Chapter 12

Transmission Control Protocol (TCP)

Objectives:

- ***Upon completion you will be able to:***
  - ***Be able to name and understand the services offered by TCP***
  - ***Understand TCP's flow and error control and congestion control***
  - ***Be familiar with the fields in a TCP segment***
  - ***Understand the phases in a connection-oriented connection***
  - ***Understand the TCP transition state diagram***
  - ***Be able to name and understand the timers used in TCP***
  - ***Be familiar with the TCP options***

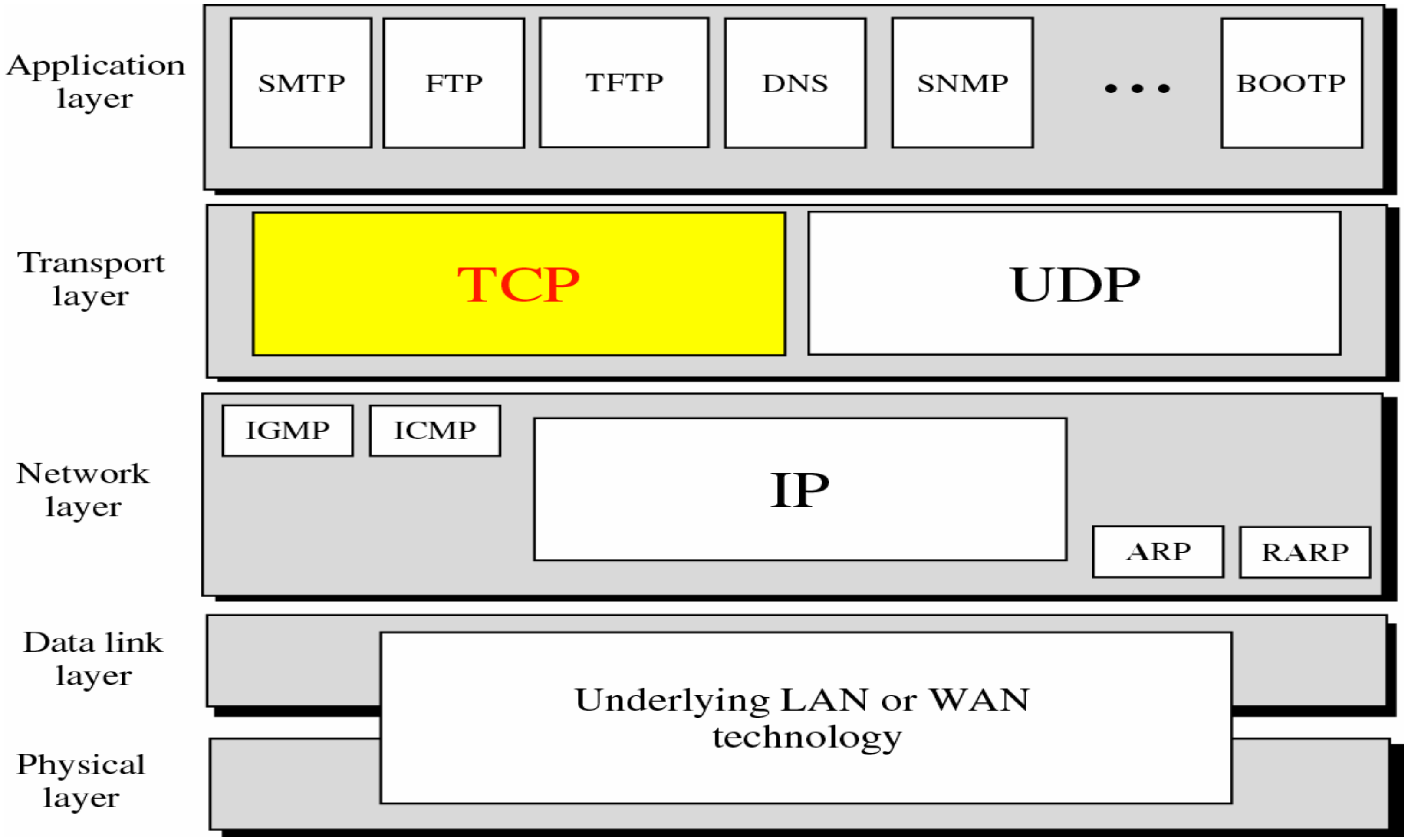
Outline

- TCP Services
- TCP Features
- Segment
- A TCP Connection
- State Transition Diagram
- Flow Control
- Error Control
- Congestion Control

Outline (Cont.)

- TCP Timers
- Options
- TCP Package

Position of TCP in TCP/IP Protocol Suite



The McGraw-Hill Companies, Inc., 2000

Introduction

- TCP
  - Like UDP, create a process-to-process (program-to-grogram) communication
    - Port numbers
  - A connection-oriented protocol
    - Create a virtual connection between two TCPs to send data
  - Add flow and error-control mechanisms at the transport layer
    - For flow control: TCP uses a *sliding window protocol*
    - For error control: TCP uses the *acknowledge packet*, *time-out*, and *retransmission* mechanisms

12.1 TCP SERVICES

We explain the services offered by TCP to the processes at the application layer.

The topics discussed in this section include:

Process-to-Process Communication
Stream Delivery Service
Full-Duplex Communication
Connection-Oriented Service
Reliable Service

TCP Services

- TCP provides services to the processes at the application layer
  - Process-to-Process Communication
  - Stream Delivery Service
  - Full-Duplex Service
  - Connection-Oriented Service
  - Reliable Service

Process-to-Process Communication

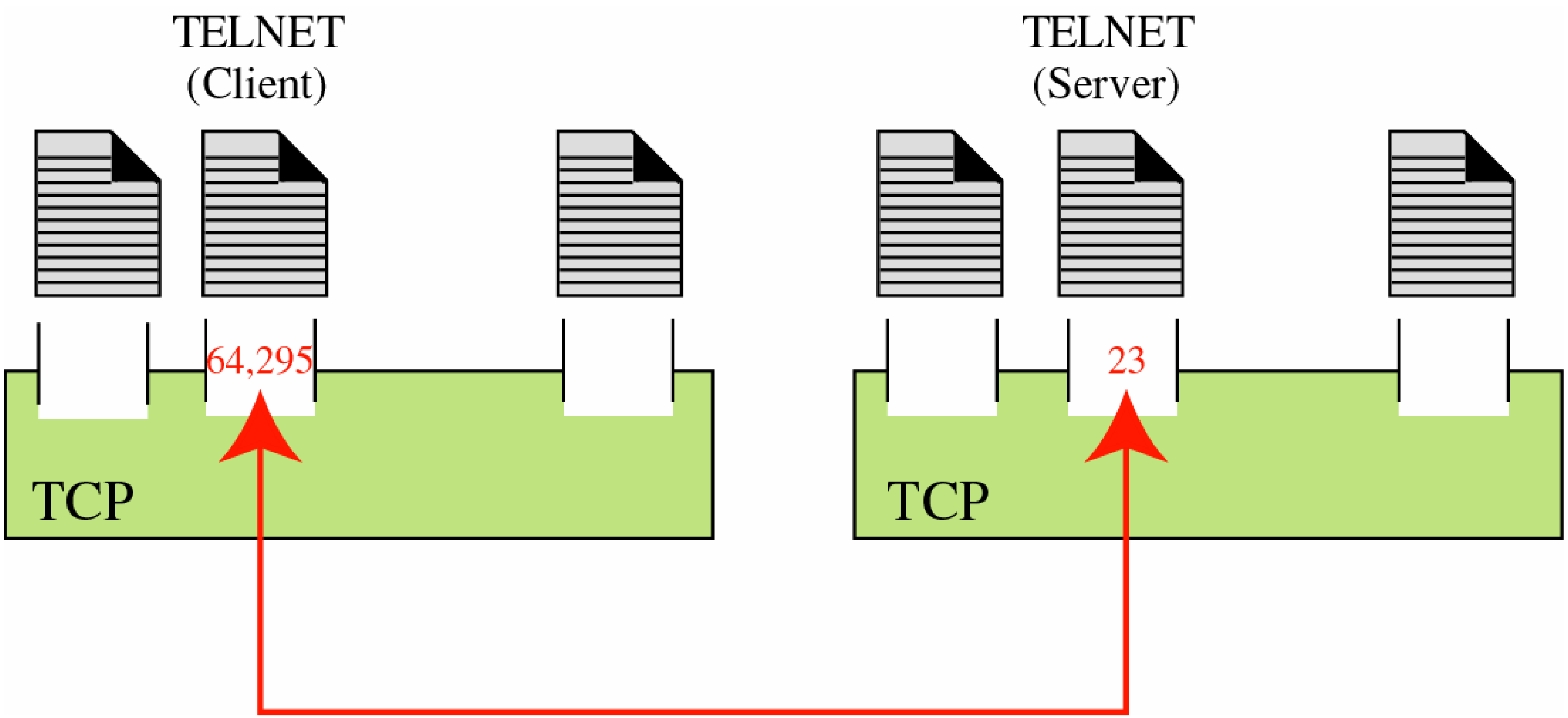
- Like UDP, TCP provides process-to-process communication using ***port numbers***

Port Number

- Client's port number
  - Chosen randomly by the TCP software running on the local host
  - Called *ephemeral port number*
- Server's port number
  - Define itself with a port number
  - Called *well-known port numbers*

Figure 12-3

Port Numbers



Well-Know Ports Used by TCP

Port	Protocol	Description
7	Echo	Echoes a received datagram back to the sender
9	Discard	Discards any datagram that is received
11	Users	Active users
13	Daytimes	Return the data and the time
17	Quote	Return a quote of the day
19	Chargen	Return a string of characters
20	FTP, Data	File Transfer Protocol (data connection)
21	FTP, Data	File Transfer Protocol (control connection)

Well-Know Ports Used by TCP

Port	Protocol	Description
23	Telnet	Terminal Network
25	SMTP	Simple Mail Transfer Protocol
53	DNS	Simple Mail Transfer Protocol
67	BOOTP	Bootstrap
79	Finger	Finger
80	HTTP	Hypertext Transfer Protocol
111	RPC	Remote Protocol Call

Example 1

As we said in Chapter 11, in UNIX, the well-known ports are stored in a file called /etc/services. Each line in this file gives the name of the server and the well-known port number. We can use the grep utility to extract the line corresponding to the desired application. The following shows the ports for FTP.

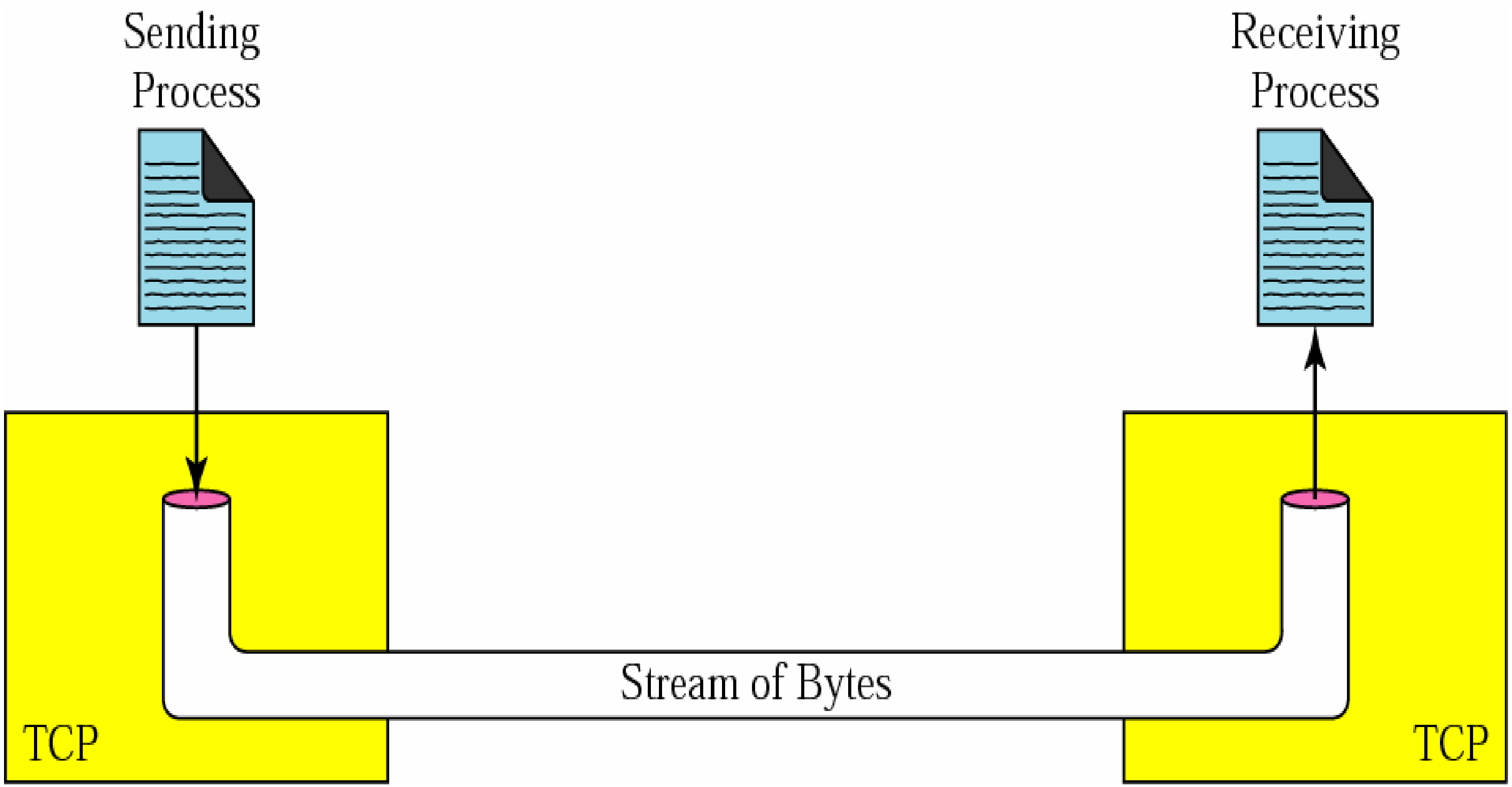
```
$ grep  ftp   /etc/services

ftp-data          20/tcp
ftp-control       21/tcp
```

Stream Delivery Service

- UDP treats each chunk independently
  - No any connection between the chucks
- In contrast, TCP allow the data be delivered/received as a stream of bytes

Stream Delivery

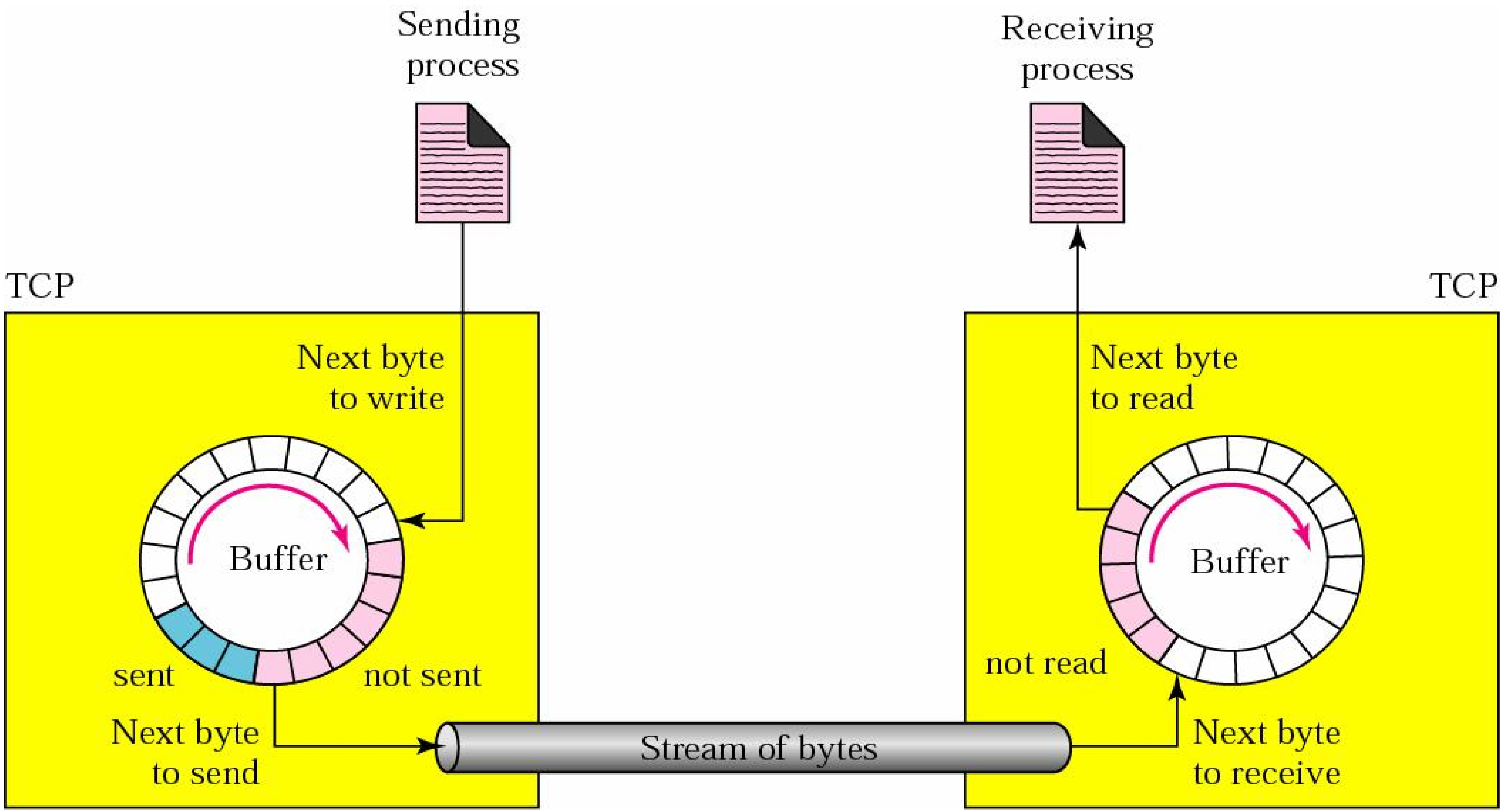


Stream Delivery Service (Cont.)

- However, the sending and receiving speed may not be the same
  - TCP needs buffers for storage

- Two buffers in TCP
  - Sending buffer and receiving buffer, one for each connection
  - Also used in flow- and error-control mechanisms

Sending and Receiving Buffers



Sending Buffers

- The sending circular buffer has ***three*** types of sections
  - White section: empty location
    - Can be filled by the sending process
  - Gray section: hold bytes that have been sent but not yet acknowledged
    - TCP keeps these bytes until it receives acknowledges
  - Color section: bytes to be sent by the sending TCP
    - TCP may be able to send only *part* of this colored section
      - The slowness of the receiving process
      - The congestion in the network

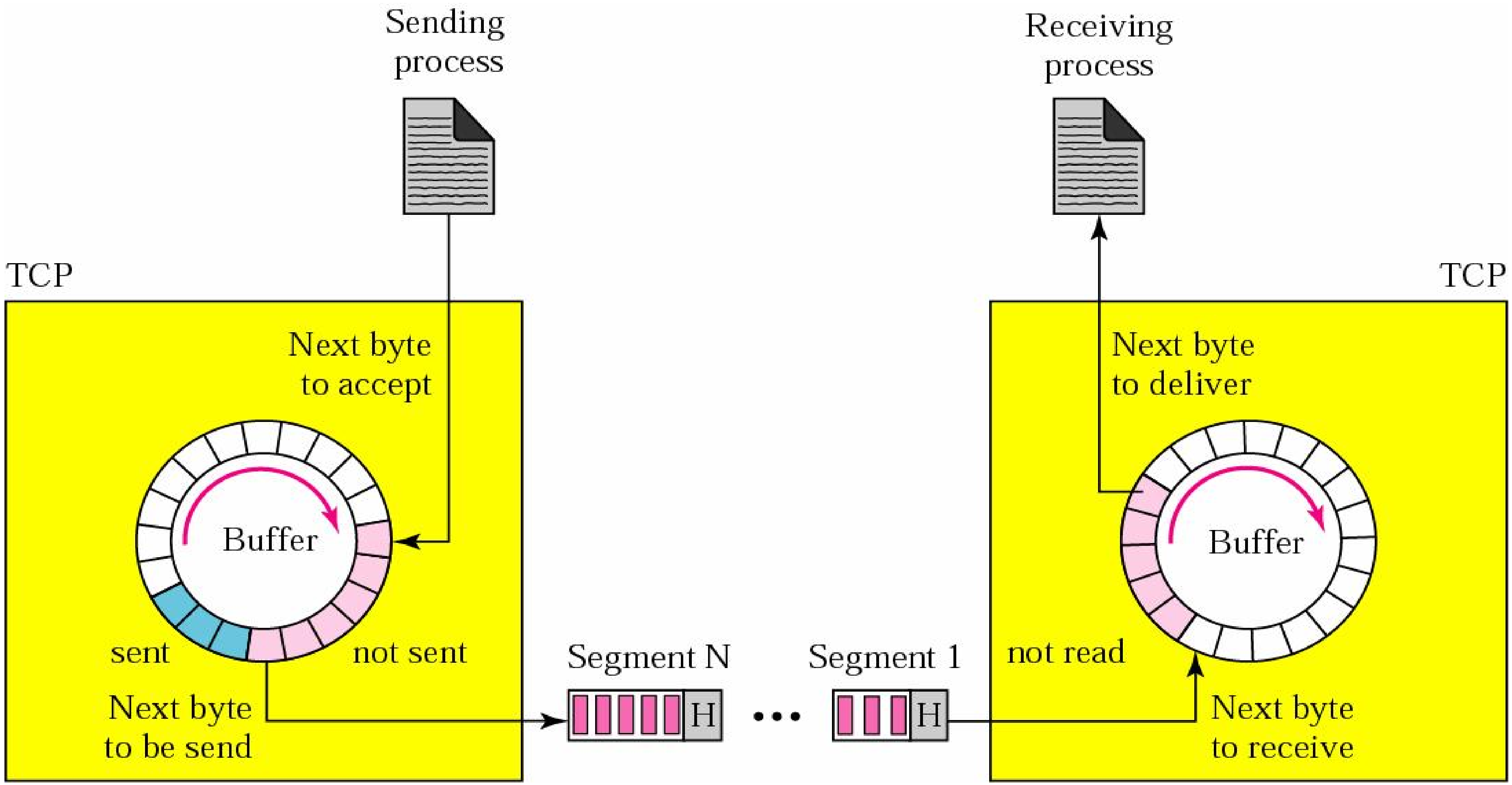
Receiving Buffer

- The receiving circular buffer is divided into ***two*** areas
  - White area:
    - Empty locations to be filled
  - Colored area:
    - Contain received bytes that can be consumed by the receiving process

Segments

- TCP groups a number of bytes together into a packet called a *segment*
  - A TCP packet is called a ***segment***
  - TCP adds a header to each segment
  - Then, the segments are encapsulated in an IP datagram
- Note: terms
  - ***UDP Datagram, TCP Segment***
  - ***IP Datagram***
  - ***MAC Frame***

TCP Segments



Full-Duplex Communication

- TCP offers full-duplex service
  - Data can flow in both directions at the same time
  - Each TCP has a sending and receiving buffer and segments are sent in both direction

Connection-Oriented Service

- TCP is a connection-oriented protocol
  - However, the connection is virtual, not a physical connection
  - Each TCP segment may use a different path to reach the destination

Reliable Service

- TCP uses an *acknowledge mechanism* to check the safe and sound arrival of data

12.2 TCP FEATURES

To provide the services mentioned in the previous section, TCP has several features that are briefly summarized in this section.

The topics discussed in this section include:

Numbering System
Flow Control
Error Control
Congestion Control

Numbering Bytes

- Although TCP use segments for transmission and reception
  - There is no field for a segment number in the segment header, i.e., TCP header

- TCP uses *sequence number* and *acknowledgement number* to keep track of the segment being transmitted or received
  - Notably, these two fields refer to the *byte number*, not the *segment number*

Byte Numbers

- TCP numbers all data bytes that are transmitted in a connection
- The numbering does not necessarily start from 0
  - *It starts randomly*
  - Between *0* and *2^32 – 1* for the number of the first byte
  - Byte numbering is used for *flow* and *error* control

Note

The bytes of data being transferred in each connection are numbered by TCP. The numbering starts with a randomly generated number.

Sequence Number

- 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

Note

The value of the sequence number field in a segment defines the number of the first data byte contained in that segment.

Example 2

Suppose a TCP connection is transferring a file of 5000 bytes. The first byte is numbered 10001.

What are the sequence numbers for each segment if data is sent in five segments, each carrying 1000 bytes?

Solution

- ***The following shows the sequence number for each segment:***

Segment 1 ➡ ***Sequence Number: 10,001 (range: 10,001 to 11,000)***

Segment 2 ➡ ***Sequence Number: 11,001 (range: 11,001 to 12,000)***

Segment 3 ➡ ***Sequence Number: 12,001 (range: 12,001 to 13,000)***

Segment 4 ➡ ***Sequence Number: 13,001 (range: 13,001 to 14,000)***

Segment 5 ➡ ***Sequence Number: 14,001 (range: 14,001 to 15,000)***

Example

- Imagine a TCP connection is transferring a file of 6000 bytes.
  - The first byte is numbered 10010

- What are the sequence numbers for each segment if data is sent in five segments with
  - The first four segments carrying 1,000 bytes
  - The last segment carrying 2,000 bytes?

Solution

The following shows the sequence number for each segment:

Segment 1 → 10,010 (10,010 to 11,009)

Segment 2 → 11,010 (11,010 to 12,009)

Segment 3 → 12,010 (12,010 to 13,009)

Segment 4 → 13,010 (13,010 to 14,009)

Segment 5 → 14,010 (14,010 to 16,009)

Acknowledgment Number

- Communication in TCP is full duplex
  - Both parties can send and receive data at the same time in a connection
- Each party numbers the bytes, usually with a different starting byte number
  - *Sequence number*: the number of the first byte carried by the segment
  - *Acknowledgment number*: the number of the next byte that *the party expects to receive*

Acknowledgment Number (Cont.)

- Acknowledgment number is *cumulative*
- For example, if a party uses 5,643 as an acknowledgment number
  - It has received all bytes from the beginning up to 5,642
  - Note that, *this does not mean that the party has received 5642 bytes*
    - The first byte number does not have to start from 0

Note

The value of the acknowledgment field in a segment defines the number of the next byte a party expects to receives. The acknowledgment number is cumulative.

Flow Control

- The receiver controls how much data are to be sent by the sender
  - Prevent the receiver from being overwhelmed with data

- The numbering system allow TCP to use a ***byte-oriented flow control***

Error Control

- TCP implements an error control mechanism
  - To provide reliable service
  - Also byte-oriented

Congestion Control

- TCP takes into account congestion in the network

- Thus, the amount of data sent by a sender is controlled both by
  - *The receiver (flow control)*
  - *The level of congestion in the network*

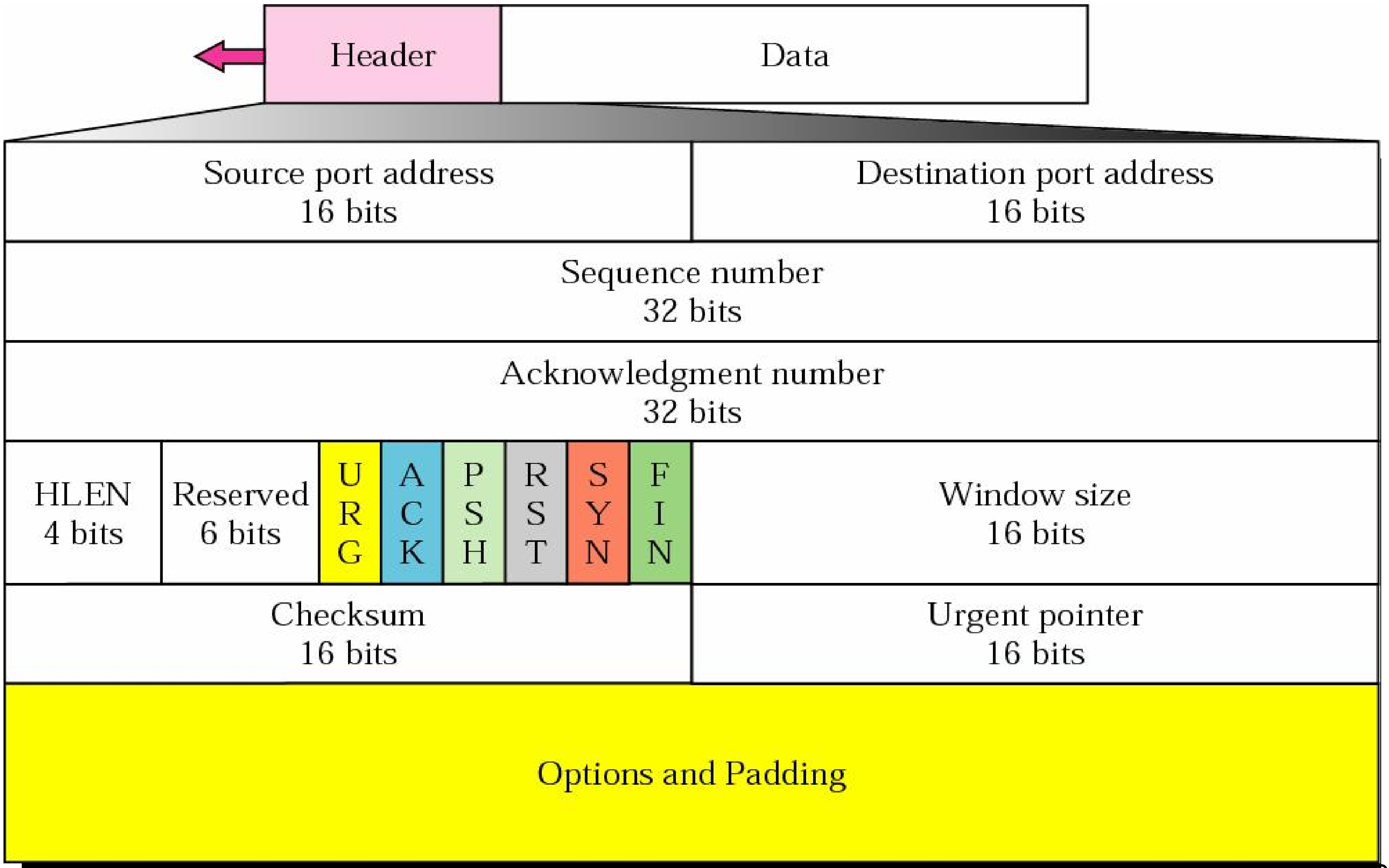
12.3 SEGMENT

A packet in TCP is called a segment

The topics discussed in this section include:

Format
Encapsulation

TCP Segment Format



TCP Segment Format

- Source port address: 16-bit
- Destination port address: 16-bit
- Sequence number: 32-bit
  - The first byte number in this segment
  - In connection establishment, each party randomly generate an ***initial sequence number (ISN)***
    - Usually different in each direction
- Acknowledgment number: 32-bit
  - The byte number that the receiver expects
    - If received byte number x, ack. number is x+1
  - *Acknowledgment and data can be piggybacked together*

TCP Segment Format (Cont.)

- Header length: 4-bit
  - The number of *4-byte words* in the TCP header
  - Value of this field is between 5 and 15
    - TCP header is between *20-60* bytes
- Reserved: 6-bit
    - Reserved for future use
- Control: 6-bits

Figure 12-20

Control Field

URG: Urgent pointer is valid
ACK: Acknowledgment is valid
PSH: Request for push
RST: Reset the connection
SYN: Synchronize sequence numbers
FIN: Terminate the connection

URG	ACK	PSH	RST	SYN	FIN

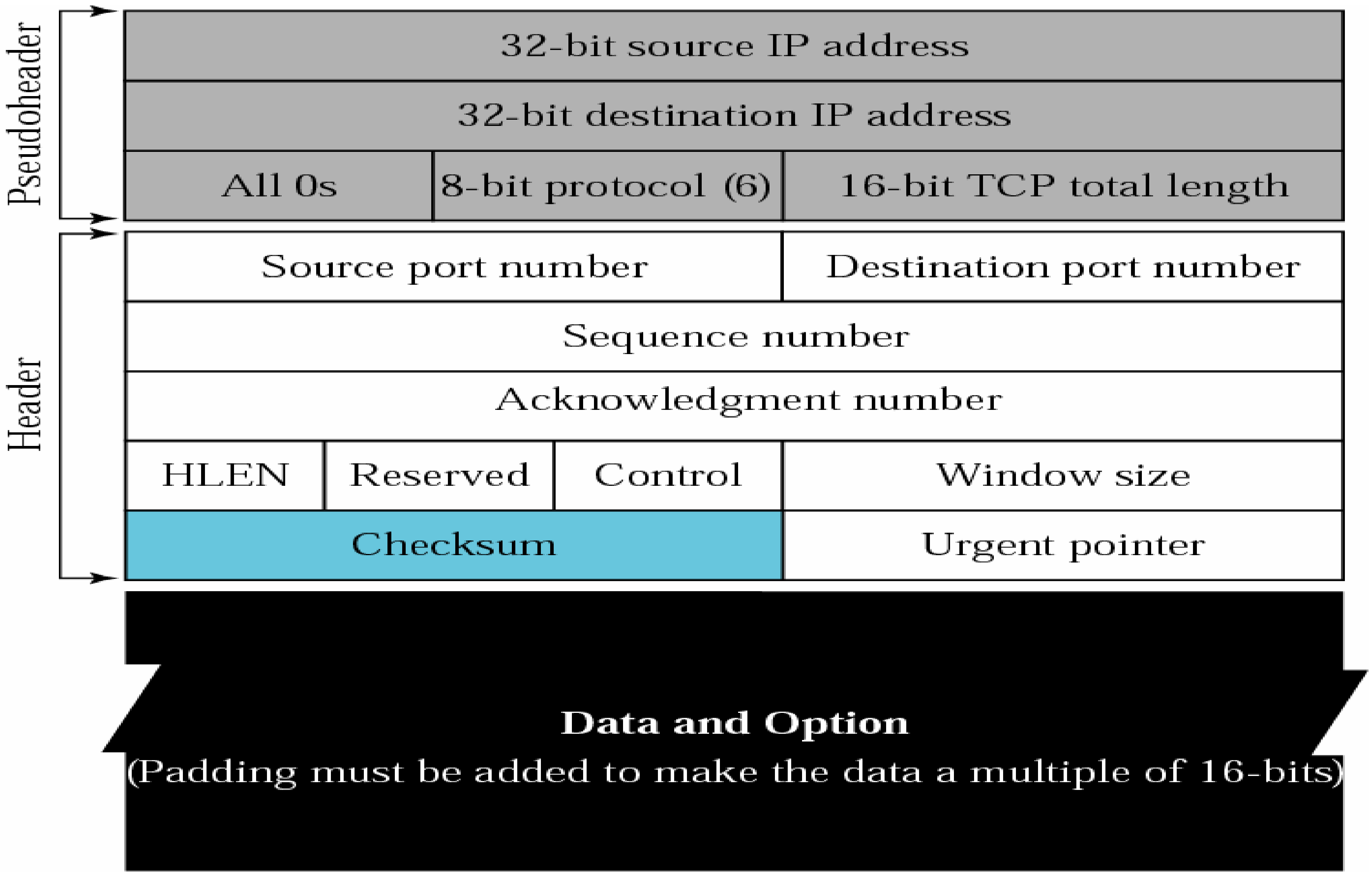
TCP Segment Format (Cont.)

- Control
  - URG: urgent pointer is valid
  - ACK: acknowledgment field is valid
  - PSH: push the data
  - RST: the connection must be reset
  - SYN: Synchronization sequence numbers during connection
  - FIN: terminate the connection

TCP Segment Format (Cont.)

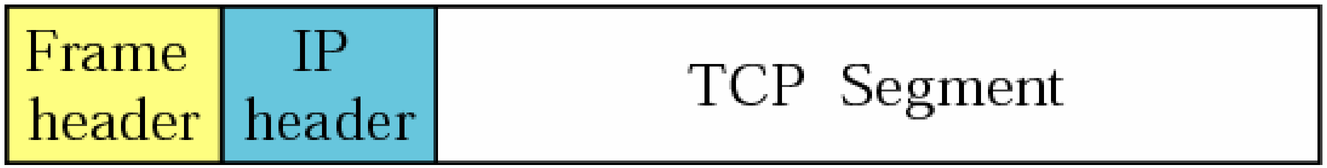
- Window size: 16-bit
  - Define the size of the receiving window, in bytes
    - Determined by the receiver
  - The maximum window size is 2^16 = 65535
- Checksum: 16-bit
  - Follow the same procedure as UDP
  - Checksum for TCP is mandatory (UDP is optional)
- Urgent pointer: 16-bit
  - Valid only if the urgent bit is set
  - Used when the segment contains urgent data
- Options: 0~40 bytes

Pseudoheader added to the TCP datagram



Encapsulation

- A TCP segment is encapsulated in an IP datagram
  - Which in turn is encapsulated in a data-link frame



12.4 A TCP CONNECTION

TCP is connection-oriented. A connection-oriented transport protocol establishes a virtual path between the source and destination. All of the segments belonging to a message are then sent over this virtual path. A connection-oriented transmission requires three phases: connection establishment, data transfer, and connection termination.

The topics discussed in this section include:

Connection Establishment
Data Transfer
Connection Termination
Connection Reset

Introduction

- TCP's connection-oriented transmission requires three phases
  - ***Connection establishment***
    - Three-way handshaking
  - ***Data transfer***
  - ***Connection termination***
    - Four-way handshaking

Connection Establishment

- Four actions are taken between host A and B
  - Host A sends a segment to announce its wish for connection and includes its initialization information
  - Host B sends a segment to acknowledge the request of A
  - Host B sends a segment that includes its initialization information
  - Host A sends a segment to acknowledge the request of B
- However
  - Step 2 and 3 can be combined into one step

Connection Establishment

- Example, a client wants to make a connection to a server
  - Server performs the *passive open*
    - Tell TCP that it is ready to accept a connection
  - Client performs the *active open*
    - Tell TCP that it needs to be connected to the server

Three-way handshaking

1. The client sends the first segment, a *SYN* segment
   - Set the *SYN* flag
   - The segment is used for synchronization of sequence number
     - *Initialization sequence number (ISN)*
   - If client wants to define MSS, add MSS option
   - If client needs a larger window
     - Define the window scale factor option
   - Does not contain any acknowledgment number
   - Does not define the window size either
     - A window size makes sense only when a segment includes an acknowledgment
   - Although a control segment and does not carry data
     - But consumes *one sequence number*

Note:

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

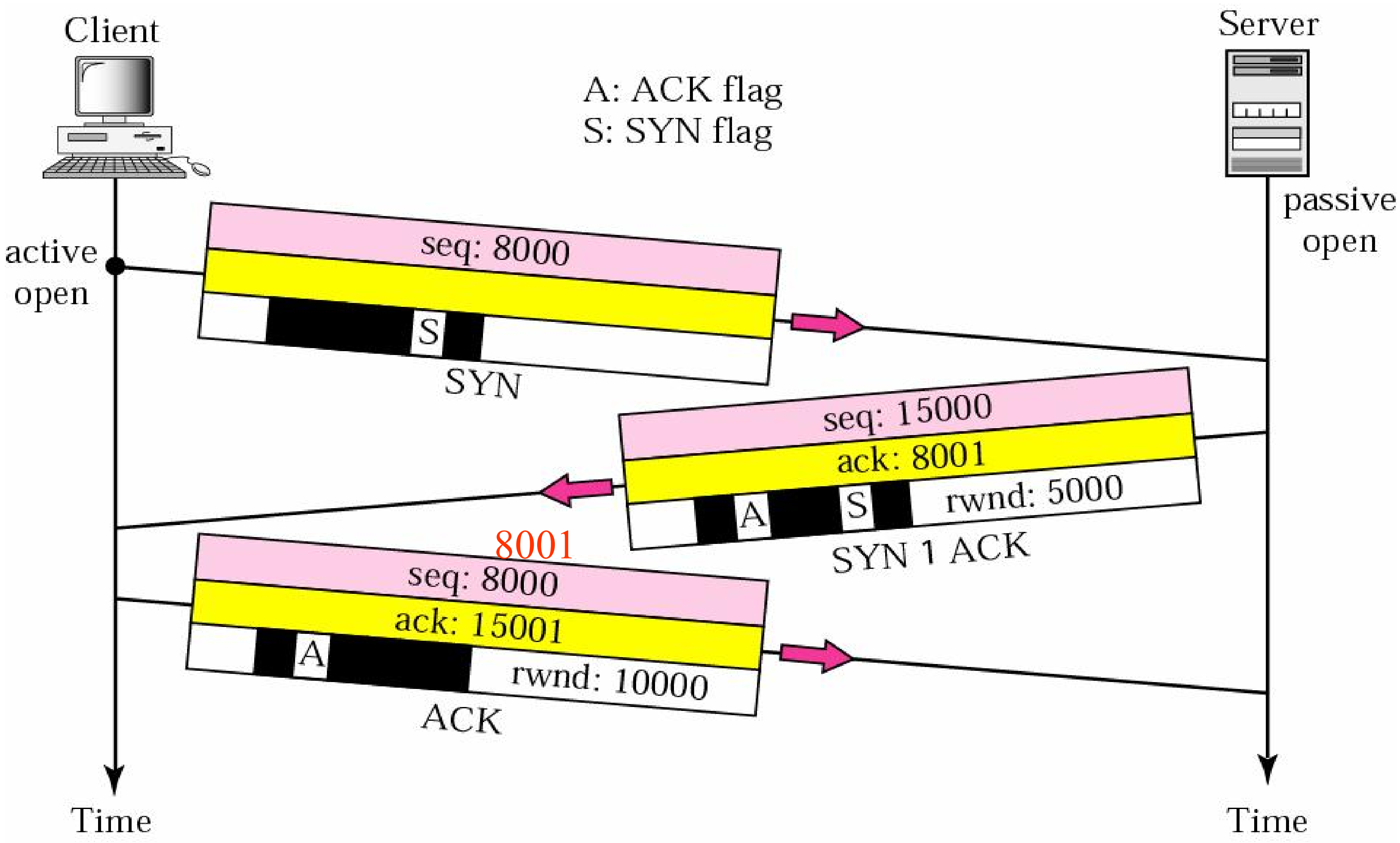
Three-way handshaking (Cont.)

2. The server sends a second segment, a ***SYN + ACK*** segment
   - Set the *SYN* and *ACK* flag
   - *Acknowledge* the receipt of the first segment using the ACK flag and acknowledgment number field
     - Acknowledgment number = client initialization sequence number + 1
     - Must also define the receiver window size for flow control
   - *SYN* information for the server
     - Initialization sequence number from server to client
     - Window scale factor if used
     - MSS is defined

Note:

A SYN + ACK segment cannot carry data, but does consume one sequence number.

Three-way Handshaking



Three-way handshaking (Cont.)

3. The client sends the third segment, ***ACK*** segment
   - *Acknowledge* the receipt of second segment
     - ACK flag is set
       - Acknowledgement number = server initialization sequence number + 1
     - Must also define the server window size
       - Set the window size field
     - The sequence number is the same as the one in the SYN segment
       - ACK segment does not consume any sequence number
   - However, in some implementation, data can be sent with the third packet
     - Must have a new sequence number showing the byte number of the first byte in the data

Note:

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

Connection Establishment (Cont.)

- Active open
  - The side that sends the first SYN
- Passive open
  - The side that receives this SYN and sends the next SYN
- Simultaneous open
  - Both processes issue an *active open*
    - But only a single connection is established (discussed later)

SYN Flooding Attack

- A malicious attacker sends a larger number of SYN segment to a server
  - Each with a fading source IP address
- The server will runs out of resource and may crash
  - *Denial of service attack*

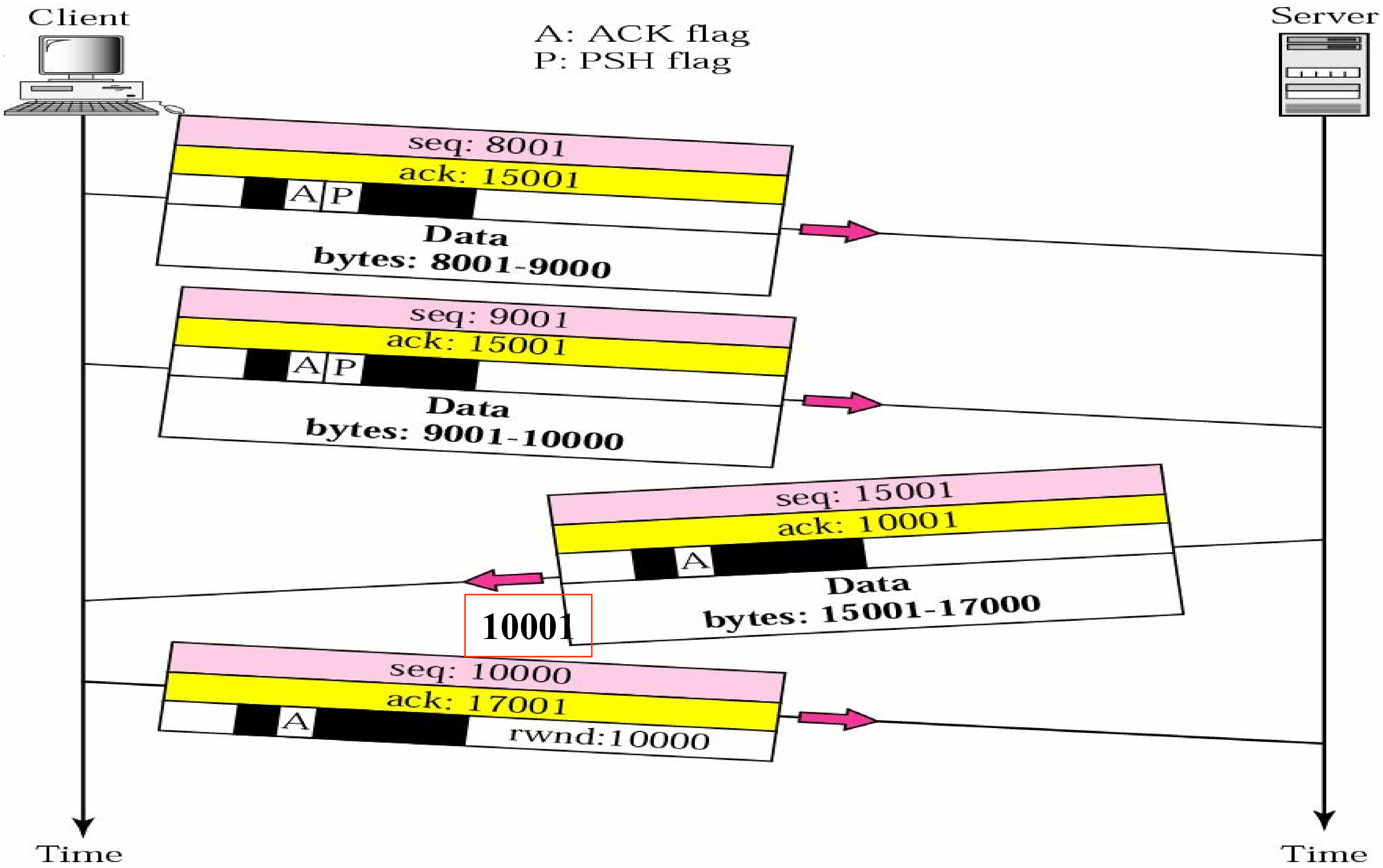
SYN Flooding Attack (Cont.)

- Possible solutions
  - Impose a limit of connections requested during a period of time
  - Filter out datagrams coming from unwanted source addresses
  - Postpone resource allocation until the entire connection is set up
    - SCTP uses strategy, called ***cookie***

Data Transfer

- ***Bidirectional*** data transfer takes place after connection is established
  - Both parties can send data and acknowledgments in both direction
  - The acknowledgment can be piggybacked with the data

Example: a Data Transfer



Pushing Data

- In TCP, both sender and receiver have buffers to hold data
  - In sender, application data to be sent is temporary hold in the buffer
  - In receiver, receiving data is temporary hold in the buffer
  - Thus, for applications, they may encounter delayed transmission and reception

Pushing Data (Cont.)

- In some cases, *delayed transmission and reception* may not be acceptable
- TCP thus support ***PUSH*** operation
  - Sending TCP must create a segment and send the data immediately
    - Must not wait for the window to be filled
  - Receiving TCP must deliver data to the application immediately
    - Does not wait for more data to come

Urgent Data

- TCP is a stream-oriented protocol
  - Data is presented as a stream of bytes

- In some cases, an application needs to send *urgent* data
  - Sender wants a piece of data to be read our of order by the receiving application

Urgent Data (Cont.)

- Solution: send a segment with ***URG*** bit set
  - Sender creates a segment, insert the urgent data at the beginning of the segment and sends the segment with the URG bit set
  - The *urgent pointer* field defines the end of the urgent data and the start of normal data

Connection Termination

- Two options
  - Three-way handshaking
  - Four-way handshaking with a half-close option

Three-Way Handshaking

1. Client TCP sends the ***FIN segment***
   - ***FIN*** flag is set
   - Two choices
     - FIN segment is only a control segment
       - Consume only one sequence number
     - FIN segment can include the last chunk of data sent by the client

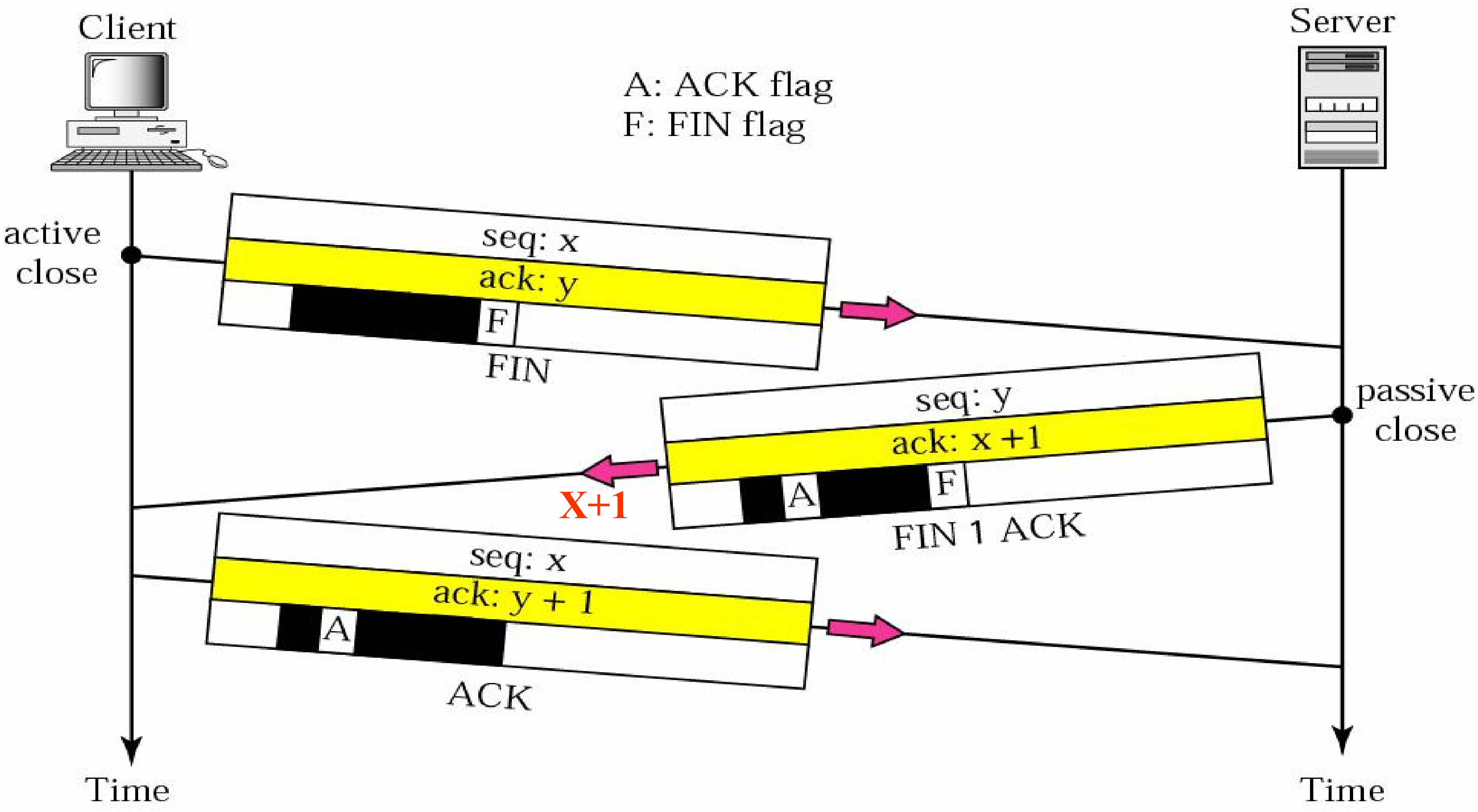
Three-Way Handshaking (Cont.)

2. The server TCP sends the ***FIN+ACK segment***
   - ***ACK*** bit is set
     - Confirm the receipt of FIN segment
   - ***FIN*** bit is set
     - Announce the closing of the connection in the other direction
   - Two choices
     - FIN+ACK segment is only a control segment
       - Consume only one sequence number
     - FIN +ACK segment can include the last chunk of data sent by the client

Three-Way Handshaking (Cont.)

- Client TCP sends the last ***ACK segment***
  - ***ACK*** bit is set
    - Confirm the receipt of the FIN+ACK segment for the TCP server
  - This segment cannot carry data and consume no sequence number
    - No further response!

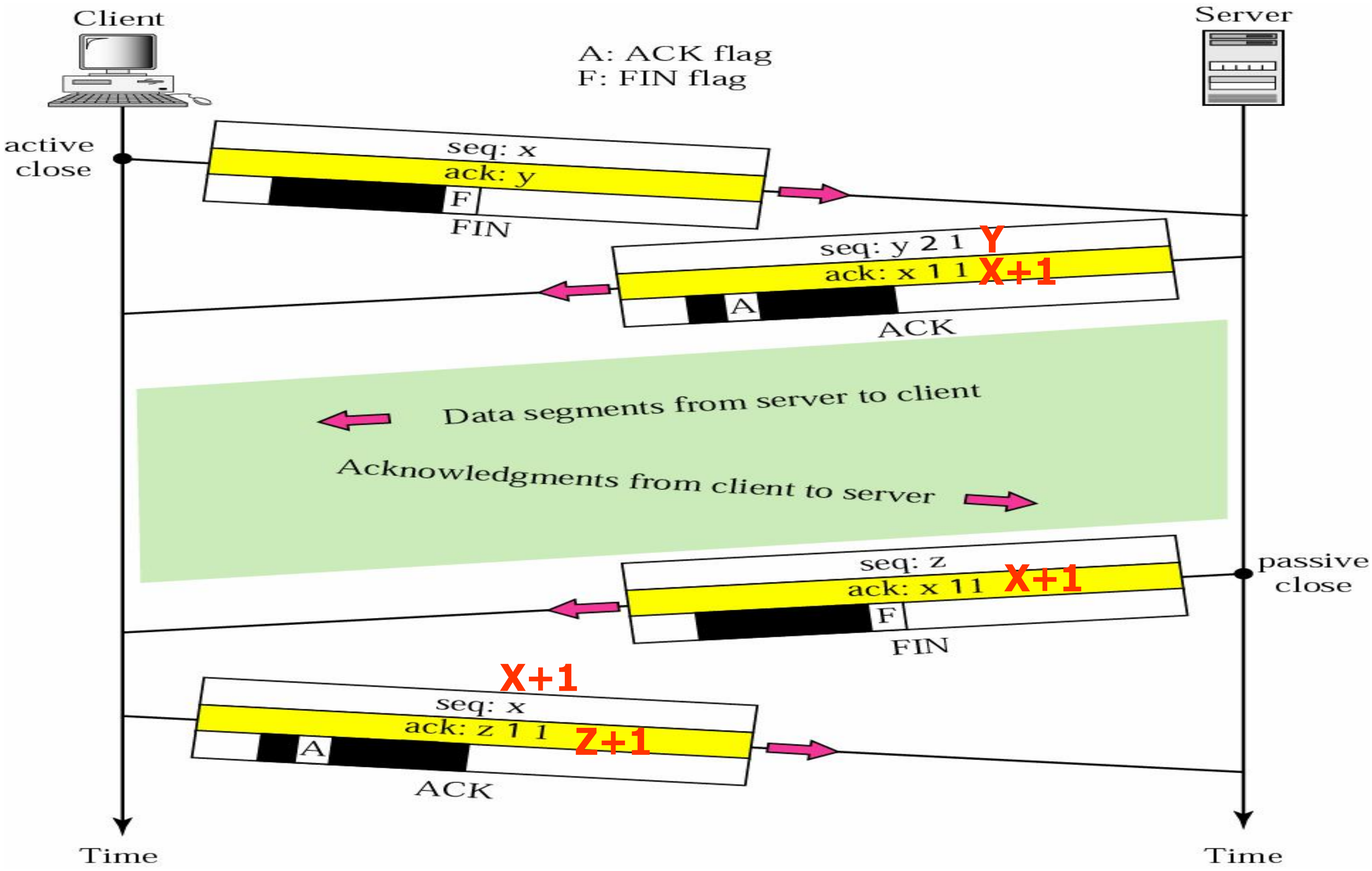
Three-Way Handshaking



Four-Way Handshaking with Half-Close

- Host A sends a ***FIN segment*** announcing its wish for connection termination
- Host B sends a ***ACK segment*** acknowledging the FIN segment from A
  - The connection is closed in one direction
  - *But host B can continue sending data to A*
- Host B sends a ***FIN segment*** to close the connection
- Host A sends a ***ACK segment*** to acknowledges the FIN segment from B

Half-Close



Client
Server
A: ACK flag
F: FIN flag
active close
seq: x
ack: y
F
FIN
seq: y 2 1
Y
ack: x 1 1
X+1
A
ACK
Data segments from server to client
Acknowledgments from client to server
passive close
seq: z
ack: x 1 1
X+1
F
FIN
X+1
seq: x
ack: z 1 1
Z+1
A
ACK
Time
Time

Connection Reset

- The TCP at one end may
  - Deny a connection request
  - Abort a connection
  - Terminate an idle connection
- How to achieve ?
  - By the ***RST (reset) flag***

Denying a Connection

- Example
  - A TCP segment is received and requested a connection to a nonexistent port
  - The receiving TCP sends a segment with the RST bit set

Aborting a Connection

- A process may want to abort a connection instead of closing it normally
  - Example, the process does not want the data in the queue to be sent
    - If closed normally, the data will be sent
- TCP may also want to abort the connection
  - Example, it receives a segment belonging to the previous connection
    - This connection uses the same source and destination port address as previous connection

Terminate an Idle Connection

- TCP on one side may discover that the TCP on the other side has been idle for a long time

- Send an RST segment to destroy the connection

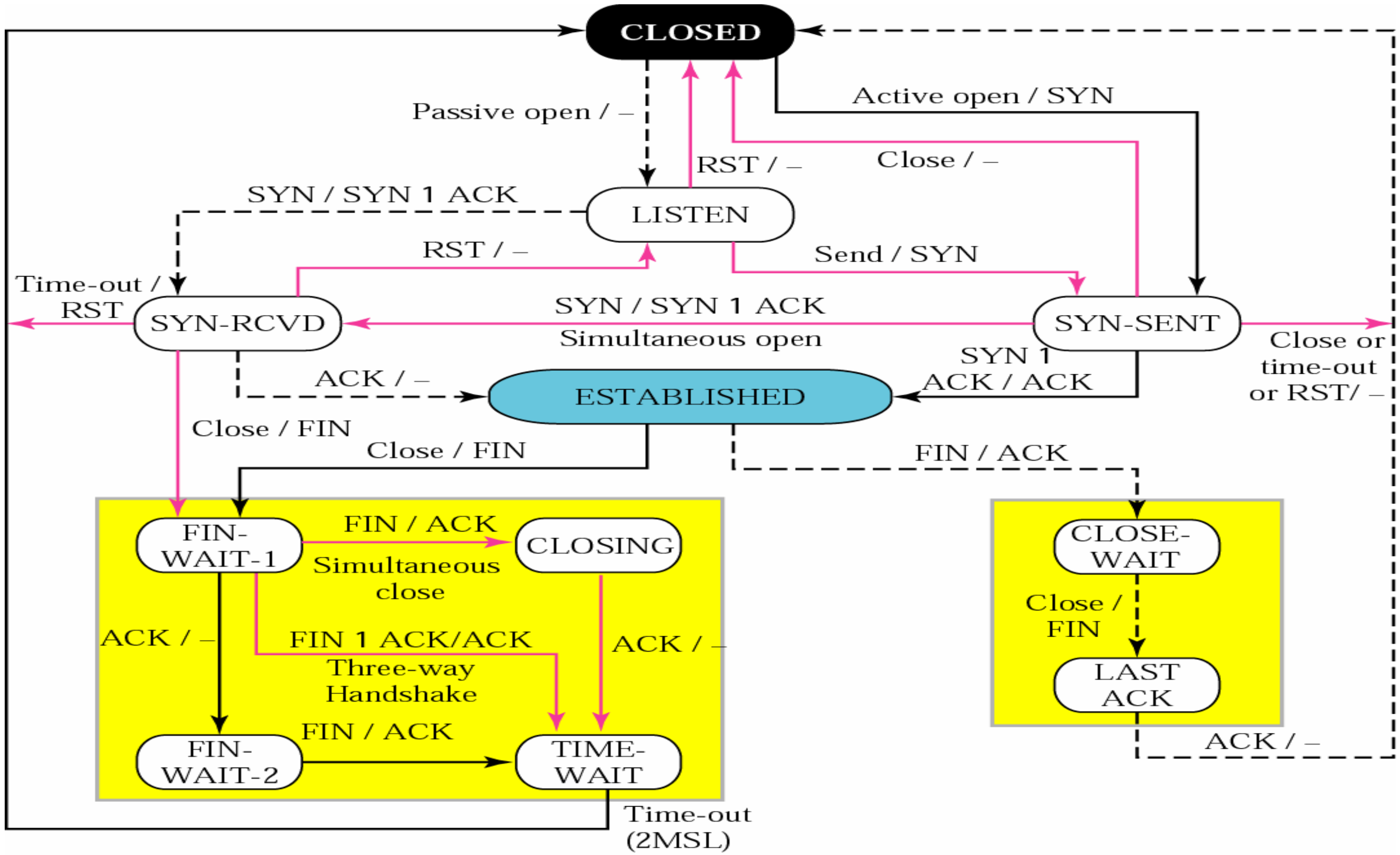
12.5 STATE TRANSITION DIAGRAM

To keep track of all the different events happening during connection establishment, connection termination, and data transfer, the TCP software is implemented as a finite state machine. .

The topics discussed in this section include:

Scenarios

State Transition Diagram



12.6 FLOW CONTROL

Flow control regulates the amount of data a source can send before receiving an acknowledgment from the destination. TCP defines a window that is imposed on the buffer of data delivered from the application program.

The topics discussed in this section include:

Sliding Window Protocol
Silly Window Syndrome

Flow Control

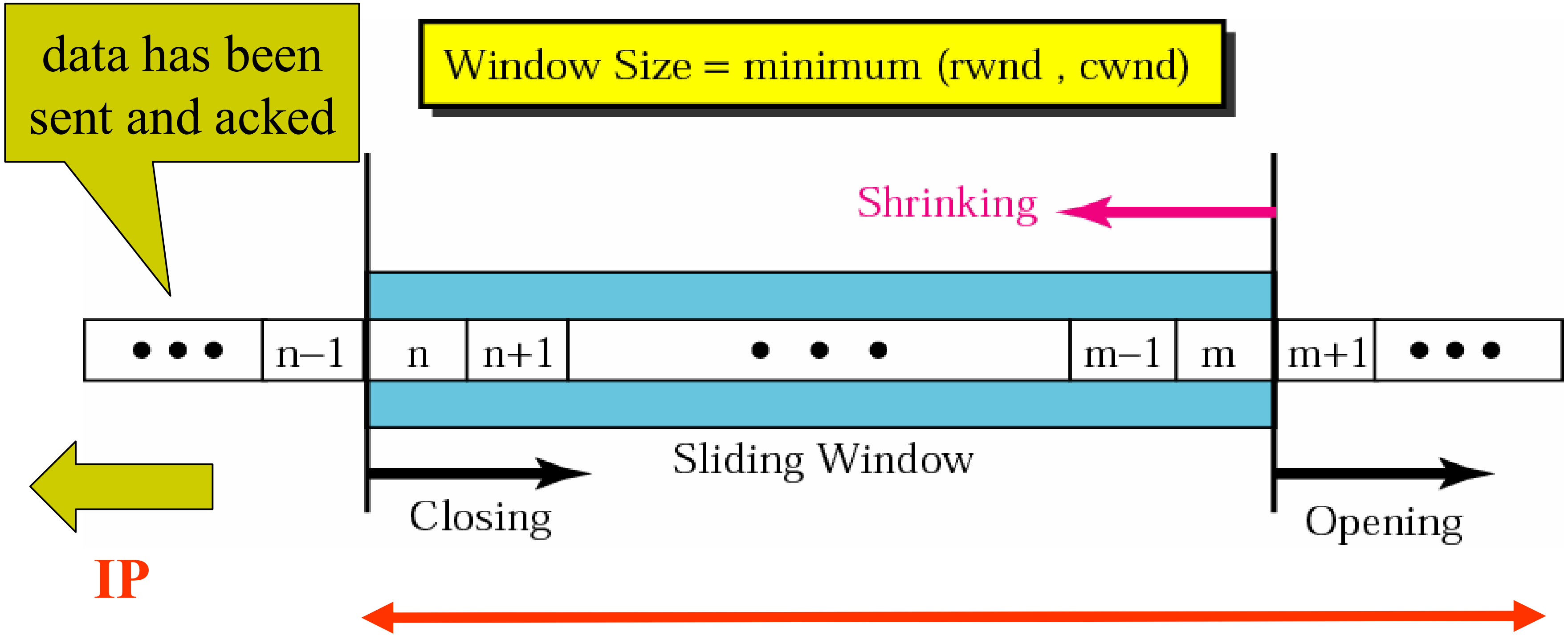
- Regulate the amount of data a source can send before receiving an acknowledgment
- Two extreme cases
  - Send 1 byte of data and wait for an acknowledge
    - Source would be idle
  - Send all of the data without worrying about acknowledge
    - May overwhelm the receiver buffer
    - Inefficient if some part of data is *lost*, *duplicated*, *received our of order* or *corrupted*
- Solution: the *sliding window protocol* by TCP

Sliding Window Protocol

- Both hosts use a window for each connection
  - Containing bytes that a host can send before worrying about an acknowledgment
- Called *sliding windows*
  - The window can slide over the buffer

- *TCP's sliding windows are byte oriented*

Sliding Window



Sliding Window Protocol (Cont.)

- The window is *opened*, *closed*, or *shrunk*
  - All in the control of the *receiver* and depend on congestion in the *network*
- ***Opening*** a window
  - Moving the right wall to the right
  - Allow more bytes are eligible for sending
- ***Closing*** a window
  - Moving the left wall to the right
    - Some bytes have been acknowledged
    - The sender needs not worry about them anymore

Sliding Window Protocol (Cont.)

- ***Shrinking*** the window
  - Moving the right wall to the left
  - Revoking the eligibility of some bytes for sending
    - Application has sent it to the TCP buffer but later wants to cannel its transmission
    - Strongly discouraged and not allowed in some implementation

Sliding Window Protocol (Cont.)

- The size of the window at one end is determined by the minimum of two values
  - ***Receiver window (rwnd)***
    - Advertised by the opposite end in a segment containing acknowledgement
  - ***Congestion window (cwnd)***
    - Determined by the network to avoid congestion

Example 3

- Suppose that Receiver B:
  - Buffer size = 5000
  - Receive 1000 bytes unprocessed data
  - What is the value of the receiver window (rwnd) for sender A?
- *Solution*
  - The value of rwnd = 5,000 − 1,000 = 4,000.
  - Host B can receive only 4,000 bytes of data before overflowing its buffer.
  - Host B advertises this value in its next segment to A.

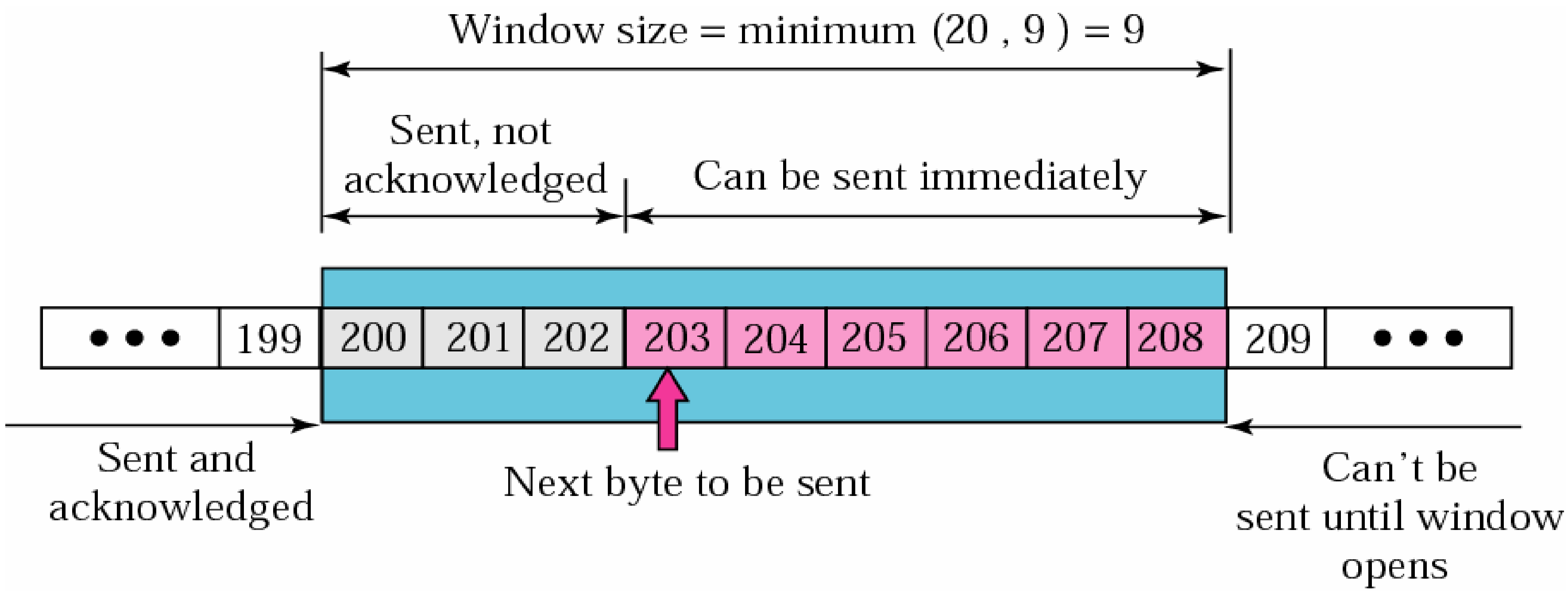
Example 4

- Suppose sender A:
  - rwnd = 3000 bytes
  - cwnd = 3500 bytes
  - What is the size of the window for host A?
- *Solution*
  - The size of the window is the smaller of rwnd and cwnd
  - Ans: 3,000 bytes

Example 5: Figure 12.21 shows an unrealistic example of a sliding window

- The sender has sent bytes up to 202.
- cwnd = 20 (in reality this value is thousands of bytes).
- The receiver has sent a ACK segment
  - Acknowledgment number = 200
  - rwnd = 9 bytes (in reality this value is thousands of bytes).
- Therefore
  - The size of the sender window
    - The minimum of rwnd and cwnd or 9 bytes.
  - Bytes 200 to 202 are sent, but not acknowledged.
  - Bytes 203 to 208 can be sent without worrying about acknowledgment.
  - Bytes 209 and above cannot be sent.

Figure 12.21


Window size = minimum (20 , 9) = 9
Sent, not acknowledged
Can be sent immediately
• • •
199
200
201
202
203
204
205
206
207
208
209
• • •
Sent and acknowledged
Next byte to be sent
Can't be sent until window opens

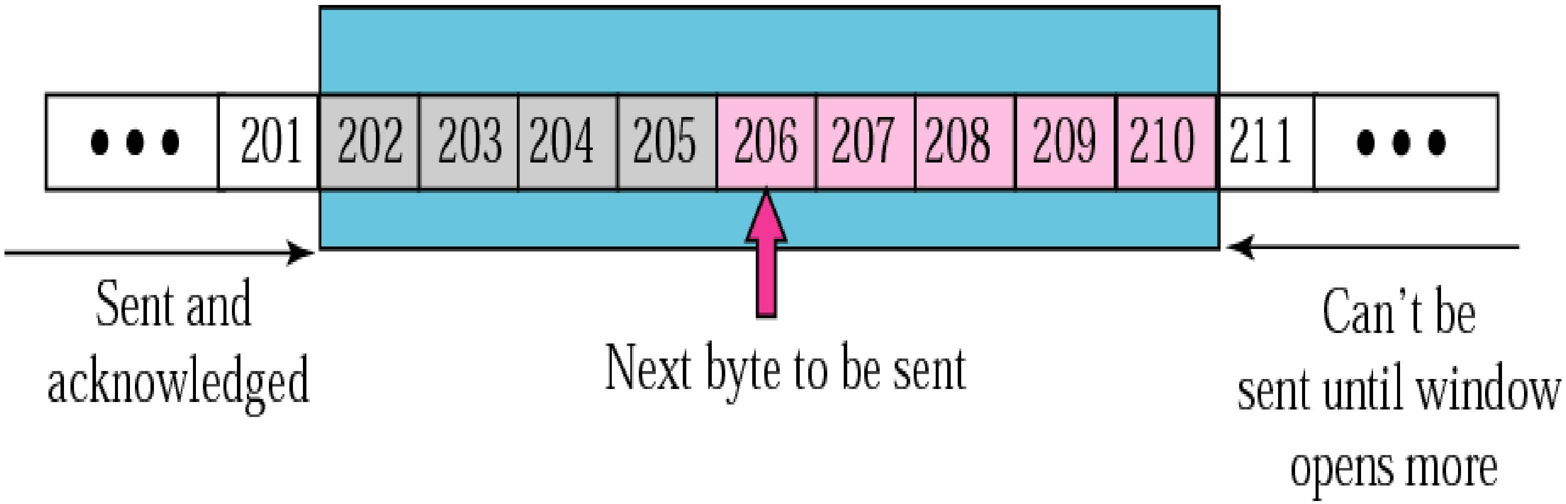
Example 6

- In Figure 12.21
  - Server receives a packet
    - Acknowledgment value = 202
    - rwnd = 9.
  - The host has already sent bytes 203, 204, and 205.
  - The value of cwnd is still 20.
  - Show the new window.

Example 6 (Cont.)

- Solution
  - Figure 12.22 shows the new window.
  - Note that this is a case in which the window
    - Closes from the left and opens from the right by an equal number of bytes
    - The size of the window has not been changed.
  - The acknowledgment value, 202, declares that bytes 200 and 201 have been received

Figure 12.22



• • •
201
202
203
204
205
206
207
208
209
210
211
• • •
Sent and acknowledged
Next byte to be sent
Can't be sent until window opens more

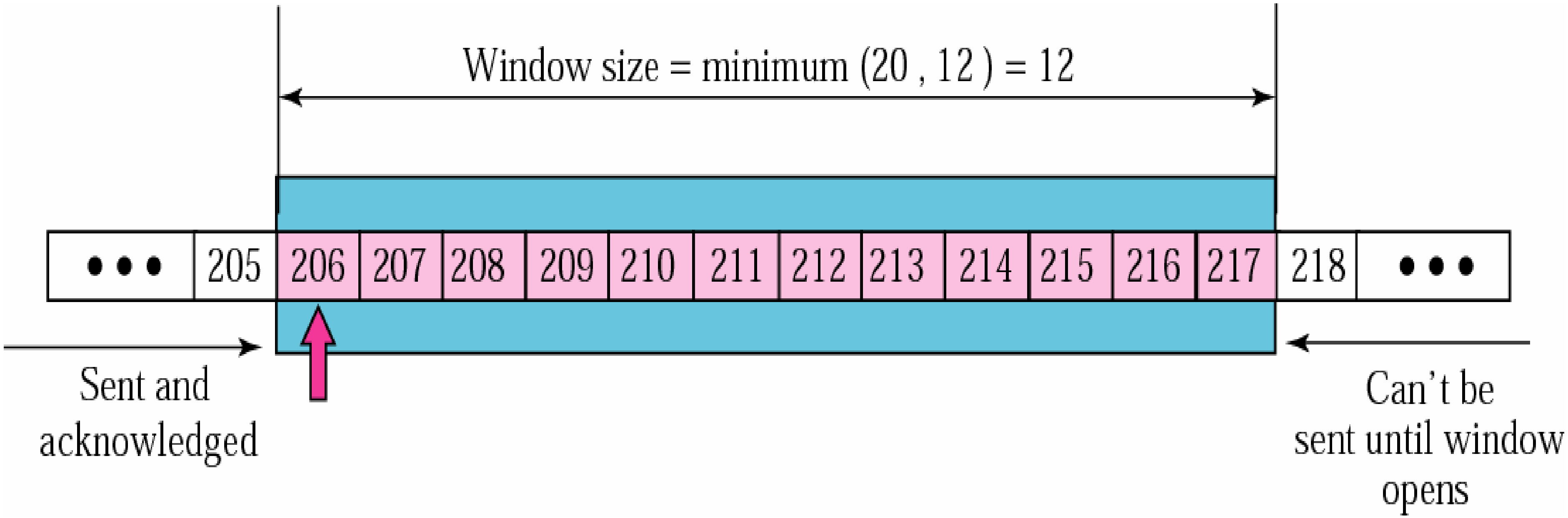
Example 7

- In Figure 12.22
  - Sender receives a packet
    - Acknowledgment value = 206
    - rwnd = 12.
  - The host has not sent any new bytes.
  - The value of cwnd is still 20.
  - Show the new window.

Example 7 (Cont.)

- Solution
  - rwnd < cwnd
    - The size of the window is 12.
  - Figure 12.23 shows the new window.
    - The window has been opened from the right by 7 and closed from the left by 4
    - The size of the window has increased.

Figure 12.23



Window size = minimum (20 , 12) = 12
• • •
205
206
207
208
209
210
211
212
213
214
215
216
217
218
• • •
Sent and acknowledged
Can't be sent until window opens

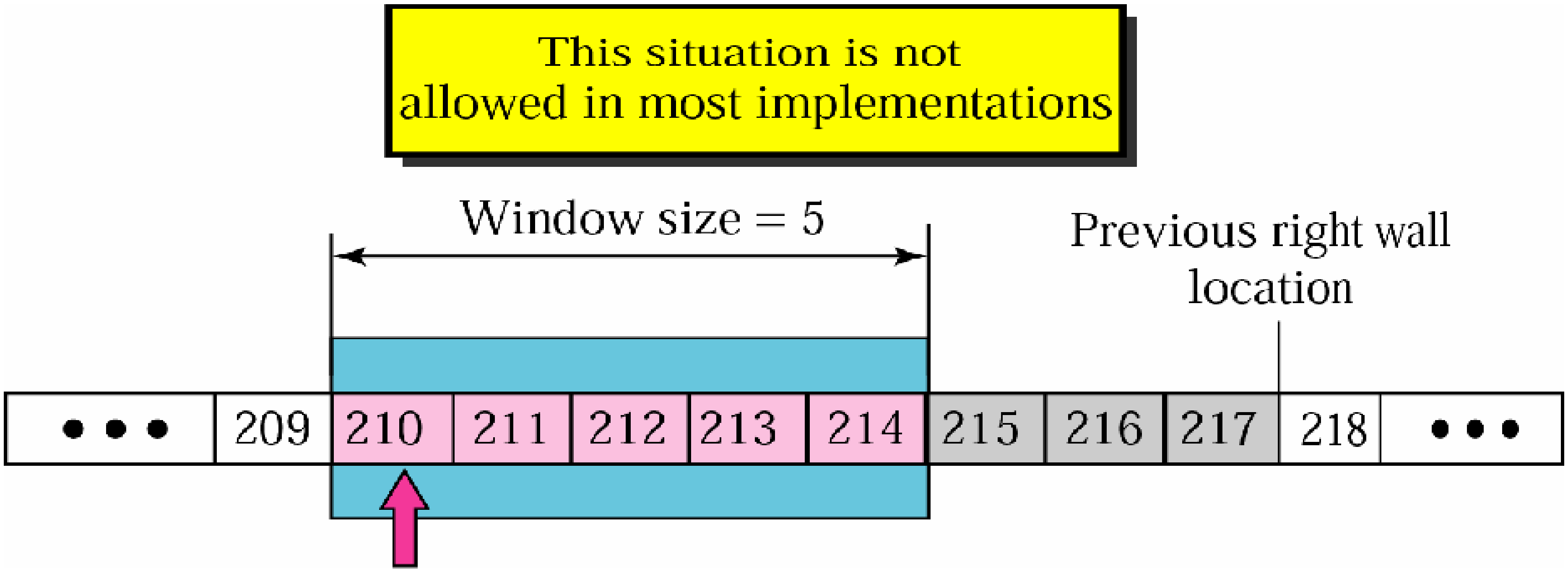
Example 8

- In Figure 12.23
  - The host receives a packet
    - Acknowledgment value = 210
    - rwnd = 5.
  - The host has sent bytes 206, 207, 208, and 209.
  - The value of cwnd is still 20.
  - Show the new window.

Example 8 (Cont.)

- Solution:
  - rwnd < cwnd,
    - The size of the window is 5.
  - Figure 12.24 shows the situation.
  - **Note that this is a case not allowed by most implementations.**

Figure 12.24


This situation is not allowed in most implementations
Window size = 5
Previous right wall location
209
210
211
212
213
214
215
216
217
218

Example 9

How can the receiver avoid shrinking the window in the previous example?

Solution

$$new\ ack + new\ rwnd \geq last\ ack + last\ rwnd$$

or

$$new\ rwnd \geq (last\ ack + last\ rwnd) - new\ ack$$

Example 9

- How can the receiver avoid shrinking the window in the example 8?

Example 9 (Cont.)

- Solution:
  - The **receiver** needs to keep track of the **last acknowledgment number** and **the last rwnd**.
    - Right wall = **acknowledgment number** + **rwnd**
  - To prevent shrinking, we must always have the following relationship.

$$\textit{new ack} + \textit{new rwnd} \geq \textit{last ack} + \textit{last rwnd}$$

or

$$\textit{new rwnd} \geq (\textit{last ack} + \textit{last rwnd}) - \textit{new ack}$$

Example 9 (Cont.)

- In example 8
  - New Ack. Num. = 210, New rwnd = 5
  - Last Ack. Num = 206, Last rwnd = 12
  - 5 < (206+12)-210; the relationship is not hold

- Thus, the receiver must want until more buffer space is free before sending an ack.
  - i.e., have a ***larger value of rwnd***

Note:

To avoid shrinking the sender window, the receiver must wait until more space is available in its buffer.

Window Shutdown

- In some cases, the receiver does not want to receive any data from the sender for a while
  - The receiver temporarily shut down the window
  - Sending a segment with the rwnd = 0
  - The sender stops until a new advertisement has arrived
- However, during window shutdown
  - The sender can always send a segment with one byte of data
  - Called ***probing*** and is used to prevent deadlock

Note

- ***In TCP, the sender window size is totally controlled by the receiver window value***

- ***However, the actual window size can be smaller if there is congestion in the network***

Note

- ***Some Points about TCP's Sliding Windows***
  - ***The size of the window is the lesser of rwnd and cwnd***
  - ***The source does not have to send a full window's worth of data***
  - ***The window can be opened or closed by the receiver, but should not be shrunk***
  - ***The destination can send an acknowledgment at any time as long as it does not result in a shrinking window***
  - ***The receiver can temporarily shut down the window; the sender, however, can always send a segment of one byte after the window is shut down***

Silly Window Syndrome

- Problem in the sliding window operation
  - The sending process creates data slowly
  - Or the receiving process consume data slowly
  - Or both
  - Result in the *sending* of data in very small segment
    - Reduce the network efficiency
- For example, send a one byte segment result in overhead of 41/1
  - Assume TCP header is 20 bytes + IP header is 20 bytes

Syndrome Created by the Sender

- Sender application create data too slowly
  - For example, only 1 byte at a time
  - Sending TCP would create segments containing 1 byte of data
- Solution: prevent the sending TCP from sending the data *byte by byte*
  - Sending TCP must be forced to wait as it collects data to send in a larger block
  - But, how long should the sending TCP wait?

Nagle's Algorithm

- Nagle found the solution to above syndrome
  - The sending TCP sends the first piece of data it receives from the sending application
    - Even if it is only 1 byte
  - After sending the first segment, the sending TCP accumulates data in the buffer and wait until
    - Either the receiving TCP sends an acknowledge
    - Or until enough data has accumulated to fill a *maximum-sized segment*
  - Step 2 is repeated. Segment 3 is sent if an acknowledgment is received for segment 2 or enough data is accumulated to fill a maximum-size segment

Nagle's Algorithm (Cont.)

- Elegance
  - Very simple
  - Take into account *the speed of the application that creates the data* and *the speed of the network that transports the data*
    - If application is faster than the network
      - The segments are larger
    - If the application is slower than the network
      - The segment are smaller

Syndrome Created by the Receiver

- The receiving TCP may also create a silly window syndrome
  - If it is serving an application that consumes data slowly
- For example
  - Sender application create data in 1K byte blocks
  - But the receiving application consumes data 1 byte at a time
  - Assume the receiver buffer is 4K bytes
    - Buffer will be full soon
    - Sender then only can send 1 byte data to the receiver

Syndrome Created by the Receiver (Cont.)

- Solution
  - Clark's solution
  - Delayed acknowledgment

Clark's Solution

- Send an acknowledgment as soon as the data arrives but to announce a window size of zero until
  - Either there is enough space to accommodate a segment of maximum size
  - Or half of the buffer is empty

Delayed Acknowledgment

- Delay sending the acknowledgment
  - When a segment arrives, it is not acknowledged immediately
  - Receiver waits until there is a decent amount of space in its incoming buffer
- Delayed acknowledgment *prevents the sending TCP from sliding it window*

Delayed Acknowledgment (Cont.)

- Another advantage
  - Reduce traffic
  - The receiver does not have to acknowledge each segment

- Disadvantage
  - The sender may retransmit the unacknowledged segment
- Solution
  - Defines the acknowledgment should not be delayed by more than 500 ms

12.7

ERROR CONTROL

12.7 ERROR CONTROL

TCP provides reliability using error control, which detects corrupted, lost, out-of-order, and duplicated segments. Error control in TCP is achieved through the use of the checksum, acknowledgment, and time-out.

The topics discussed in this section include:

Checksum
Acknowledgment
Acknowledgment Type
Retransmission
Out-of-Order Segments
Some Scenarios

Error Control

- TCP provides reliability using error control
  - Detect corrupted segment, lost segment, out-of-order segment, and duplicated segment

- Error detection and correction is achieved by
  - ***Checksum***
  - ***Acknowledgment***
  - ***Time-out***

Checksum

- TCP uses 16-bit checksum
  - Mandatory in every segment
- Actually, 16-bit checksum is considered inadequate for SCTP
  - Mentioned later

Acknowledgment

- TCP uses acknowledgment to confirm the receipt of ***data segment***
- ***Control segments*** that carry no data but consume a sequence number are also acknowledged
- ACK segment are never acknowledged

Note:

ACK segments do not consume sequence numbers and are not acknowledged.

Generating Acknowledgments: When Does a Receiver Generate Acknowledgment?

1. When one end sends a data segment
   - It must piggyback an acknowledgment
   - Decrease the number of segments needed
2. The receiver delays sending an ACK
   - When the following three conditions hold
     - When the receiver has no data to send
     - It receives an in-order segment
     - The previous segment has already been acknowledged
   - The receiver delays sending an ACK
     - until another segment arrives or
     - Until a period of time (normally 500 ms) has passed
   - Thus, if only one outstanding in-order segment
     - Delaying sending an ACK
   - Prevent ACK segments from creating extra traffic

Generating Acknowledgments: When Does a Receiver Generate Acknowledgment? (Cont.)

3. The receiver immediately sends an ACK
   - When the following two conditions hold
     - A in-order segment arrives
     - The previous in-order segment has not been acknowledged
   - Thus, there should not be more than two in-order unacknowledged segment at any time
   - Prevent unnecessary retransmission

Generating Acknowledgments: When Does a Receiver Generate Acknowledgment? (Cont.)

4. The receiver immediately sends an ACK
   - When an out-of-order segment with higher sequence number is received
   - Enable fast retransmission of any missing segment
5. When a missing segment arrives, the receiver sends an ACK
   - Inform the receiver that segments reported missing have been received
6. The receiver immediately sends an ACK if a duplicate segment arrives
   - Solve some problems when an ACK segment itself is lost

Acknowledgment Type

- Acknowledgment type
  - ***Accumulative Acknowledgment (ACK)***
    - In past, TCP only uses this ACK
  - ***Selective Acknowledgment (SACK)***
    - Newly added feature

Accumulative Acknowledgment (ACK)

- The receiver advertises the next byte it expects to receive
  - Ignore all segments received out-of-order

- Also called "positive" accumulative acknowledgement
  - Discarded, lost, or duplicated segments are not reported

Selective Acknowledgment (SACK)

- Does not replace ACK
- But report additional information to the sender
  - The block of data that is out-of-order
  - The block of segments that is duplicated
- SACK is implemented as an ***option***
  - Since there is no provision in the TCP header for SACK

Retransmission

- When to retransmit a segment
  - When a *retransmission timer* expires
  - When the sender receives *three duplicate ACK*
- No retransmission occurs for segments
  - If it does not consume sequence number
  - If it is an ACK segment

Retransmission After RTO

- Sender TCP starts a ***retransmission time-out (RTO)*** timer for each segment sent
- If timer matures
  - Retransmit the segment
- RTO value is dynamic
  - Updated based on the round trip time (RTT)

Retransmission After Three Duplicated ACK Segments

- A segment is lost but the receiver receives so many out-of-order segments
  - Buffer may overflow

- Solution: ***fast retransmission***
  - Retransmit the missing segment immediately if three duplicate ACK received

Out-of-Order Segment

- Out-of-order
  - When a segment is delayed, lost, or discarded
- Previous solution in TCP
  - Does not acknowledge an out-of-order segment
  - Discard all out-of-order segment
    - Result in the retransmission of the missing segment and the following segment

Out-of-Order Segment (Cont.)

- Current implementation of TCP
  - Store out-of-order segments temporarily
  - Until the missing segment arrives
- Note
  - The out-of-order segment are not delivered to the process
  - TCP guarantees in-order delivery

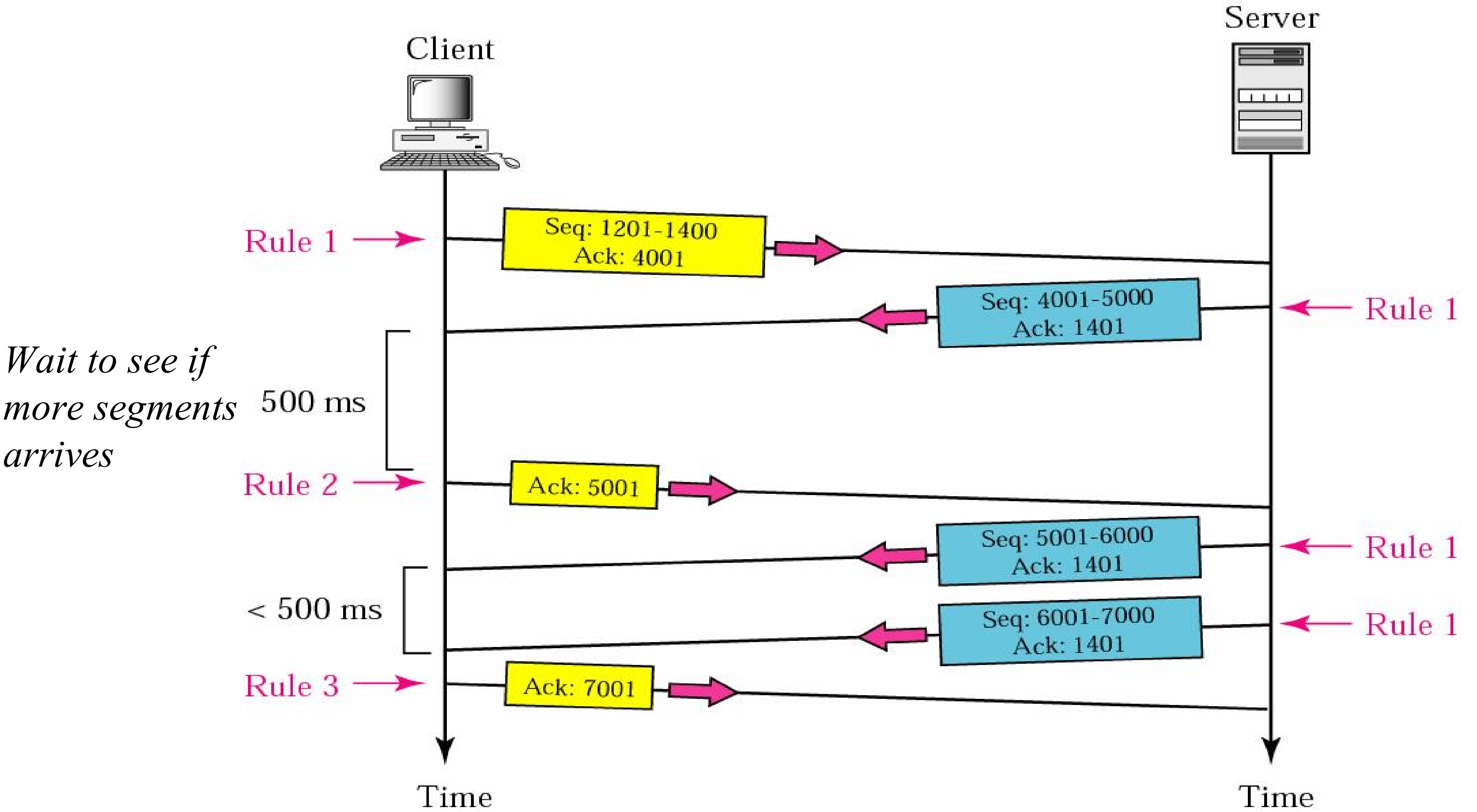
Note:

Data may arrive out of order and be temporarily stored by the receiving TCP, but TCP guarantees that no out-of-order segment is delivered to the process.

Some Scenarios

- Some scenarios occurs during the operation of TCP
  - Normal operation
  - Lost segment
  - Fast retransmission
  - Delayed segment
  - Duplicate segment
  - Automatically corrected lost ACK
  - Lost acknowledgment corrected by resending a segment
  - Deadlock created by lost acknowledgment

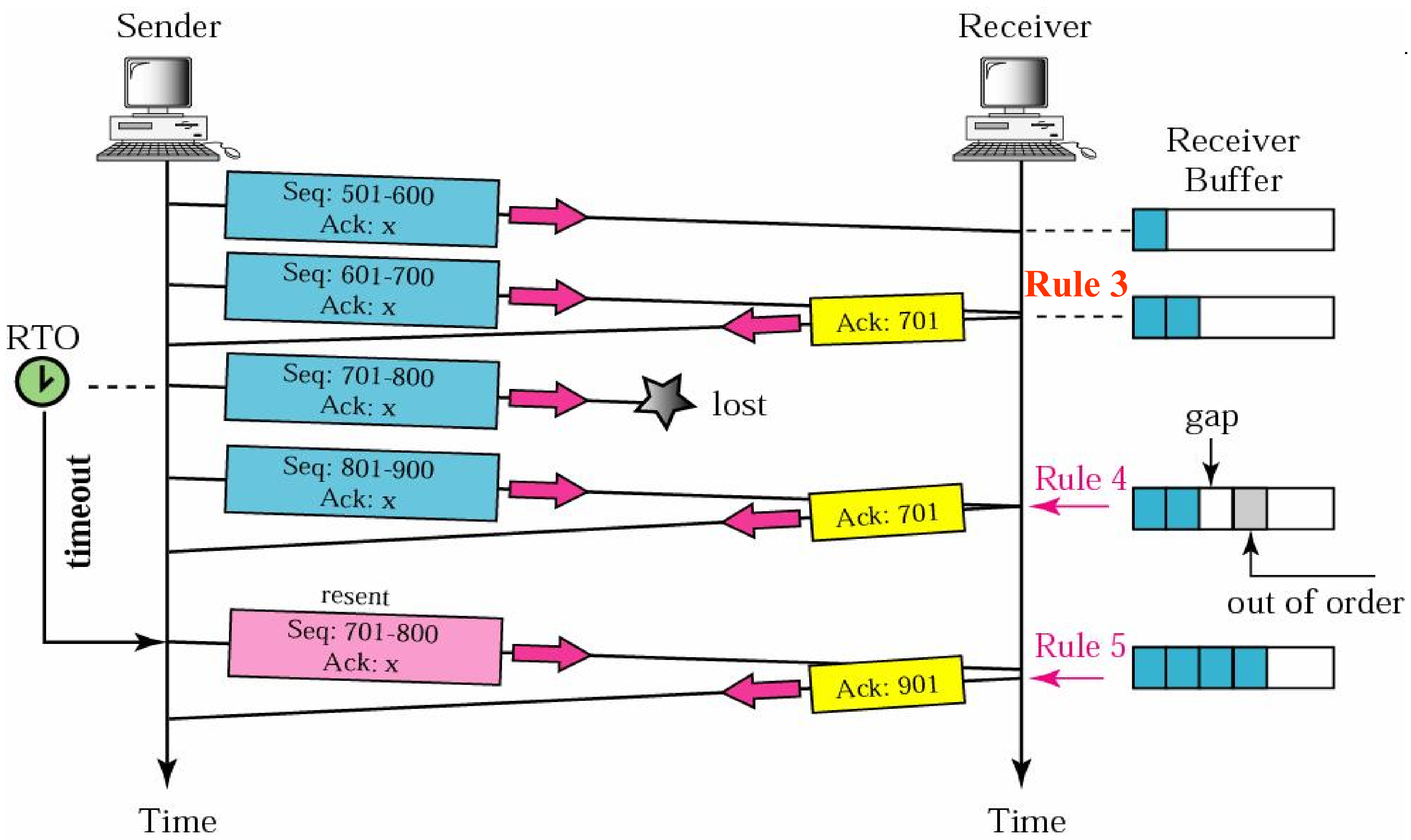
Normal Operation



Lost or Corrupted Segment

- A lost or corrupted segment is treated the same way by the receiver TCP
  - ***Lost segment***: discarded somewhere in the network
  - ***Corrupted segment***: discarded by the receiver itself
- In following feature, segment 3 is lost
  - Receiver receives a out-of-order segment (segment 4)
    - Store it temporary and leave a gap
    - Send an ACK immediately (ACK number = 701)
  - Sender resent segment 3 when the RTO timer matures

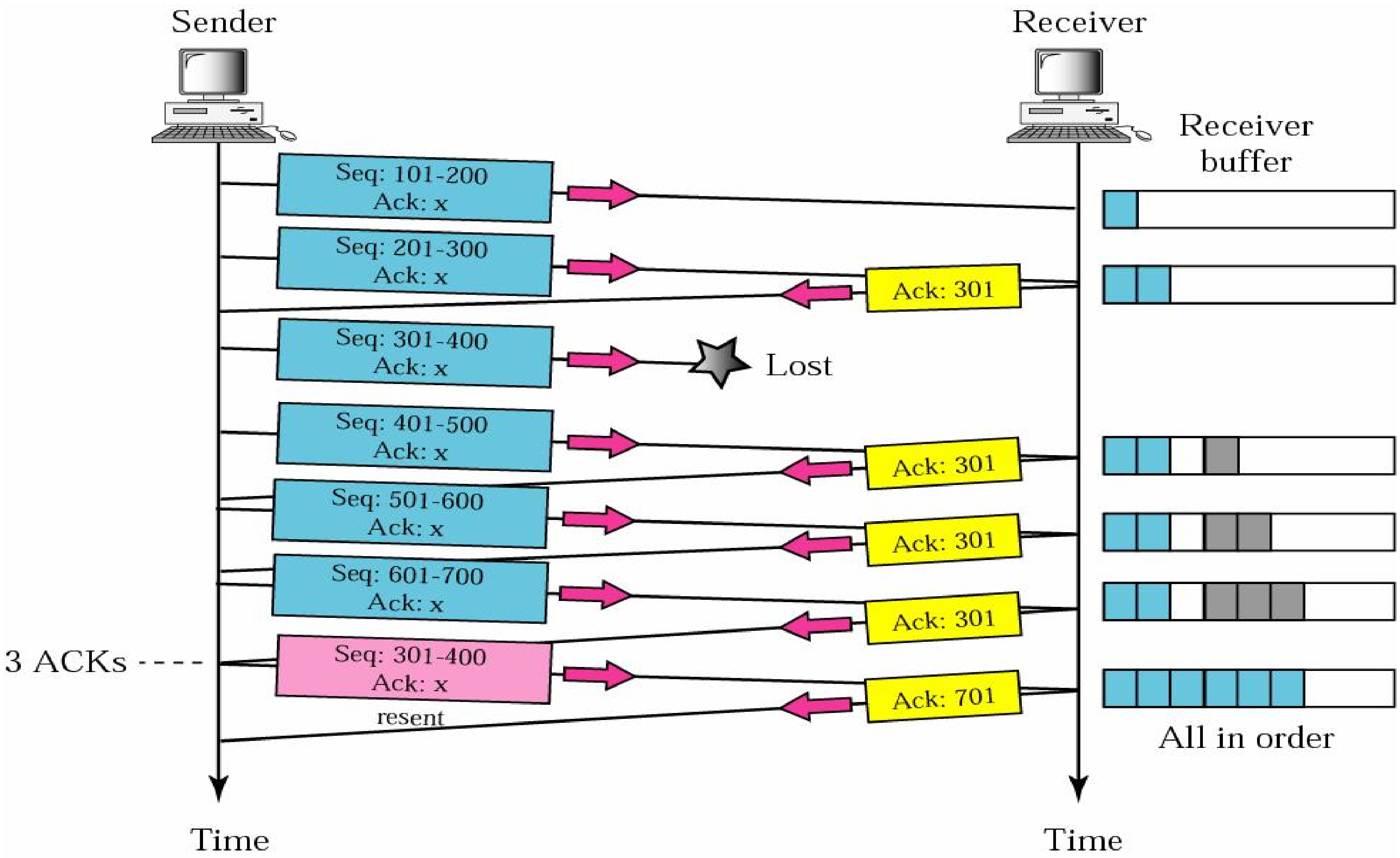
Lost Segment



Fast Retransmission

- In the following feature
  - When the receiver receives the 4th, 5th, 6th segment, it triggers an acknowledgment
  - Thus, four acknowledgment are the same value
    - ***Three duplicated***
  - Although the RTO timer for segment 3 has not yet matured
    - Invoke fast retransmit for segment 3

Fast Retransmission



The McGraw-Hill Companies, Inc., 2000

Delayed Segment

- Each TCP segment is encapsulated in an IP datagram
  - IP datagram is routed independently
  - A TCP segment may be delayed
- A delayed segment is treated the same way as lost or corrupted segment by the receiver
- Note
  - A delayed segment may arrive after it has been ***resent***
  - Cause a ***duplicate segment***

Duplicate Segment

- Created by a sending TCP when a segment is delayed and treated as lost by the receiver

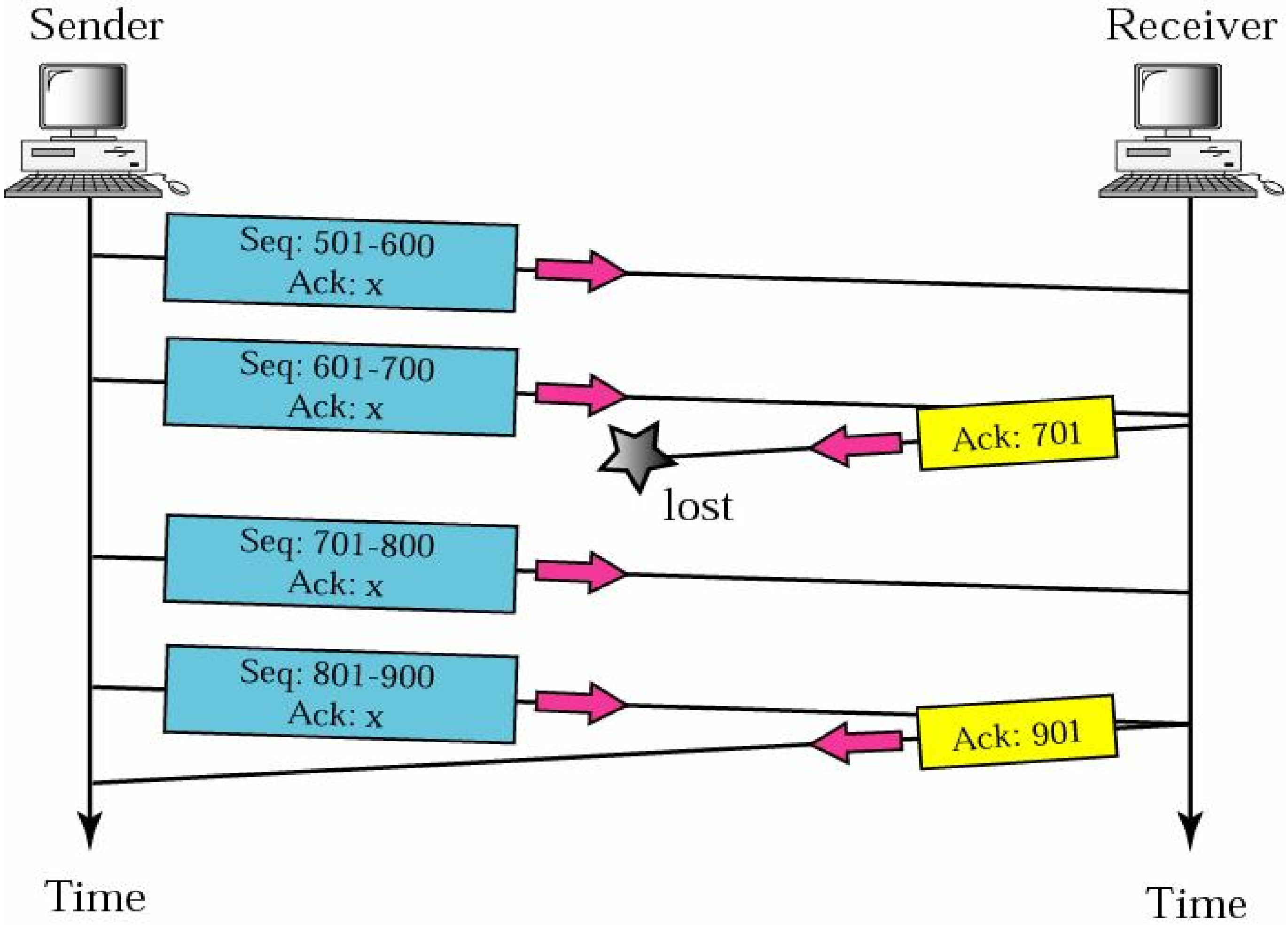
- Destination detects duplicate segment since they have the same sequence number
  - *Discard the later segment*

Automatically Corrected Lost ACK

- A lost acknowledgment is automatically replaced by the next
  - Since ACK is accumulative

- In the following feature
  - The next ACK automatically correct the lost of the acknowledgment

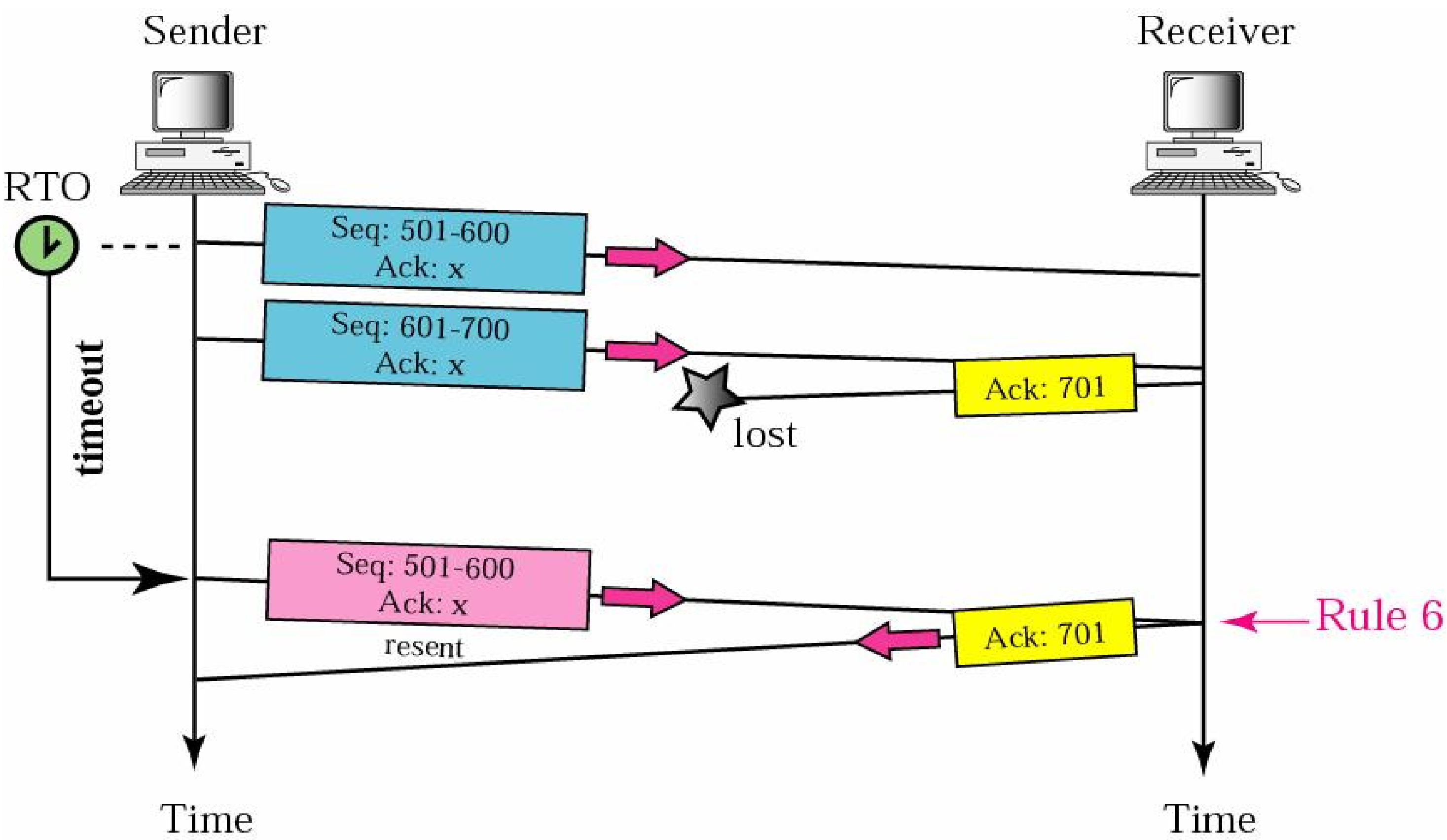
Lost Acknowledgment



Lost Acknowledgment Corrected by Resending a Segment

- If an ACK is lost
  - But the next ACK is delayed for a long time or
  - There is no next acknowledgment
- How to correct ?
  - By the RTO timer and resent the data segment
- Result in a duplicate ACK
  - Receiver just discards it
  - Resent the ACK immediately (rule 6)

Lost Acknowledgment Corrected by Resending a Segment



Deadlock Created by Lost Acknowledgment

- A lose of an acknowledgment may result in system deadlock
- Example
  - Receiver sends an ACK with rwnd = 0
    - Request the sender to shut down its window temporarily
  - The sending TCP stops transmitting segments
  - After a while, receiver sends an ACK and rwnd <>0
    - Announce it can receive data again
    - However, this ACK is lost
  - As a result, both sender and receiver continue to wait for each other forever
- Solution: a ***persistent timer***

12.8

CONGESTION CONTROL

12.8 CONGESTION CONTROL

Congestion control refers to the mechanisms and techniques to keep the load below the capacity.

The topics discussed in this section include:

Network Performance
Congestion Control Mechanisms
Congestion Control in TCP

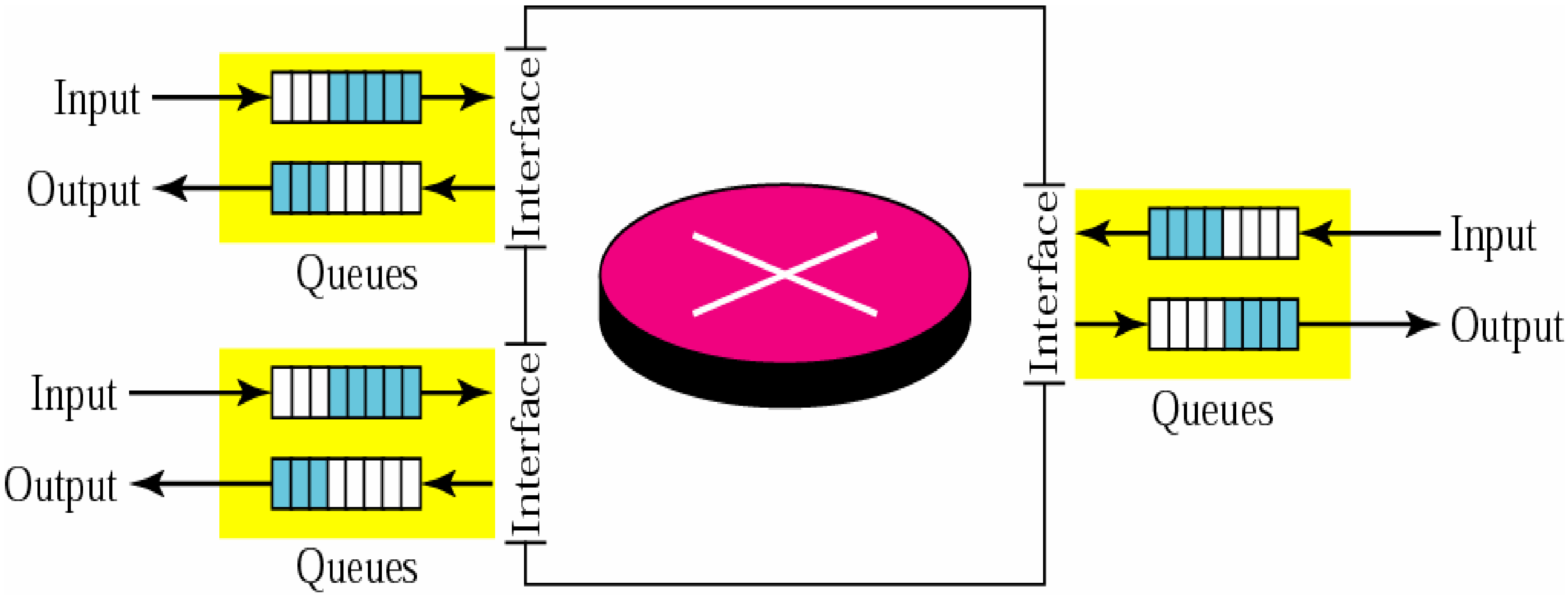
Congestion Control

- Congestion
  - The load on the network is greater than the *capacity* of the network
- Why?
  - An internet is a combination of networks and connecting devices, e.g., routers and switches
  - A router has a *buffer* that *stores the incoming packets*, *processes them*, and *forward them.*

Congestion Control (Cont.)

- In the following features,
  - Input queue may be congested
    - Packet arrival rate > packet processing rate
  - Output queue may be congested
    - Packet departure rate < packet processing rate

Router Queue



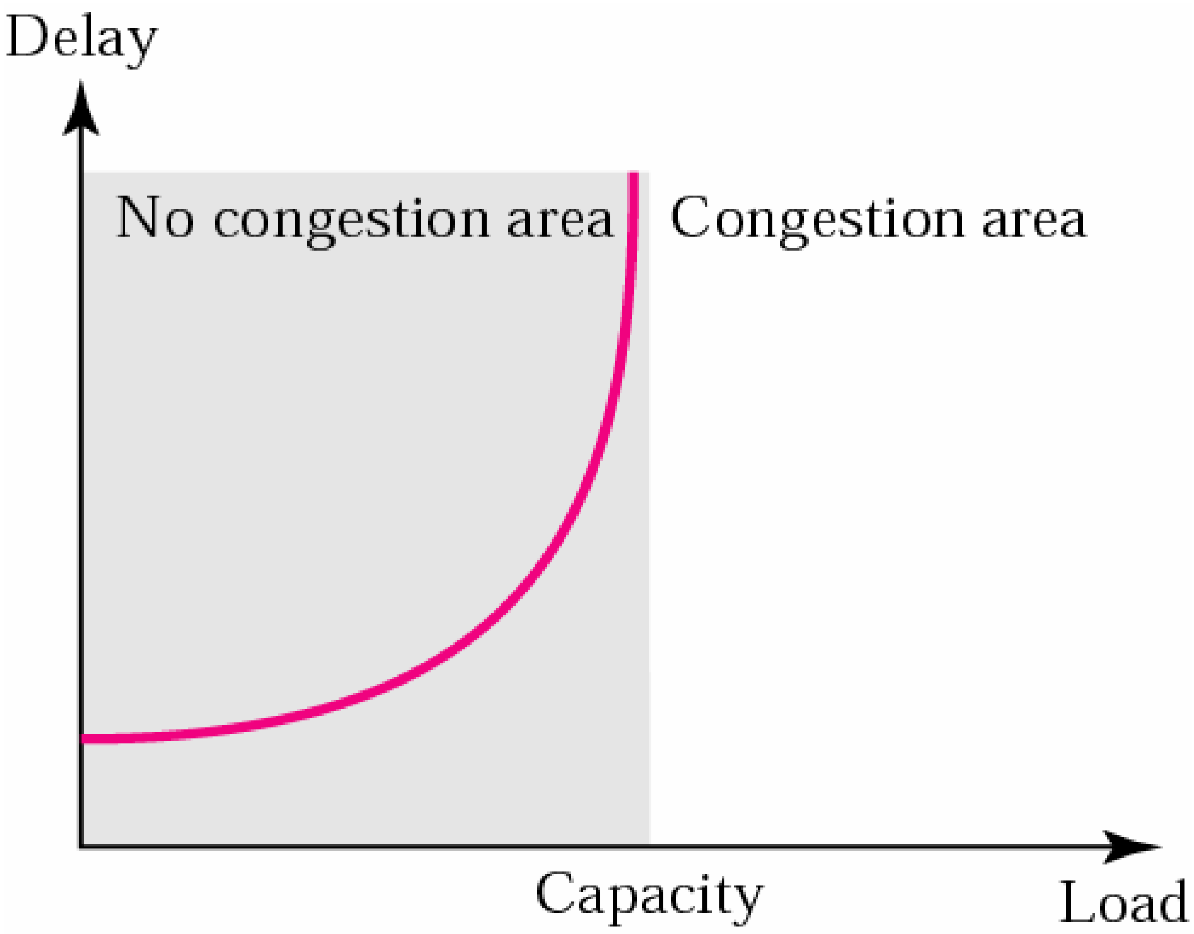
Network Performance

- Network performance
  - *Delay*
  - *Throughput*

Delay versus Load

- In Fig. 12.31
  - When the load is small, delay is minimum
    - Delay = propagation delay + processing delay
    - Can be negligible
  - When the load becomes larger
    - Delay = propagation delay + processing delay + waiting time in all routers' queues along the path
  - When the load is greater than the capacity
    - Delay becomes infinite

Packet Delay and Network Load



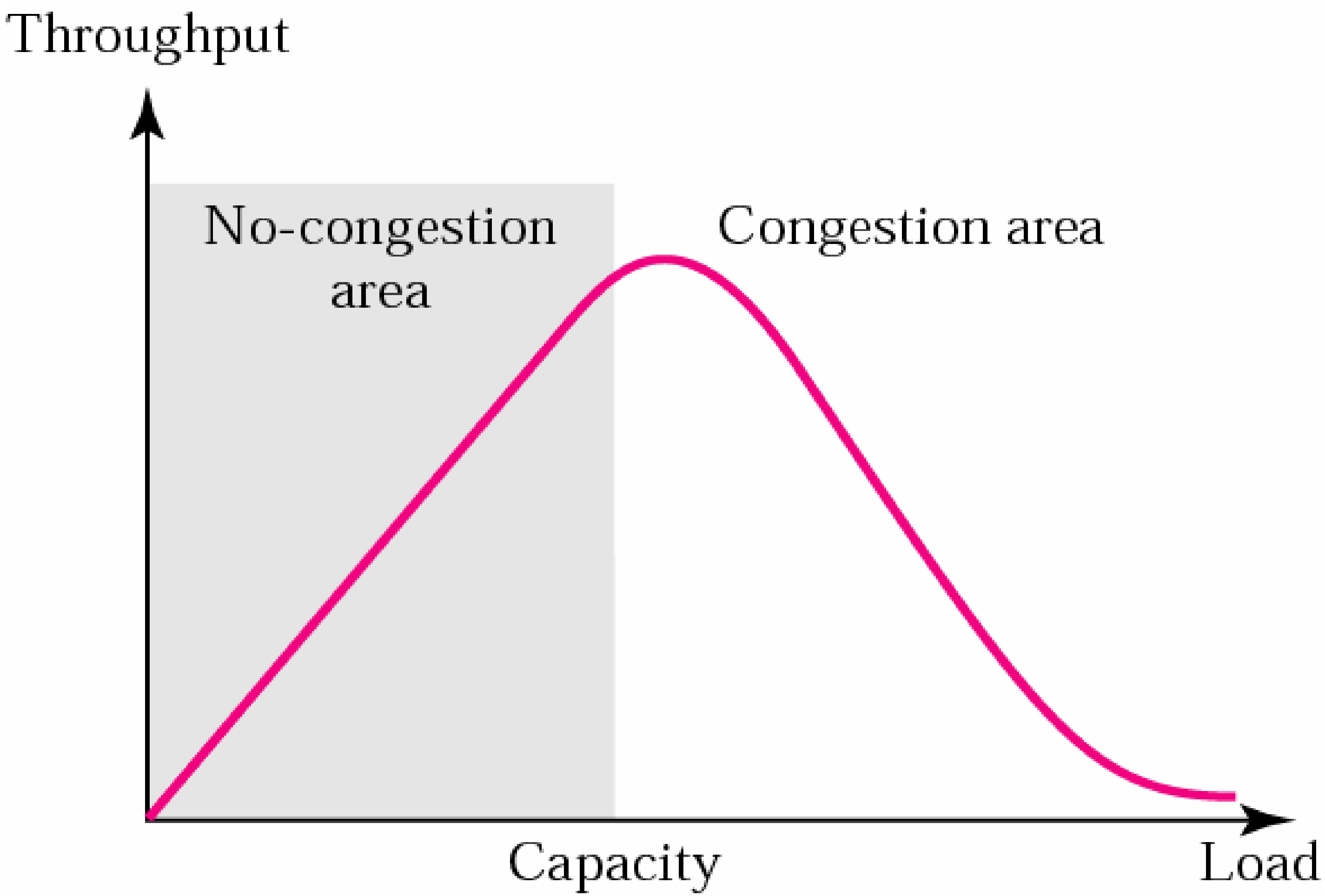
Delay versus Load (Cont.)

- Delay has a negative effect on the load/congestion
  - When a packet is delayed or dropped in the router
    - No acknowledgement is received by the sender
    - The sender will retransmit the packet
      - Create more congestion and more delaying/dropping of packets

Throughput versus Load

- When the load is below the capacity of the network
  - Throughput increases proportionally with the load
- When the load reaches the capacity
  - The throughput declines sharply
    - Since more segments are discarded by the router
    - However, discard segments cause more segments to be transmitted
      - Since TCP retransmission scheme

Throughput versus Network Load



Congestion Control Mechanism

- Congestion control
  - Prevent congestion before it happens
  - Remove congestion after it happens

- Two categories
  - ***Open-loop congestion control (prevention)***
  - ***Closed-loop congestion control (removal)***

Open-Loop Congestion Control

- Prevent congestion before it happens
- Possible policies
  - *Retransmission policy*
    - Retransmission policy and retransmission timer should be designed to optimize efficiency
  - *Acknowledgment policy*
    - Does not ACK every packet it receives
  - *Discard policy*
    - Router should adopt good discard policy

Closed-Loop Congestion Control

- Try to alleviate congestion after it happens
- Possible mechanisms
  - *Back pressure*
    - When a router is congested, it can inform the previous unstream router to reduce it outgoing rate
    - The action can be recursive all the way to the router *just prior to the source*
  - *Choke Point*
    - A router sends a packet to the source to inform congestion
    - This packet is called chock point, like ICMP's source quench packet
  - *Implicit signaling*
    - Source can detect an implicit signal warning of congestion
      - For example, the delay in receiving an acknowledgment
  - *Explicit signaling*
    - Router can send an explicit signal to the sender or receiver of congestion
      - For example, set a bit in a packet

Congestion Control in TCP

- Outline
  - Congestion window
  - Congestion policy
    - *Slow start: exponential increase*
    - *Congestion avoidance: additive increase*
    - *Congestion detection: multiplicative decrease*

Congestion Window

- *Flow control*
  - Sender window size is determined by the available buffer space in the *receiver*
- However, in addition to the *receiver*, the *network* should be a second entity that determines the size of the sender's window
- *Congestion control*
  - Determine the sender window size by the congestion condition in the network

Congestion Window (Cont.)

- Thus, the sender's window size is determined by both
  - *Receiver*
  - *Congestion in the network*
- The sender has two pieces of information
  - The receiver-advertised window size
  - The congestion widow size
- The actual size of the window is the minimum of these two
  - *Actual window size = minimum (receiver window size, congestion window size) = minimum (rwnd, cwnd)*

Congestion Policy

- Three phase in TCP's congestion policy
  - Slow start
  - Congestion avoidance
  - Congestion detection
    - When a congestion is detected, sender return to the *slow start* or *congestion avoidance*

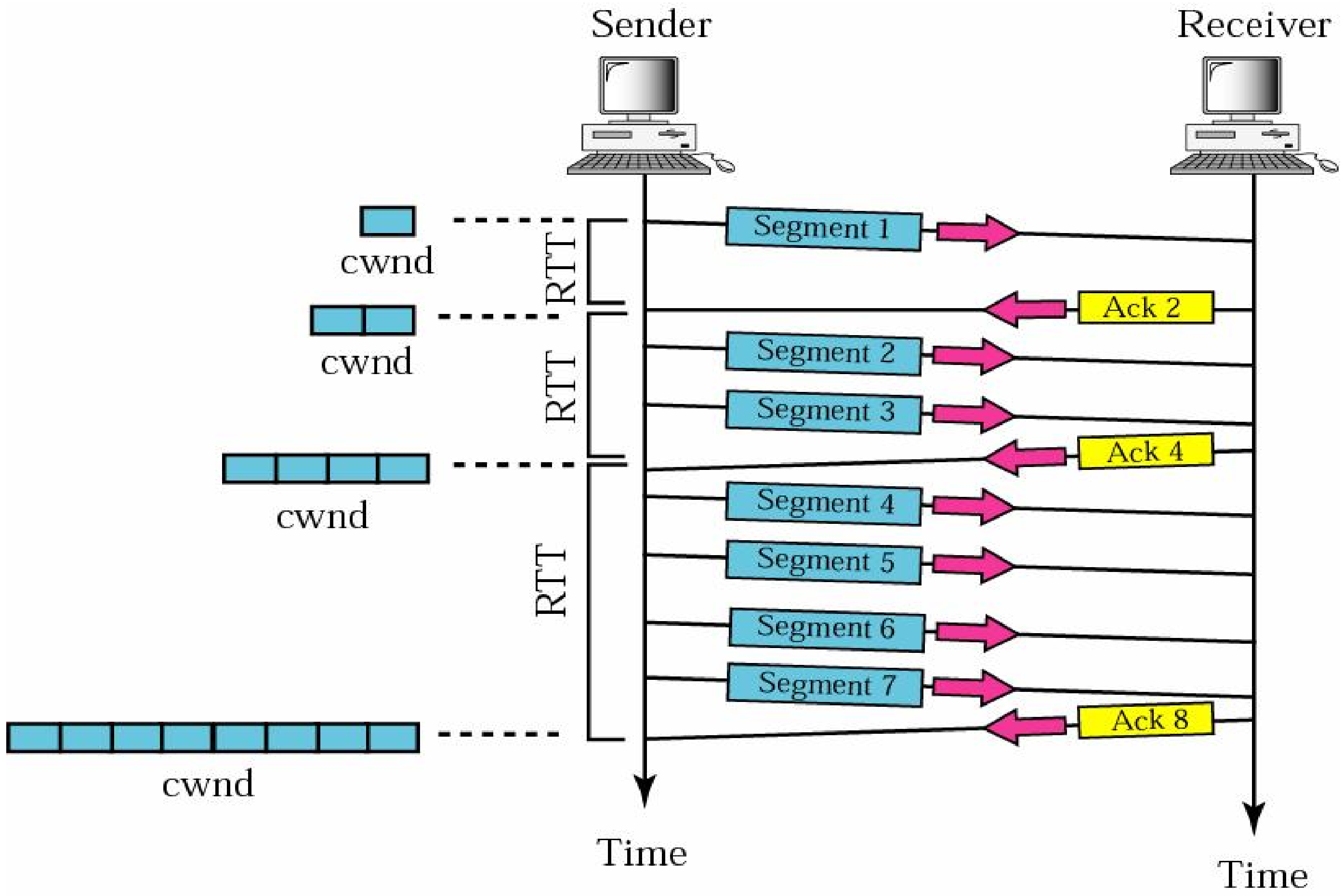
Slow Start: Exponential Increase

- At the beginning,
  - congestion window size = maximum segment size (MSS)
- MSS is determined during connection establishment using an option (mentioned later)
- For each segment that is acknowledged
  - Increase the congestion window size by one maximum segment unit
  - Until it reaches a *threshold, called ssthresh (show start threshold)*
    - Usually, *ssthresh* = 65535 bytes

Slow Start: Exponential Increase (Cont.)

- However, it is not actually “slow start”
  - The congestion window size increases *exponentially*
  - Start => cwnd = 1 = 2^0
  - After 1 RTT => cwnd = 2 = 2^1
  - After 2 RTT => cwnd = 4 = 2^2
  - After 3 RTT => cwnd = 8 = 2^3

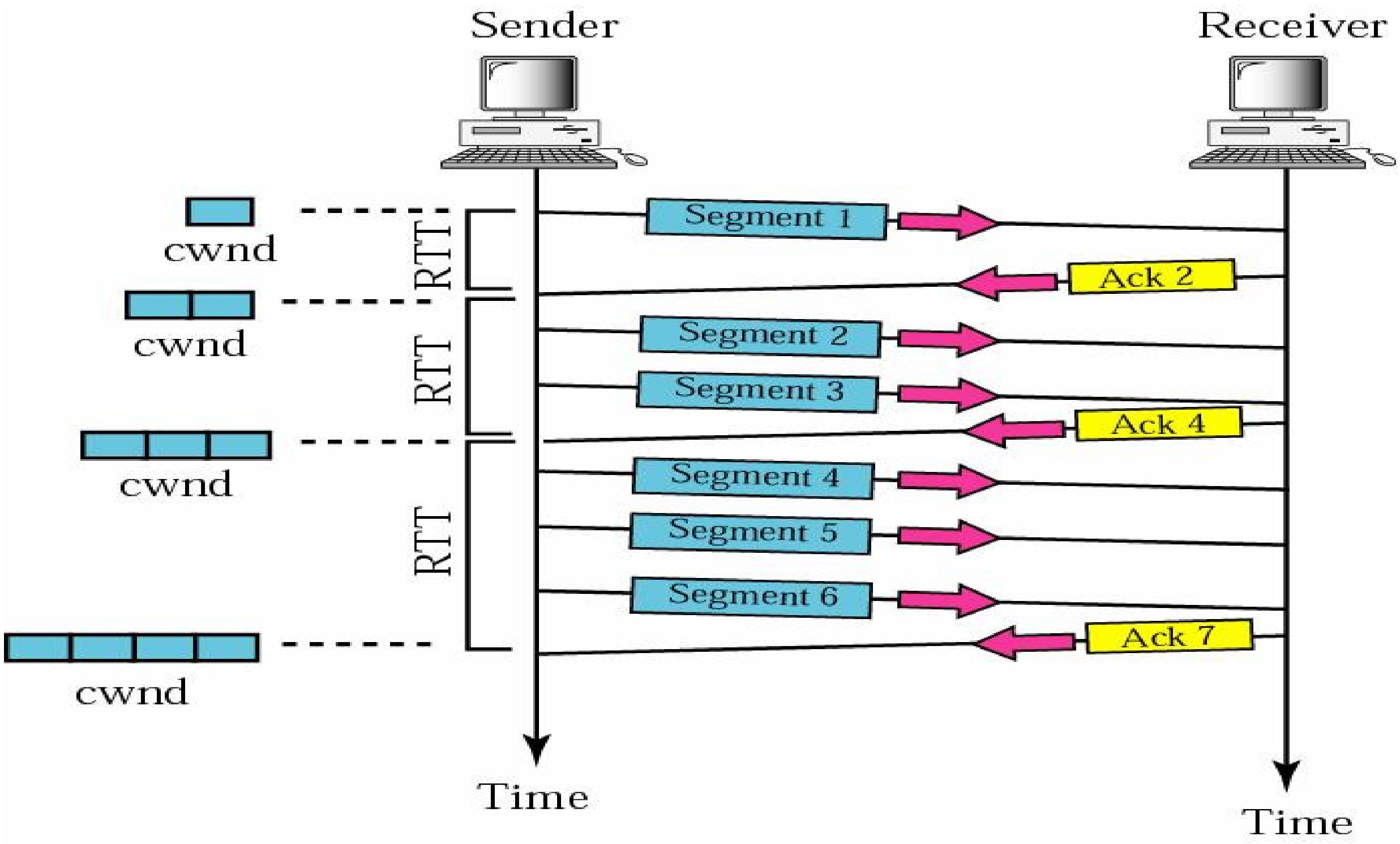
Slow Start: Exponential Increase



Congestion Avoidance: Additive Increase

- Started after the congestion window size reaches the *ssthresh threshold*
- When the *whole window of segments* is acknowledged
  - The size of congestion window is increased *one*
  - *Note, the whole window size is usually larger than one in congestion avoidance*

Congestion Avoidance, Additive Increase



Congestion Avoidance: Additive Increase (Cont.)

- In the above figure
  - Start => cwnd = 1
  - After 1 RTT => cwnd = 1 + 1 = 2
  - After 2 RTT => cwnd = 2 + 1 = 3
  - After 3 RTT => cwnd = 3 + 1 = 4

Congestion Detection: Multiplicative Decrease

- If congestion occurs, the congestion window size must be decreased
- How to detect a congestion?
  - *The need to retransmit a segment*
- When to retransmit a segment
  - *When an RTO timer out*
  - *When three duplicate ACKs are received*

Congestion Detection: Multiplicative Decrease (Cont.)

- In both cases, the size of the threshold is ***half of the current congestion window size***
  - ***multiplicative decrease***
- However, different actions are taken
  1. If a time-out occurs: a strongly possibility of congestion
     - *The threshold should be set to half of the current congestion window size*
       - *Multiplicative decrease*
     - *The congestion window size should start from one again, i.e., cwnd = 1*
     - *The sender return to the slow start phase*

Congestion Detection: Multiplicative Decrease (Cont.)

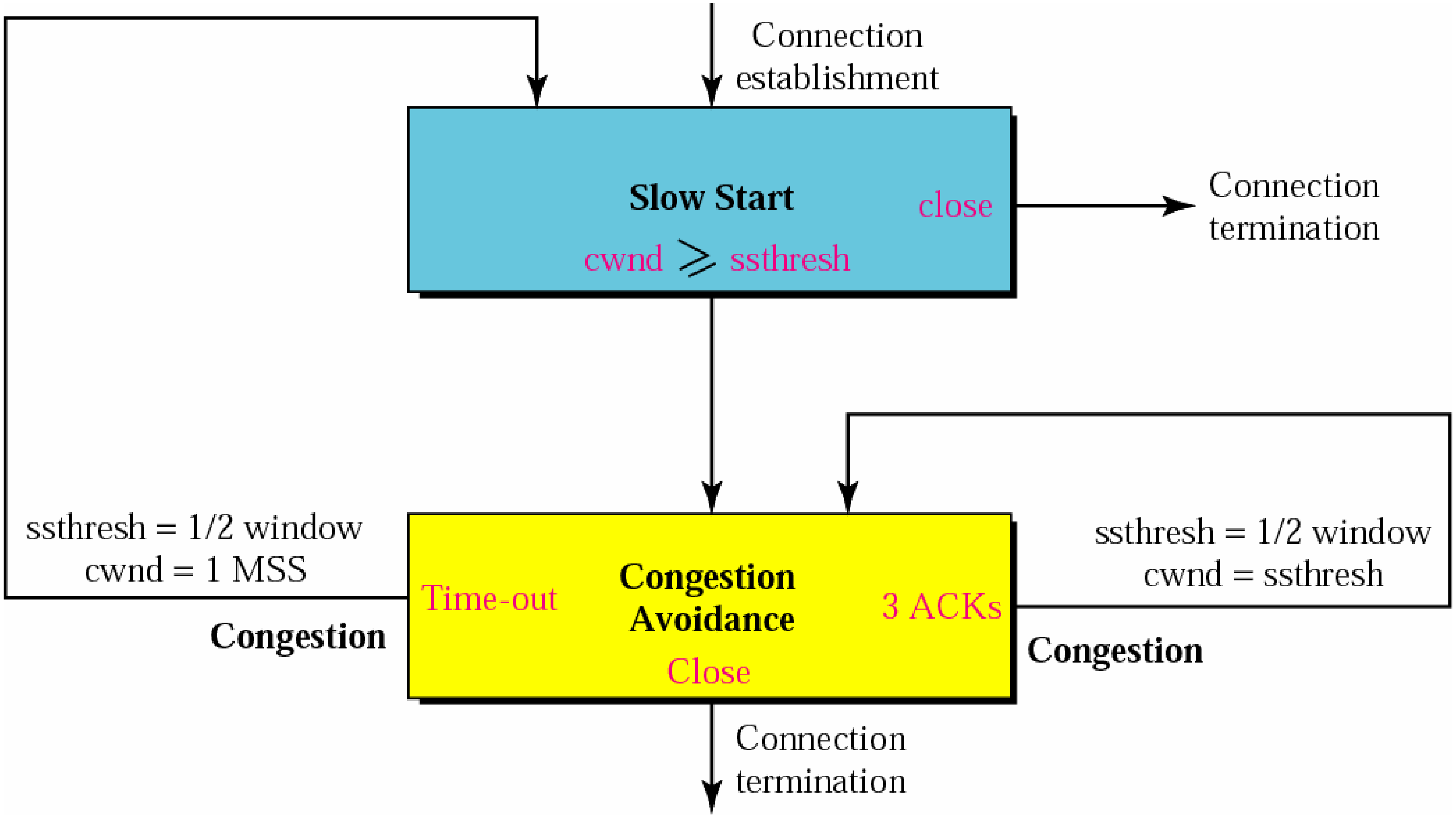
- If three duplicated ACKs are received: a weaker possibility of congestion
- Invoke ***fast retransmission and fast recovery***
  - *The threshold should be set to half of the current congestion window size*
    - *Multiplicative decrease*
  - *The congestion window size = threshold again, i.e., cwnd = ssthresh*
  - *The sender starts the congestion avoidance phase*

Note:

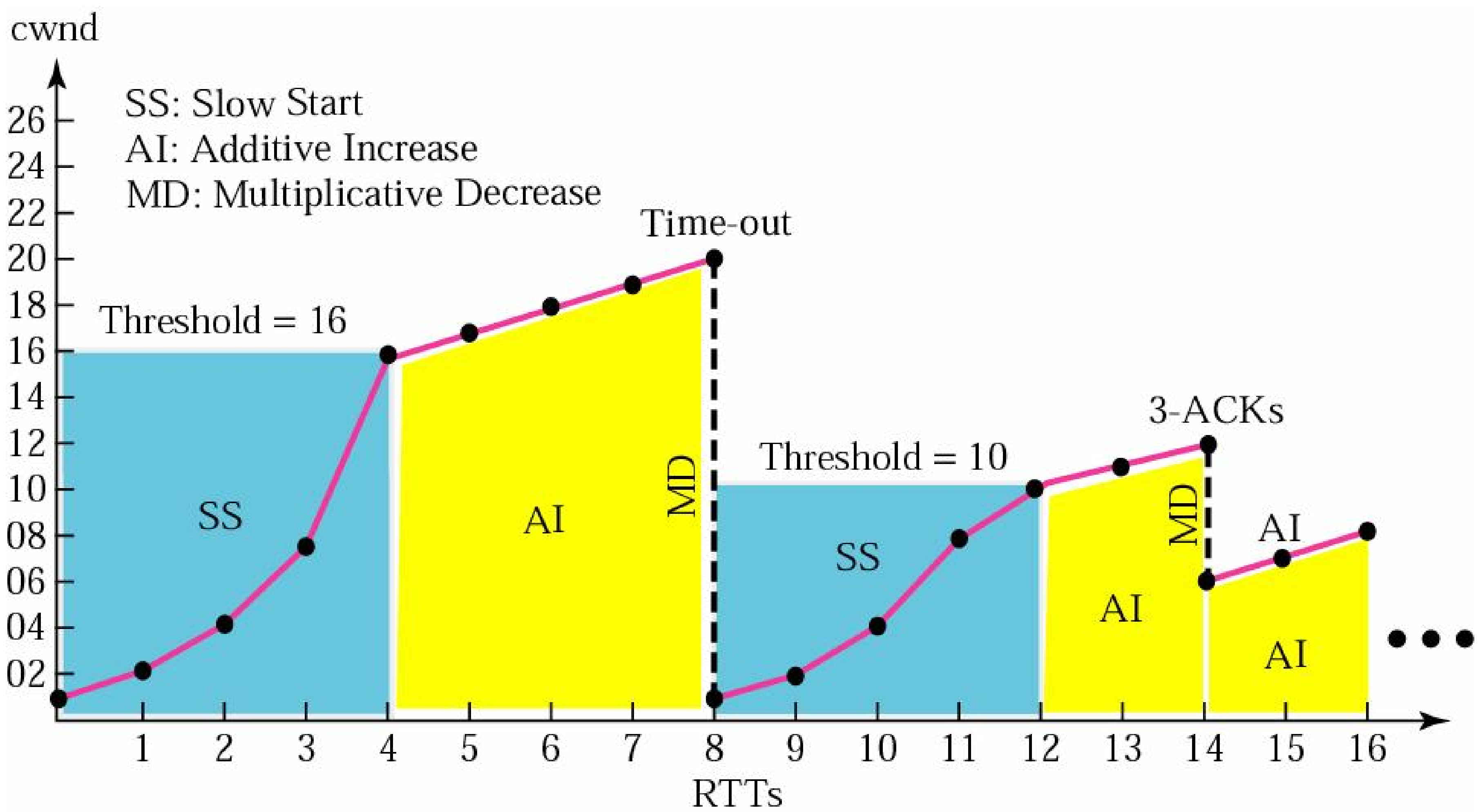
Most implementations react differently to congestion detection:

- ***If detection is by time-out, a new slow start phase starts.***
- ***If detection is by three ACKs, a new congestion avoidance phase starts.***

TCP Congestion Policy Summary



Congestion Example



12.9

TCP TIMERS

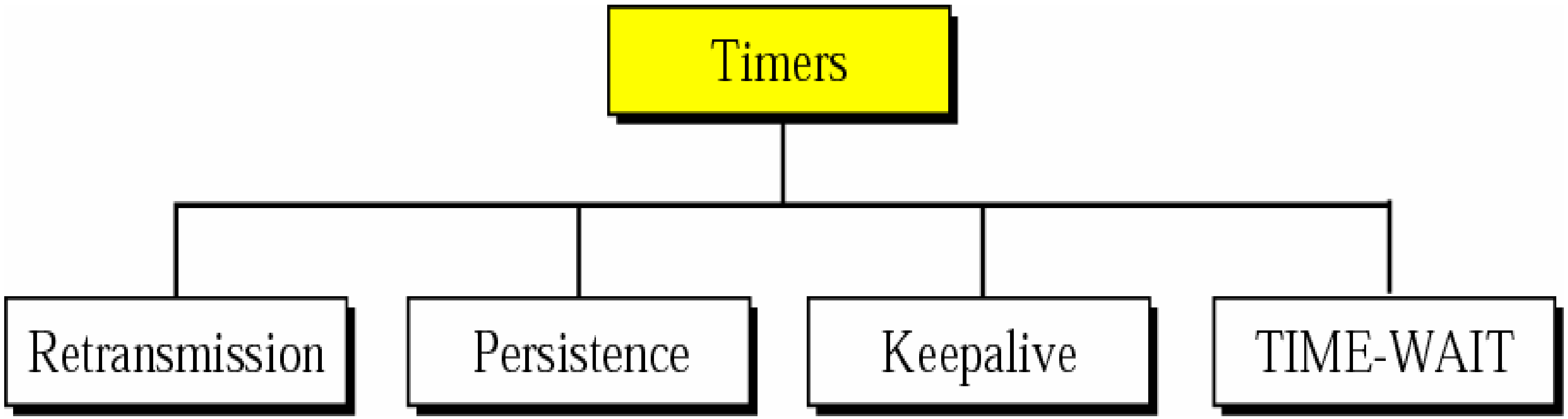
12.9 TCP TIMERS

To perform its operation smoothly, most TCP implementations use at least four timers.

The topics discussed in this section include:

Retransmission Timer
Persistence Timer
Keepalive Timer
TIME-WAIT Timer

TCP Timers



Retransmission Timer

- When TCP sends a segment, it creates a *retransmission timer* to that segment
  - If an acknowledgment is received before the timer goes off
    - The timer is destroyed
  - If timer goes off before an acknowledgment arrives
    - The segment is retransmitted and the timer is reset

Calculation of Retransmission Time

- TCP cannot use the same retransmission time for all connections
  - Each connection has different length and network characteristics
- Furthermore, TCP cannot use the same retransmission time for one single connection
  - The network behavior is dynamic
- Thus, TCP uses the *dynamic* retransmission time
  - Different for each connection and may be changed during the same connection

Calculation of Retransmission Time (Cont.)

- Retransmission time-out (RTO) is calculated based on RTT
- But, how to calculate RTT?

Calculation of RTT

- Measured RTT
- Smoothed RTT
- RTT Deviation

Measured RTT

- Measured RTT: $\boldsymbol{RTT_M}$
  - How long it takes to send a segment and receive an acknowledgment
- Note, segments and their ACKs do not have a one-to-one relationship
  - Several segments may be acknowledged together
- In TCP, *only one RTT measurement can be in process at any time*

Smoothed RTT

- The measured RTT may fluctuation very highly
  - Cannot be used for RTO purpose
- Smooth RTT: $\boldsymbol{RTT_S}$
- $\mathbf{RTT_S} = (1\text{-}a)\ \mathbf{RTT_S} + a \times \mathbf{RTT_M}$
  - a is usually 1/8 percent

RTT Deviation

- Most implementation also calculate the RTT deviation, called RTT_D
- Original => No value
- After first measurement
  - $\mathbf{RTT_D} = \mathbf{RTT_M}/2$
- After any other measurement
  - $\mathbf{RTT_D}=(1-B)\mathbf{RTT_D}+B\text{x } |\mathbf{RTT_S}-\mathbf{RTT_M}|$
  - B is usually 1/4

Retransmission Timeout (RTO)

- Original => Initial value
- After any measurement
  - $\mathbf{RTO} = \mathbf{RTT_S} + 4 \times \mathbf{RTT_D}$

Example 10

Figure 12.38 shows part of a connection. The figure shows the connection establishment and part of the data transfer phases.

1. When the SYN segment is sent, there is no value for RTT_M, RTT_S, or RTT_D. The value of RTO is set to 6.00 seconds.

$$RTO = 6$$

2. When the SYN+ACK segment arrives, RTT_M is measured and is equal to 1.5 seconds.

$RTT_M = 1.5$

$RTT_S = 1.5$

$RTT_D = 1.5 / 2 = 0.75$

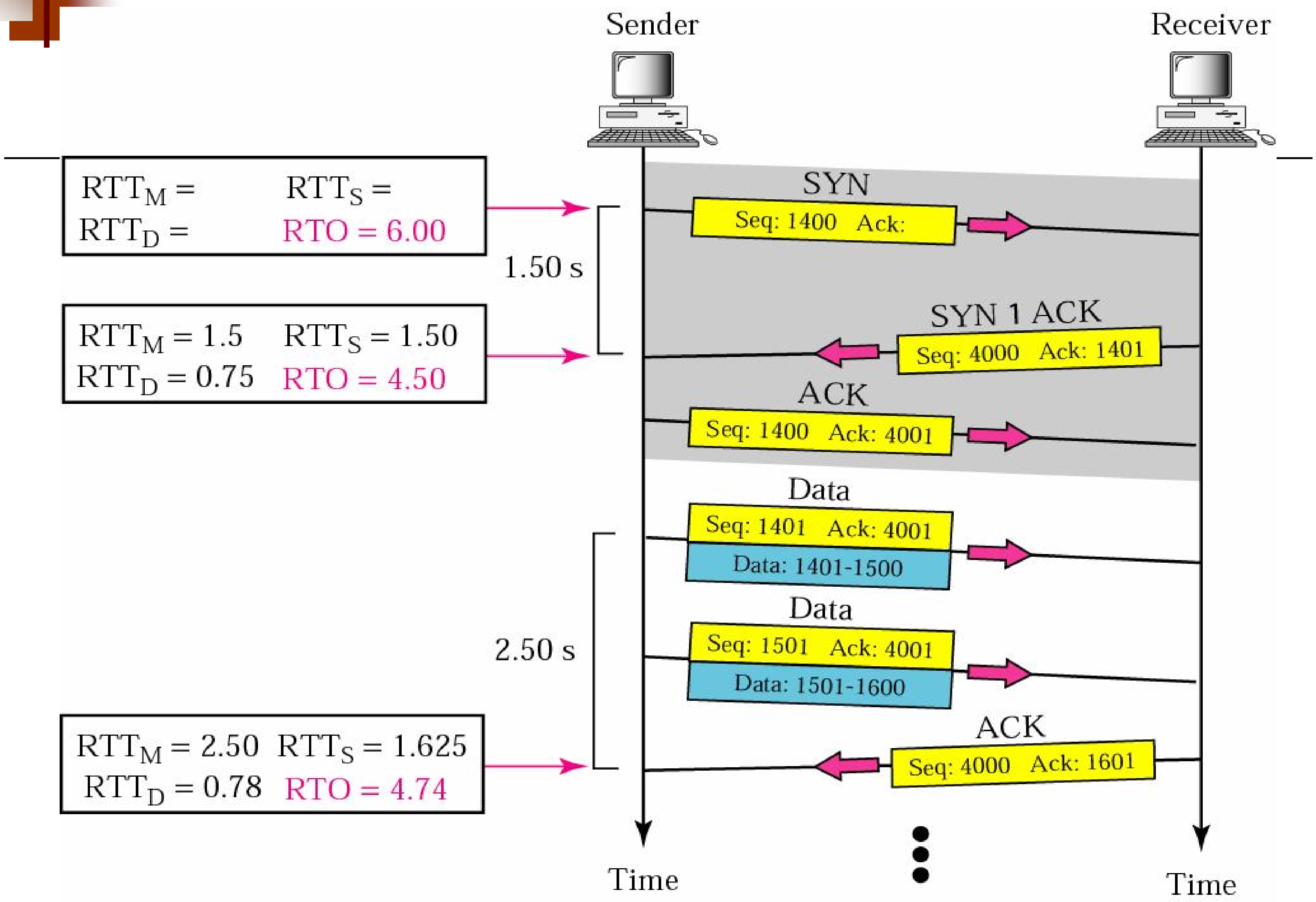
$RTO = 1.5 + 4 . 0.75 = 4.5$

Example 10 (continued)

***3.** When the first data segment is sent, a new RTT measurement starts. Note that the sender does not start an RTT measurement when it sends the ACK segment, because it does not consume a sequence number and there is no time-out. No RTT measurement starts for the second data segment because a measurement is already in progress.*

$RTT_M = 2.5$
$RTT_S = 7/8\ (1.5) + 1/8\ (2.5) = 1.625$
$RTT_D = 3/4\ (7.5) + 1/4\ |1.625 - 2.5| = 0.78$
$RTO = 1.625 + 4\ (0.78) = 4.74$

Figure 12.38 *Example 10*



Karn’s Algorithm

- Problem
  - If a segment is not acknowledged during the retransmission period and it is retransmitted
  - When the sending TCP receives an ACK.
    - It does not know this acknowledgment is for the first one or for the retransmitted one?
- Solution: ***Karn’s algorithm***
  - Do not consider the RTT of a retransmitted segment in the calculation of the new RTT

Exponential Backoff

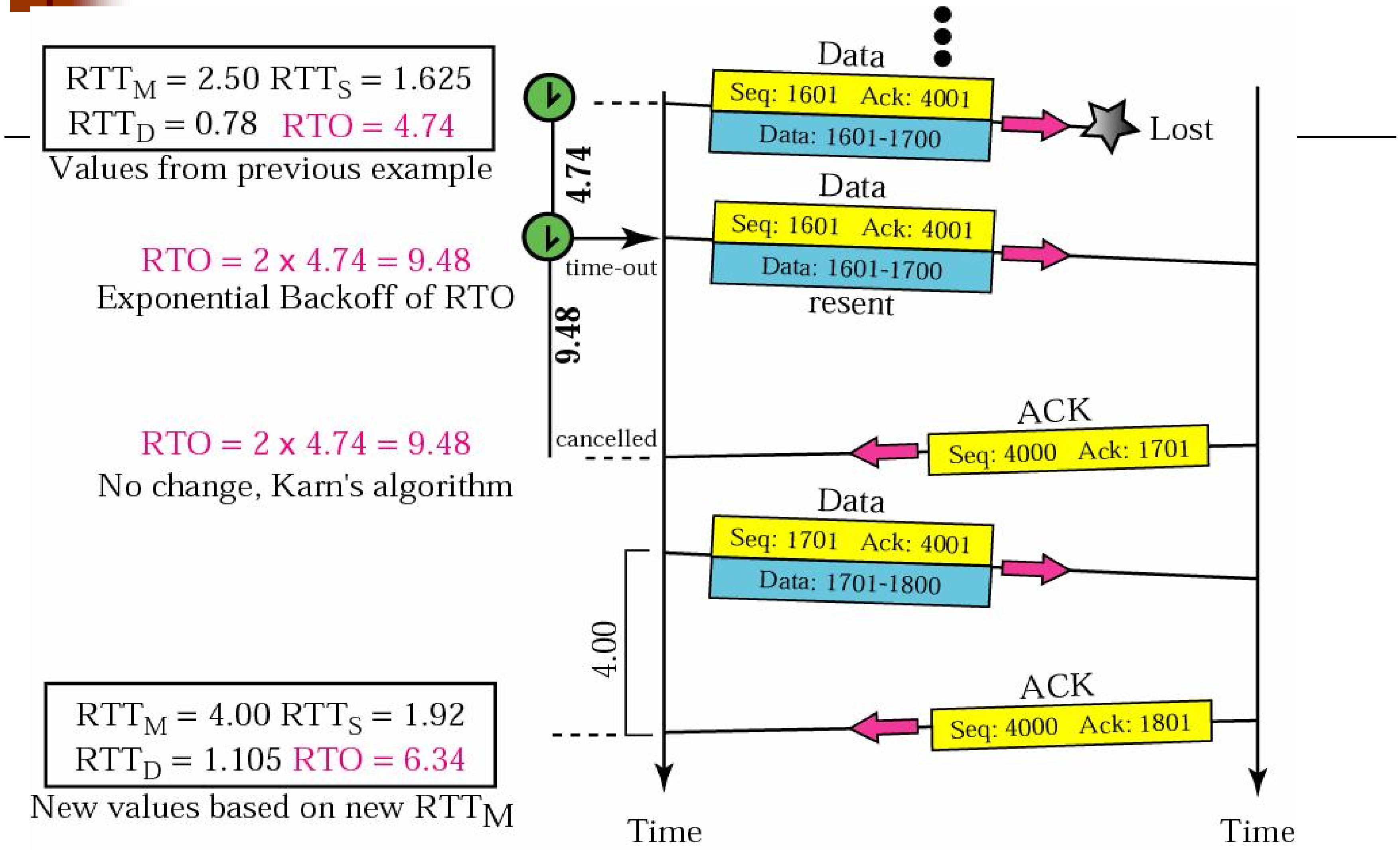
- What is the value of RTO if a retransmission occurs ?

- ***Exponential backoff*** in TCP
  - RTO is double for each retransmission

Example

- Figure 12.39 is a continuation of the previous example
- There is retransmission and Karn's algorithm is applied.
- The first segment in the figure is sent, but lost.
  - The RTO timer expires after 4.74 seconds.
  - The segment is retransmitted and the timer is set to 9.48, twice the previous value of RTO.
- This time an ACK is received before the time-out.
  - Wait until we send a new segment and receive the ACK for it before recalculating the RTO (Karn's algorithm).

Figure 12.39 *Example 11*



Persistence Timer

- TCP needs another timer to deal with the *zero window-size advertisement*
- Example
  - Receiving TCP announces a window size of zero
  - The sending TCP stops transmitting segments
  - After a while, receiving TCP sends an acknowledgment announcing a non-zero window size
    - However, this acknowledgment was lost
  - As a result, both sender and receiver continue to wait for each other forever

Persistence Timer (Cont.)

- Solution: TCP uses a persistence timer for each connection
- When the sending TCP receives an acknowledgment with a window size of zero
  - It starts a persistence timer
- When the timer goes off
  - The sending TCP sends a special segment called a ***probe***
    - Contain only 1 byte of data and is never acknowledged
    - Alert the receiving TCP that the acknowledgment may by lost and should be resent

Persistence Timer (Cont.)

- Value of persistence timer is set to the value of the retransmission timer
- However, if a response is not received from the receiver
  - Another probe segment is sent
  - The value of the persistence timer is double
- Above process is repeated until the persistence timer reaches a threshold
  - Usually 60 seconds

Keepalive Timer

- If a client has crashed
  - A TCP connection will be remain open forever
- Solution: Keepalive timer
  - The time-out is usually 2 hours
  - If a server does not hear from the client after two hours
    - Send a probe segment
  - If there is no response after 10 probes, each of which is 75 seconds apart
    - It assumes that client is down and terminates the connection

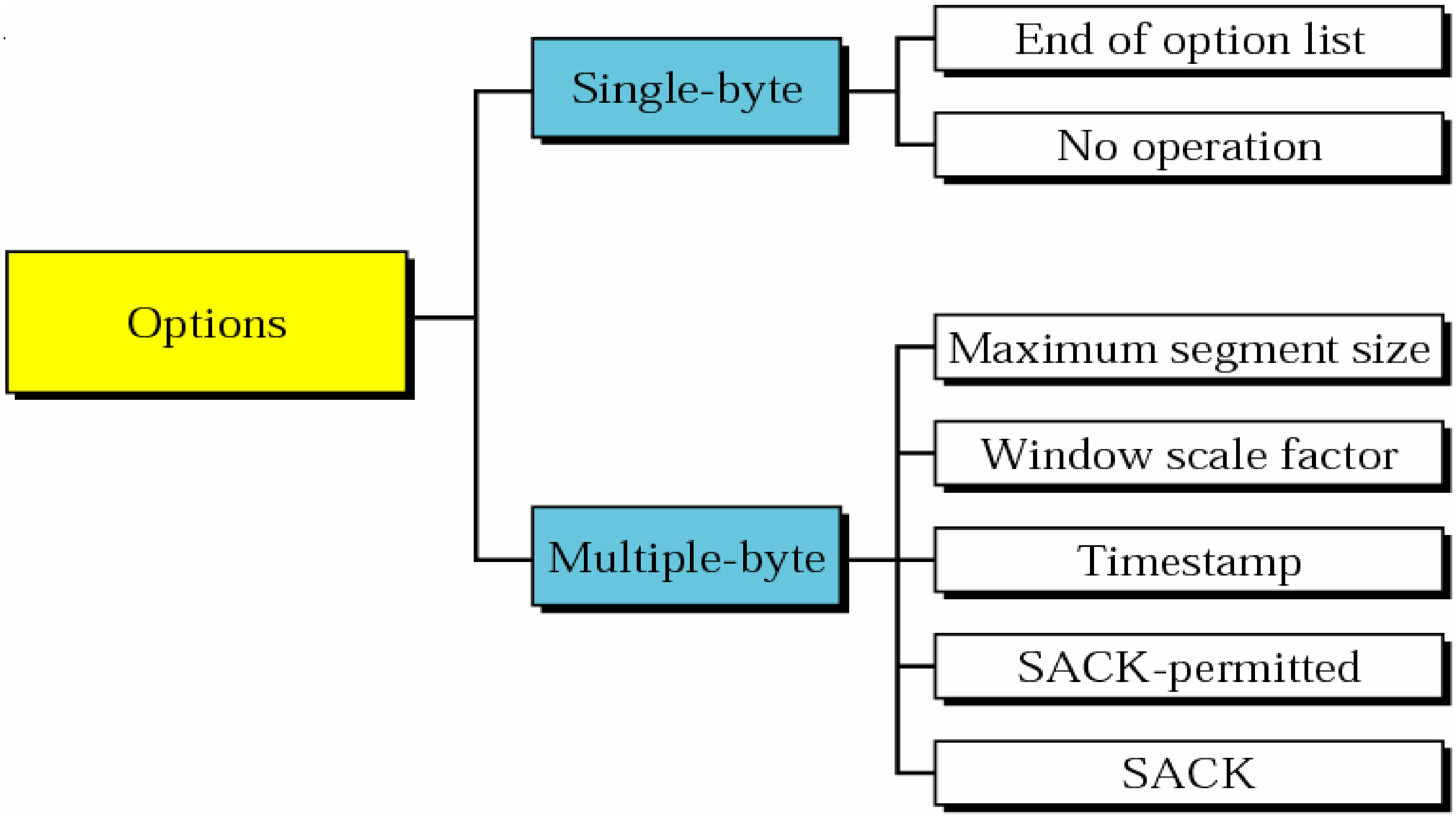
Time-Waited Timer

- Used during connection termination
- Mentioned later but ignore

12.10

OPTIONS

Options

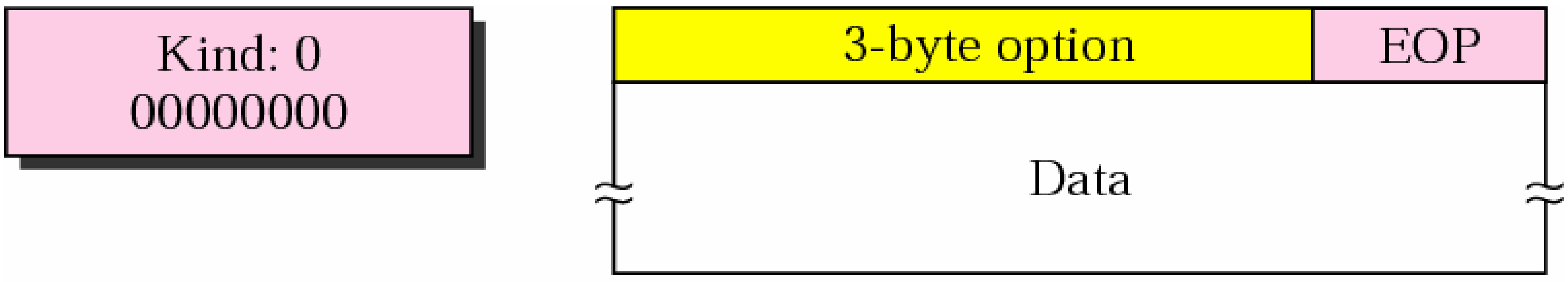


Options
Single-byte
Multiple-byte
End of option list
No operation
Maximum segment size
Window scale factor
Timestamp
SACK-permitted
SACK

End of Option

- Used for padding at the end of the option field
  - Can only be used as the *last option*
  - *Can be used only once*
- Only one end of option can be used
  - If more than 1 byte is needed to align the option field, use some *no-operation option* followed by an *end of option*
- Three pieces of information to the destination
  - No more options in the header
  - Data from the application program starts at the beginning of the next 32-bit word

End of option Option



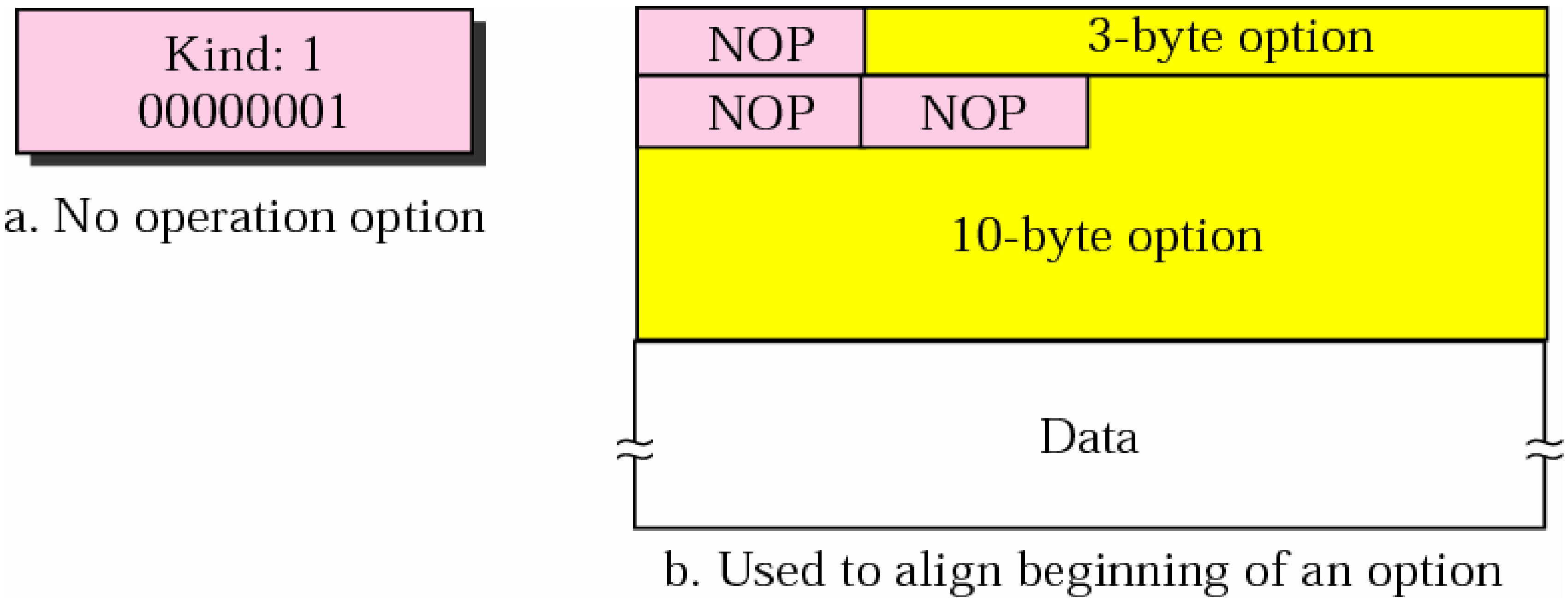
a. End of option list

b. Used for padding

No Operation

- Used to align the next option on a 32-bit boundary

No operation Option



Maximum Segment Size (MSS)

- Define the size of the biggest chunk of data that can be received by the destination
- Notably, it actually defines the maximum size of data, not the maximum size of segment
- Determined during the connection establishment phase
  - Once determined, it does not change during the connection
  - If neither party defines the size, the default is chosen
  - Default value is 536

Maximum segment size Option

Kind: 2 00000010	Length: 4 00000100	Maximum segment size
1 byte	1 byte	2 bytes

Window Scale Factor

- The *window size* field in the header defines the size of the sliding window
  - The size of the windows: 0 ~ 65535
  - However, it may not be sufficient in some networks
- To increase the window size, the *window scale factor* is used
  - New window size = *window size defined in the header* x $2^{\textit{window scale factor}}$
- However, the window size cannot be greater than the maximum value for the sequence number

Window Scale Factor (Cont.)

- Window scale factor can be determined only during the connection setup phase
- Thus, during data transfer, the size of the window may be changed
  - But it *must* be multiplied by the same scale factor
  - Cannot be changed during the connection
- The scale factor is also called ***shift count***
  - Multiplying a number by power of 2 = left shift

Figure 12-25

Window scale factor Option

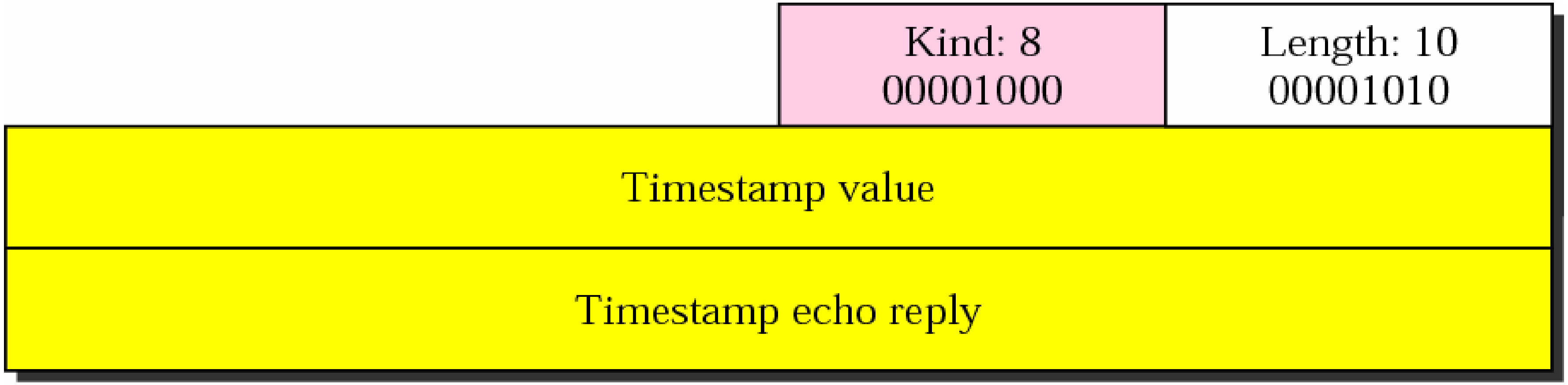
Kind: 3 00000011	Length: 3 00000011	Scale factor
1 byte	1 byte	1 byte

Timestamp

- Two applications
  - Measure the round trip time
  - Prevent wrap around sequence number

Figure 12-26

Timestamp Option



Measuring RTT

- Timestamp value field
  - Filled by the source when a segment leaves
- Timestamp echo reply field
  - When destination sends an acknowledgement, copy the received timestamp value into the *timestamp echo reply* field
- The source, when it receives acknowledgment
  - Calculate the *round-trip time*
- Thus, there is no need for clock synchronization
  - All calculation is based on the sender clock

Measuring RTT (Cont.)

- The receiver needs to keep two variables
  - *lastack*: the value of the last acknowledgment number sent
  - *tsrecent*: the value of recent timestamp that has not yet echoed

  - Detailed operation is shown in the next example

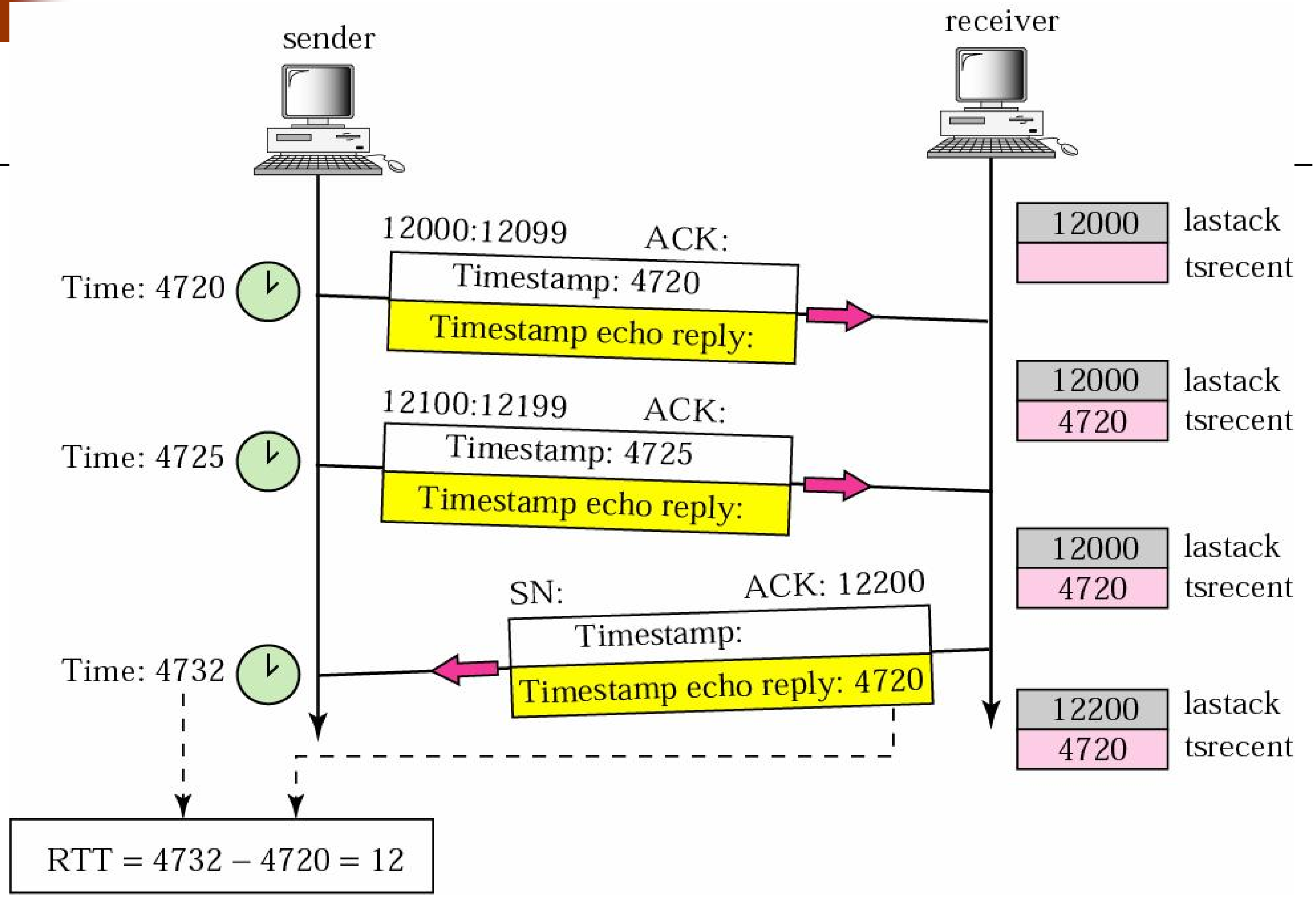
Example 12

- The sender simply inserts the value of its system clock in the *timestamp* field for the first and second segment.
- When an acknowledgment comes (the third segment)
  - The value of the clock is checked and the value of the echo reply field is subtracted from the current time.
  - RTT is *12 s* in this scenario.

Example 12 (Cont.)

- The receiver's function is more involved.
- It keeps track of the last acknowledgment sent (12000).
- When the first segment arrives (bytes 12000 to 12099)
  - The first byte is the same as the value of *lastack*.
  - Copy the timestamp value (4720) into the *tsrecent* variable.
- When the second segment arrives
  - None of the byte numbers in this segment include the value of *lastack*
    - The value of the timestamp field is ignored.
- When the receiver decides to send an accumulative acknowledgment with acknowledgment 12200
  - Changes the value of *lastack* to 12200
  - Inserts the value of *tsrecent* in the echo reply field.

Figure 12.46 *Example 12*



Example 12 (Cont.)

- In this example
  - RTT is calculated the time difference between *sending the first segment* and *receiving the third segment.*

- This is actually the meaning of RTT:
  - *The time difference between a packet sent and the acknowledgment received.*

PAWS

- Timestamp is also used for another application
  - ***Protection against wrapped around sequence number (PAWS)***
- Although sequence number is 32 bits
  - It could be wrapped around in a high-speed connection
  - $\boldsymbol{T}$=0, a sequence number is n
  - After $\boldsymbol{T}$=t, the sequence number is also n in the same connection

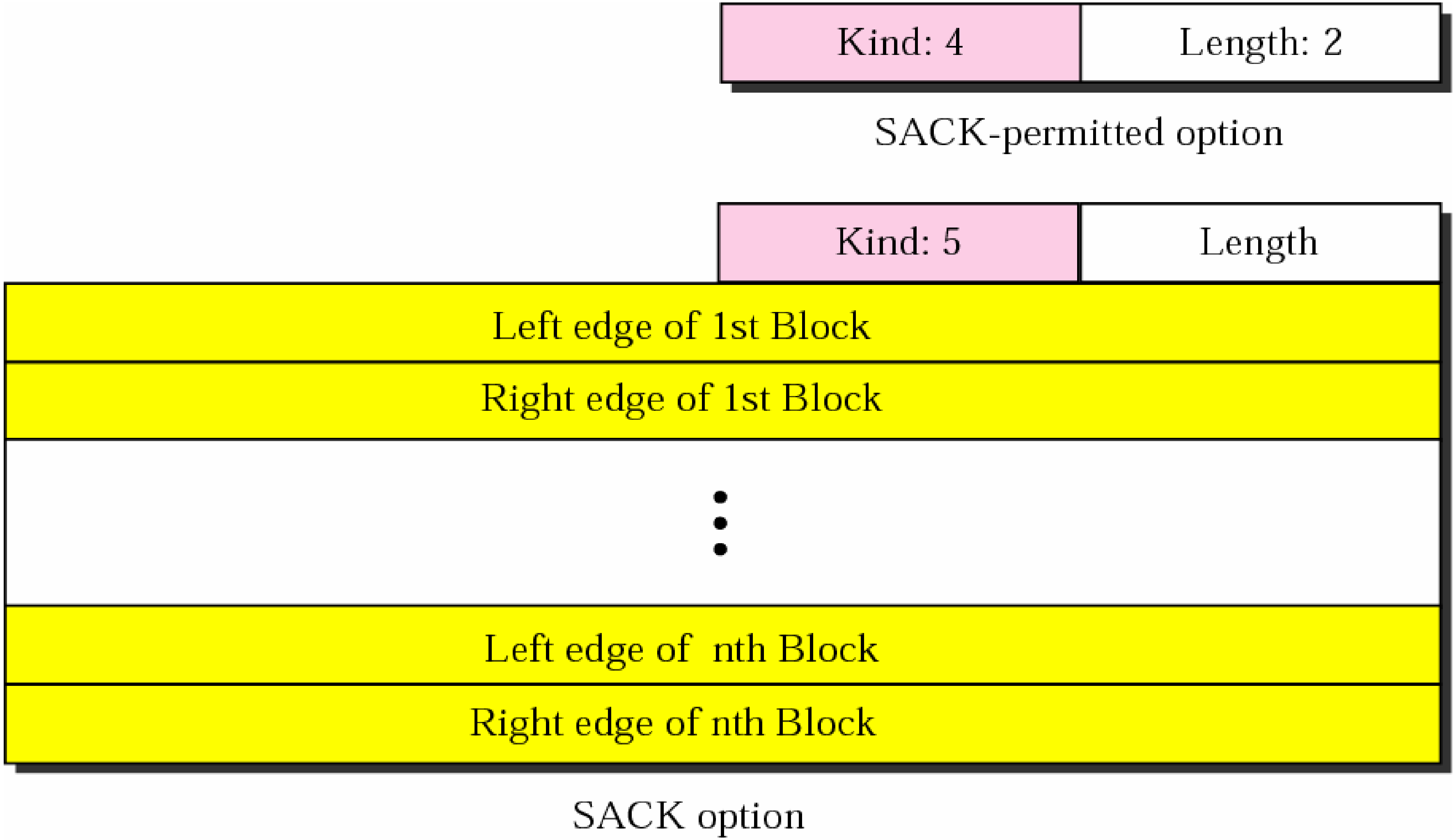
PAWS (Cont.)

- Problem:
  - If the fist segment is duplicated and arrives during the second round of sequence number
  - The segment will be wrongly considered belonging to the second run
- Solution:
  - *Increase the size of sequence number*
    - Change the window size and segment format
  - *Include the timestamp*
    - The identity of a segment is the combination of *timestamp* and *sequence number*

SACK-Permitted and SACK Options

- TCP's ACK is accumulative
- Problems
  - Does not report the bytes that have arrived out of order
  - Does not report about duplicate segments
- Solutions
  - *Selective acknowledgment (SACK)*
- Thus, two new options
  - SACK permitted
  - SACK

SACK


Kind: 4
Length: 2
SACK-permitted option
Kind: 5
Length
Left edge of 1st Block
Right edge of 1st Block
Left edge of nth Block
Right edge of nth Block
SACK option

SACK-Permitted Option

- Two bytes used ***only*** during connection establishment
  - Not allowed during the data transfer phase
- Sender
  - SYN segment with SACK-permitted option
- Receiver
  - SYN+ACK segment also with SACK-permitted option

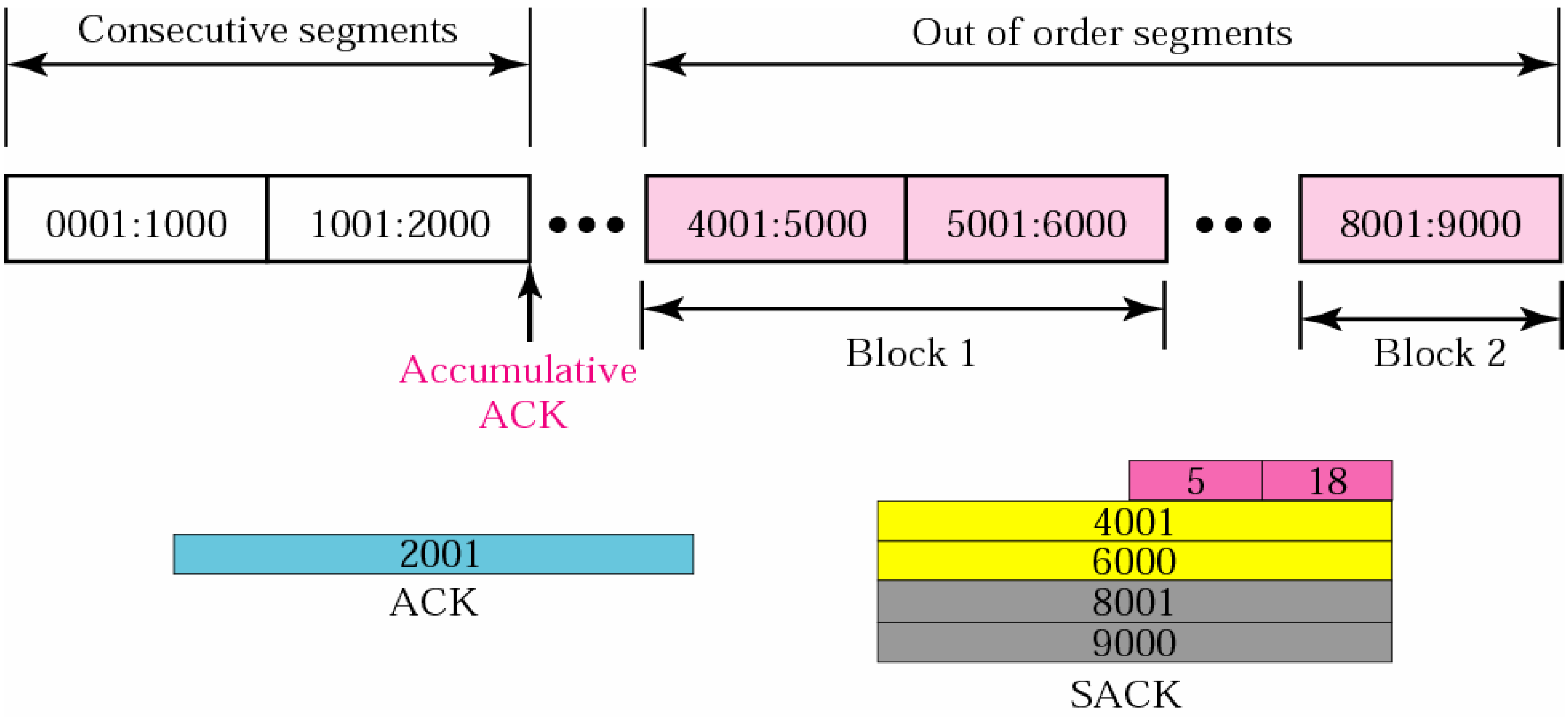
SACK Option

- Variable length
- Include a list of blocks arriving out-of-order
  - The first block can also be used to report the duplicates
  - Each block occupies two 32-bit number
    - Defining the beginning and the end of the block
  - SACK option cannot define more than 4 blocks
    - The allowed size of an option in TCP is 40 bytes
    - If 5 blocks, $(5 \times 2) \times 4 + 2 = 42 > 40$

Example 13

- The first and second segments are in consecutive order.
- Segments 3, 4, and 5 are out of order
  - A gap between the *second and third*
  - Another gap between the *fourth and the fifth*.
- An ACK and a SACK together can easily clear the situation for the sender.
  - The value of ACK is 2001
    - Sender need not worry about bytes 1 to 2000.
  - The SACK has two blocks.
    - The first block announces that bytes 4001 to 6000 have arrived out of order.
    - The second block shows that bytes 8001 to 9000 have also arrived out of order.
    - This means that bytes *2001 to 4000* and bytes *6001 to 8000* are lost or discarded.
      - The sender can resend only these bytes.

Example 13



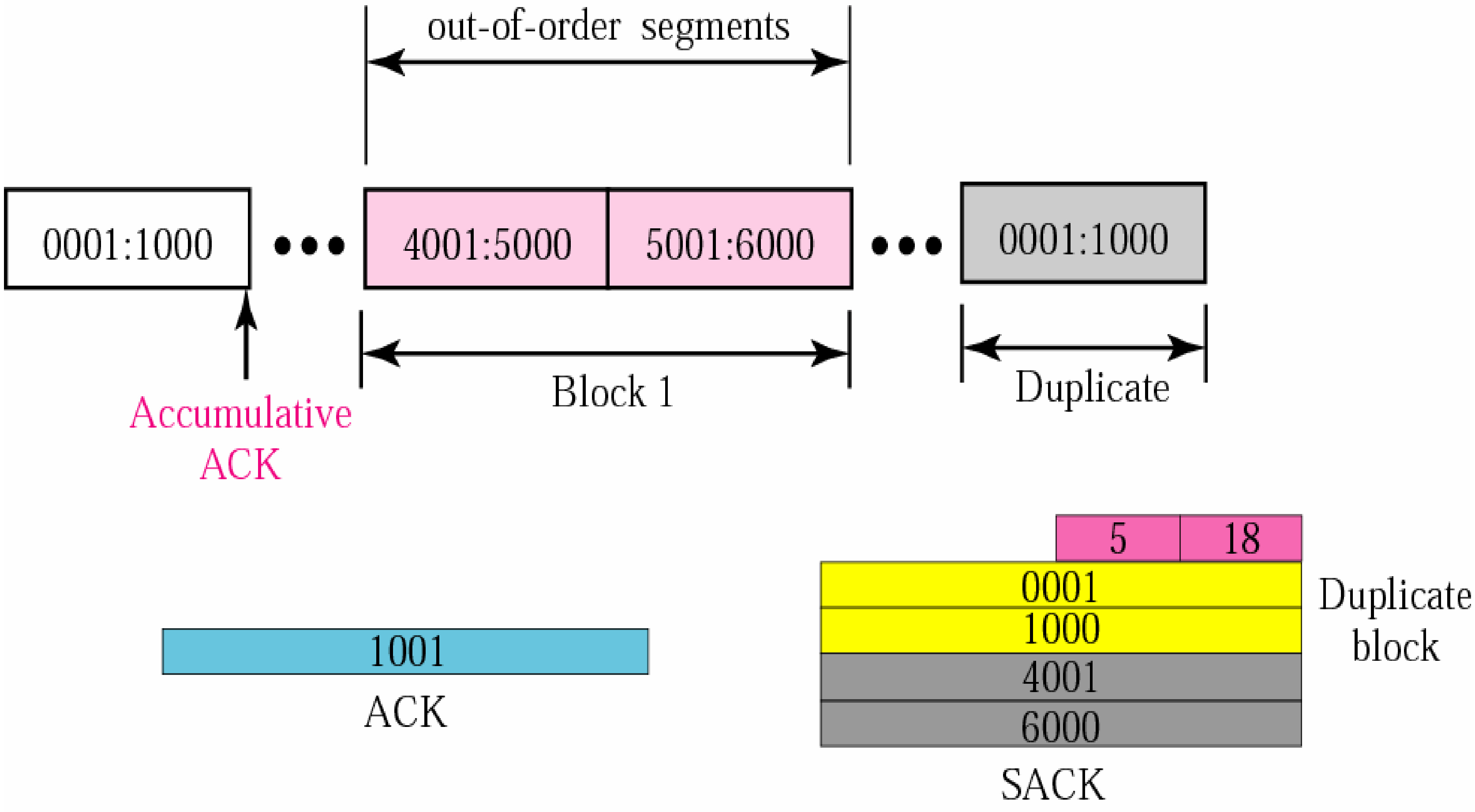
An end has received five segment of data

Example 14

- Figure 12.49 shows how a duplicate segment can be detected with a combination of ACK and SACK.

- In the figure, we have two out-of-order segments (in one block) and one duplicate segment.
  - SACK uses the first block to show the duplicate data
    - Note that only the first block can be used for duplicate data.
  - The other blocks to show out-of-order data.

Example 14

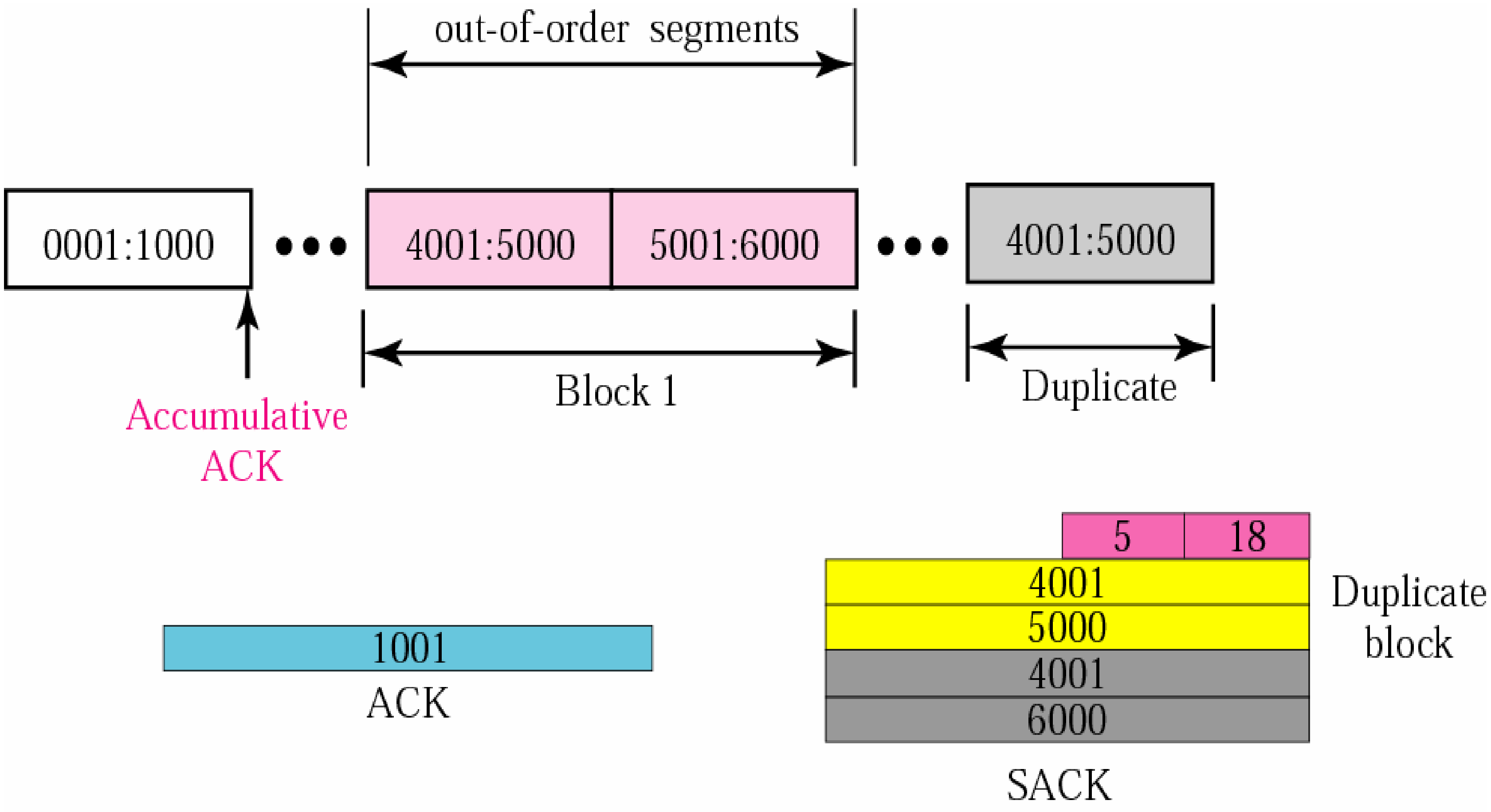


out-of-order segments
0001:1000
4001:5000
5001:6000
0001:1000
Block 1
Duplicate
Accumulative ACK
5
18
0001
1000
4001
6000
Duplicate block
1001
ACK
SACK

Example 15

- The example shows what happens if one of the segments in the out-of-order section is also duplicated.
- One of the segments (4001:5000) is duplicated.
  - The SACK option announces this duplicate data first
  - Then the out-of-order block.

Example 15



out-of-order segments
0001:1000
4001:5000
5001:6000
4001:5000
Block 1
Duplicate
Accumulative
ACK
5
18
4001
5000
4001
6000
Duplicate
block
1001
ACK
SACK