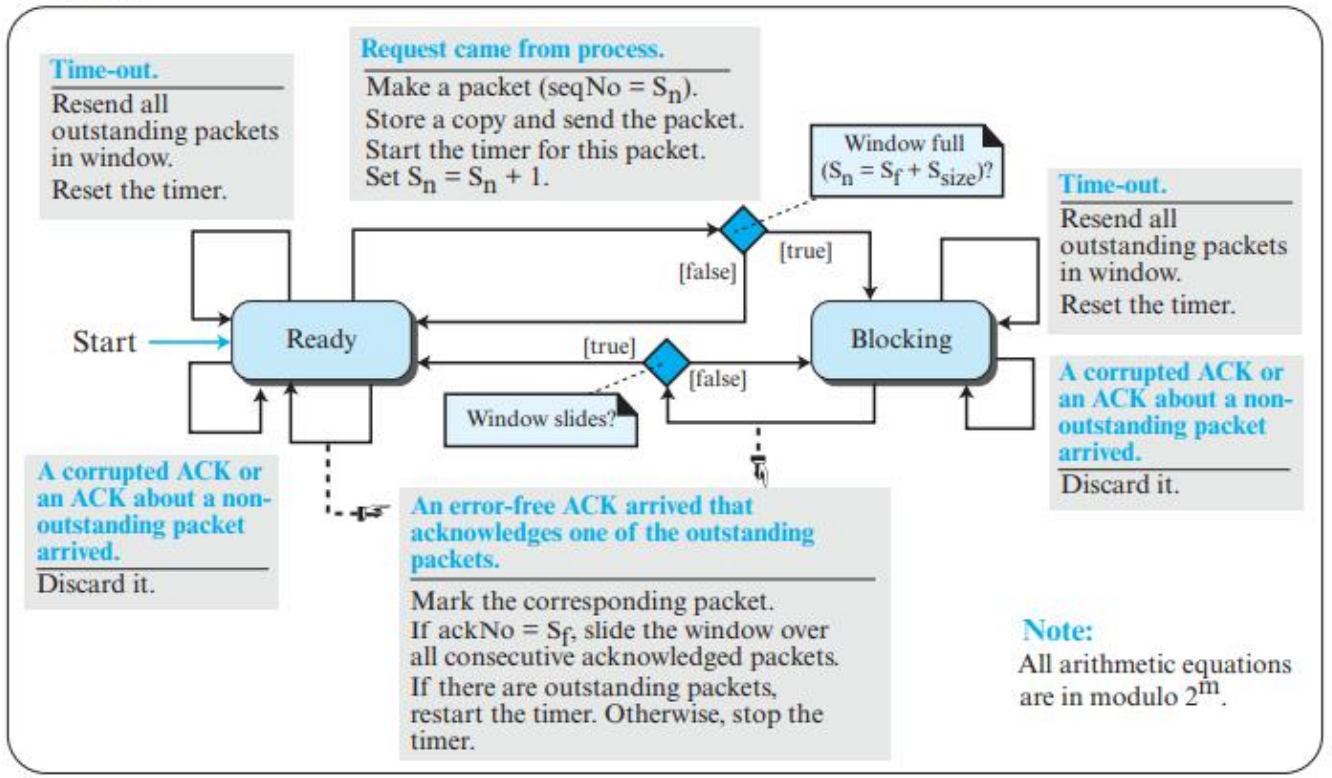
Assume a sender sends 6 packets: packets 0, 1, 2, 3, 4, and 5. The sender receives an ACK with ackNo = 3. What is the interpretation if the system is using GBN or SR?

Solution

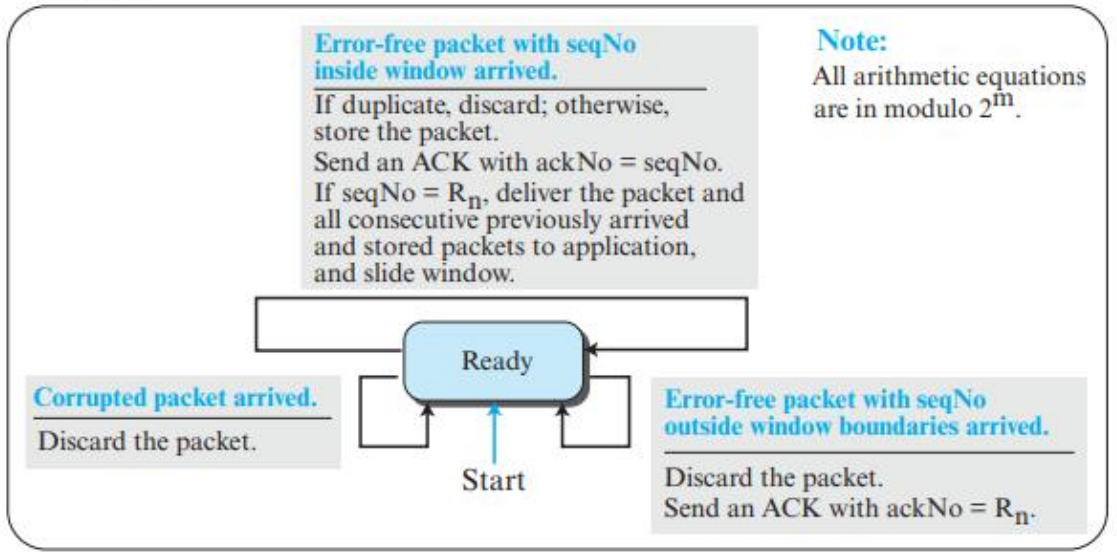
If the system is using GBN, it means that packets 0, 1, and 2 have been received uncorrupted and the receiver is expecting packet 3. If the system is using SR, it means that packet 3 has been received uncorrupted; the ACK does not say anything about other packets.

Selective-Repeat: FSMs

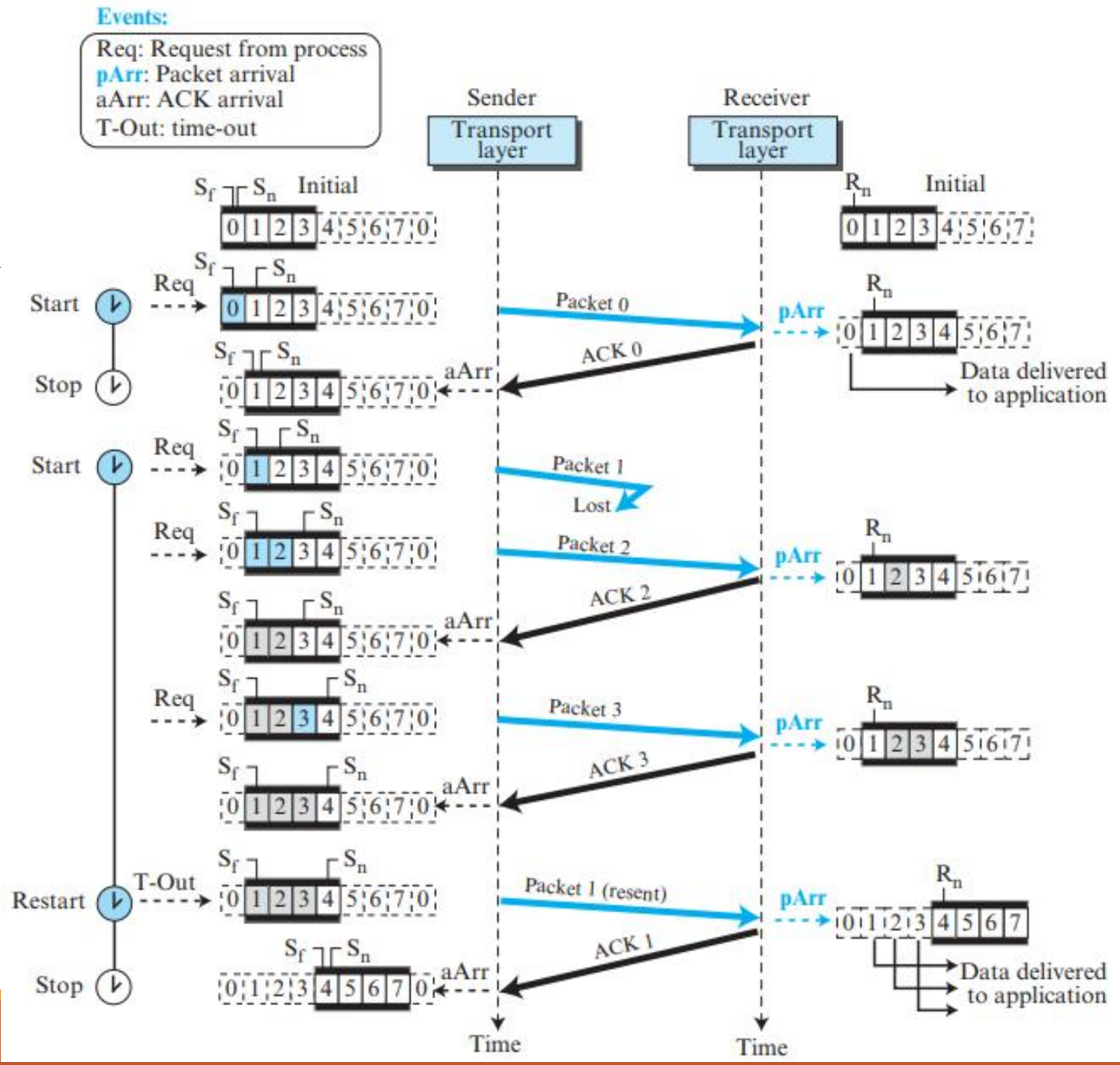
Sender



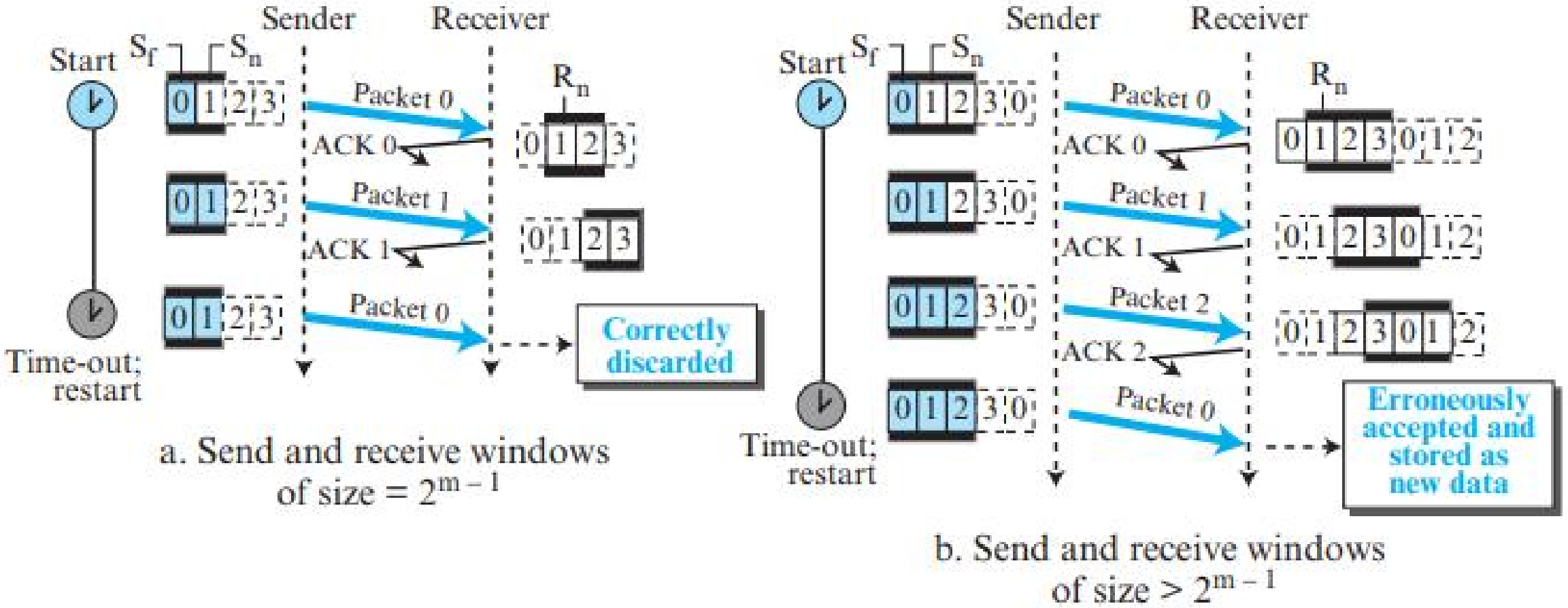
Receiver



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.

QUESTION

Host A wants to send 10 frames to Host B. The hosts agreed to go with SR ARQ. How many number of frames are transmitted by Host A if every 6th frame that is transmitted by host A is either corrupted or lost? Also compare the number of transmissions of SR ARQ with Go-Back-4 ARQ.

ANSWER

Go-Back-N: Number of frames transmitted by Host A: 17

Selective Repeat: Number of frames transmitted by Host A: 11

In SR protocol, suppose frames through 0 to 4 have been transmitted. Now, imagine that 0 times out, 5 (a new frame) is transmitted, 1 times out, 2 times out and 6 (another new frame) is transmitted.

At this point, what will be the outstanding packets in sender's window?

Step-01:

Frames through 0 to 4 have been transmitted-

4 , 3 , 2 , 1 , 0

Step-02:

0 times out. So, sender retransmits it-

0 , 4 , 3 , 2 , 1

Step-03:

5 (a new frame) is transmitted-

5 , 0 , 4 , 3 , 2 , 1

Step-04:

1times out. So, sender retransmits it-

1 , 5 , 0 , 4 , 3 , 2

2times out. So, sender retransmits it-

2 , 1 , 5 , 0 , 4 , 3

Step-06:

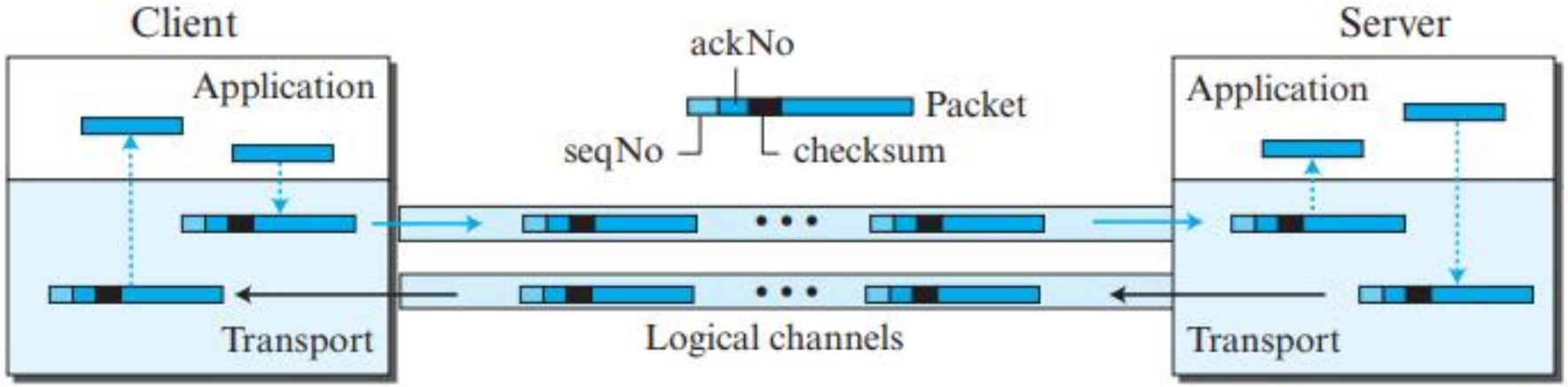
6 (another new frame) is transmitted-

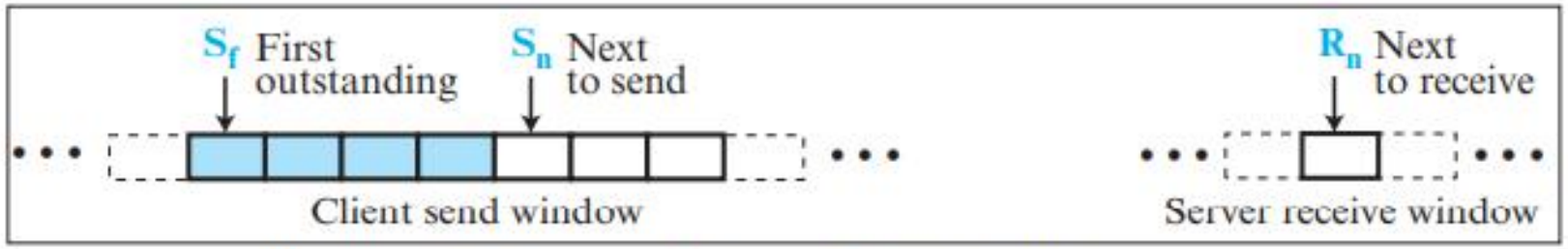
6 , 2 , 1 , 5 , 0 , 4 , 3

Bidirectional Protocols: Piggybacking

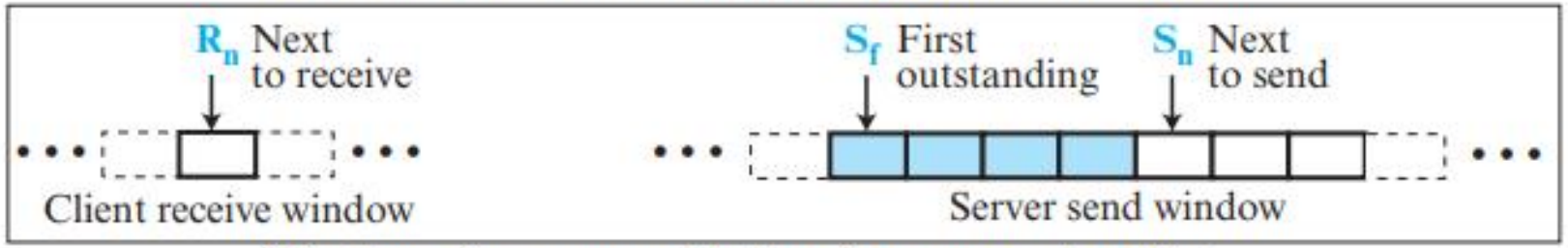
The protocols we have discussed so far are all unidirectional or simplex protocols in which data packets flow in only one direction and acknowledgements travel in the other direction.
In real life, data packets are normally flowing in both directions: from client to server and from server to client. 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

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.

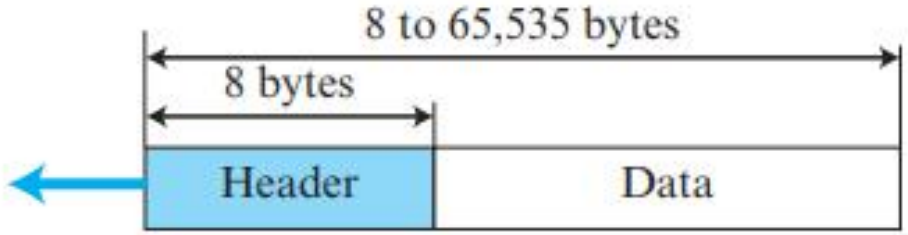
No connection establishment - TCP uses a three-way handshake before it starts to transfer data. UDP just blasts away without any formal preliminaries. Thus UDP does not introduce any delay to establish a connection. This is probably the principal reason why DNS runs over UDP rather than TCP—DNS would 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.

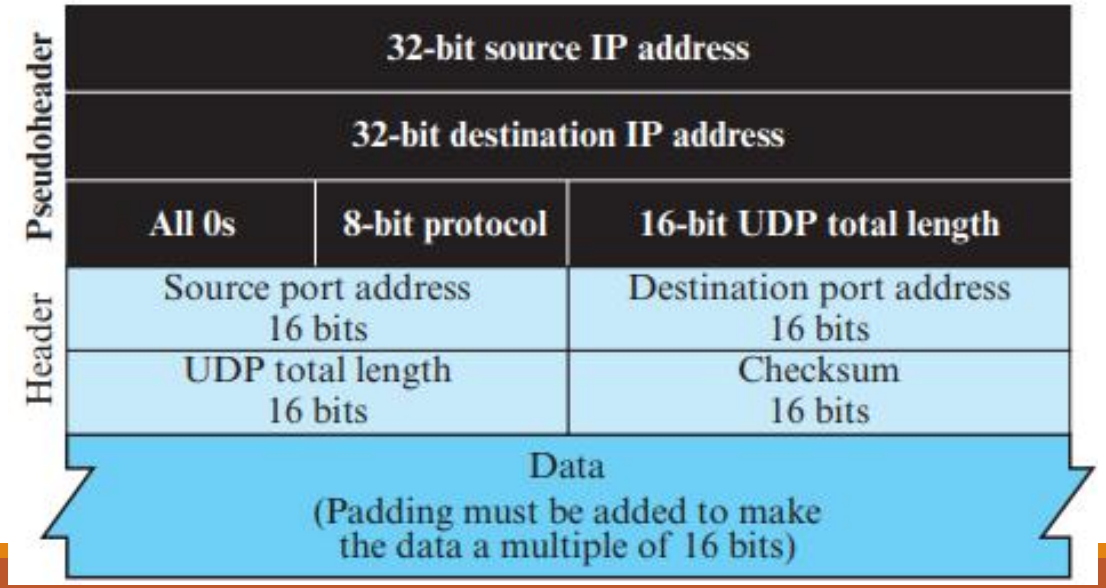
The checksum calculation involves the UDP header, the data (payload), and a pseudo-header derived from the IP header.

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

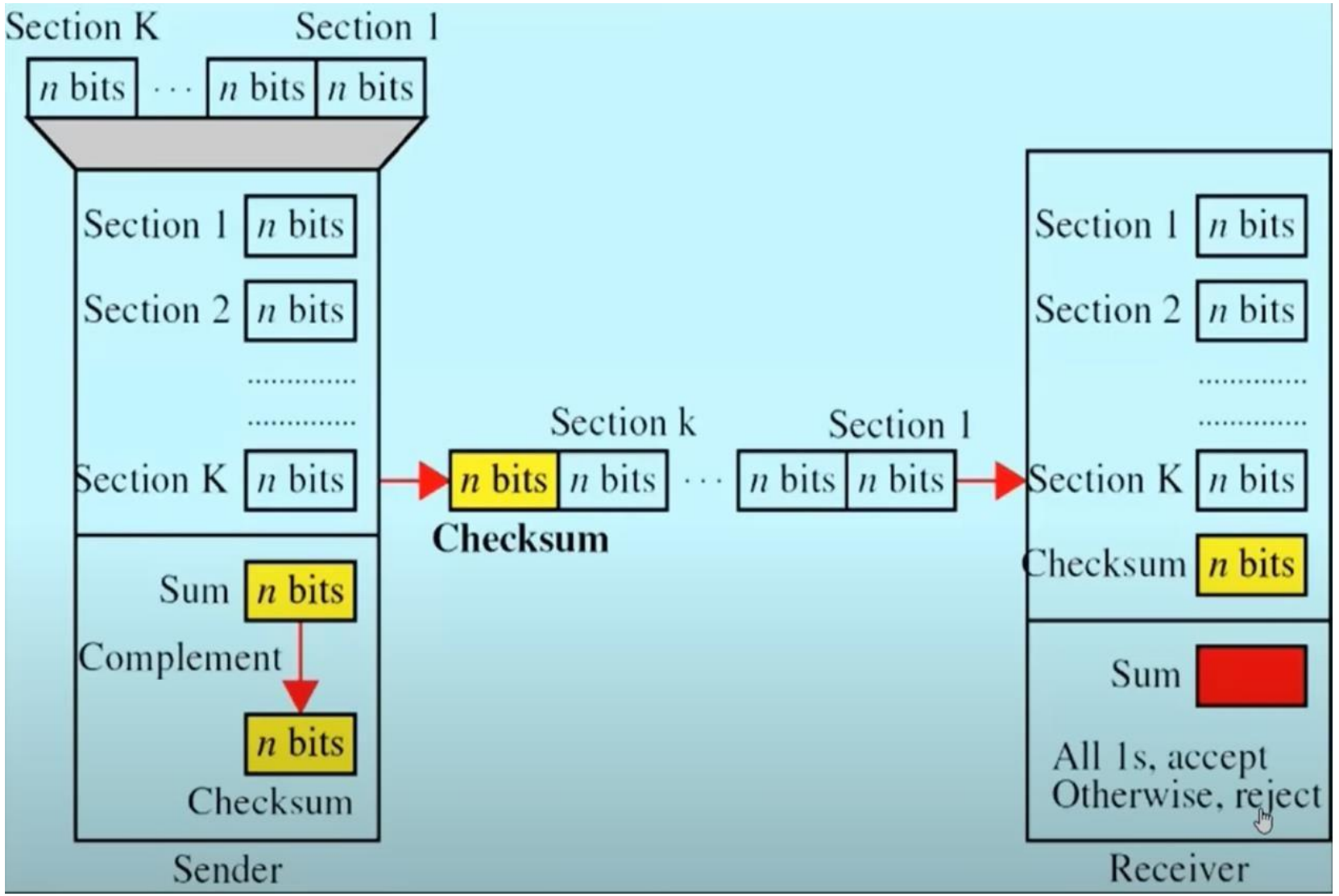
When the receiver gets the UDP packet, it performs the same checksum calculation over the received UDP header, payload, and pseudo-header.
If the resulting checksum is all ones (0xFFFF), it means the packet is valid (i.e., no errors were detected).
If the result is anything other than 0xFFFF, it indicates 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 – Operation at sender side

1. Break the original message in to 'k' number of blocks with 'n' bits in each block.
2. Sum all the 'k' data blocks.
3. Add the carry to the sum, if any.
4. Do 1's complement to the sum = Checksum.


Section K
Section 1
n bits
n bits
n bits
Section 1
n bits
Section 2
n bits
Section K
n bits
Sum
n bits
Complement
n bits
Checksum
Sender
Section k
Section 1
n bits
n bits
n bits
n bits
Checksum
Section 1
n bits
Section 2
n bits
Section K
n bits
Checksum
n bits
Sum
All 1s, accept
Otherwise, reject
Receiver

Checksum – Example

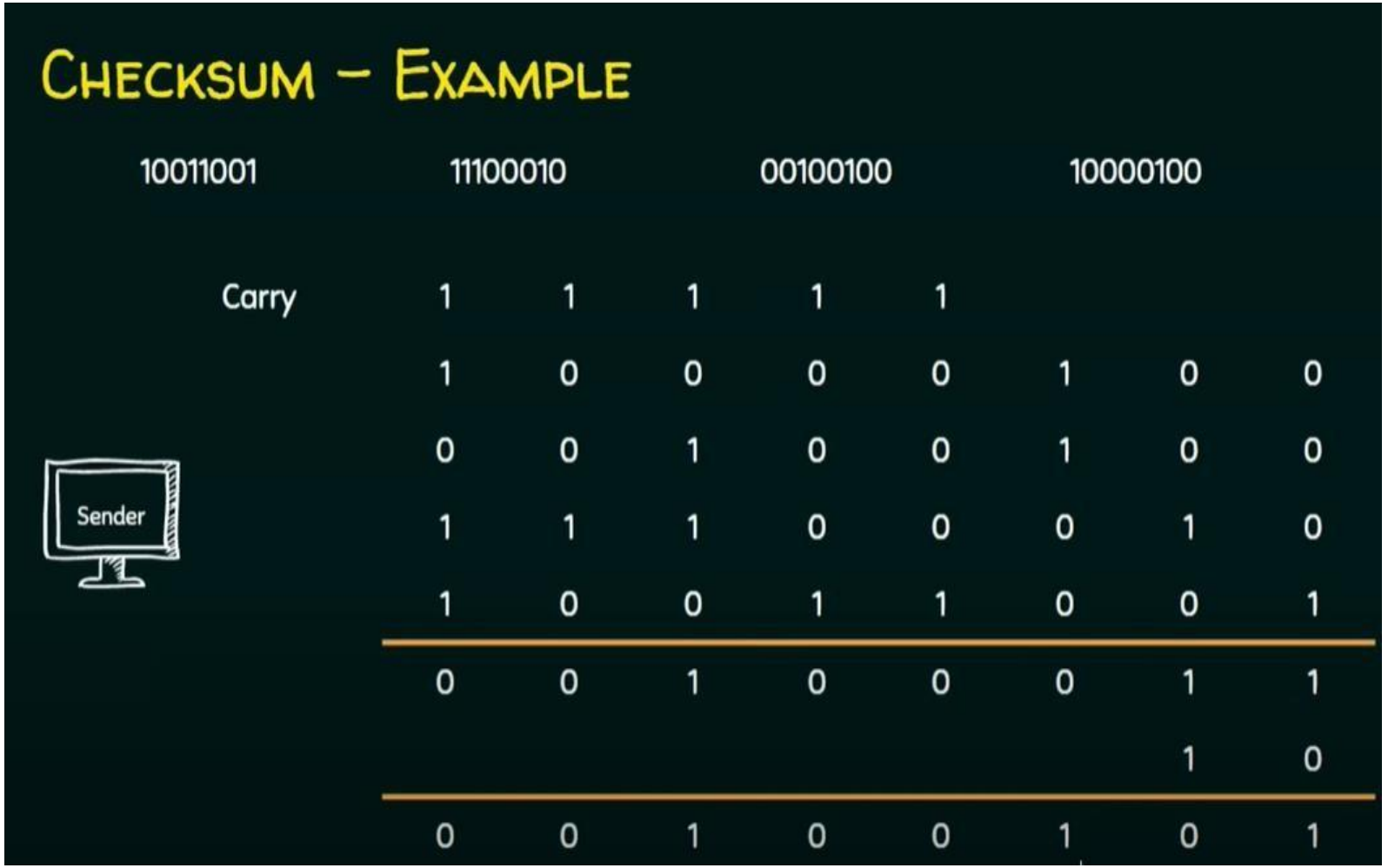
Consider the data unit to be transmitted is:

10011001111000100010010010000100

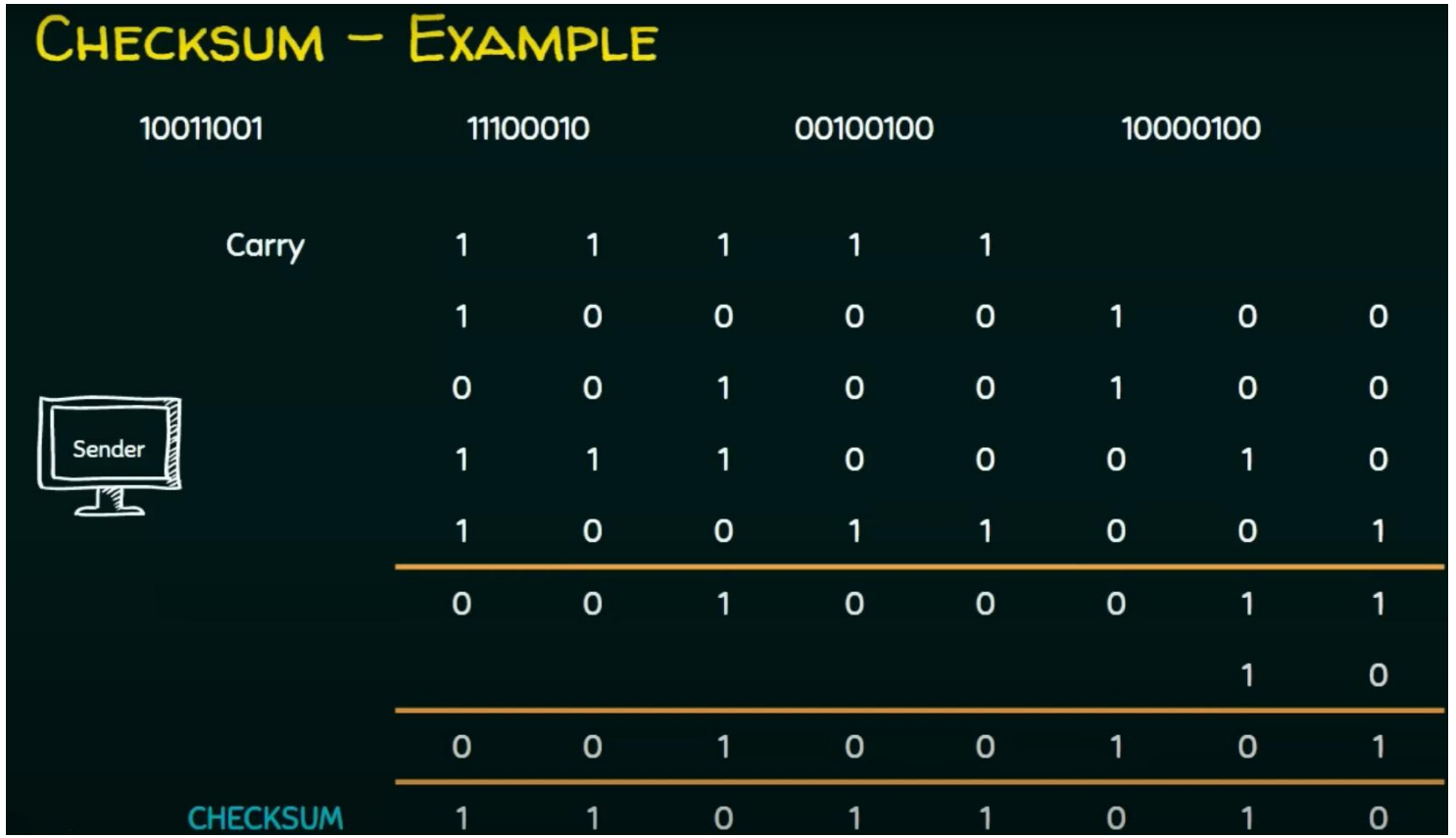
Checksum – Example

10011001		11100010			00100100			10000100	
	Carry	1	1	1	1	1			
		1	0	0	0	0	1	0	0
		0	0	1	0	0	1	0	0
		1	1	1	0	0	0	1	0
		1	0	0	1	1	0	0	1
1	0	0	0	1	0	0	0	1	1

Now, the carry needs to be added in the sum



Now, perform the 1's compliment of the sum to get the Checksum.



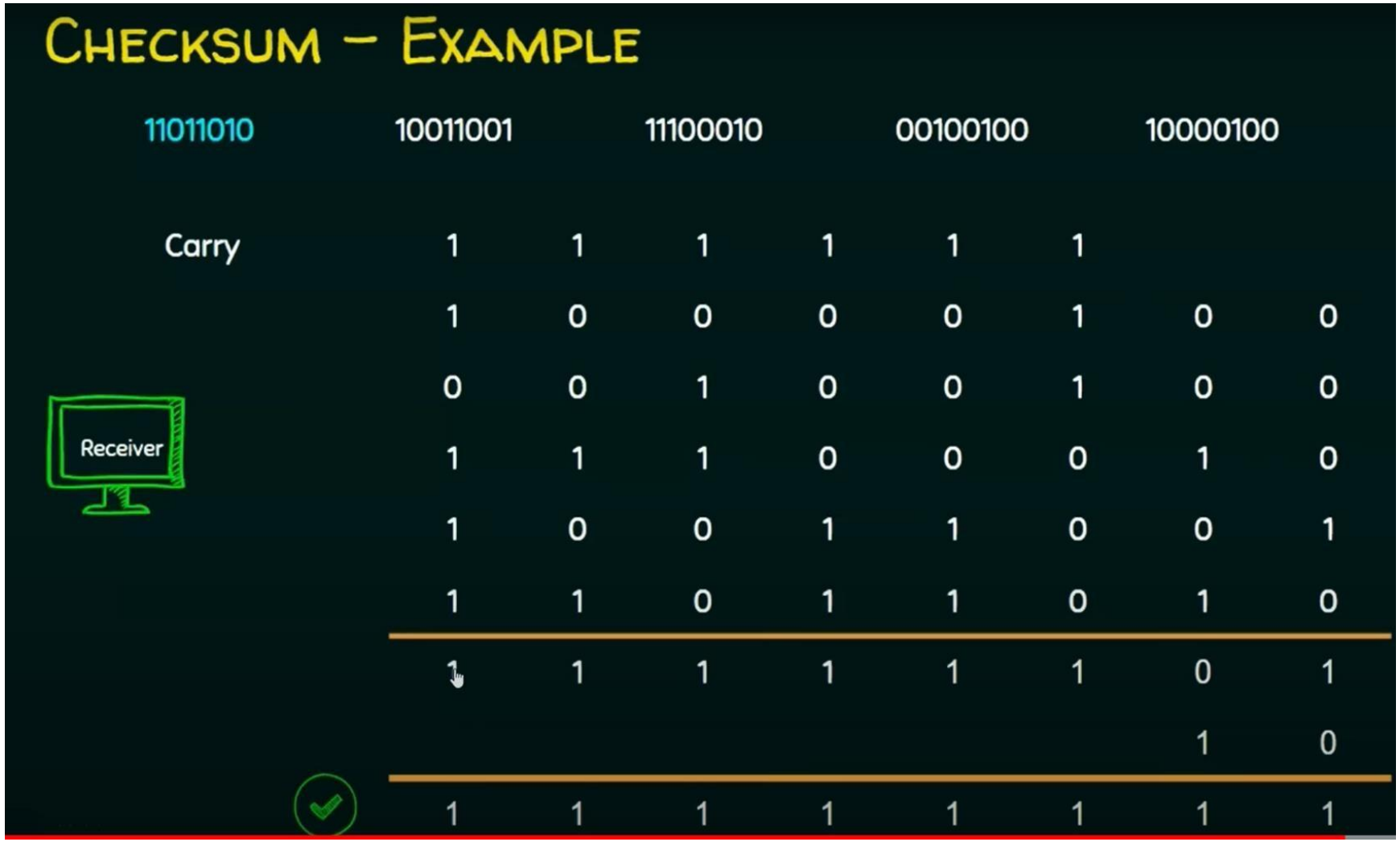
Now, sender appends the checksum with data blocks and send it to receiver

Checksum – Operation at Receiver Side

- ★ Collect all the data blocks including the checksum.
- ★ Sum all the data blocks and checksum
- ★ If the result is all 1's, ACCEPT; Else, REJECT.

Checksum – Example

	11011010	10011001	11100010	00100100	10000100
Carry		1 1 1 1 1 1			
		1 0 0 0 0 1 0 0			
		0 0 1 0 0 1 0 0			
		1 1 1 0 0 0 1 0			
		1 0 0 1 1 0 0 1			
		1 1 0 1 1 0 1 0			
	1 0	1 1 1 1 1 1 0 1			



As the result is all 1, so, there is no error and the receiver accepts the packet.

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.

UDP Operation:

Connectionless Services:

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

There is no relationship between the different user datagrams even if they are coming from the same source process and going to the same destination program.

The user datagrams are not numbered. Also, there is no connection establishment and no connection termination, as is the case for TCP. This means that each user datagram can travel on a different path.

Flow and Error Control:

UDP is a very simple, unreliable transport protocol. There is no flow control and hence no window mechanism. The receiver may overflow with incoming messages.

There is no error control mechanism in UDP except for the checksum.

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

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

Encapsulation and Decapsulation:

To send a message from one process to another, the UDP protocol encapsulates and decapsulates messages in an IP datagram.

Process-to-Process Communication

UDP provides process-to-process communication using socket addresses, a combination of IP addresses and port numbers.

Congestion Control

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

Queuing

We have talked about ports without discussing the actual implementation of them. In UDP, queues are associated with ports.

At the client site, when a process starts, it requests a port number from the operating system. Some implementations create both an incoming and an outgoing queue associated with each process. Other implementations create only an incoming queue associated with each process.

Applications

Connectionless Service

UDP is a connectionless protocol, meaning each packet it sends is independent. This can be an advantage or disadvantage depending on the application. It's advantageous when you need to quickly send short requests and responses because it avoids the overhead of setting up and closing connections. For instance, in connection-oriented services, it takes at least 9 packets to achieve a similar task, while in connectionless services, only 2 packets are exchanged. Connectionless services are faster with less delay, making them preferable when minimizing delay is crucial for an application.

Lack of error control

UDP (User Datagram Protocol) doesn't guarantee error control, making it unreliable. Many applications prefer reliability in communication. However, reliable services can introduce delays when lost or corrupted data needs to be resent. Some applications can tolerate these delays, but for others, especially time-sensitive ones, such variations in delay can be problematic.

Lack of Congestion Control

UDP does not provide congestion control. However, UDP does not create additional
traffic in an error-prone network. TCP may resend a packet several times and thus contribute to the creation of congestion or worsen a congested situation. Therefore, in
some cases, lack of error control in UDP can be considered an advantage when congestion is a big issue.

Use of UDP:

The following lists some uses of the UDP protocol:

UDP is suitable for a process that requires simple request-response communication with little concern for flow and error control.

UDP is suitable for a process with internal flow and error control mechanisms. For example, the Trivial File Transfer Protocol (TFTP) process includes flow and error control. It can easily use UDP.

UDP is a suitable transport protocol for multicasting. Multicasting capability is embedded in the UDP software but not in the TCP software.

UDP is used for management processes such as SNMP.

UDP is used for some route updating protocols such as Routing Information Protocol.

TRANSMISSION CONTROL PROTOCOL (TCP)

TCP, like UDP, is a **process-to-process** (program-to-program) protocol. TCP, therefore, like UDP, uses **port numbers**.

Unlike UDP, TCP is a connection-oriented protocol; it creates a **virtual connection** between two TCPs to send data.

In addition, TCP uses flow and error control mechanisms at the transport level.

In brief, TCP is called a ***connection-oriented, reliable transport protocol***.

It adds connection-oriented and reliability features to the services of IP.

TCP Services:

Process-to-Process Communication:

Like UDP, TCP provides process-to-process communication using port numbers. Table below lists some well-known port numbers used by TCP.

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

Stream Delivery Service:

TCP, on the other hand, allows the sending process to deliver data as a stream of bytes and allows the receiving process to obtain data as a stream of bytes.

TCP creates an environment in which the two processes seem to be connected by an imaginary "tube“ that carries their data across the Internet.

The sending process produces (writes to) the stream of bytes, and the receiving process consumes (reads from) them.

Sending and Receiving Buffers: Because the sending and the receiving processes may not write or read data at the same speed, TCP needs buffers for storage.

There are two buffers, the **sending buffer,** and the **receiving buffer**, one for each direction.

Segments: Although buffering handles the disparity between the speed of the producing and consuming processes, we need one more step before we can send data.

The IP 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 and delivers the segment to the IP layer for transmission.

The segments are encapsulated in IP datagrams and transmitted.

Full-Duplex Communication:

TCP offers full-duplex service, in which data can flow in both directions at the same time.

Each TCP then has a sending and receiving buffer, and segments move in both directions.

Connection-Oriented Service:

TCP, unlike UDP, is a connection-oriented protocol.

When a process at site A wants to send and receive data from another process at site B, the following occurs:

1. The two TCPs establish a connection between them.
2. Data are exchanged in both directions.
3. The connection is terminated.

Reliable Service:

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

TCP Features:

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 the **byte number** and not the segment number.

Byte Number: TCP numbers all data bytes that are transmitted in a connection.

Numbering is independent in each direction.

When TCP receives bytes of data from a process, it stores them in the sending buffer and numbers them.

Sequence Number: After the bytes have been numbered, TCP assigns a sequence number to each segment that is being sent.

The sequence number for each segment is the number of the first byte carried in that segment.

When a segment carries a combination of data and control information (control information means acknowledgement) (**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.

Piggybacking: In reliable full - duplex data transmission, the technique of hooking up acknowledgments onto outgoing data frames is called piggybacking.

With piggybacking, a substantial gain is obtained in reducing bandwidth requirement.

Acknowledgment Number: The value of the acknowledgment field in a segment defines the number of the next byte a party expects to receive.

The acknowledgment number is cumulative. The term cumulative here means that if a party uses 5643 as an acknowledgment number, it has received all bytes from the beginning up to 5642.

Flow Control:

TCP, unlike UDP, provides *flow control*. The receiver of the data controls the amount of data that are to be sent by the sender. This is done to prevent the receiver from being overwhelmed with data.

The numbering system allows TCP to use a byte-oriented flow control.

Error Control:

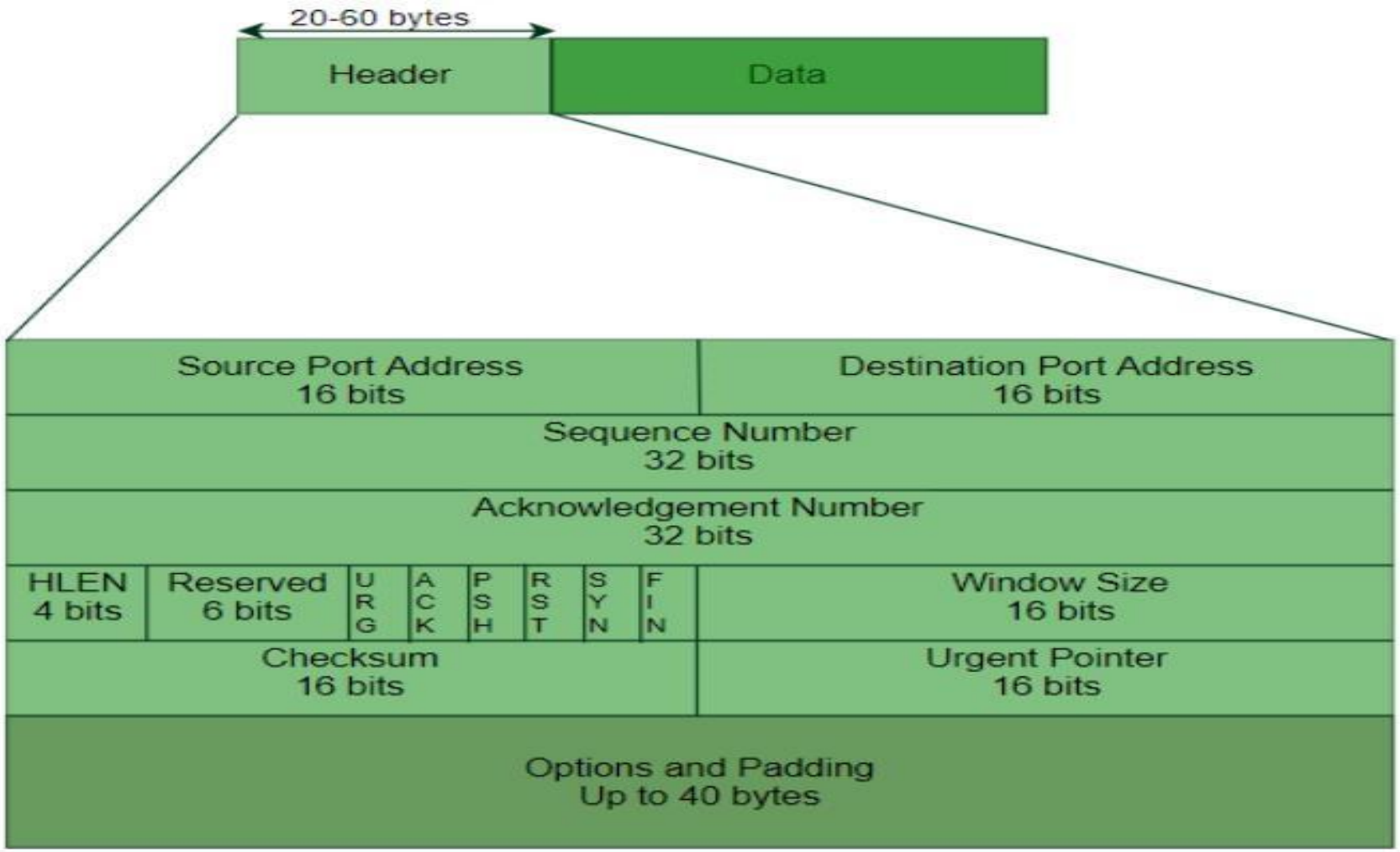
To provide reliable service, TCP implements an error control mechanism. Although error control considers a segment as the unit of data for error detection (loss or corrupted segments), error control is byte-oriented.

Congestion Control:

TCP, unlike UDP, takes into account congestion in the network.

The amount of data sent by a sender is not only controlled by the receiver (flow control), but is also determined by the level of congestion in the network.

TCP Segment format:



The segment consists of a **20- to 60-byte header**, followed by data from the application program.

The header is 20 bytes if there are no options and up to 60 bytes if it contains options.

Source port address: This is a 16-bit field that defines the port number of the application program in the host that is sending the segment.

Destination port address: This is a 16-bit field that defines the port number of the application program in the host that is receiving the segment.

Sequence number: This 32-bit field defines the number assigned to the first byte of data contained in this segment.

As we said before, TCP is a stream transport protocol.

To ensure connectivity, each byte to be transmitted is numbered. The sequence number tells the destination which byte in this sequence comprises the first byte in the segment.

Acknowledgment number: This 32-bit field defines the byte number that the receiver of the segment is expecting to receive from the other party.

If the receiver of the segment has successfully received byte number x from the other party, it defines x + 1 as the acknowledgment number.

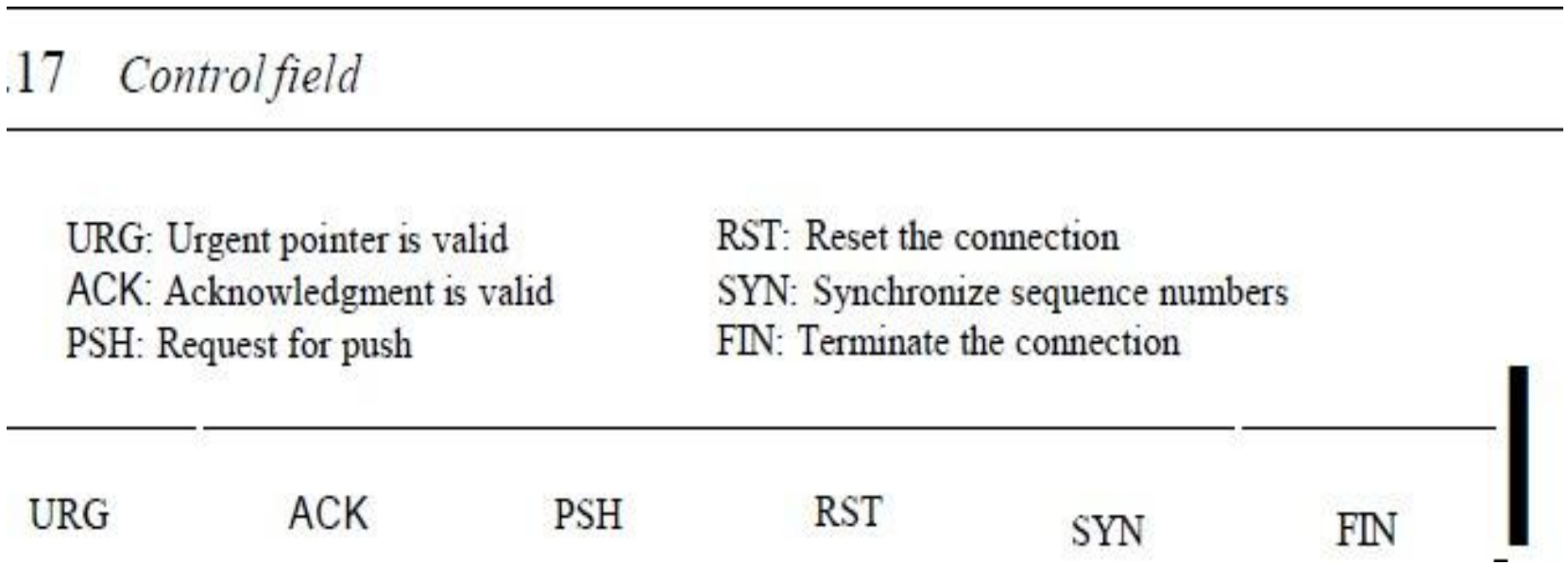
Acknowledgment and data can be **piggybacked** together.

Header length: This 4-bit field indicates the number of 4-byte words in the TCP header.

The length of the header can be between 20 and 60 bytes. Therefore, the value of this field can be between 5 (5 x 4 =20) and 15 (15 x 4 =60).

Reserved: This is a 6-bit field reserved for future use.

Control: This field defines 6 different **control bits or flags** as shown in figure below. One or more of these bits can be set at a time.



These bits enable flow control, connection establishment and termination, connection

abortion, and the mode of data transfer in TCP. A brief description of each bit is shown in the Table in next slide.

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

Description of flags in the control field

Window size: This field defines the size of the window, in bytes, that the other

party must maintain.

Note that the length of this field is 16 bits, which means that the maximum size of the window is 65,535 bytes.

This value is normally referred to as the receiving window (rwnd) and is determined by the receiver.

The sender must obey the dictation of the receiver in this case.

Checksum: This 16-bit field contains the checksum.

The calculation of the checksum for TCP follows the same procedure as the one described for UDP.

However, the inclusion of the checksum in the UDP datagram is optional, whereas the inclusion of the checksum for TCP is mandatory.

The same pseudoheader, serving the same purpose, is added to the segment.

For the TCP pseudoheader, the value for the protocol field is 6.

Urgent pointer: This l6-bit field, which is valid only if the urgent flag is set, and is used when the segment contains urgent data.

It defines the number that must be added to the sequence number to obtain the number of the last urgent byte in the data section of the segment.

Options: There can be up to 40 bytes of optional information in the TCP header.

A TCP Connection:

In TCP, connection-oriented transmission requires three phases: connection establishment, data transfer, and connection termination.

First Phase: *Connection Establishment:*

TCP transmits data in full-duplex mode.

When two TCPs in two machines are connected, they are able to send segments to each other simultaneously.

This implies that each party must initialize communication and get approval from the other party before any data are transferred.

Three-Way Handshaking: The connection establishment in TCP is called three-way handshaking.

In our example, an application program, called the client, wants to make a connection with another application program, called the server, using TCP as the transport layer protocol.

The process starts with the server. The server program tells its TCP that it is ready to accept a connection. This is called a request for a ***passive open***.

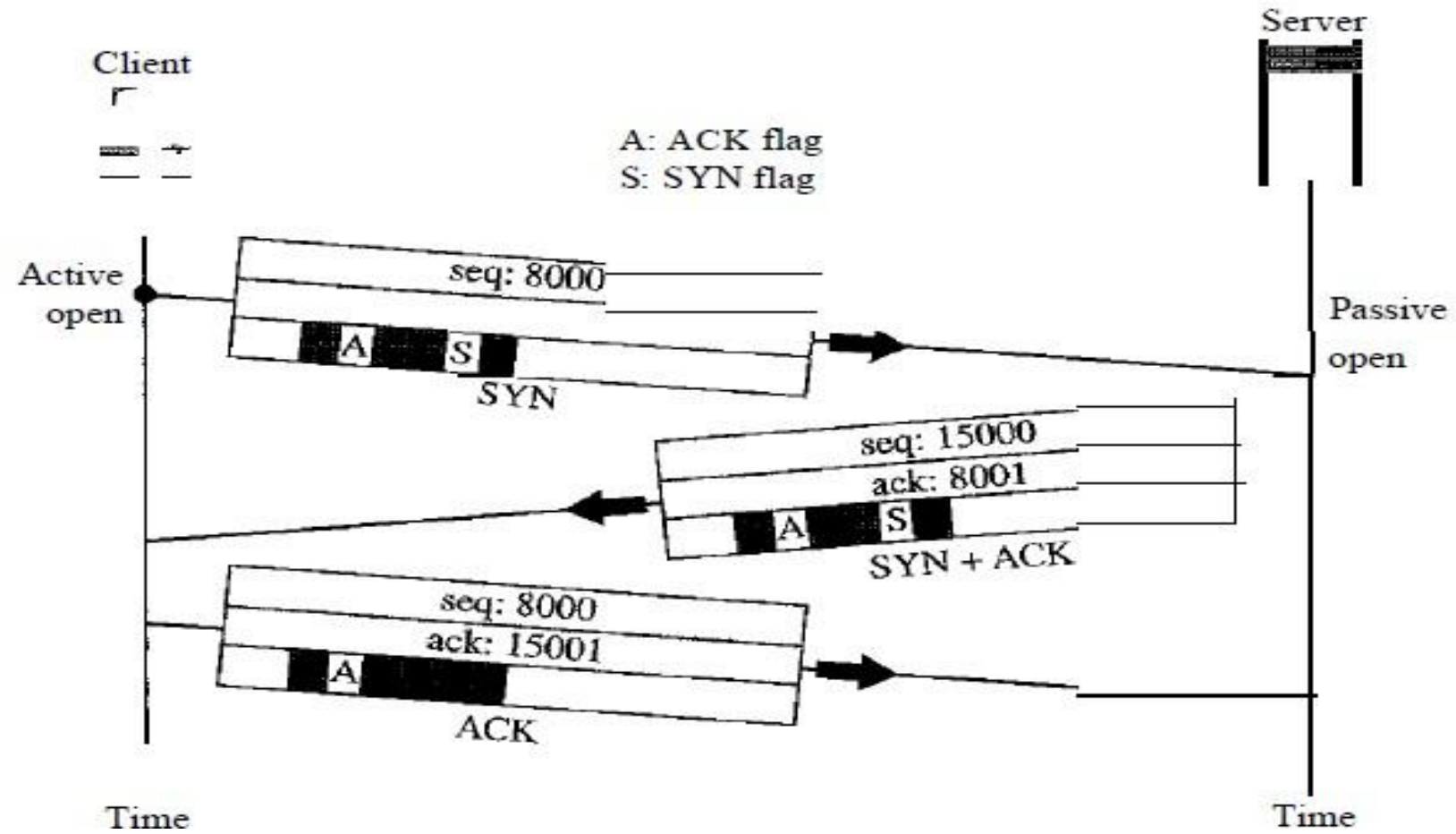
Although the server TCP is ready to accept any connection from any machine in the world, it cannot make the connection itself.

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

A client that wishes to connect to an open server tells its TCP that it needs to be connected to that particular server.

TCP can now start the three-way handshaking process as shown in Figure in the next slide.

.8 *Connection establishment using three-way handshaking*



The three steps in this phase are as follows.

1.The client sends the first segment, a **SYN segment**, in which only the SYN flag is set.

This segment is for synchronization of sequence numbers. It consumes one sequence number.

When the data transfer starts, the sequence number is incremented by 1.

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

2.The server sends the second segment, a SYN + ACK segment, with 2 flag bits set: SYN and ACK.

This segment has a dual purpose. It is a SYN segment for communication in the other direction and serves as the acknowledgment for the SYN segment.

It consumes one sequence number.

3. The client sends the third segment. This is just an ACK segment.

It acknowledges the receipt of the second segment with the ACK flag and acknowledgment number field.

Note that the sequence number in this segment is the same as the one in the SYN segment; the ACK segment does not consume any sequence numbers.

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

Simultaneous Open: A rare situation, called a simultaneous open, may occur when **both processes issue an active open**.

In this case, both TCPs transmit a SYN + ACK segment to each other, and one single connection is established between them.

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.

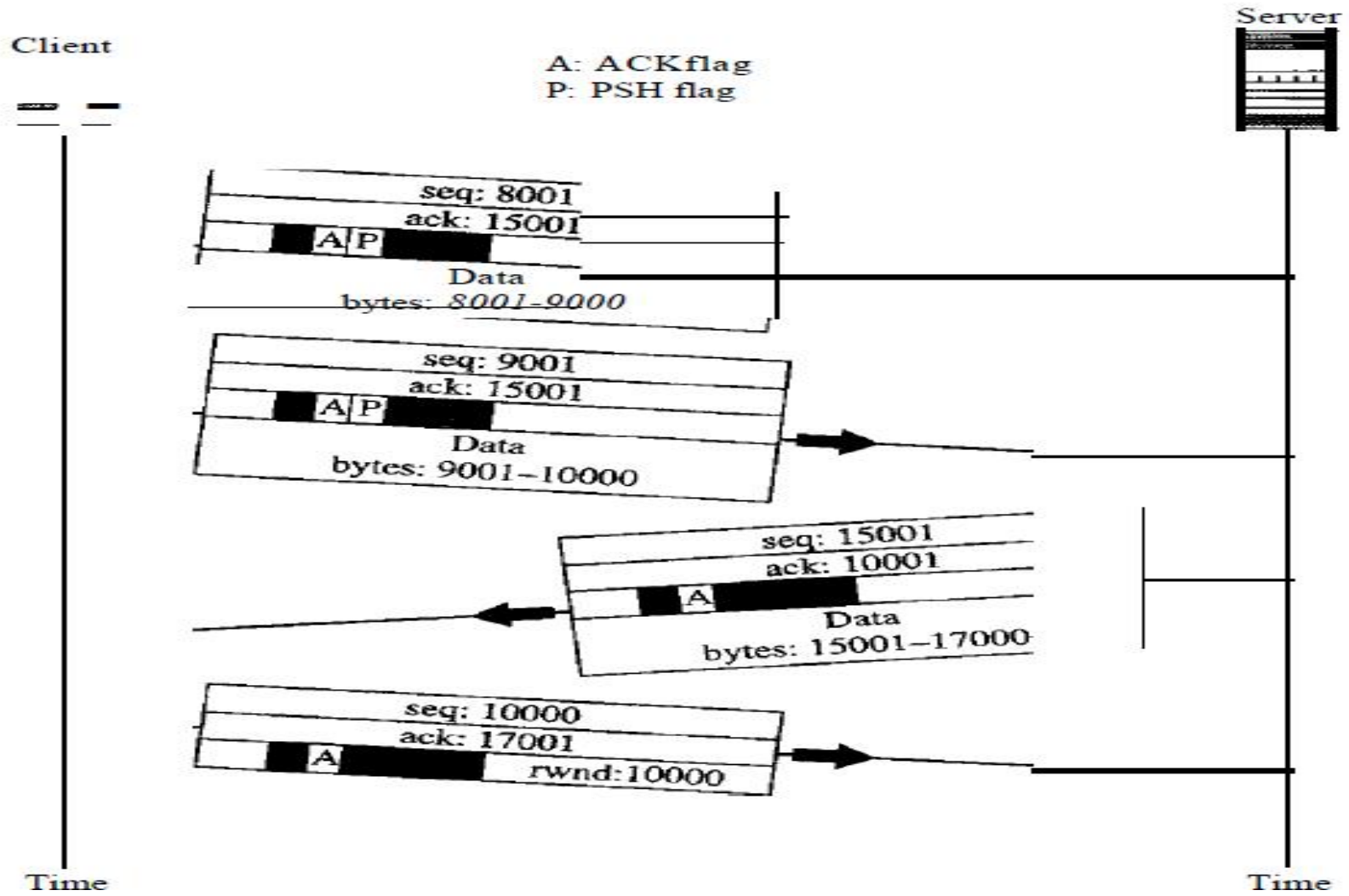
If, during this short time, the number of SYN segments is large, the server eventually runs out of resources and may crash.

This SYN flooding attack belongs to a type of security attack known as a **denial-of-service attack**, in which an attacker monopolizes a system with so many service requests that the system collapses and denies service to every request.

Second Phase: ***Data Transfer:*** After connection is established, **bidirectional** data transfer can take place. The client and server can both send data and acknowledgments.

The acknowledgment is piggybacked with the data.

Data transfer



In the previous figure, after connection is established, the client sends 2000 bytes of data in two segments.

The server then sends 2000 bytes in one segment.

The client sends one more segment.

The first three segments carry both data and acknowledgment, but the last segment carries only an acknowledgment because there are no more data to be sent.

The data segments sent by the client have the **PSH (push) flag set** so that the server TCP knows to deliver data to the server process as soon as they are received.

The segment from the server, on the other hand, does not set the push flag.

Urgent Data: TCP is a stream-oriented protocol. This means that the data are presented from the application program to TCP as a stream of bytes.

Each byte of data has a position in the stream. However, on occasion an application program needs to send *urgent* bytes.

This means that the sending application program wants a piece of data to be read out of order by the receiving application program.

The solution is to send a segment with the URG bit set.

The sending application program tells the sending TCP that the piece of data is urgent. The sending 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.

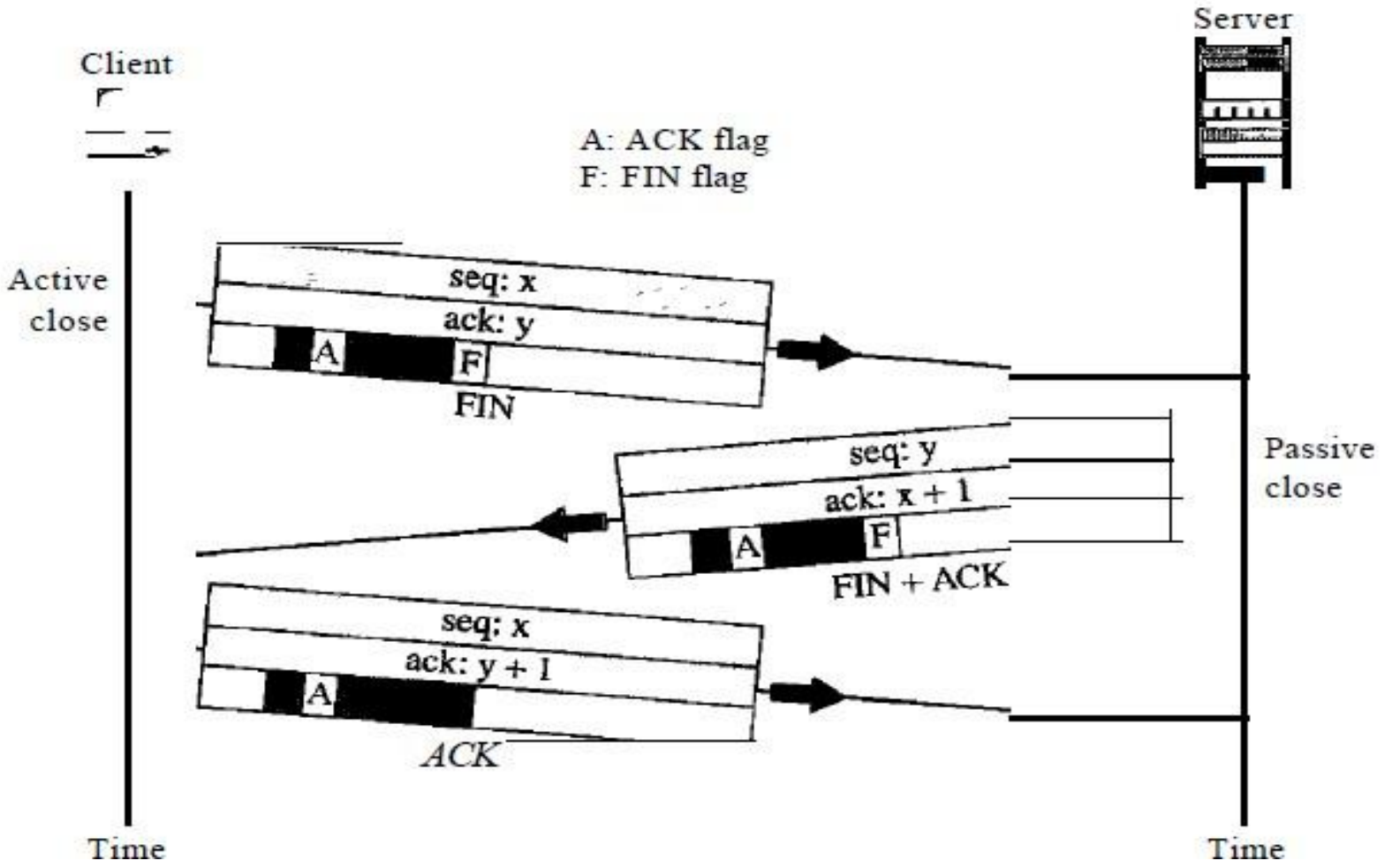
Third Phase: ***Connection Termination:*** Any of the two parties involved in exchanging data (client or server) can close the connection, although it is usually initiated by the client.

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

Three-Way Handshaking for connection termination:

1. In a normal 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 Figure in the next slide.


Client
Server
A: ACK flag
F: FIN flag
Active close
seq: x
ack: y
A
F
FIN
seq: y
ack: x + 1
A
F
FIN + ACK
Passive close
seq: x
ack: y + 1
A
ACK
Time
Time

2.The server TCP, after receiving the FIN segment, informs its process of the situation and sends the second segment, a FIN +ACK segment, to confirm the receipt of the FIN segment from the client and at the same time to announce the closing of the connection in the other direction.

This segment can also contain the last chunk of data from the server.

If it does not carry data, it consumes only one sequence number.

3.The client TCP sends the last segment, an ACK segment, to confirm the receipt of the FIN segment from the TCP server.

This segment contains the acknowledgment number, which is 1 plus the sequence number received in the FIN segment from the server. This segment cannot carry data and consumes no sequence numbers.

Half-Close: In TCP, one end can stop sending data while still receiving data. This is called a half-close.

Although either end can issue a half-close, it is normally initiated by the client.

It can occur when the server needs all the data before processing can begin.

Figure in the next slide shows an example of a half-close.

The client half-closes the connection by sending a FIN segment. The server accepts the half-close by sending the ACK segment.

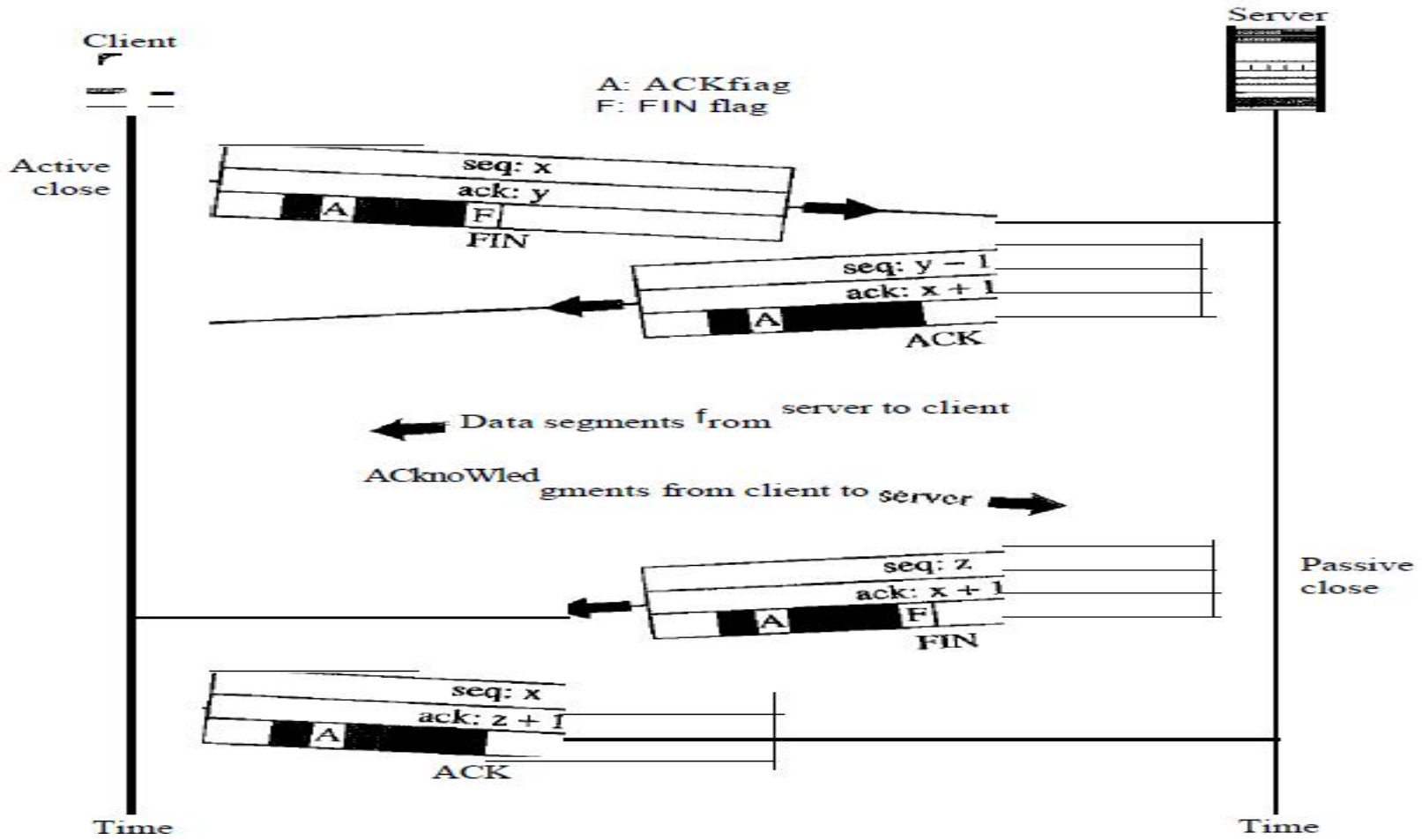
The data transfer from the client to the server stops. The server, however, can still send data.

When the server has sent all the processed data, it sends a FIN segment, which is acknowledged by an ACK from the client.

After half-closing of the connection, data can travel from the server to the client and acknowledgments can travel from the client to the server.

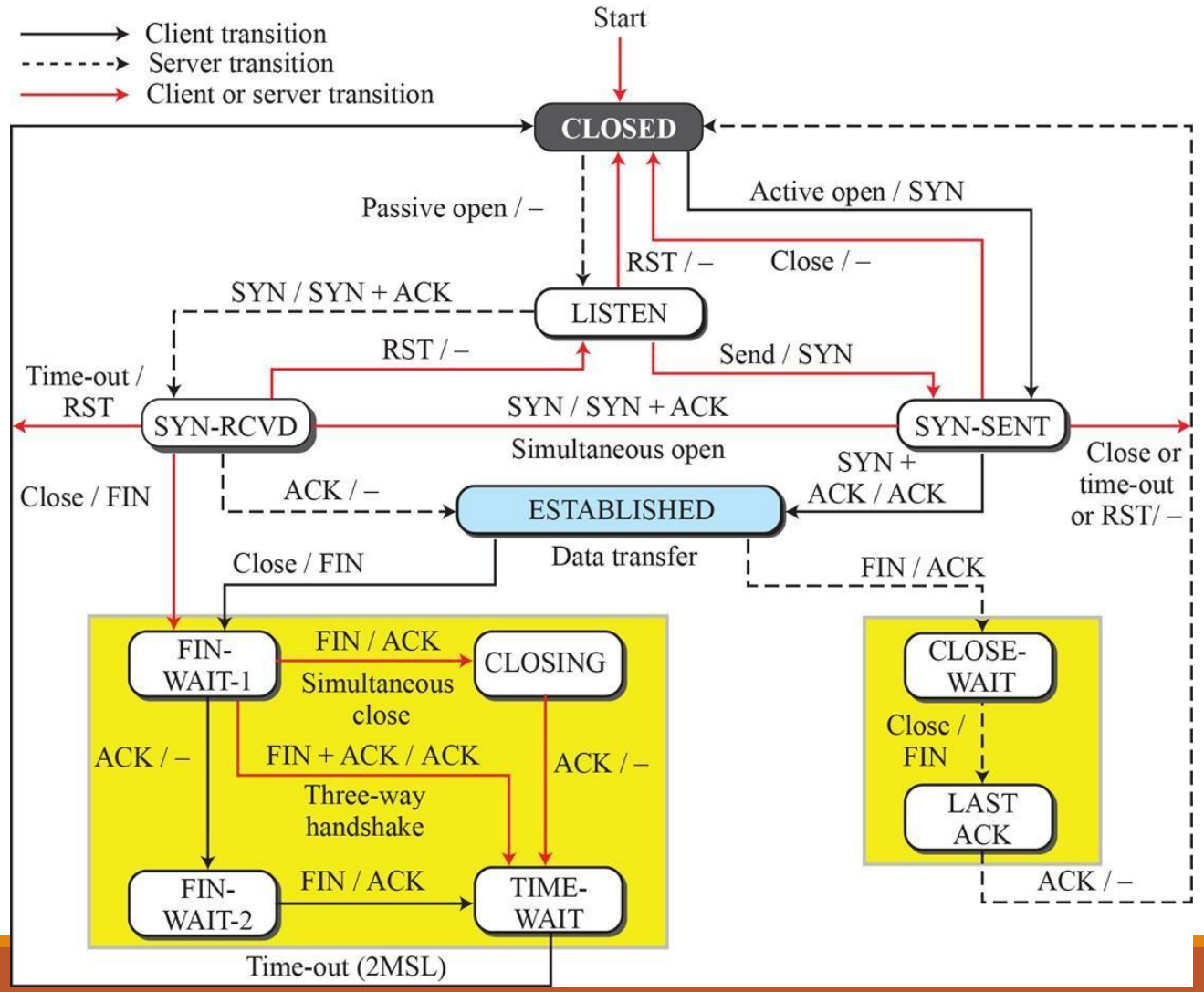
The client cannot send any more data to the server.

.21 *Half-close*

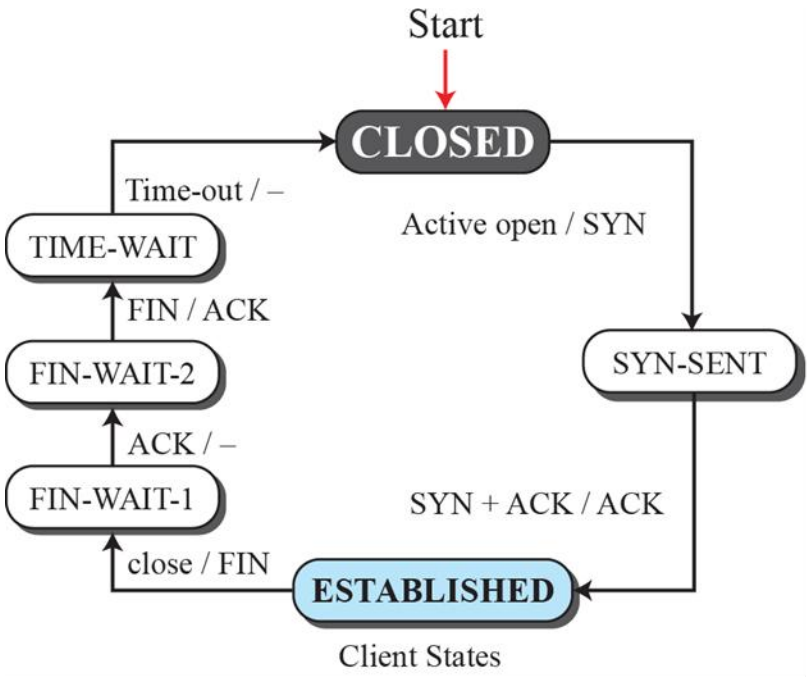


State Transition Diagram

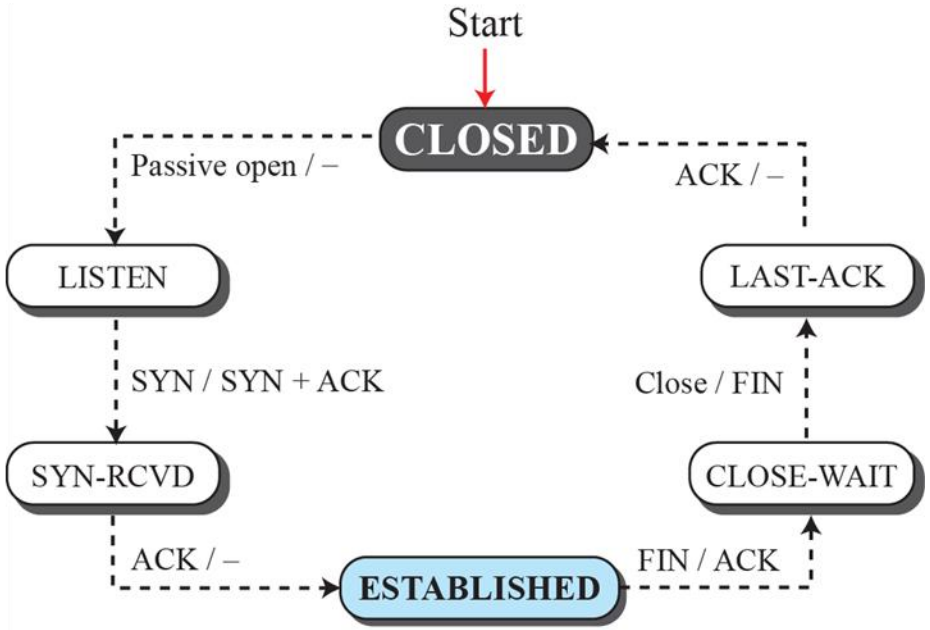
To keep track of all the different events happening during connection establishment, connection termination, and data transfer, TCP is specified as the finite state machine (FSM)



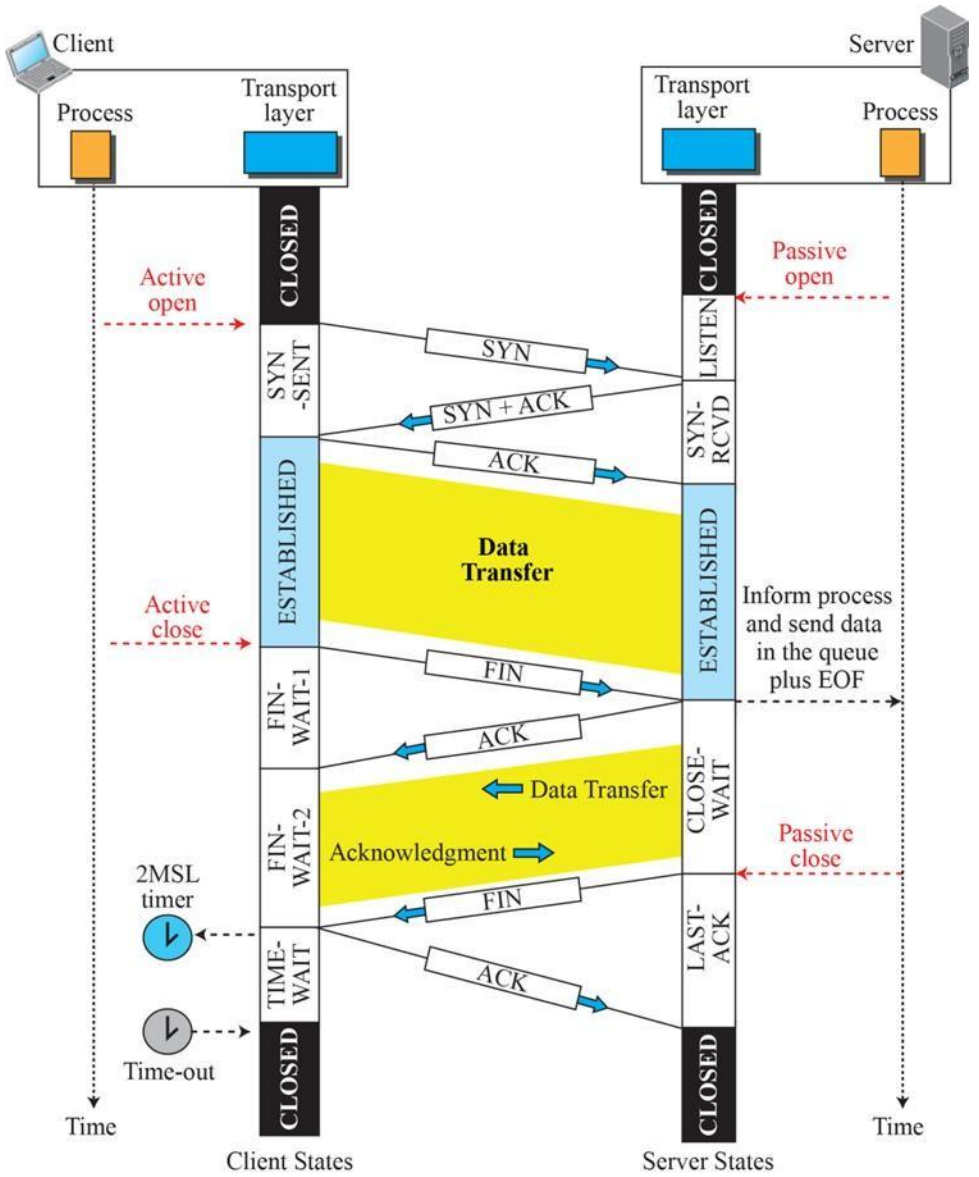
State	*Description*
CLOSED	No connection exists
LISTEN	Passive open received; waiting for SYN
SYN-SENT	SYN sent; waiting for ACK
SYN-RCVD	SYN+ACK sent; waiting for ACK
ESTABLISHED	Connection established; data transfer in progress
FIN-WAIT-1	First FIN sent; waiting for ACK
FIN-WAIT-2	ACK to first FIN 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



Client States

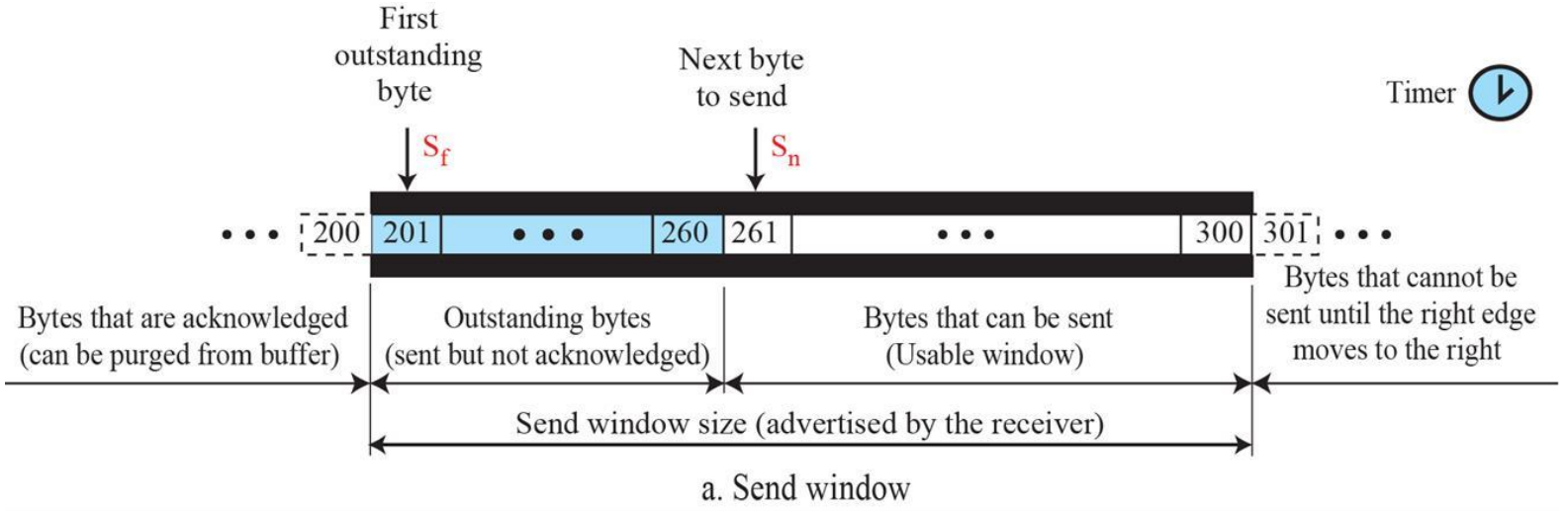


Server States


Client
Server
Process
Transport layer
Transport layer
Process
CLOSED
CLOSED
Active open
Passive open
SYN
SYN-SENT
LISTEN
SYN + ACK
SYN-RCVD
ACK
ESTABLISHED
Data Transfer
ESTABLISHED
Active close
Inform process and send data in the queue plus EOF
FIN
FIN-WAIT-1
ACK
Data Transfer
CLOSE-WAIT
FIN-WAIT-2
Acknowledgment
Passive close
2MSL timer
FIN
LAST-ACK
TIME-WAIT
ACK
Time-out
CLOSED
CLOSED
Time
Time
Client States
Server States

Windows in TCP

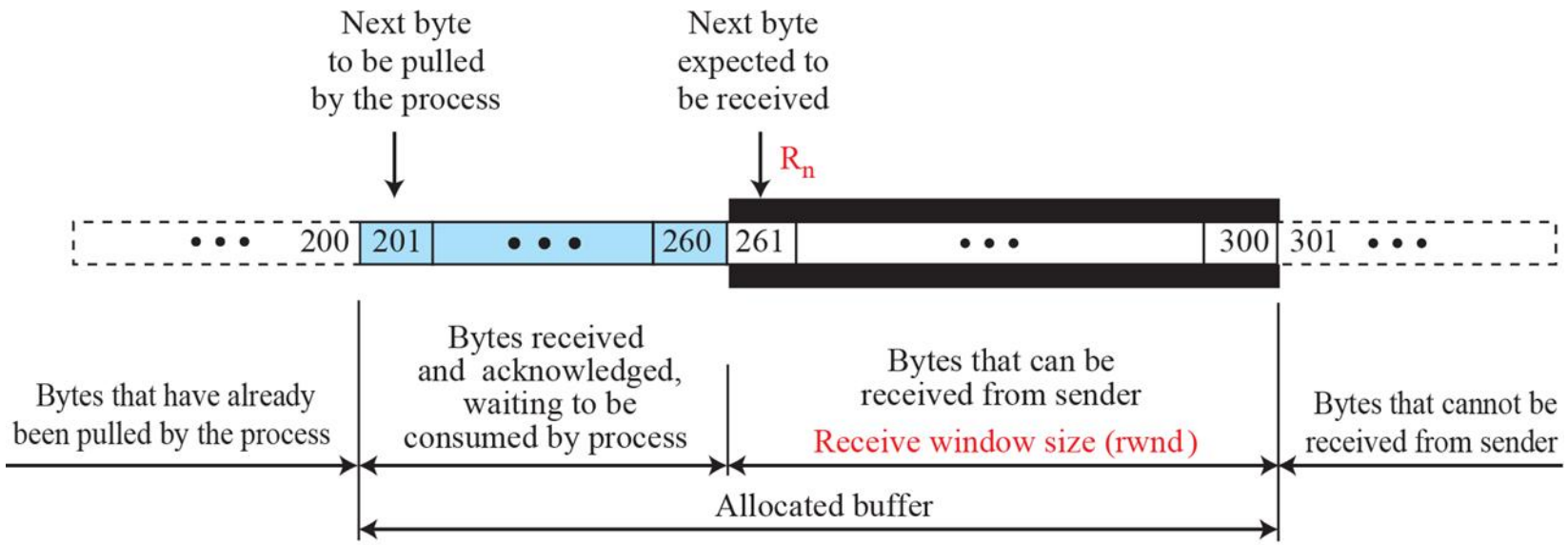
TCP uses two windows (send window and receive window) for each direction of data transfer, which means four windows for a bidirectional communication. To make the discussion simple, we make an unrealistic unidirectional; the bidirectional communication can be inferred using two unidirectional communications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



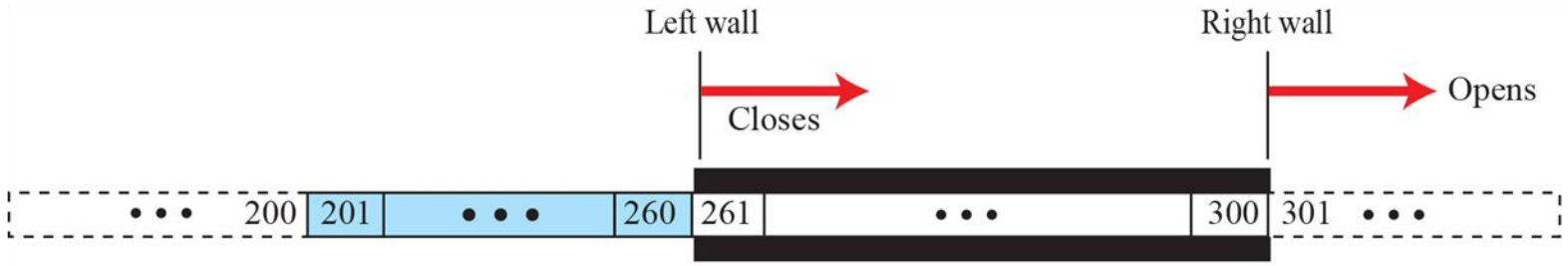
a. Send window

b. Opening, closing, and shrinking send window

Send window in TCP



a. Receive window and allocated buffer



b. Opening and closing of receive window

Receive window in TCP

Flow control in TCP:

TCP uses a sliding window, to handle flow control.

The sliding window protocol in TCP looks like the Go-Back-N protocol because it does not use NAKs; it looks like Selective Repeat because the receiver holds the out-of-order segments until the missing ones arrives.

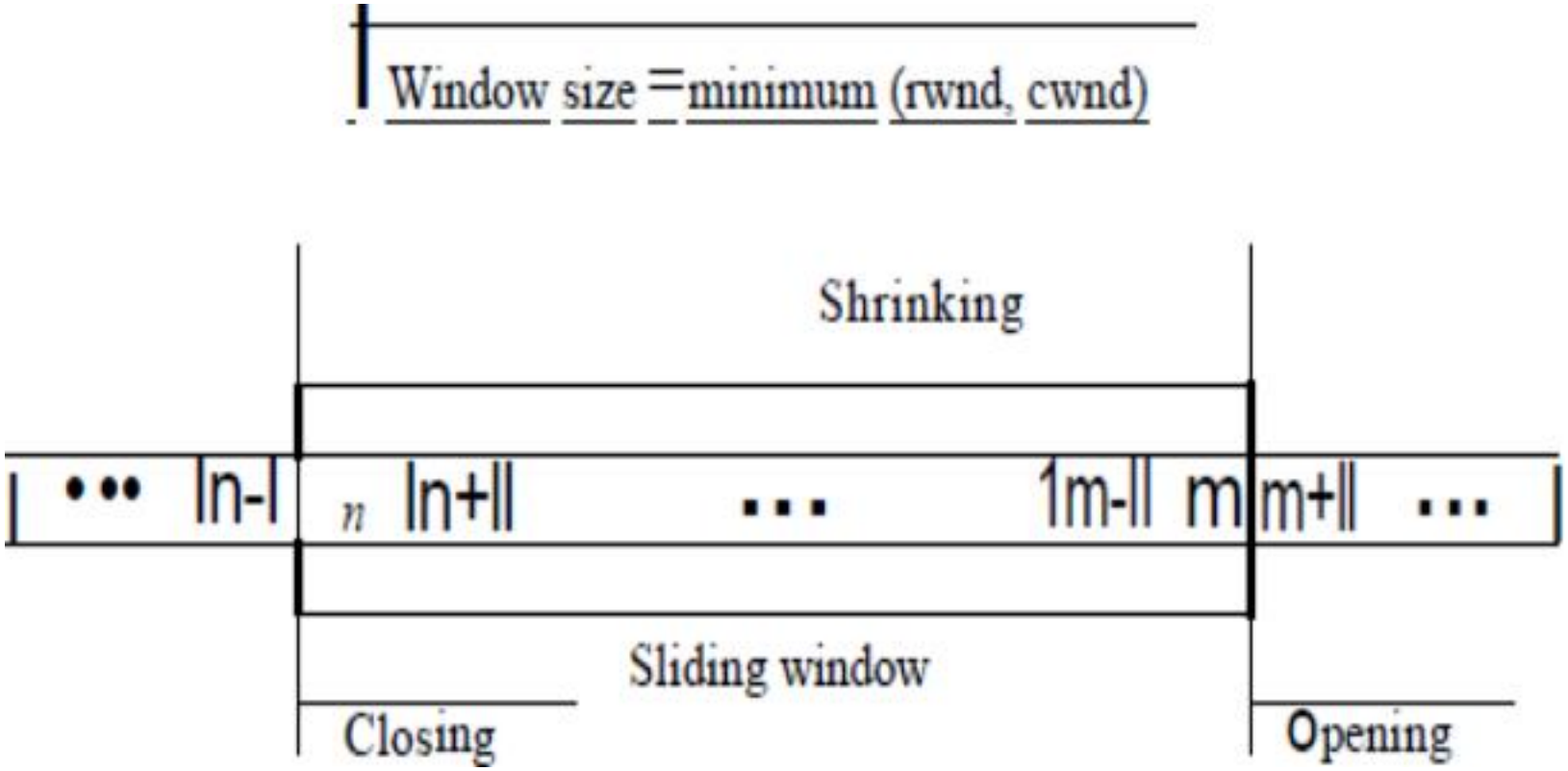
The sliding window of TCP is **byte-oriented** and it is of **variable size**.

Figure in the next slide shows the sliding window in TCP.

The window spans a portion of the buffer containing bytes received from the process.

The bytes inside the window are the bytes that can be in transit; they can be sent without worrying about acknowledgment.

The imaginary window has **two walls**: one **left** and one **right**.



Sliding Window in TCP

The window is ***opened***, ***closed***, *or* ***shrunk***. These three activities, are in the control of the receiver (and depend on congestion in the network), not the sender.

The sender must obey the commands of the receiver in this matter.

Opening a window means moving the right wall to the right. This allows more new bytes in the buffer that are eligible for sending.

Closing the window means moving the left wall to the right. This means that some bytes have been acknowledged and the sender need not worry about them anymore.

Shrinking the window means moving the **right wall to the left**. This is strongly discouraged and not allowed in some implementations because it means revoking the eligibility of some bytes for sending. This is a problem if the sender has already sent these bytes.

Note that the left wall cannot move to the left because this would revoke some of the previously sent acknowledgments.

The size of the window at one end is determined by the lesser of two values: ***receiver window (rwnd)*** *or* ***congestion window (cwnd).***

The receiver window is the value advertised by the opposite end in a segment containing acknowledgment.

It is the number of bytes the other end can accept before its buffer overflows and data are discarded.

The congestion window is a value determined by the network to avoid congestion.

Q. What is the value of the receiver window (rwnd) for host A if the receiver, host B, has a buffer size of 5000 bytes and 1000 bytes of received and unprocessed data?

Solution:

The value of rwnd =5000 - 1000 =4000. Host B can receive only 4000 bytes of data before overflowing its buffer.

Host B advertises this value in its next segment to A.

Q. What is the size of the window for host A if the value of *rwnd* is 3000 bytes and the value of *cwnd* is 3500 bytes?

Solution:

The size of the window is the smaller of *rwnd* and *cwnd,* which is 3000 bytes.

TCP Congestion Control

Topics to cover

1. Congestion Window
2. Congestion Detection
3. Congestion Policies
   - Slow start
   - Congection avoidance
   - Fast recovery

Congestion Window

Imagine you're sending data over the internet using TCP (Transmission Control Protocol). TCP wants to make sure that it sends data efficiently without causing network congestion or losing too much data.

1.Flow Control with TCP:

- TCP uses a "send window" to control the amount of data it sends.
- There are **two factors determining this window size: rwnd (receiver window) and cwnd (congestion window)**.
- rwnd is set by the receiver to avoid overflow at the receiving end.
- However, even if the receiving end is fine, **the routers in the middle might get congested, dropping some segments.**

2. Congestion in the Network:

- TCP, being an end-to-end protocol, relies on IP for services.
- But **IP doesn't handle congestion well, so TCP has to deal with it**.
- **TCP can't be too aggressive (flooding the network) or too conservative (underutilizing bandwidth)**. It needs a balance.

3. Congestion Window (cwnd):

- TCP uses a variable called cwnd to control the number of segments it sends.
- **cwnd size depends on the congestion situation in the network.**
- **The actual send window size is the minimum of rwnd and cwnd.**

4. Detecting Congestion:

- TCP sender looks for signs of congestion, like timeout or receiving three duplicate ACKs.
- **Timeout occurs if the sender doesn't get an acknowledgment within a set time**, indicating possible congestion.
- **Three duplicate ACKs suggest a missing segment,** possibly due to congestion.
- TCP needs to adjust its behavior based on these signs to avoid making congestion worse.

Congestion Detection

When TCP is sending data over the internet, it needs to make sure it's not overwhelming the network. So, it has a way to detect if there's congestion:

1. Timeout Event:
- If the sender doesn't get an acknowledgment (ACK) for a segment within a certain time (timeout), it assumes that the 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.

2. Three Duplicate ACKs Event:
- If the sender receives three duplicate ACKs (four ACKs with the same acknowledgment number), it means one segment is missing, but the receiver got the other three.
- This could indicate a less severe congestion or that the network has recovered a bit.
- **It's a sign of weaker congestion compared to a timeout.**

3. Different Treatments in TCP Versions:

- Older TCP versions treated both timeout and three duplicate ACKs similarly.
- Newer versions, like Reno TCP, treat these two signs differently based on their severity.

4. TCP Relies on ACKs:

- TCP relies on feedback from the other end in the form of ACKs to detect congestion.
- Lack of regular, timely ACKs leading to a timeout suggests strong congestion.
- Receiving three duplicate ACKs indicates weaker congestion in the network.

Congestion Policies

TCP's general policy for handling congestion is based on three algorithms:

1. Slow start
2. Congestion avoidance
3. 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.

1. Starting Slowly:
- At the beginning, 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.

2. 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.

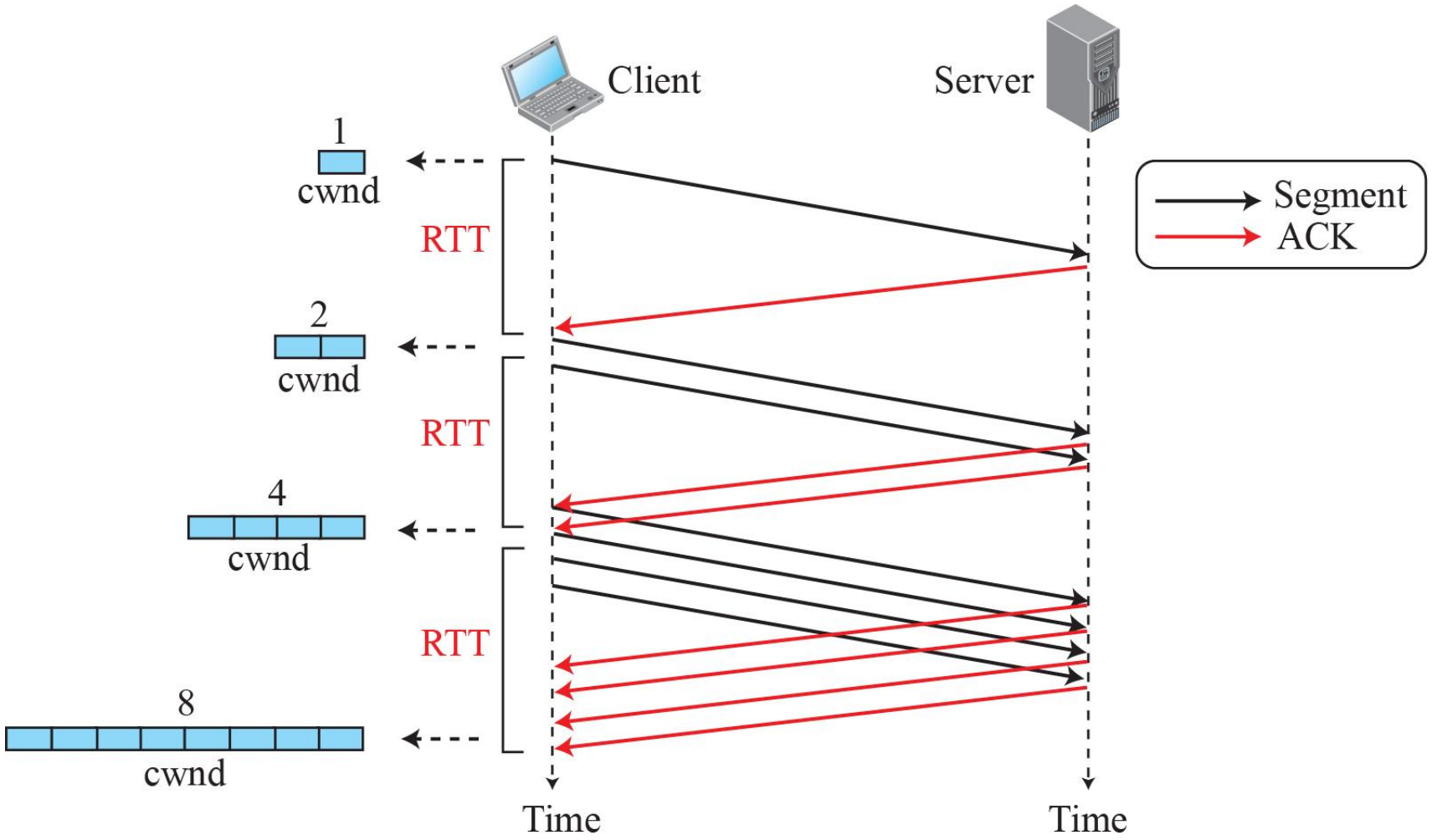
Stopping Slow Start:

- It can't go on forever, though. **There's a limit called the slow-start threshold (ssthresh)**.
- **When the window size reaches this threshold, slow start stops, and a new phase begins**.

Consideration for Delayed Acknowledgments:

- 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.


Client
Server
1
cwnd
RTT
2
cwnd
RTT
4
cwnd
RTT
8
cwnd
Segment
ACK
Time
Time

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

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

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).

3. Example:

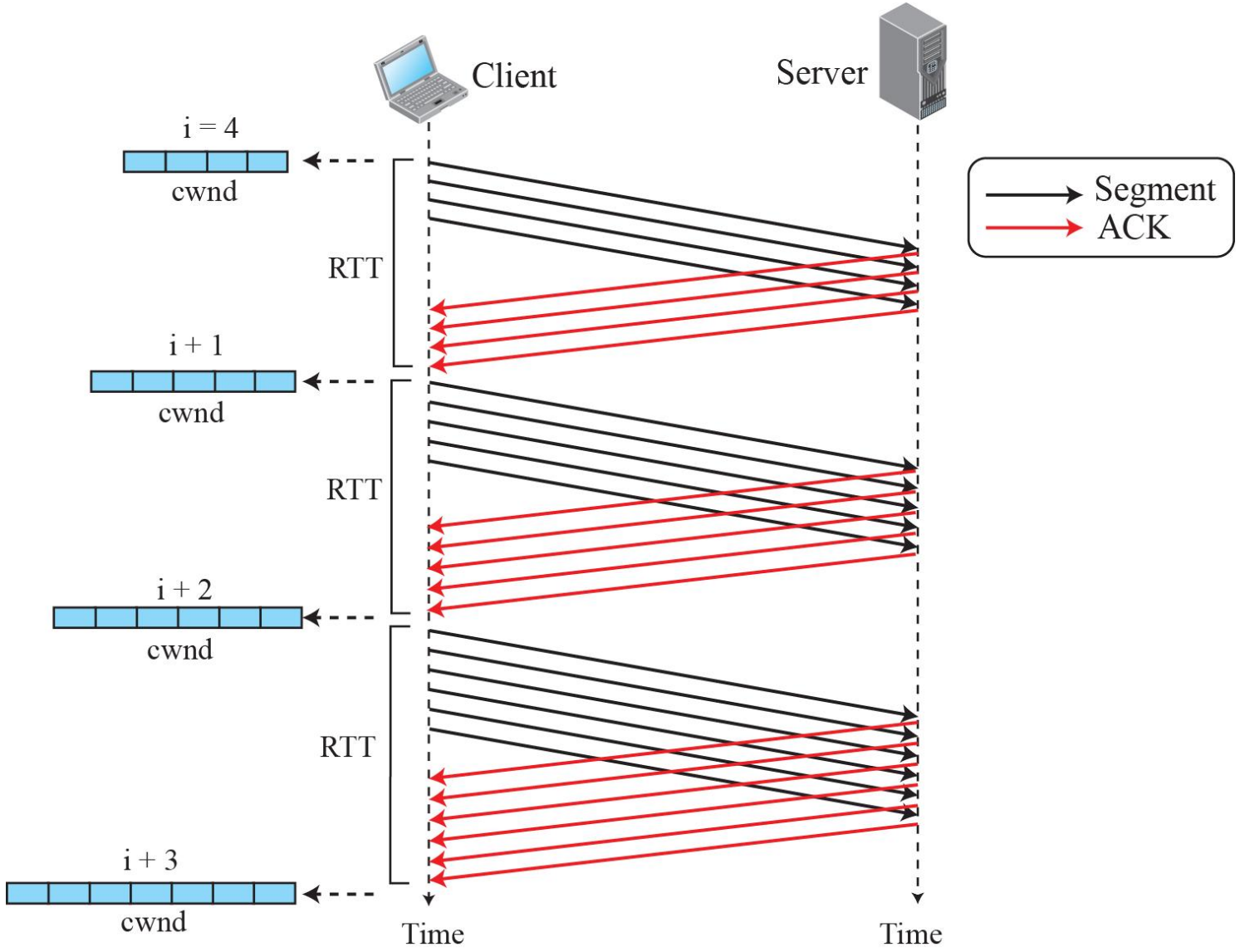
- Let's say the sender starts with cwnd = 4, meaning it can send four segments initially.
- After getting four acknowledgments, it can send one more segment, and the window becomes 5.
- The process continues, increasing the window each time the whole window is acknowledged.

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

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)**. We can say

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

Q. let the size of congestion window of a TCP Connection in two cases when
Case1: timeout occur
case2: 3 ACK Recived
is 32KB. the RTT of a connection is 100m sec and MSS = 2KB. the time taken (m sec.) by TCP connection to get back to 32KB congestion Window is _____ and _____ respectively.

Policy Transition

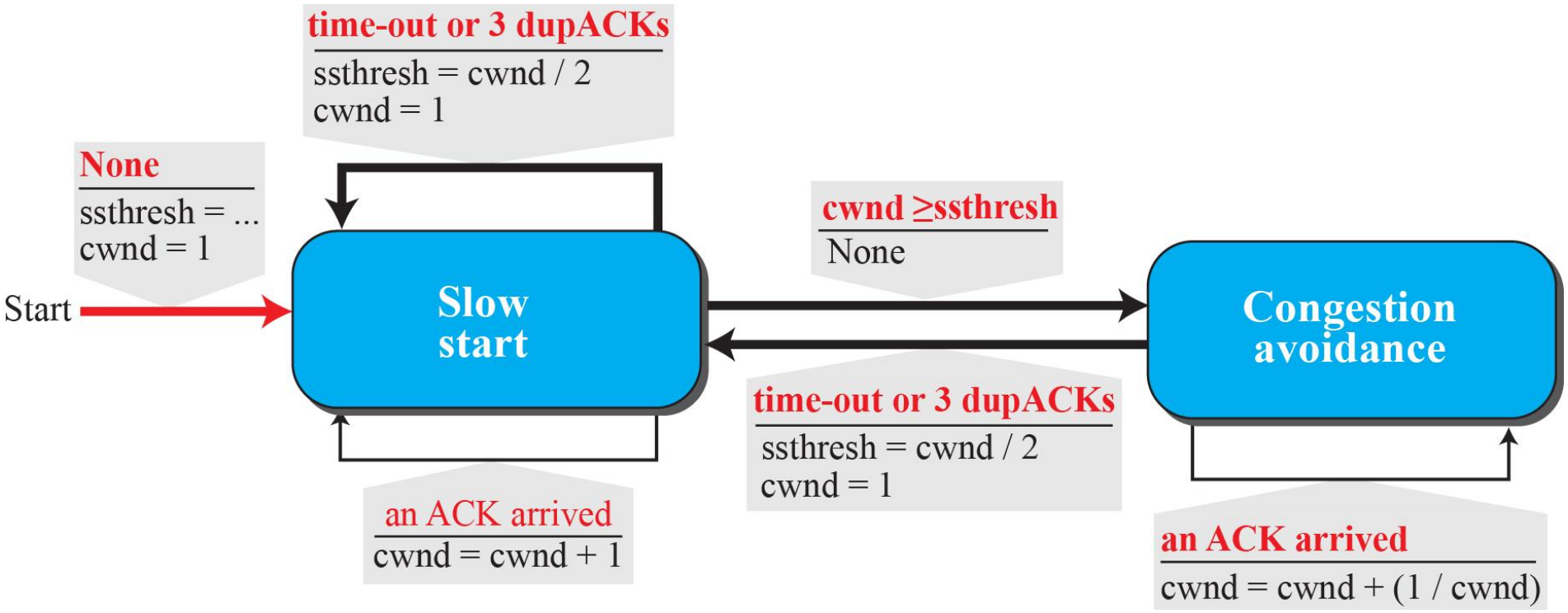
We discussed three congestion policies in TCP. Now the question is when each of these policies are used and when TCP moves from one policy to another. To answer these questions, we need to refer to three versions of TCP:

1. Taho TCP
2. Reno TCP
3. New Reno TCP

Taho TCP

Taho TCP, an early version of TCP, implemented a congestion control mechanism with two algorithms: slow start and congestion avoidance. During connection establishment, it initiated the slow-start algorithm, incrementing the congestion window (cwnd) by 1 for each incoming acknowledgment (ACK). This aggressive growth, while efficient, could lead to congestion. Upon detecting congestion through time-out or three duplicate ACKs, Taho TCP immediately interrupted its aggressive growth and initiated a new slow start, setting the threshold (ssthresh) to half of the current cwnd and resetting cwnd to 1. In congestion avoidance state, cwnd increased by 1 for every set of ACKs equal to the current window size, with no ceiling for growth unless congestion was detected. If congestion occurred in this state, Taho TCP reset ssthresh to half of the current cwnd and returned to slow start. The continuous adjustment of ssthresh with each congestion detection did not necessarily result in a decrease; it could increase based on the circumstances. Overall, Taho TCP aimed to balance aggressive growth, learn from congestion events, and adapt its behavior for effective congestion control.

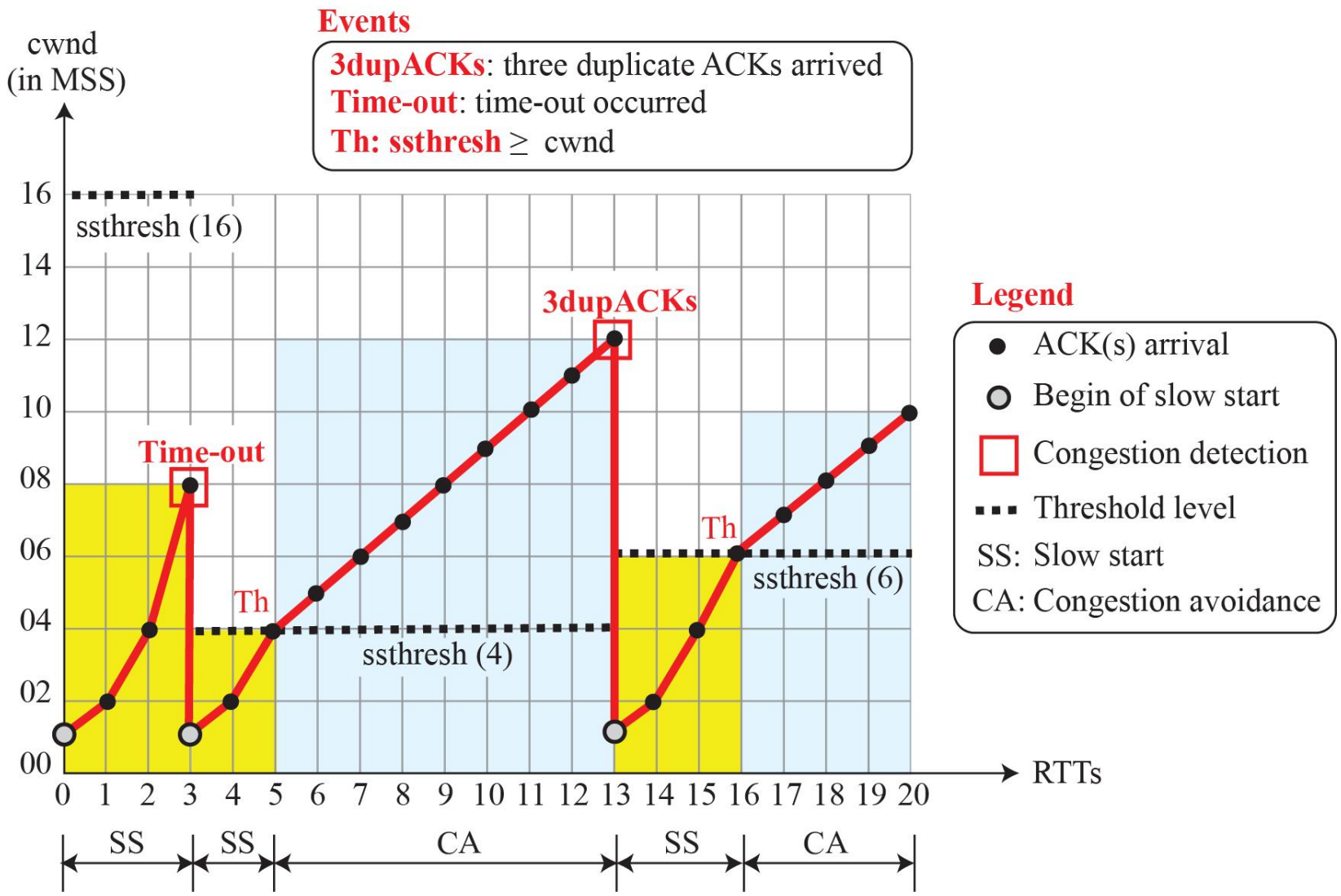
In other words, not only does TCP restart from scratch, but it also learns how to adjust the threshold.


time-out or 3 dupACKs
ssthresh = cwnd / 2
cwnd = 1
None
ssthresh = ...
cwnd = 1
Start
Slow start
cwnd ≥ssthresh
None
Congestion avoidance
time-out or 3 dupACKs
ssthresh = cwnd / 2
cwnd = 1
an ACK arrived
cwnd = cwnd + 1
an ACK arrived
cwnd = cwnd + (1 / cwnd)

Example

Figure 3.69 shows an example of congestion control in a Taho TCP. TCP starts data transfer and sets the ssthresh variable to an ambitious value of 16 MSS. TCP begins at the slow-start (SS) state with the cwnd = 1. The congestion window grows exponentially, but a time-out occurs after the third RTT (before reaching the threshold). TCP assumes that there is congestion in the network. It immediately sets the new ssthresh = 4 MSS (half of the current cwnd, which is 8) and begins a new slow start (SA) state with cwnd = 1 MSS. The congestion grows exponentially until it reaches the newly set threshold. TCP now moves to the congestion avoidance (CA) state and the congestion window grows additively until it reaches cwnd = 12 MSS.

At this moment, three
duplicate ACKs arrive, another indication of the congestion in the network. TCP again halves the value of ssthresh to 6 MSS and begins a new slow-start (SS) state. The exponential growth of the cwnd continues. After RTT 15, the size of cwnd is 4 MSS. After sending four segments and receiving only two ACKs, the size of the window reaches the ssthresh (6) and the TCP moves to the congestion avoidance state. The data transfer now continues in the congestion avoidance (CA) state until the connection is terminated after RTT 20.


Events
3dupACKs: three duplicate ACKs arrived
Time-out: time-out occurred
Th: ssthresh ≥ cwnd
cwnd
(in MSS)
16
14
12
10
08
06
04
02
00
ssthresh (16)
Time-out
Th
ssthresh (4)
3dupACKs
Th
ssthresh (6)
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
RTTs
SS
SS
CA
SS
CA
Legend
ACK(s) arrival
Begin of slow start
Congestion detection
Threshold level
SS: Slow start
CA: Congestion avoidance

Reno TCP

1. Introduction of Fast-Recovery State:
- In the newer version of TCP, known as Reno TCP, a fast-recovery state was added to the congestion control Finite State Machine (FSM).

2. Handling Congestion Signals:
- Reno TCP treated congestion signals differently: time-out and arrival of three duplicate ACKs.
- If a time-out occurred, TCP moved to the slow-start state (or started a new round if already in slow start).
- If three duplicate ACKs arrived, TCP moved to the fast-recovery state and stayed there as long as more duplicate ACKs were received.

3. Fast-Recovery State Behavior:
- The fast-recovery state was positioned between slow start and congestion avoidance.
- It behaved like slow start, with the congestion window (cwnd) growing exponentially, but it started with the value of ssthresh plus 3 Maximum Segment Sizes (MSS) instead of 1.

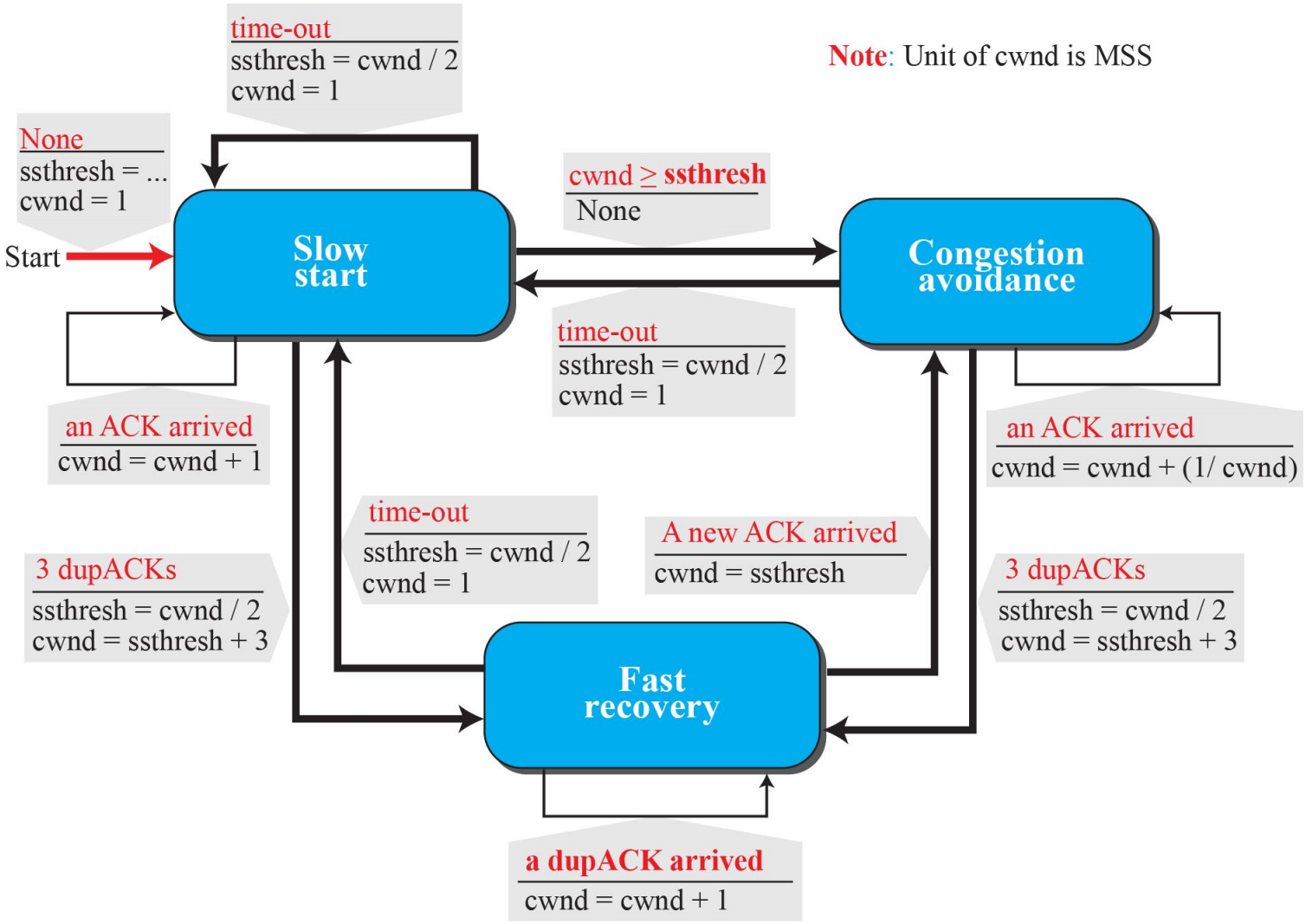
4. Events in Fast-Recovery State:

In the fast-recovery state, several events could occur.

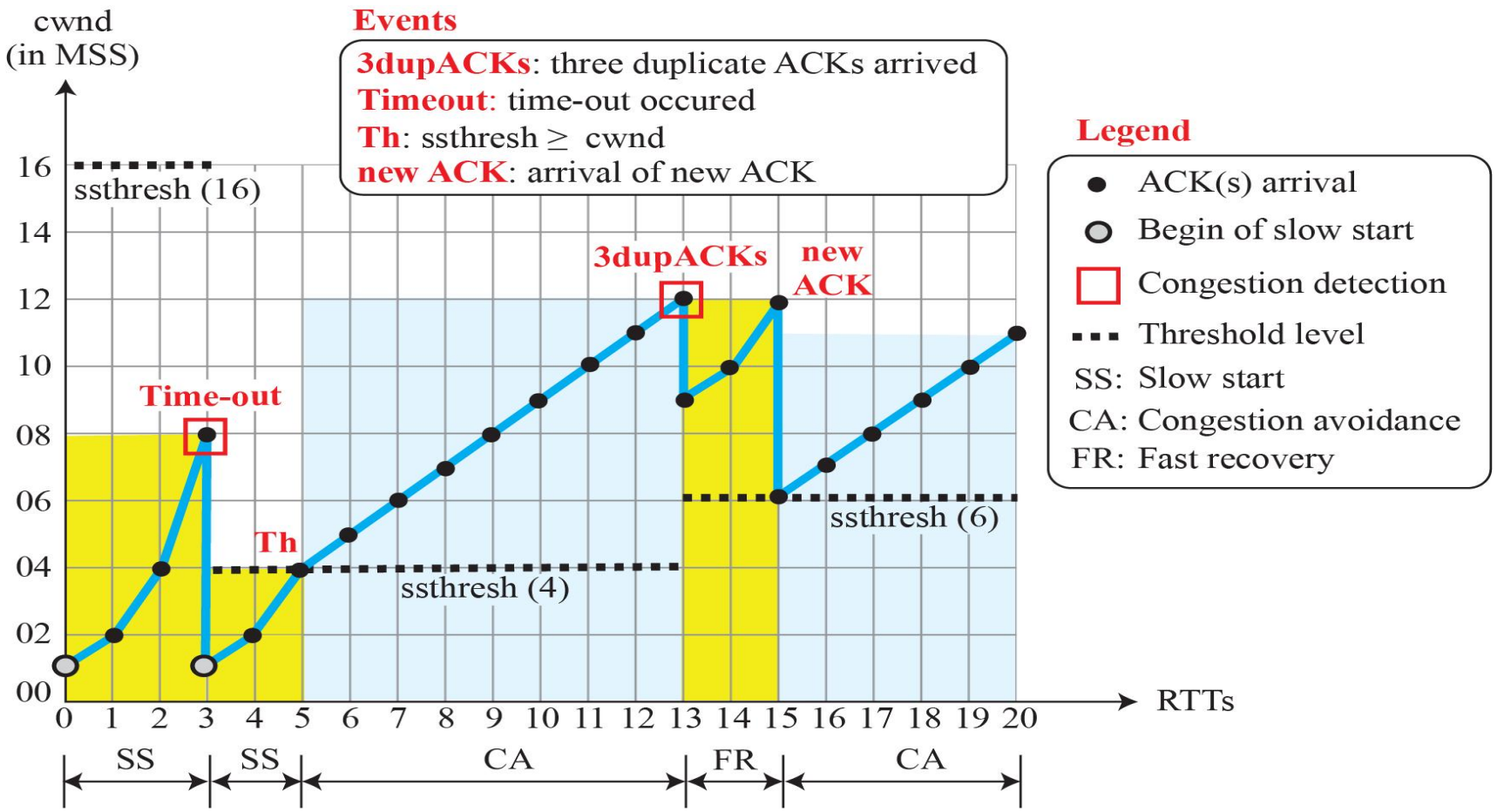
- If duplicate ACKs continued to arrive, TCP stayed in this state, and cwnd grew exponentially.
- If a time-out occurred, TCP assumed real congestion and moved to the slow-start state.
- If a new (non-duplicate) ACK arrived, TCP moved to the congestion-avoidance state but reduced the cwnd size to the ssthresh value, as if the three duplicate ACKs had not occurred.

5. Transition in FSM:

- Figure displayed the simplified FSM for Reno TCP, with some trivial events removed for simplicity.


Note: Unit of cwnd is MSS
time-out
ssthresh = cwnd / 2
cwnd = 1
None
ssthresh = ...
cwnd = 1
Start
Slow start
cwnd ≥ ssthresh
None
Congestion avoidance
time-out
ssthresh = cwnd / 2
cwnd = 1
an ACK arrived
cwnd = cwnd + 1
an ACK arrived
cwnd = cwnd + (1/ cwnd)
time-out
ssthresh = cwnd / 2
cwnd = 1
A new ACK arrived
cwnd = ssthresh
3 dupACKs
ssthresh = cwnd / 2
cwnd = ssthresh + 3
3 dupACKs
ssthresh = cwnd / 2
cwnd = ssthresh + 3
Fast recovery
a dupACK arrived
cwnd = cwnd + 1

Figure(on next page) shows the same situation as Figure(taho tcp), but in Reno TCP. The changes in the congestion window are the same until RTT 13 when three duplicate ACKs arrive. At this moment, Reno TCP drops the ssthresh to 6 MSS, but it sets the cwnd to a much higher value (ssthresh + 3 = 9 MSS) instead of 1 MSS. It now moves to the fast recovery state. We assume that two more duplicate ACKs arrive until RTT 15, where cwndgrows exponentially. In this moment, a new ACK (not duplicate) arrives that announces the receipt of the lost segment. It now moves to the congestion avoidance state, but first deflates the congestion window to 6 MSS as though ignoring the whole fast-recovery state and moving back to the previous track.


cwnd
(in MSS)
Events
3dupACKs: three duplicate ACKs arrived
Timeout: time-out occured
Th: ssthresh ≥ cwnd
new ACK: arrival of new ACK
Legend
ACK(s) arrival
Begin of slow start
Congestion detection
Threshold level
SS: Slow start
CA: Congestion avoidance
FR: Fast recovery
ssthresh (16)
Time-out
Th
ssthresh (4)
3dupACKs
new ACK
ssthresh (6)
00
02
04
06
08
10
12
14
16
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
RTTs
SS
SS
CA
FR
CA

NewReno TCP

imagine you're sending a bunch of letters to your friend, and you want to make sure they get every letter you send. Now, let's say your friend receives three identical letters from you. Instead of just acknowledging the receipt of those three letters, they let you know that they've got them by sending you a duplicate of one of the letters.

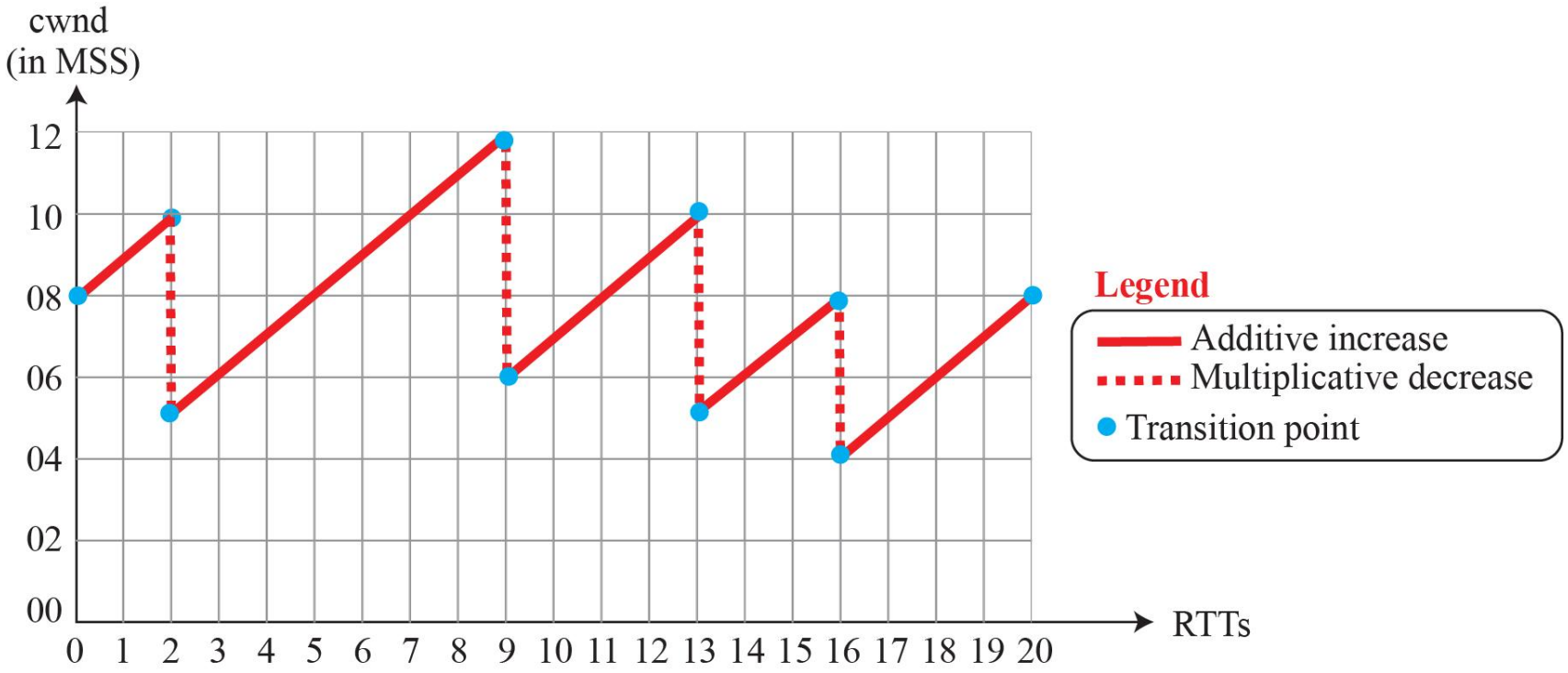
Now, in the regular TCP (Transmission Control Protocol), if your friend gets three duplicate letters, you assume one letter got lost somewhere in the mail. So, you resend that missing letter.

NewReno TCP is like a smarter version of this. If your friend gets three duplicate letters, it checks to see if more than one letter is missing. If it is sure that only one letter is lost, it just resends that missing letter. But, if it's not sure and thinks there might be more missing, it resends the one it thinks might be lost, even if there's a chance it's not. This way, it's being extra cautious to make sure all your letters get through without any missing.

Additive Increase, Multiplicative Decrease

Of the three TCP versions, the Reno version is the most common nowadays. In this version, congestion is mostly identified and addressed by recognizing three duplicate ACKs. Even when there are occasional time-out events, TCP rebounds from them through rapid and aggressive growth. Simply put, in a prolonged TCP connection, disregarding the initial slow-start phases and brief exponential growth during fast recovery, the TCP congestion window increases by a fraction (congestion avoidance) when an ACK arrives, and it is halved when congestion is detected. This occurs as if the slow-start phase doesn't exist, and the fast-recovery phase is reduced to zero. The first action is termed additive increase, while the second is known as multiplicative decrease. Essentially, after surpassing the initial slow start, the congestion window size follows a pattern resembling a sawtooth: additive increase, multiplicative decrease (AIMD).

if we ignore the slow-start states and short exponential growth during fast recovery, the TCP congestion window is cwnd = cwnd + (1/cwnd) when an ACK arrives (congestion avoidance), and cwnd = cwnd / 2 when congestion is detected


cwnd
(in MSS)
12
10
08
06
04
02
00
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
RTTs
Legend
Additive increase
Multiplicative decrease
Transition point

TCP Throughput

The throughput for TCP, which is based on the congestion window behavior, can be easily found if the cwnd is a constant (flat line) function of RTT. The throughput with this unrealistic assumption is throughput = cwnd / RTT. In this assumption, TCP sends a cwnd bytes of data and receives acknowledgement for them in RTT time. The behavior of TCP, as shown in Figure, is not a flat line; it is like saw teeth, with many minimum and maximum values. If each tooth were exactly the same, we could say that the throughput = [(maximum + minimum) / 2] / RTT. However, we know that the value of the maximum is twice the value of the minimum because in each congestion detection the value of cwnd is set to half of its previous value. So the throughput can be better calculated as

$$\text{Throughput} = (0.75)\ W_{max} / RTT$$

in which Wmax is the average of window sizes when the congestion occurs.

Example

If MSS = 10 KB (kilobytes) and RTT = 100 ms in Figure 3.72, we can calculate the throughput as shown below.

$$W_{max} = (10 + 12 + 10 + 8 + 8) / 5 = 9.6 \text{ MSS}$$

$$\text{Throughput} = (0.75\ W_{max} / \text{RTT}) = 0.75 \times 960 \text{ kbps} / 100 \text{ ms} = 7.2 \text{ Mbps}$$