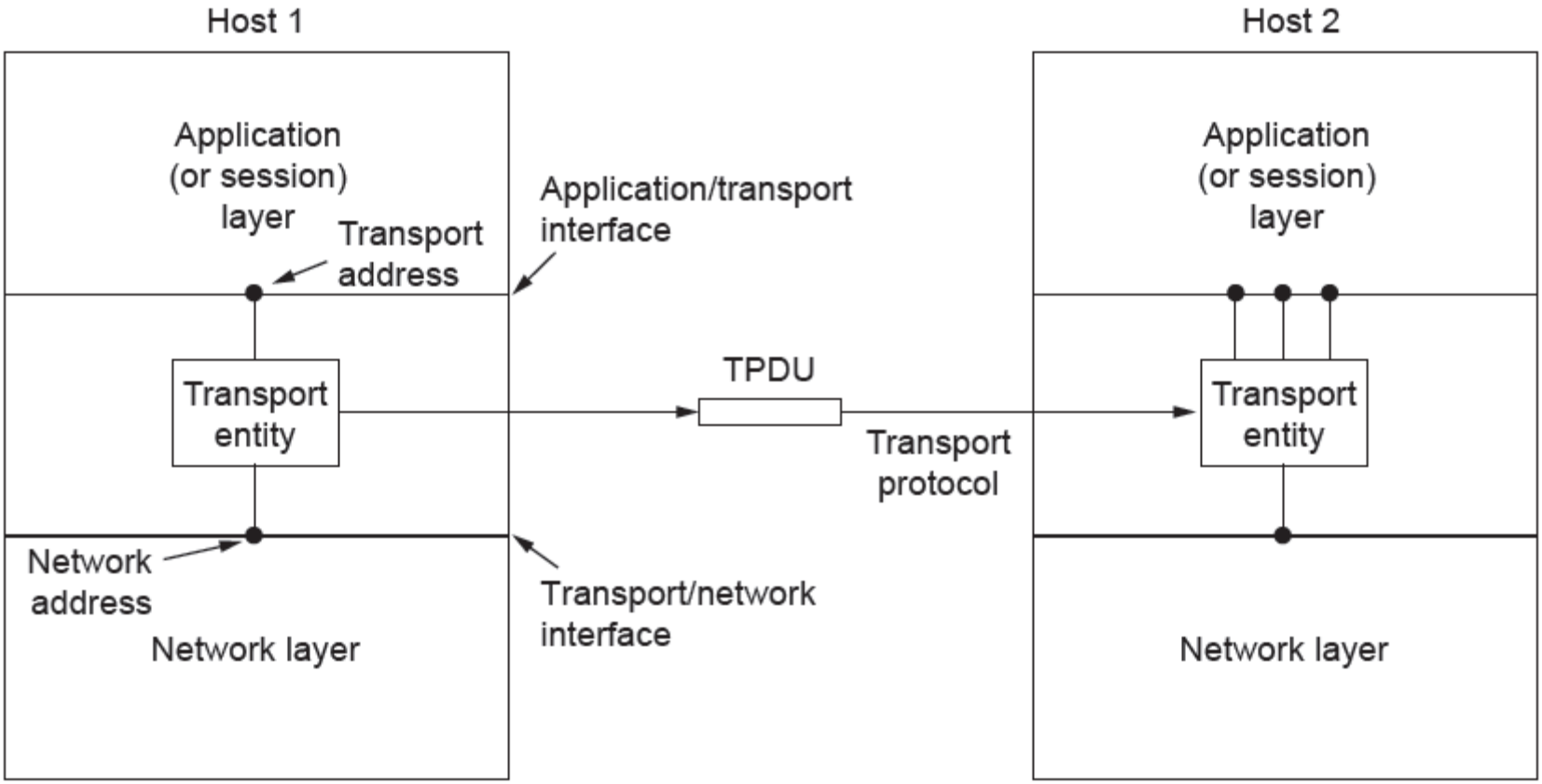
The Transport Layer

Chapter 6

Transport Service

- Upper Layer Services
- Transport Service Primitives
- Berkeley Sockets

Services Provided to the Upper Layers



The network, transport, and application layers

Services Provided to the Upper Layers

- Qualitative distinction between layers 1 through 4 and layer(s) above 4
- The bottom four layers can be seen as the **transport service provider**, whereas the upper layer(s) are the **transport service user**
- It forms the major boundary between the provider and user of the reliable data transmission service

Transport Service Primitives (1)

- To allow users to access the transport service, the transport layer must provide some operations to application programs, that is, a transport service interface
- Each transport service has its own interface
- The connection-oriented transport service, in contrast, is reliable. Of course, real networks are not error-free, but that is precisely the purpose of the transport layer—to provide a reliable service on top of an unreliable network
- As an example, consider two processes on a single machine connected by a pipe in UNIX (or any other interprocess communication facility). They assume the connection between them is 100% perfect. They do not want to know about acknowledgements, lost packets, congestion, or anything at all like that.
- The connection-oriented transport service is all about—hiding the imperfections of the network service so that user processes can just assume the existence of an error-free bit stream even when they are on different machines.
- The transport layer can also provide unreliable (datagram) service

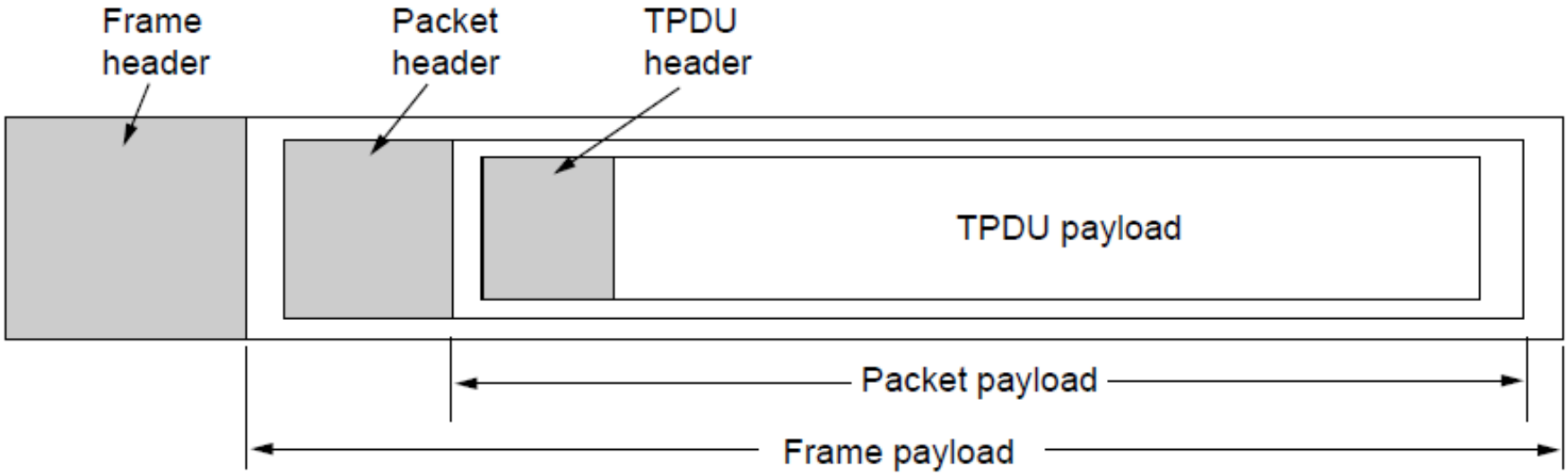
Transport Service Primitives (2)

- A second difference between the network service and transport service is whom the services are intended for
- Very few users write their own transport entities, and thus few users or programs ever see the bare network service. In contrast, many programs (and thus programmers wiritng socket applications) see the transport primitives

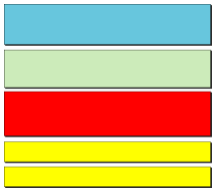
Primitive	Packet sent	Meaning
LISTEN	(none)	Block until some process tries to connect
CONNECT	CONNECTION REQ.	Actively attempt to establish a connection
SEND	DATA	Send information
RECEIVE	(none)	Block until a DATA packet arrives
DISCONNECT	DISCONNECTION REQ.	This side wants to release the connection

The primitives for a simple transport service

Transport Service Primitives (3)

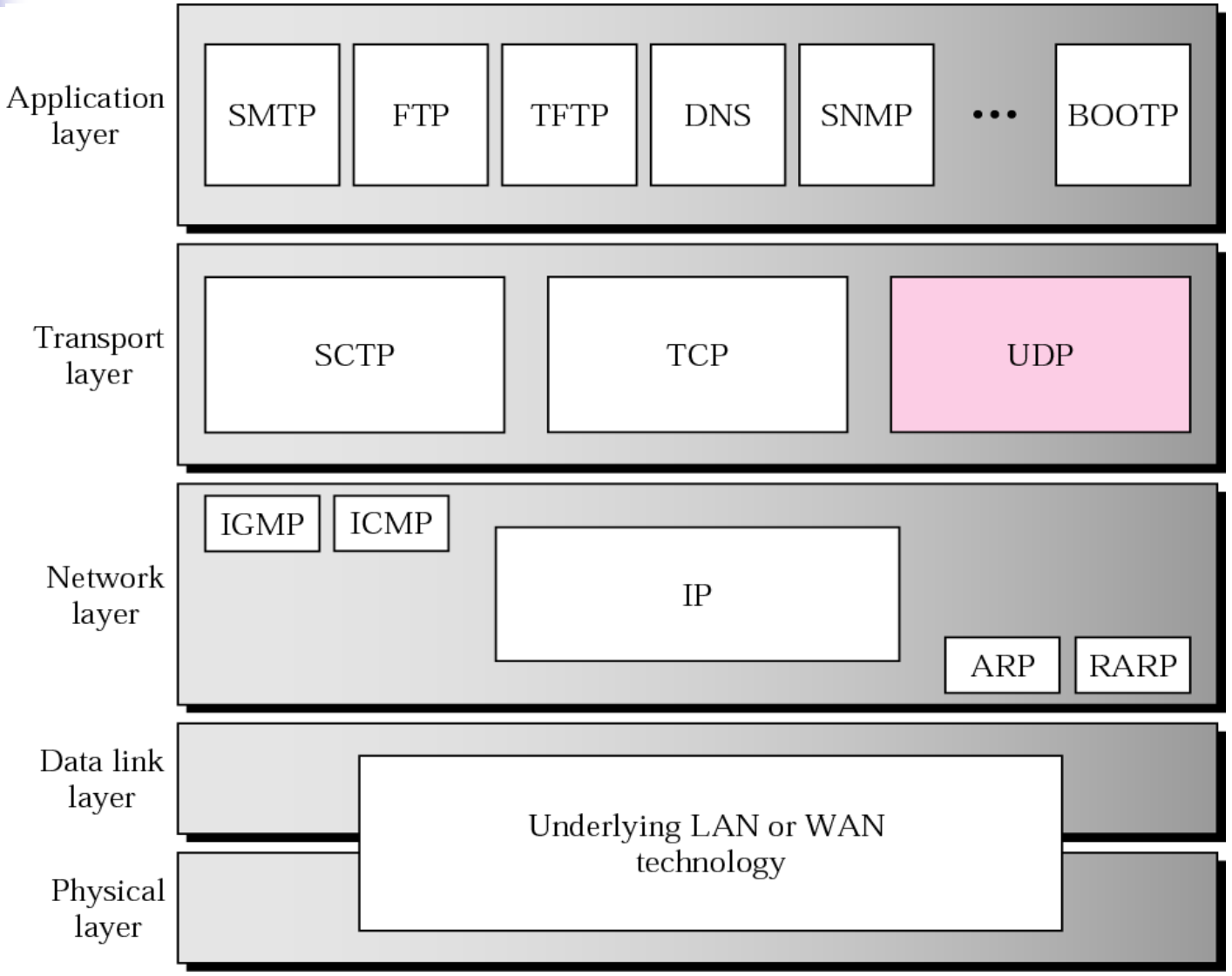


Nesting of TPDUs, packets, and frames.



User Datagram Protocol (UDP)

Figure 11.1 *Position of UDP in the TCP/IP protocol suite*



11.1 PROCESS-TO-PROCESS COMMUNICATION

Before we examine UDP, we must first understand host-to-host communication and process-to-process communication and the difference between them.

The topics discussed in this section include:

Port Numbers
Socket Addresses

Figure 11.2 *UDP versus IP*

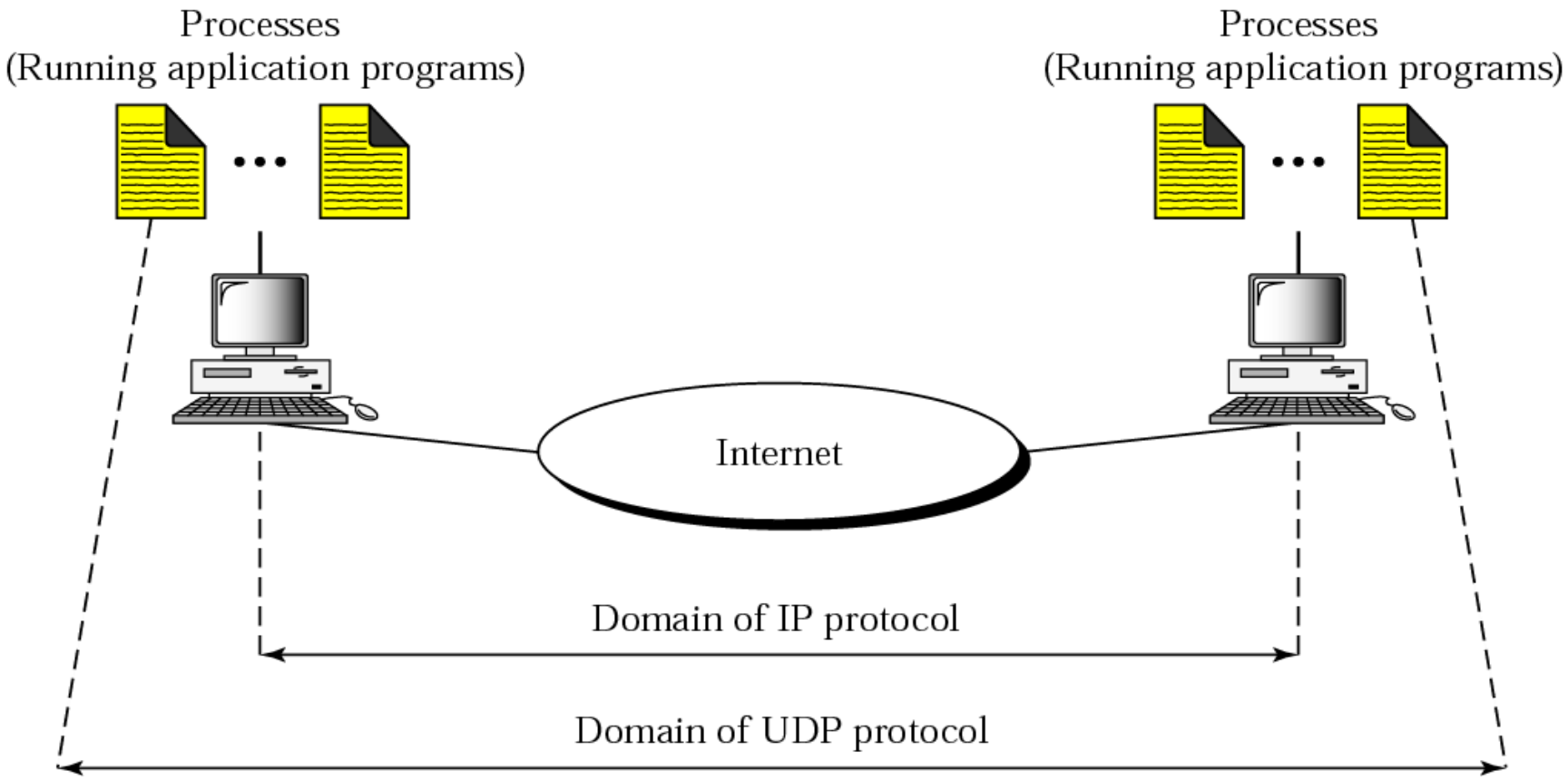


Figure 11.3 *Port numbers*

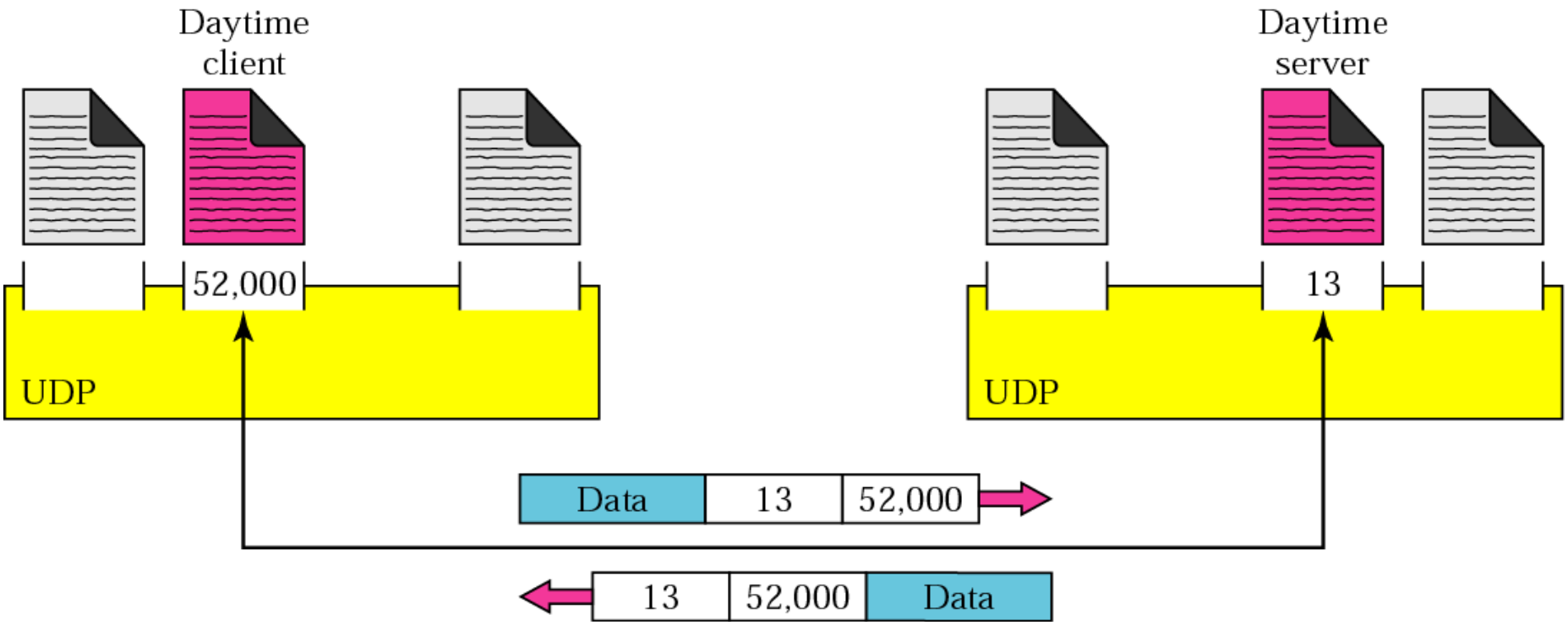


Figure 11.4 *IP addresses versus port numbers*

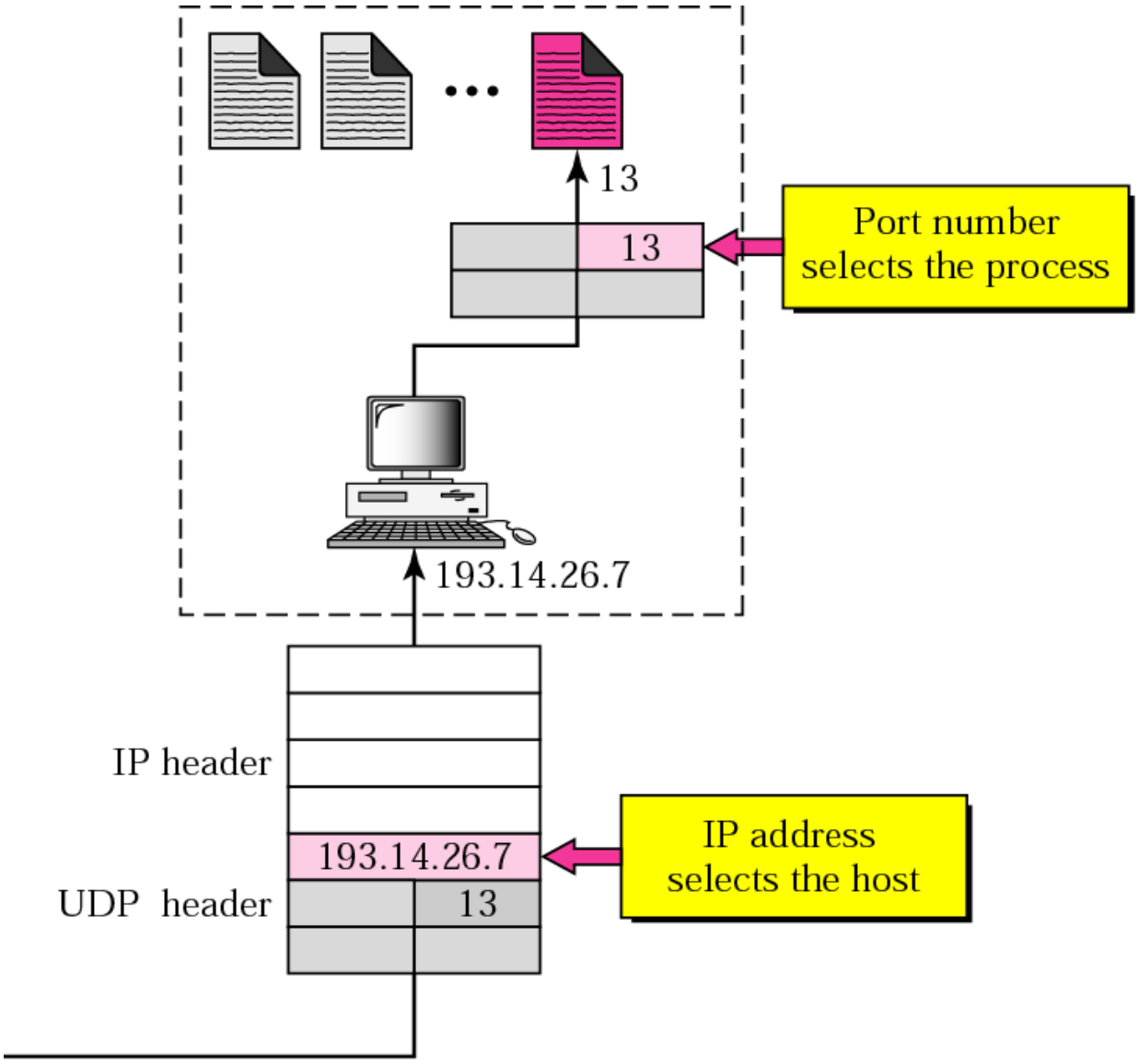
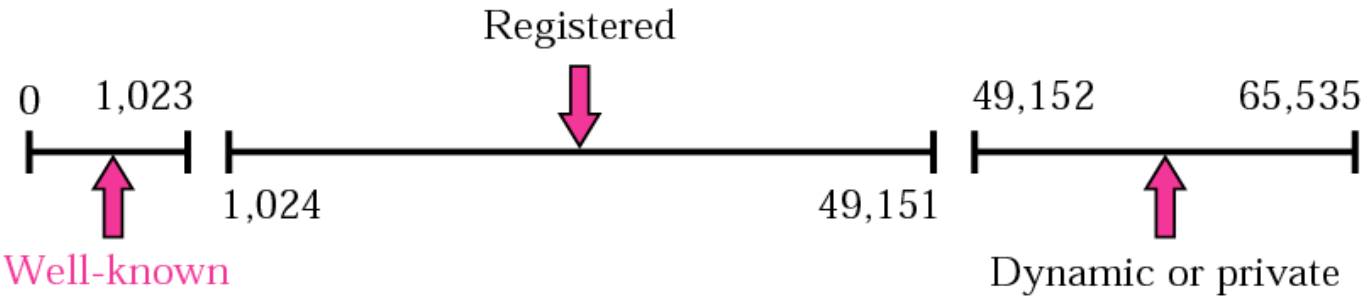


Figure 11.5 *ICANN ranges*



The well-known port numbers are less than 1024.

Table 11.1 Well-known ports used with UDP

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

EXAMPLE 1

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 port for TFTP. Note TFTP can use port 69 on either UDP or TCP.

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

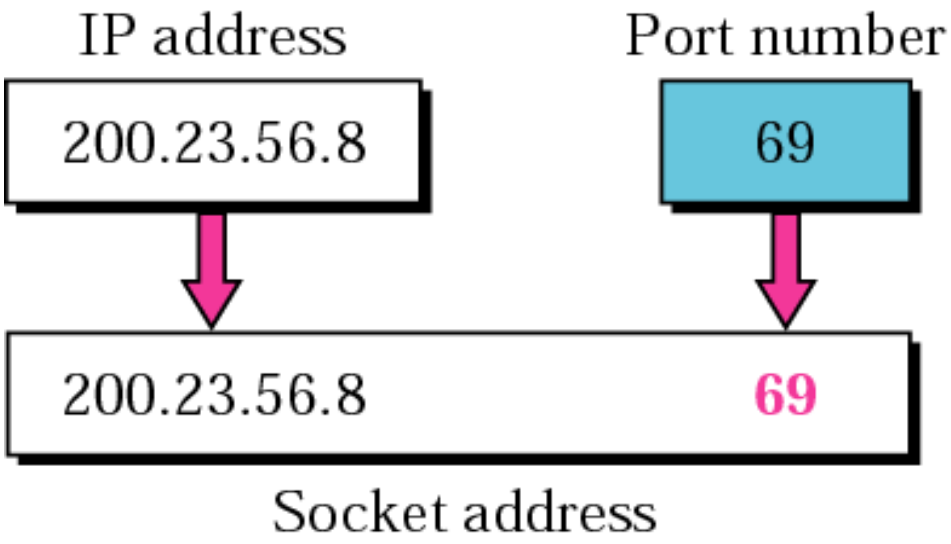
See Next Slide

EXAMPLE 1 (CONTINUED)

SNMP uses two port numbers (161 and 162), each for a different purpose, as we will see in Chapter 21.

```
$ grep snmp /etc/services
   snmp          161/tcp          #Simple Net Mgmt Proto
   snmp          161/udp          #Simple Net Mgmt Proto
   snmptrap      162/udp          #Traps for SNMP
```

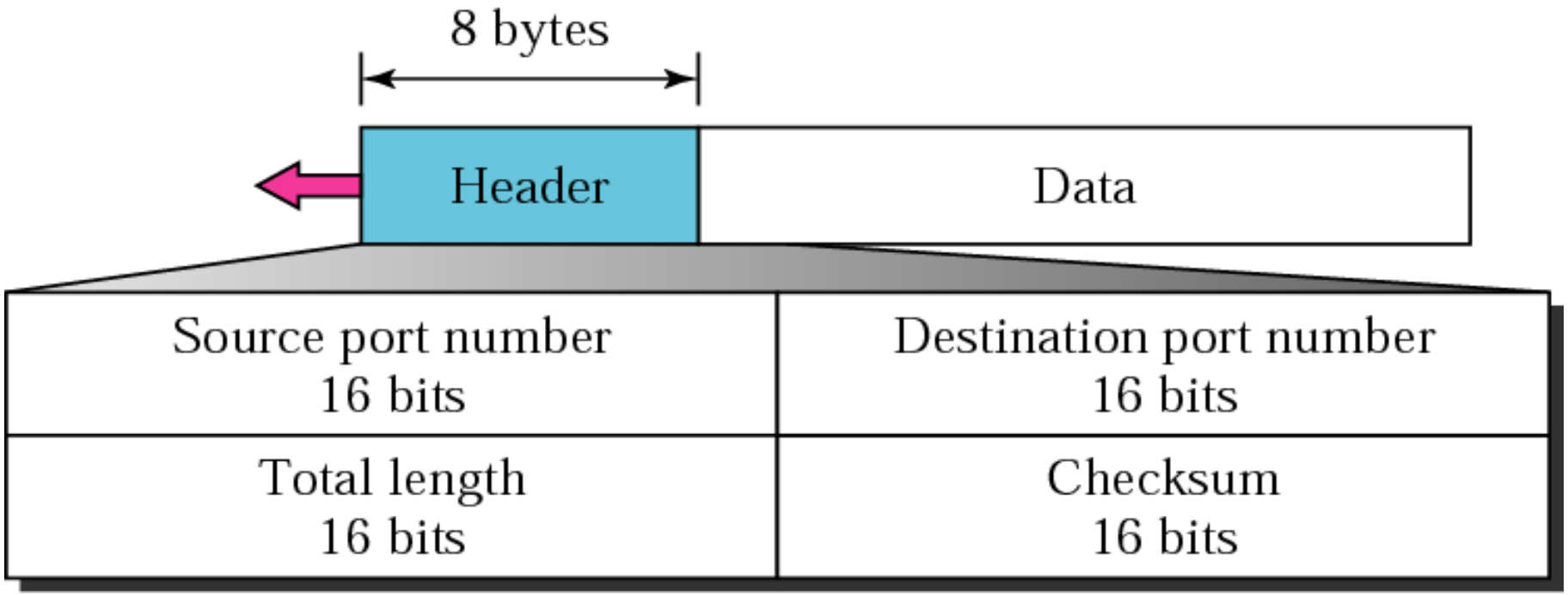
Figure 11.6 *Socket address*

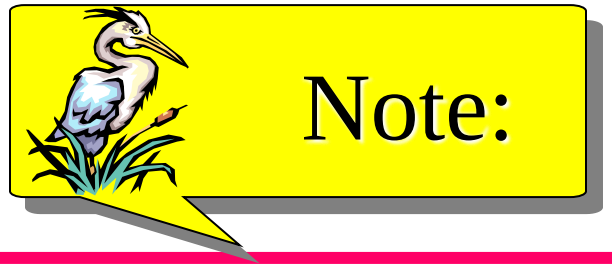


11.2 USER DATAGRAM

UDP packets are called user datagrams and have a fixed-size header of 8 bytes.

Figure 11.7 *User datagram format*





UDP length =
IP length − IP header's length

11.3 CHECKSUM

UDP checksum calculation is different from the one for IP and ICMP. Here the checksum includes three sections: a pseudoheader, the UDP header, and the data coming from the application layer.

The topics discussed in this section include:

Checksum Calculation at Sender
Checksum Calculation at Receiver
Optional Use of the Checksum

Figure 11.8 *Pseudoheader for checksum calculation*

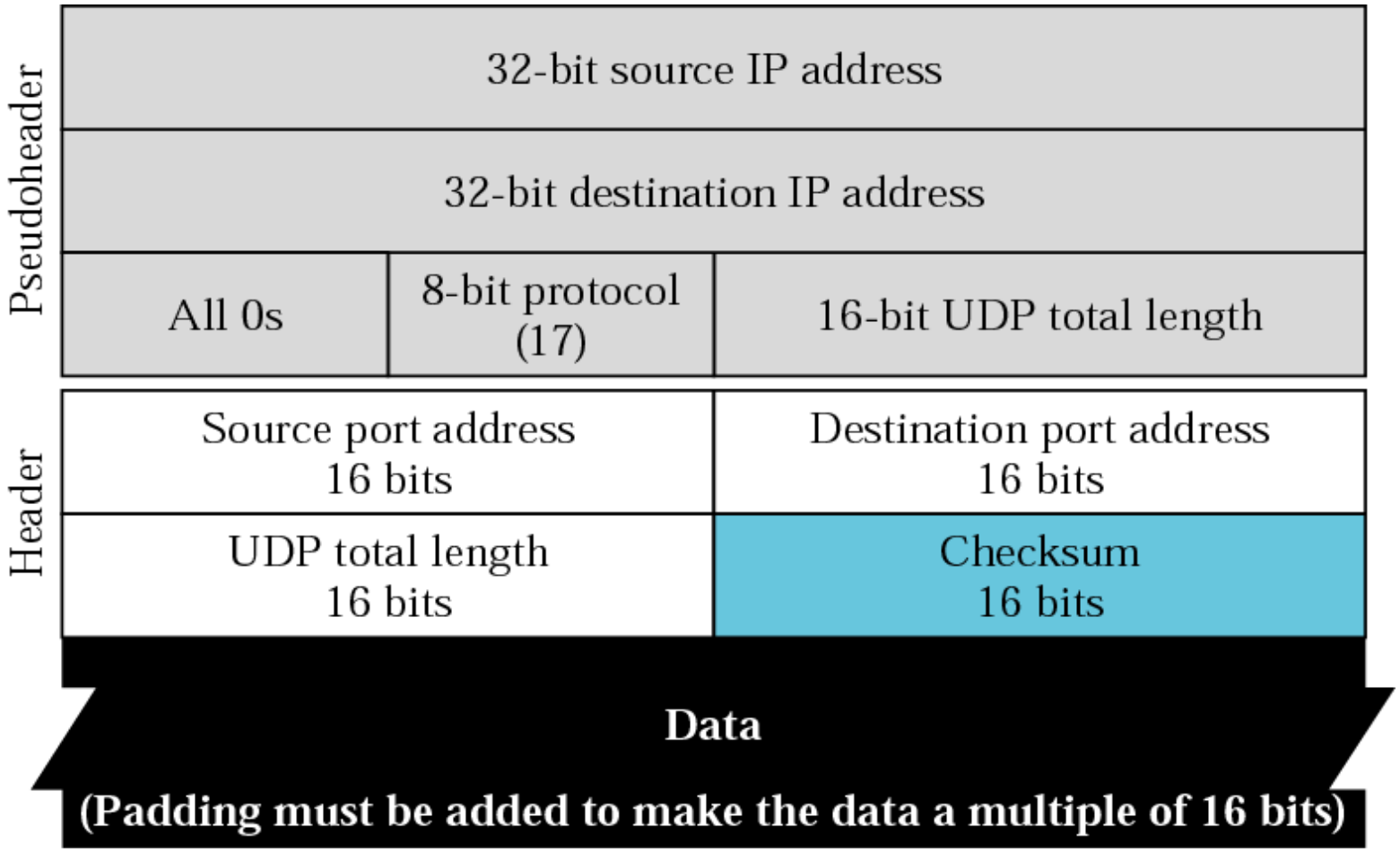
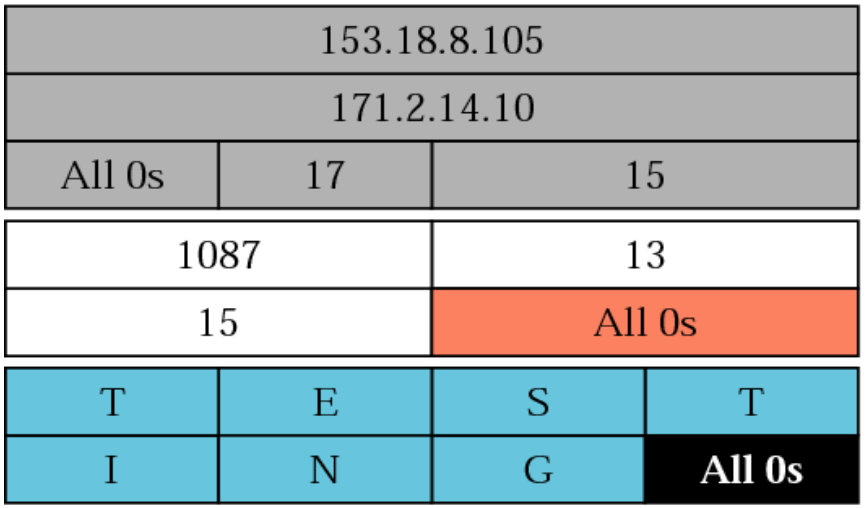


Figure 11.9 *Checksum calculation of a simple UDP user datagram*



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

11.4 UDP OPERATION

UDP uses concepts common to the transport layer. These concepts will be discussed here briefly, and then expanded in the next chapter on the TCP protocol.

The topics discussed in this section include:

Connectionless Services
Flow and Error Control
Encapsulation and Decapsulation
Queuing
Multiplexing and Demultiplexing

Figure 11.10 *Encapsulation and decapsulation*

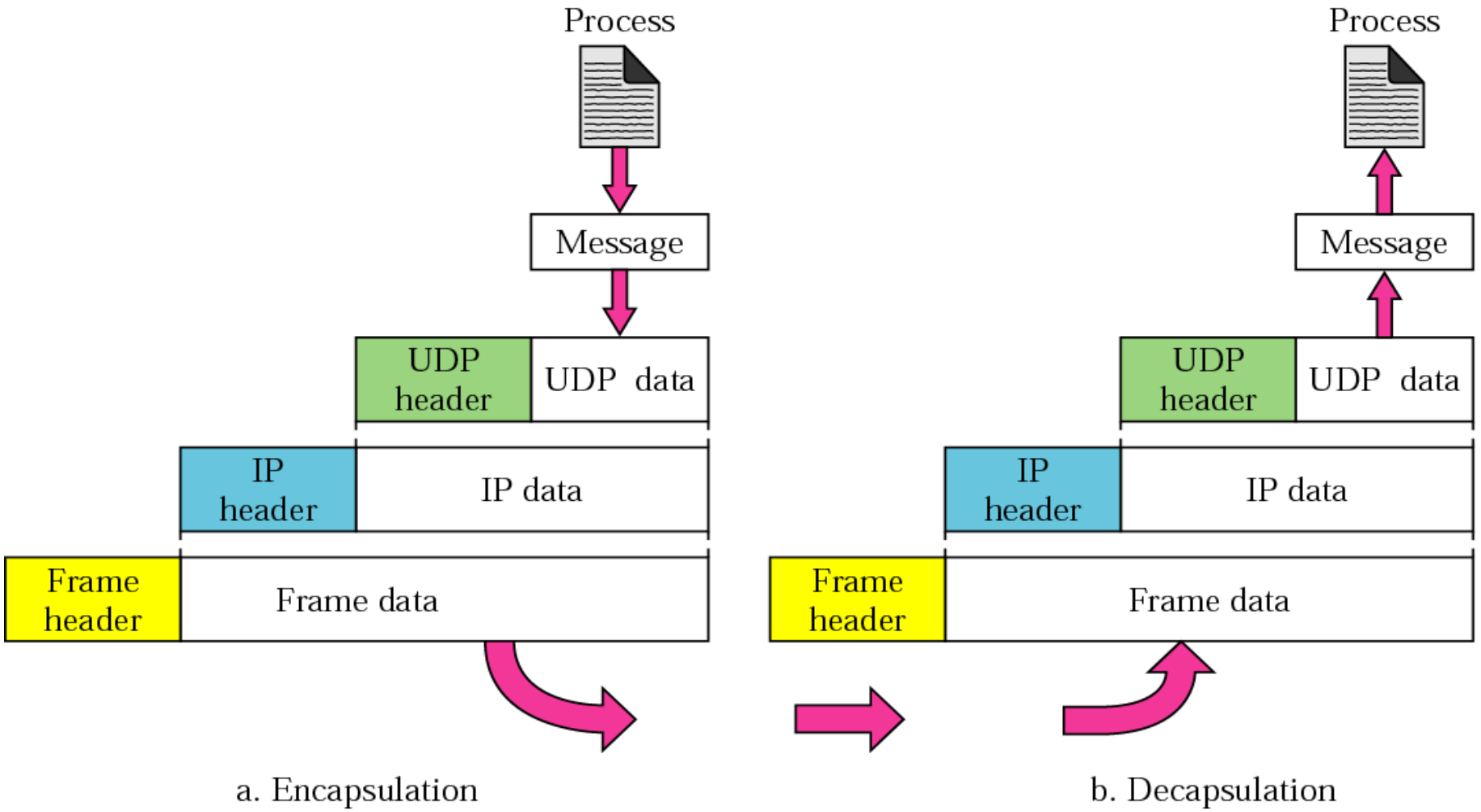


Figure 11.11 *Queues in UDP*

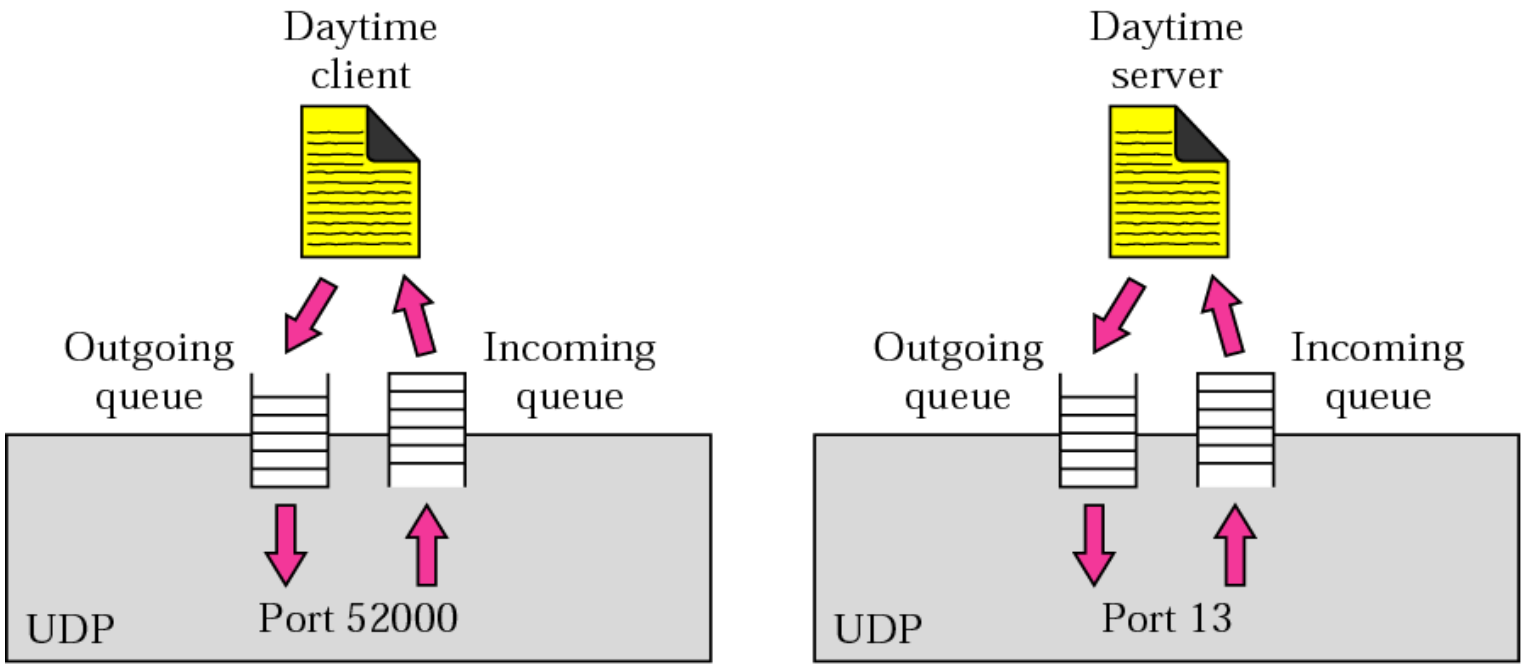
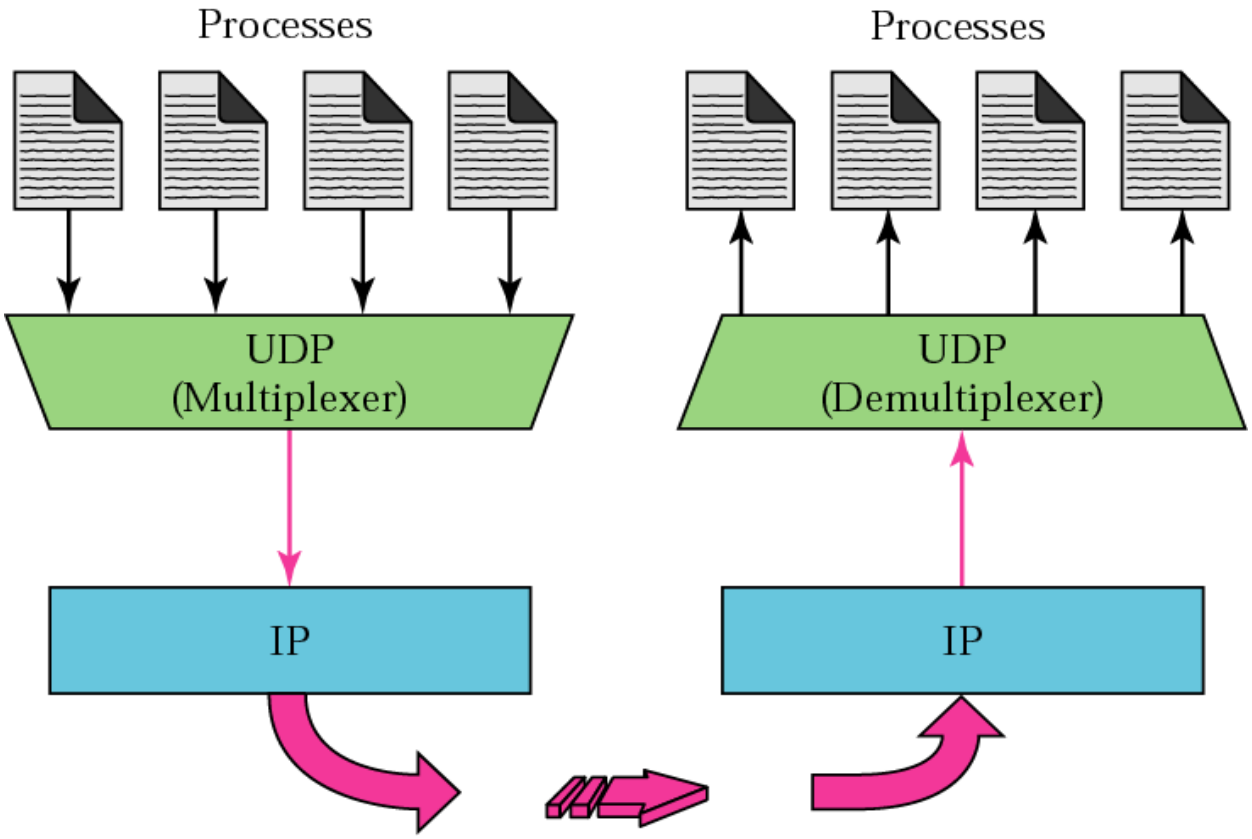
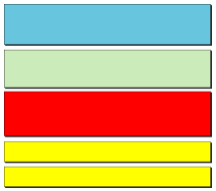


Figure 11.12 *Multiplexing and demultiplexing*



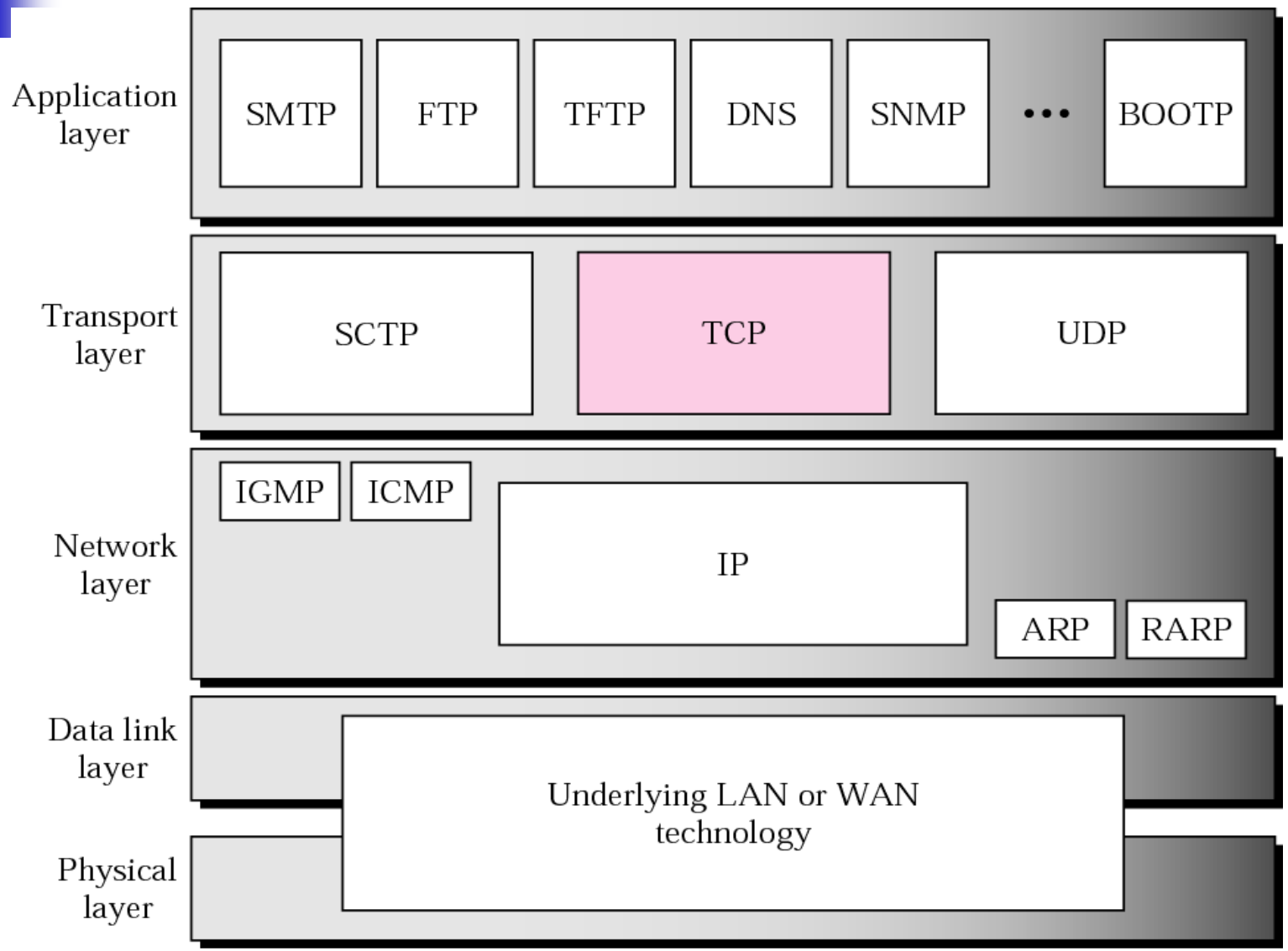
11.5 USE OF UDP

- *UDP is a low overhead protocol*
- *Primarily suitable for applications which can tolerate packet loss*
- *Suitable for applications which needs timely but may not be reliable delivery like multimedia applications*



Transmission Control Protocol (TCP)

Figure 12.1 *TCP/IP protocol suite*



12.1 TCP SERVICES

The services offered by TCP to the processes at the application layer.

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

Table 12.1 Well-known ports used by TCP

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

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
```

Figure 12.2 *Stream delivery*

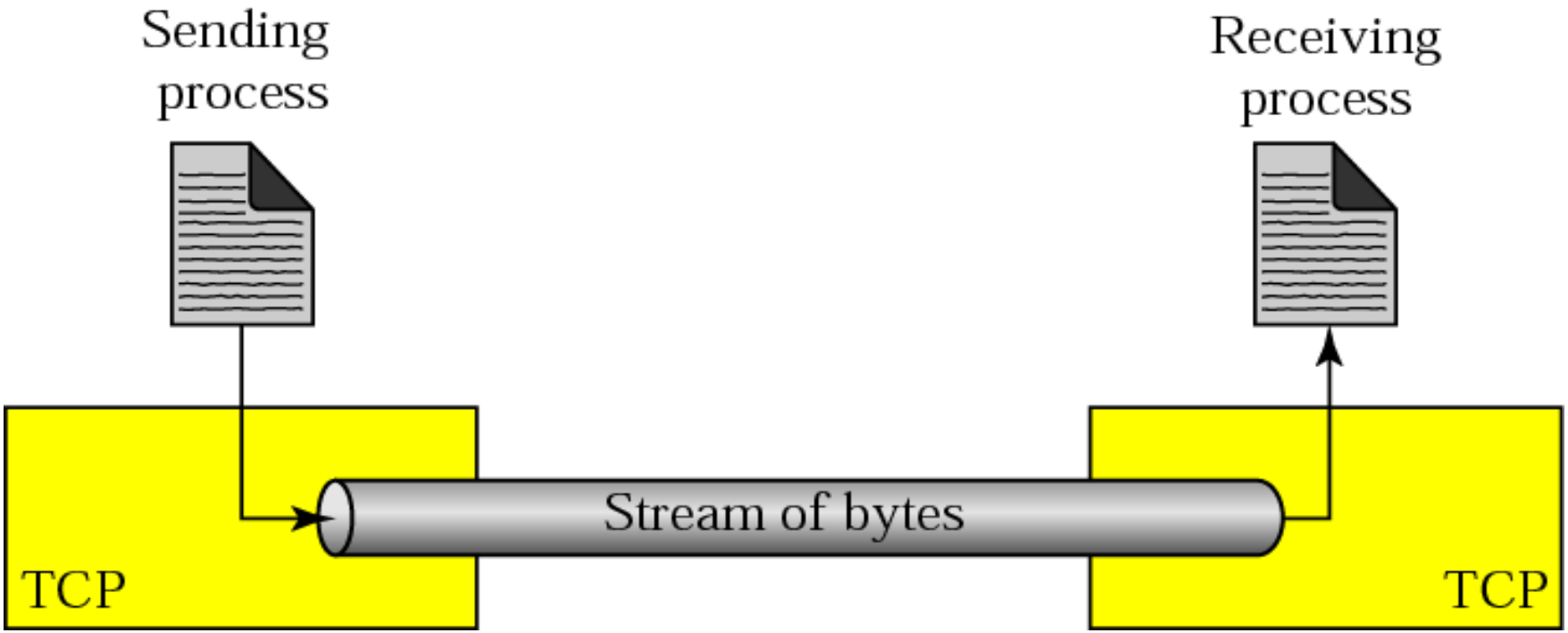


Figure 12.3 *Sending and receiving buffers*

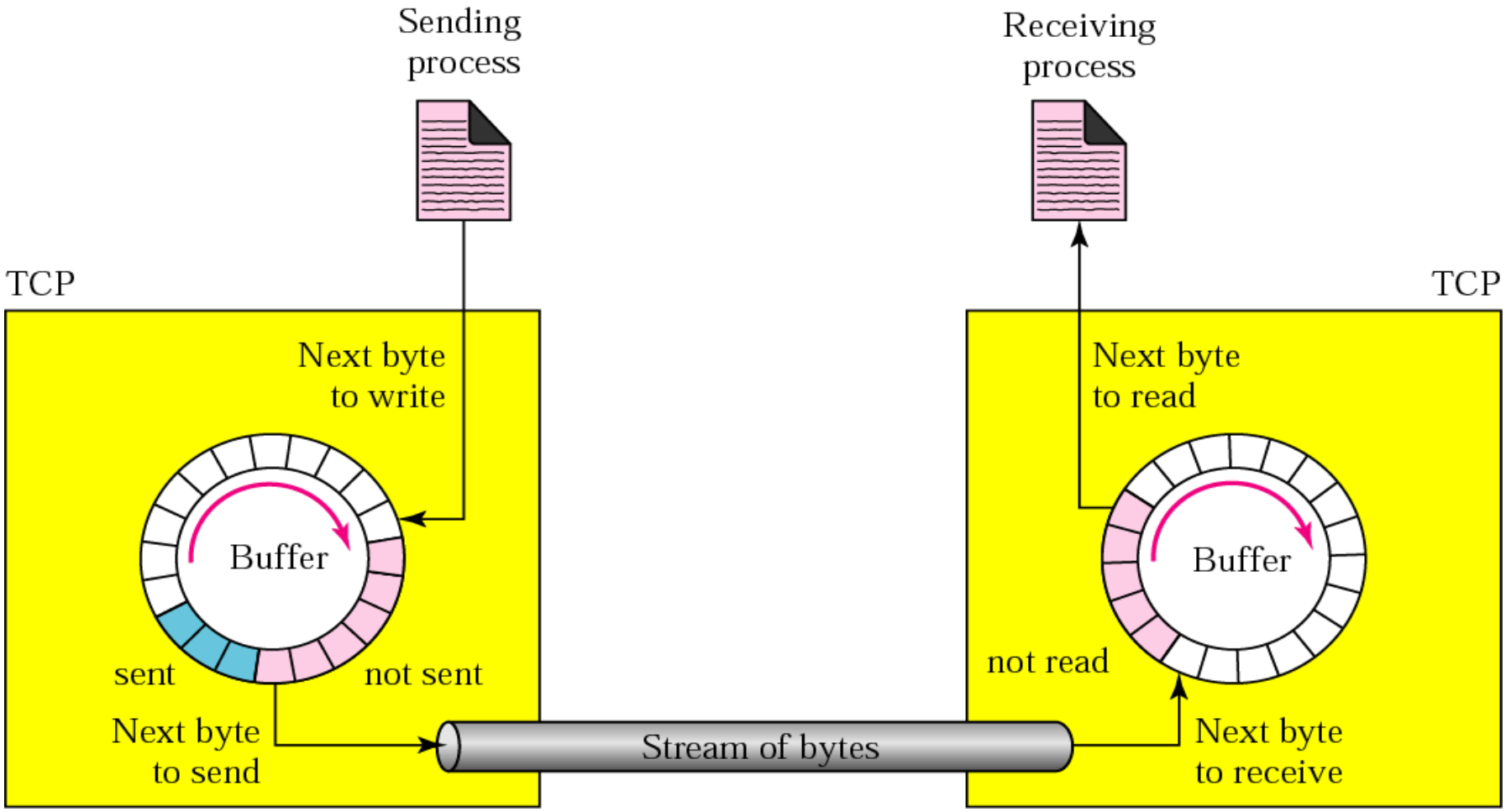
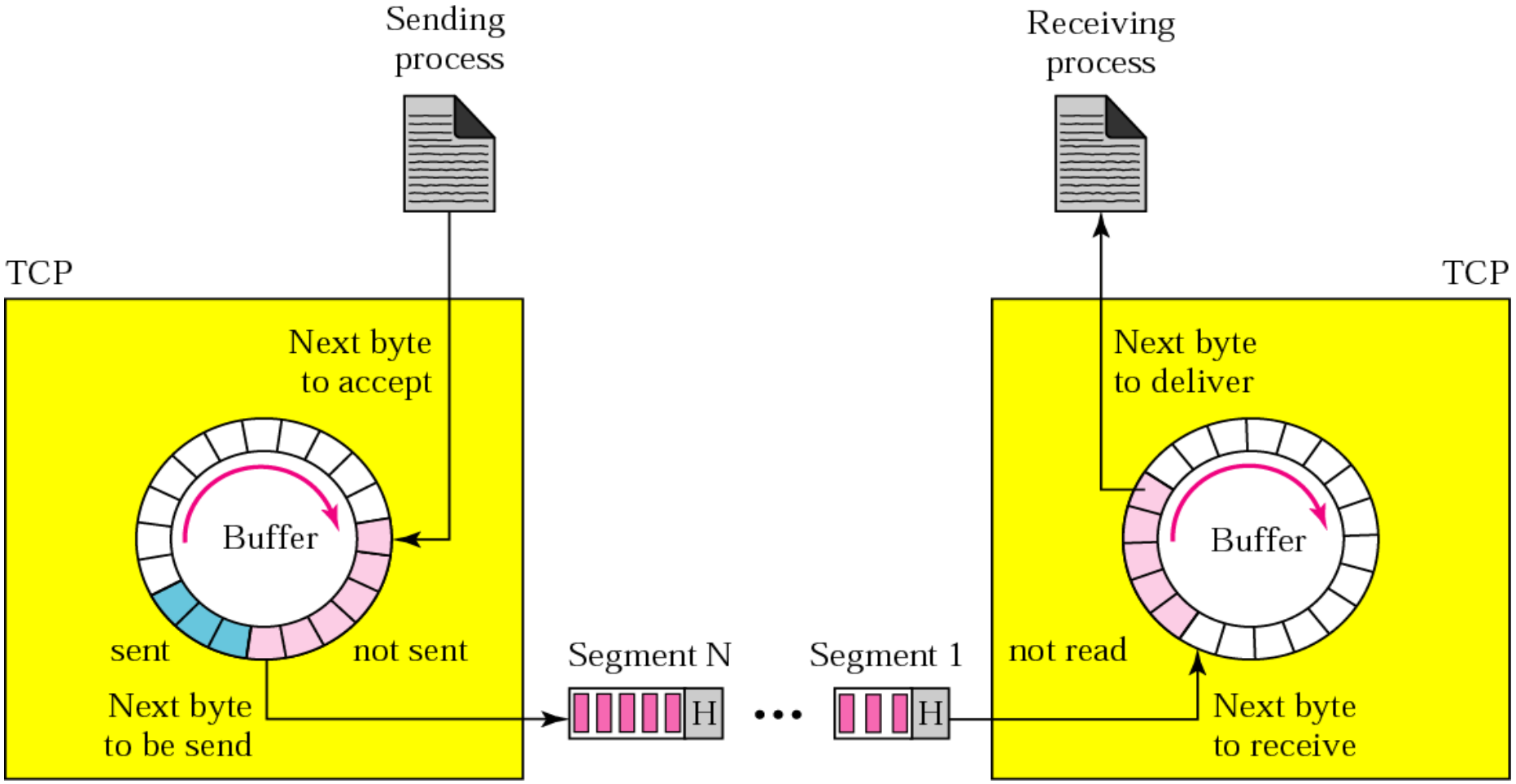


Figure 12.4 *TCP segments*



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

Note:

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

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:

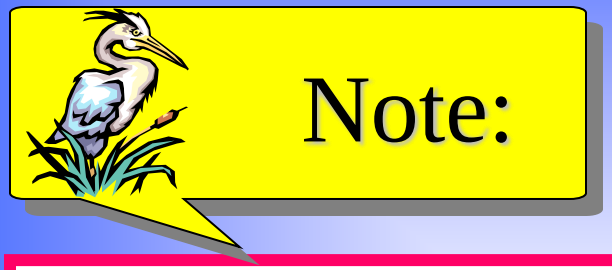
Segment 1 ➡ *Sequence Number: 10,001 (range: 10,001 to 11,000)*

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)*



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

Note:

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.

12.3 SEGMENT

A packet in TCP is called a segment

The topics discussed in this section include:

Format
Encapsulation

Figure 12.5 *TCP segment format*

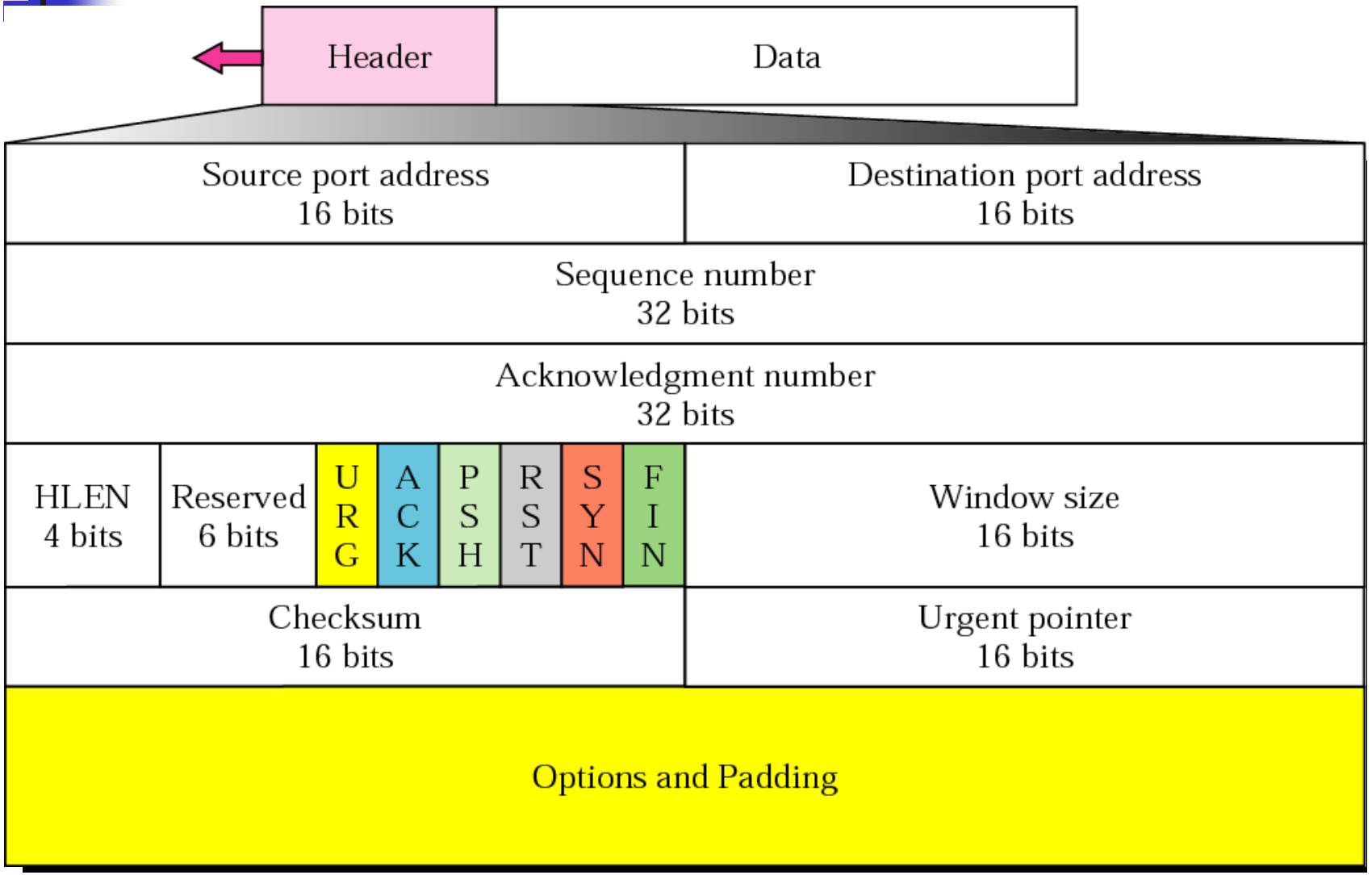


Figure 12.6 *Control field*

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

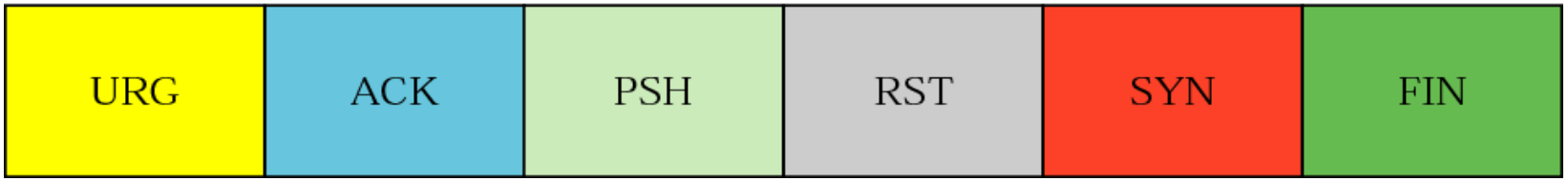
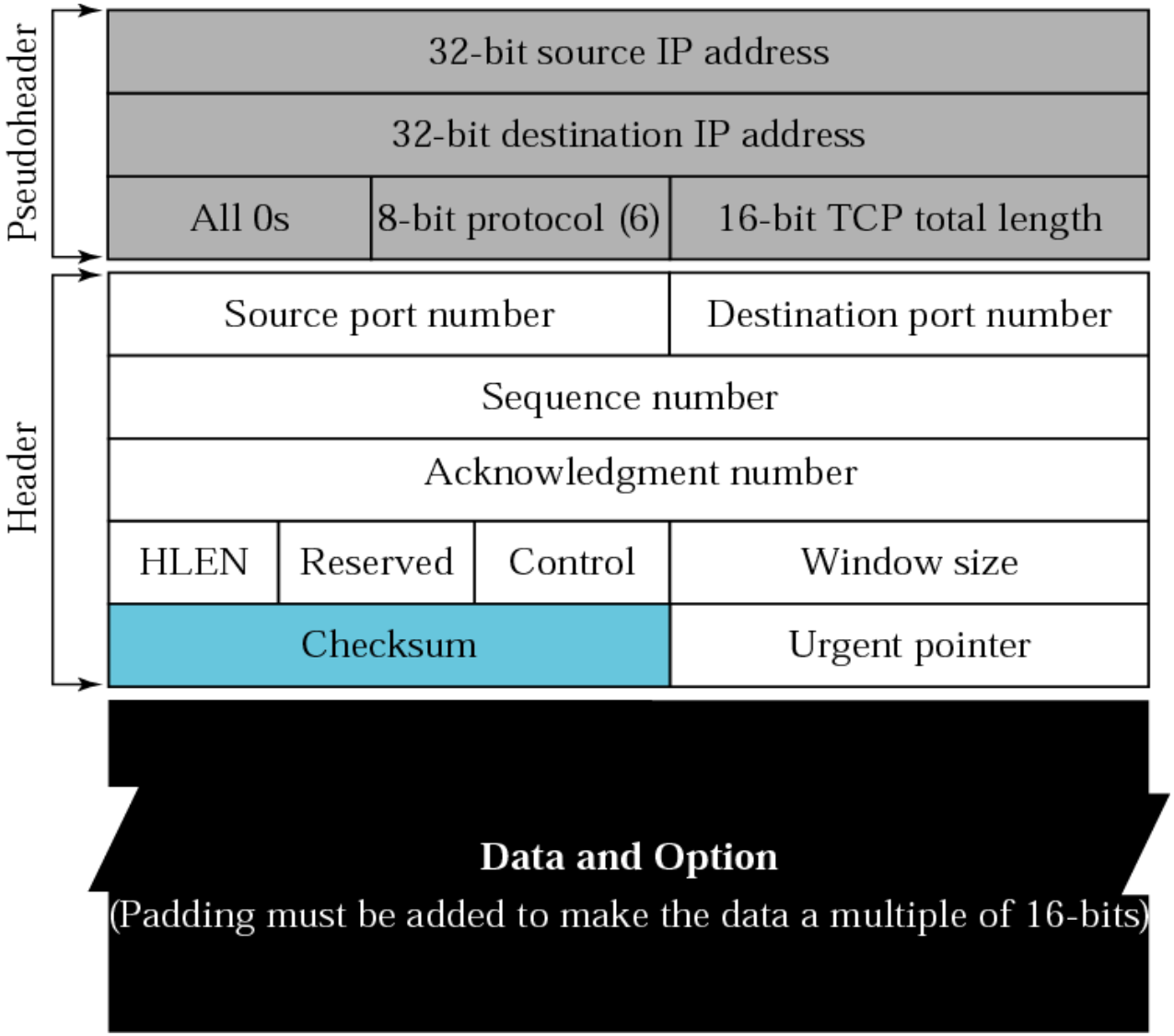


Table 12.2 Description of flags in the control field

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

Figure 12.7 *Pseudoheader added to the TCP datagram*



The inclusion of the checksum in TCP is mandatory.

Figure 12.8 *Encapsulation and decapsulation*

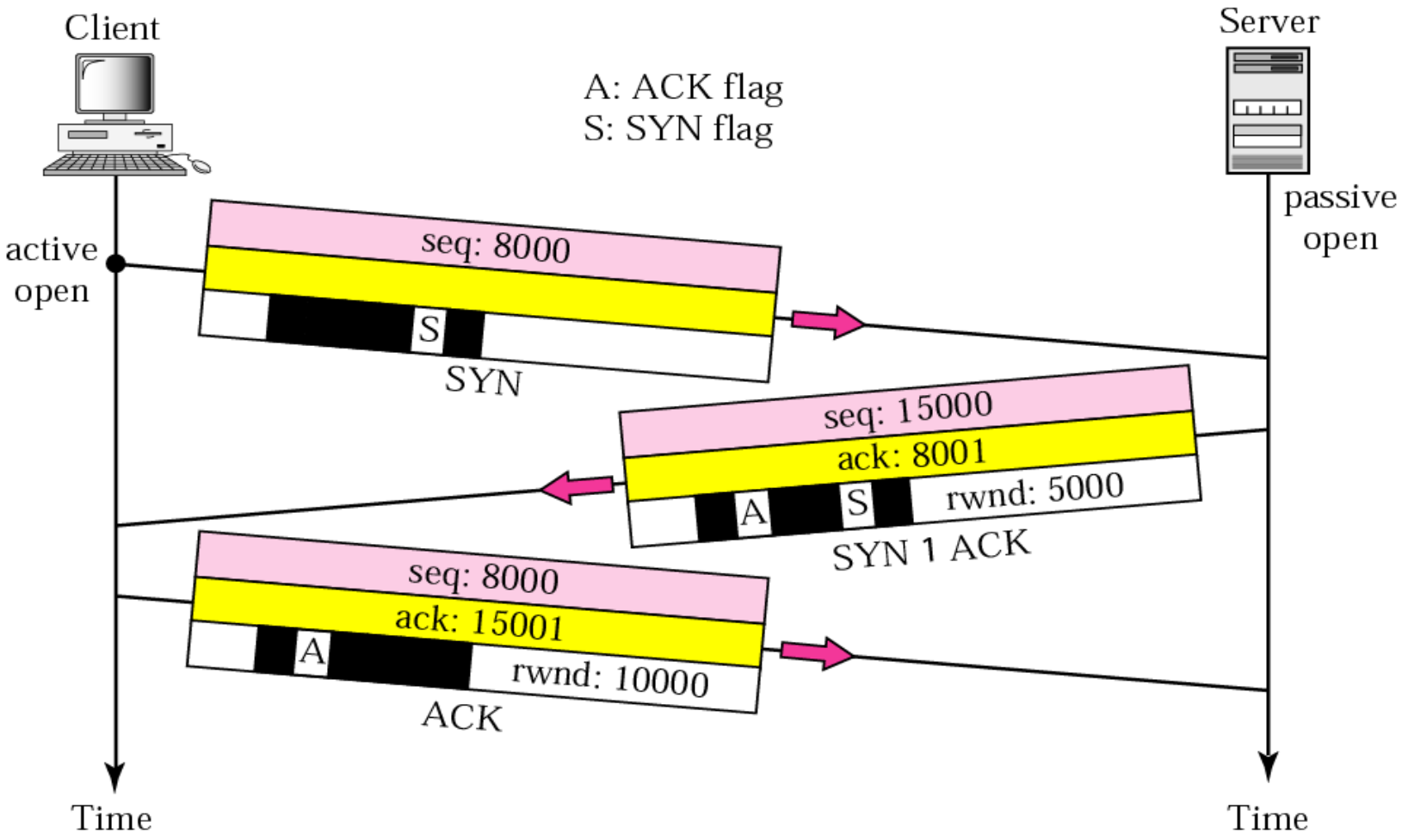
12.4 A TCP CONNECTION

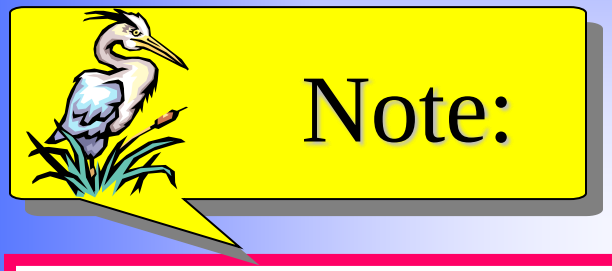
TCP is connection-oriented. A connection-oriented transport protocol establishes a virtual path between the source and destination. All of the segments belonging to a message are then sent over this virtual path. A connection-oriented transmission requires three phases: connection establishment, data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

The topics discussed in this section include:

Connection Establishment
Data Transfer
Connection Termination
Connection Reset

Figure 12.9 *Connection establishment using three-way handshaking*





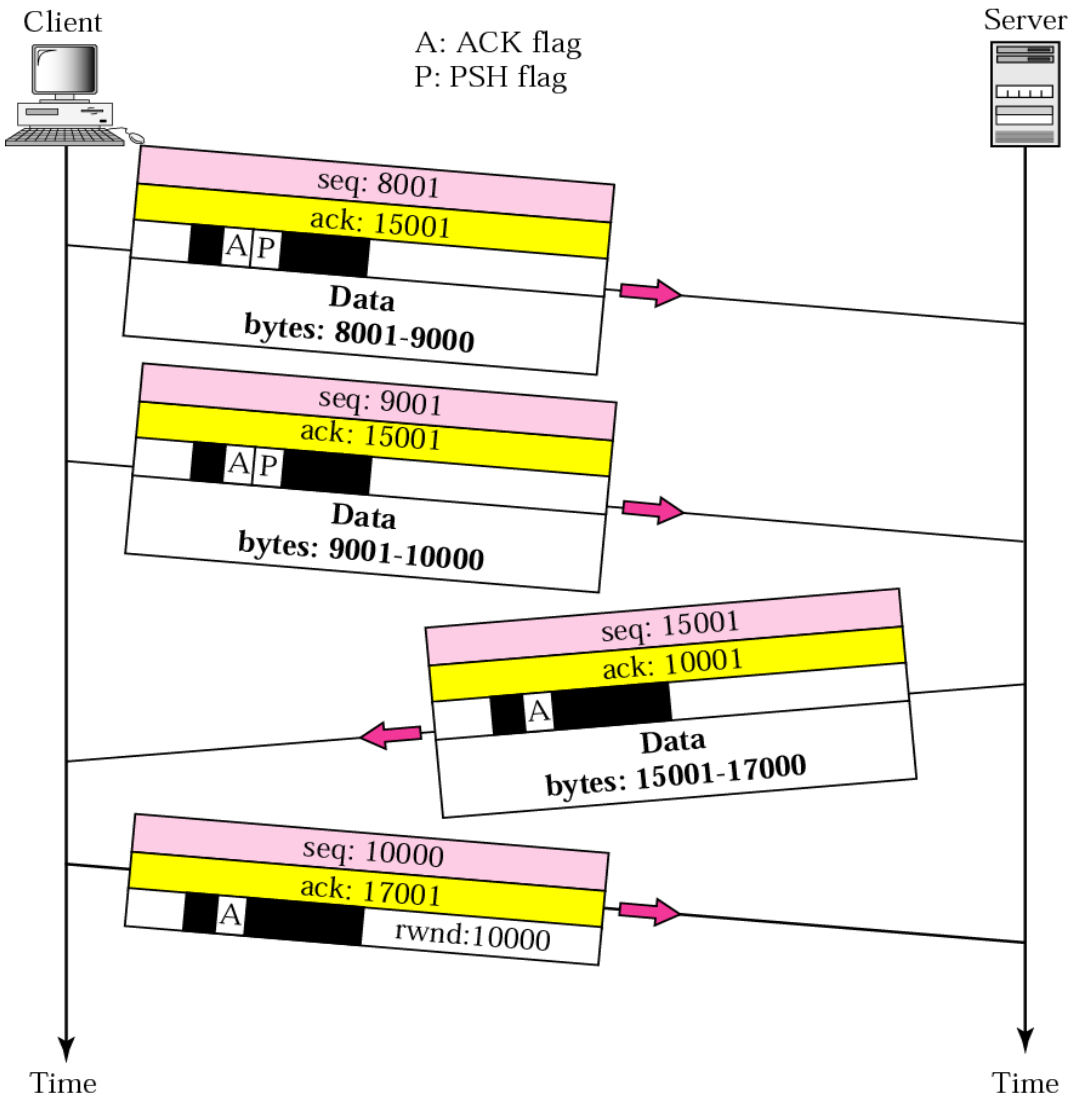
A SYN segment cannot carry data, but it consumes one sequence number.

Note:

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

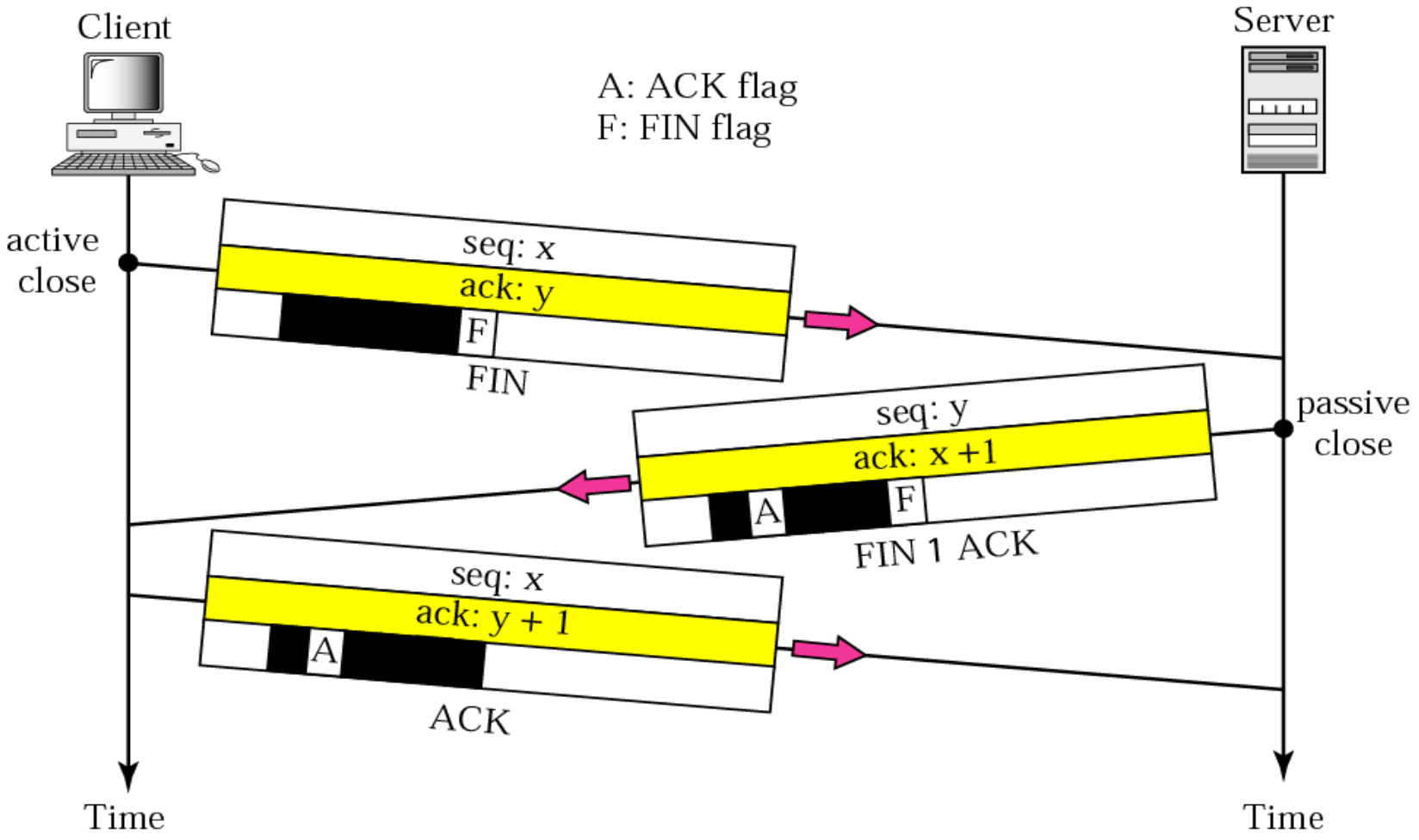
An ACK segment, if carrying no data, consumes no sequence number.

Figure 12.10 *Data transfer*



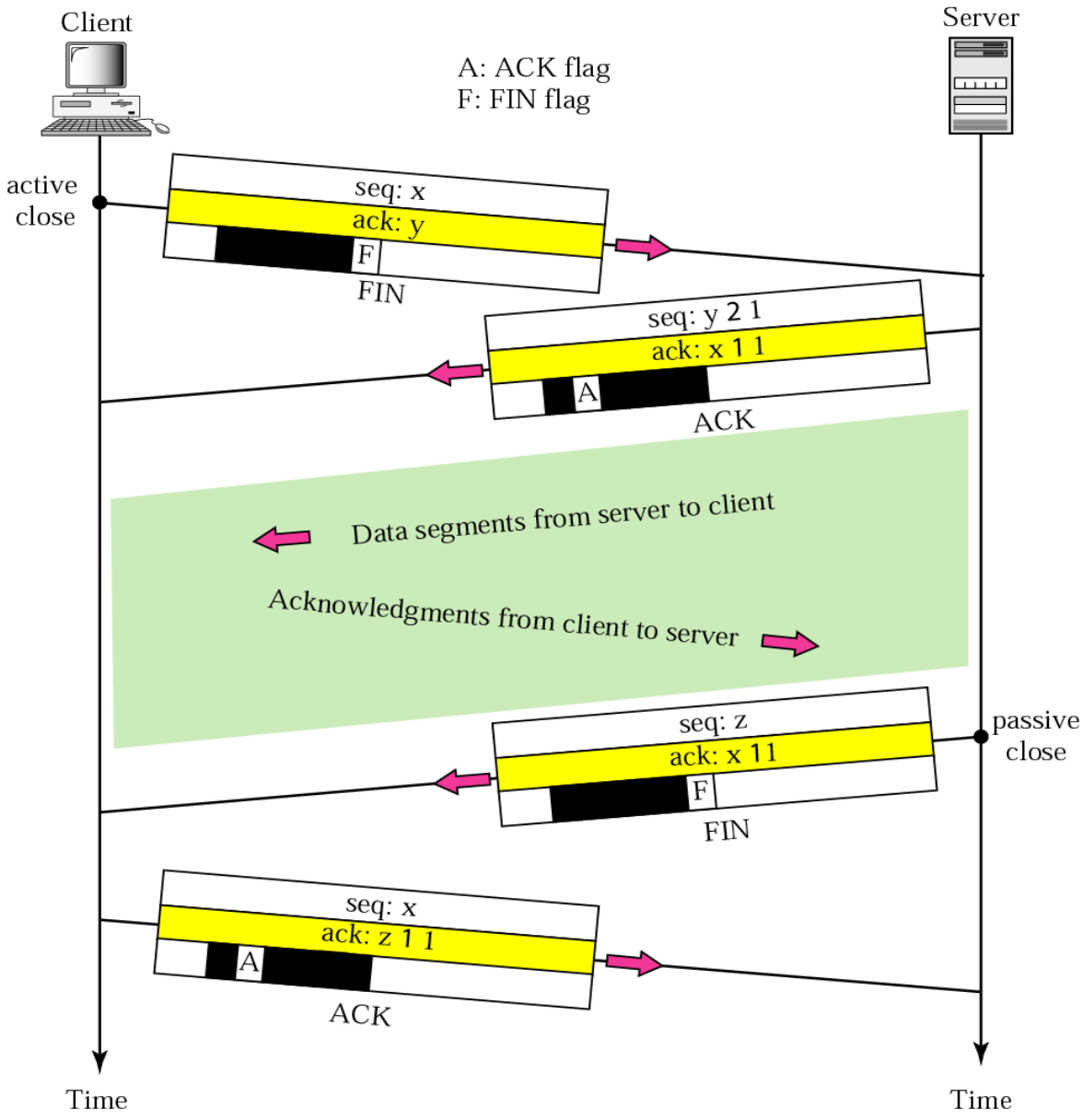
The FIN segment consumes one sequence number if it does not carry data.

Figure 12.11 *Connection termination using three-way handshaking*



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

Figure 12.12 *Half-close*



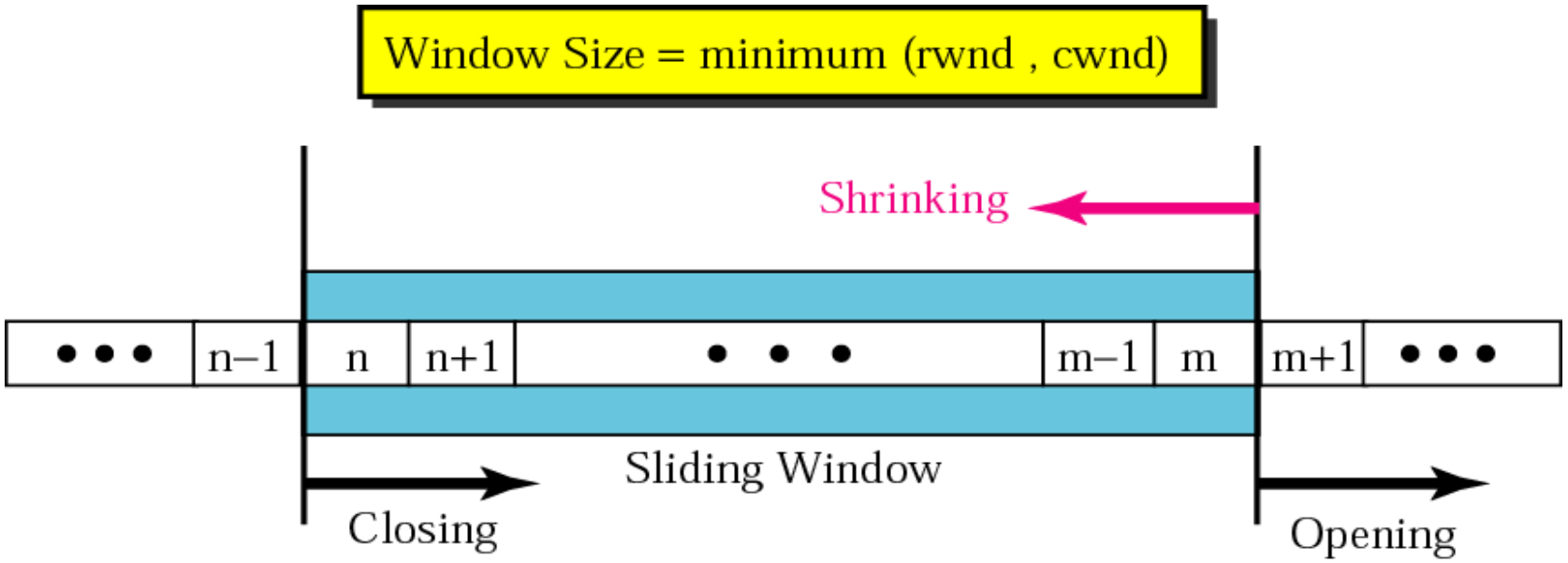
12.5 FLOW CONTROL

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

Figure 12.20 *Sliding window*



Note:

A sliding window is used to make transmission more efficient as well as to control the flow of data so that the destination does not become overwhelmed with data.

TCP's sliding windows are byte oriented.

EXAMPLE 3

What is the value of the receiver window (rwnd) for host A if the receiver, host B, has a buffer size of 5,000 bytes and 1,000 bytes of received and unprocessed data?

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

What is the size of the window for host A if the value of rwnd is 3,000 bytes and the value of cwnd is 3,500 bytes?

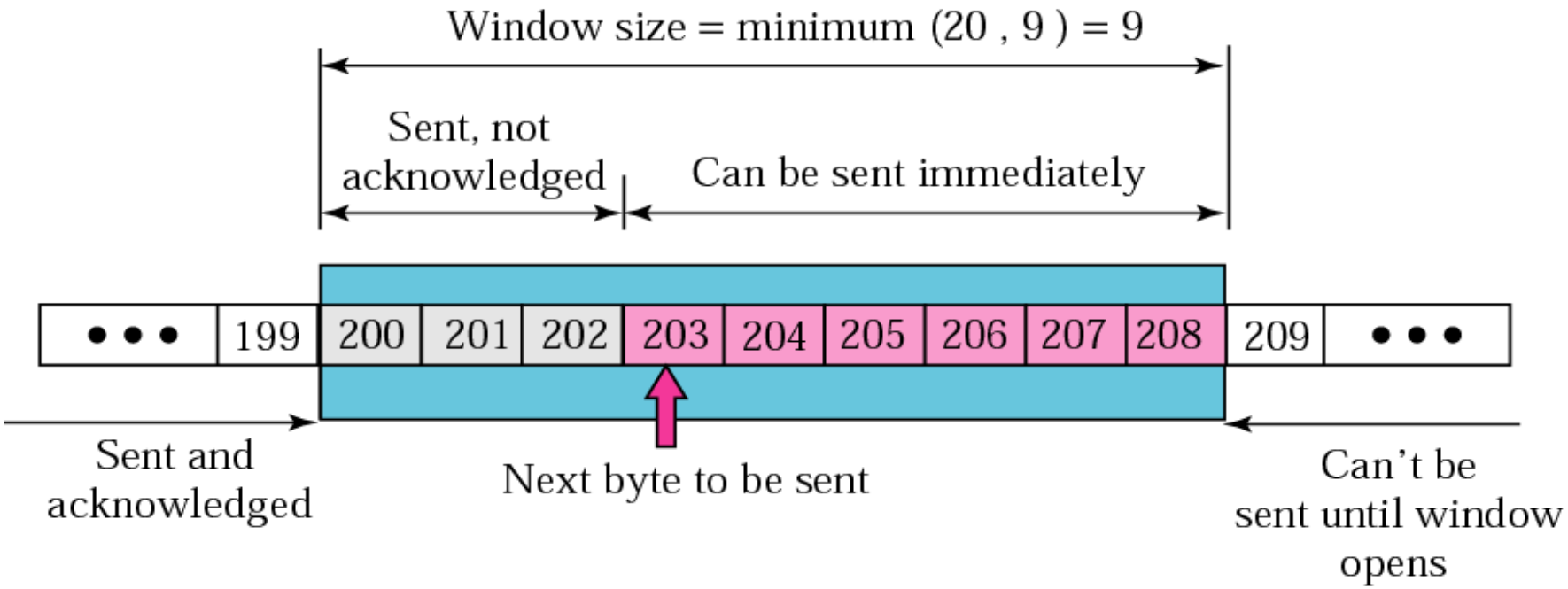
Solution

The size of the window is the smaller of rwnd and cwnd, which is 3,000 bytes.

EXAMPLE 5

Figure 12.21 shows an unrealistic example of a sliding window. The sender has sent bytes up to 202. We assume that cwnd is 20 (in reality this value is thousands of bytes). The receiver has sent an acknowledgment number of 200 with an rwnd of 9 bytes (in reality this value is thousands of bytes). The size of the sender window is 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 *Example 5*



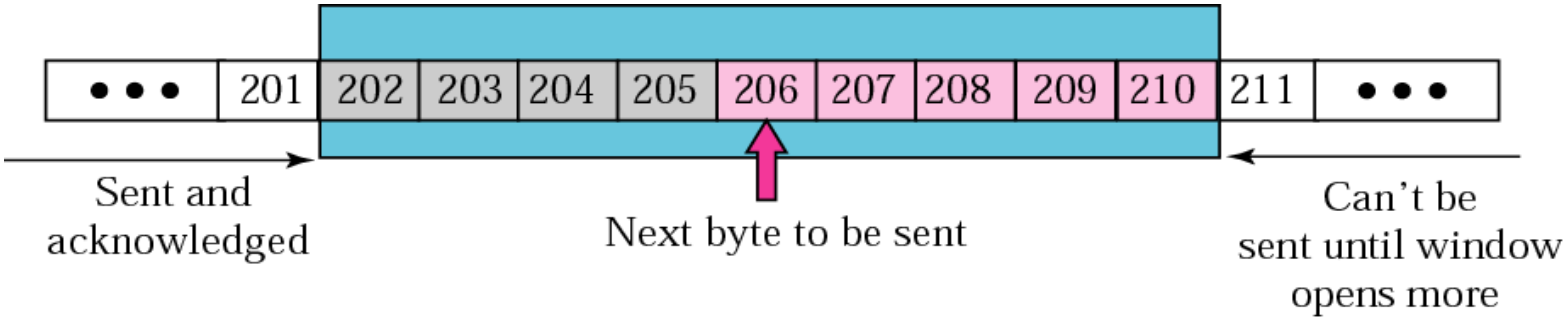
EXAMPLE 6

In Figure 12.21 the server receives a packet with an acknowledgment value of 202 and an rwnd of 9. The host has already sent bytes 203, 204, and 205. The value of cwnd is still 20. Show the new window.

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 and the sender needs not worry about them; the window can slide over them.

Figure 12.22 *Example 6*



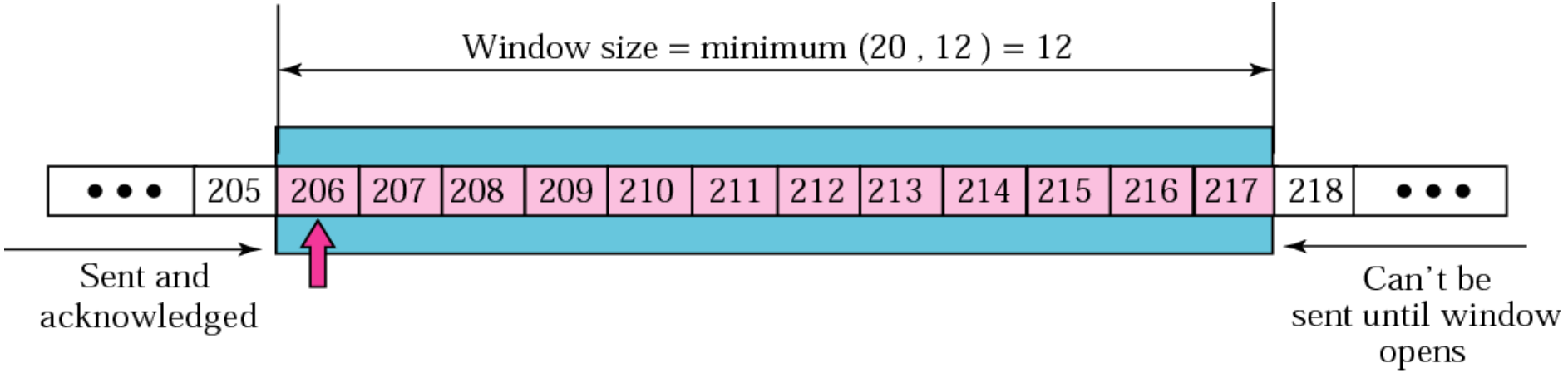
EXAMPLE 7

In Figure 12.22 the sender receives a packet with an acknowledgment value of 206 and an rwnd of 12. The host has not sent any new bytes. The value of cwnd is still 20. Show the new window.

Solution

The value of rwnd is less than cwnd, so the size of the window is 12. Figure 12.23 shows the new window. Note that 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 *Example 7*

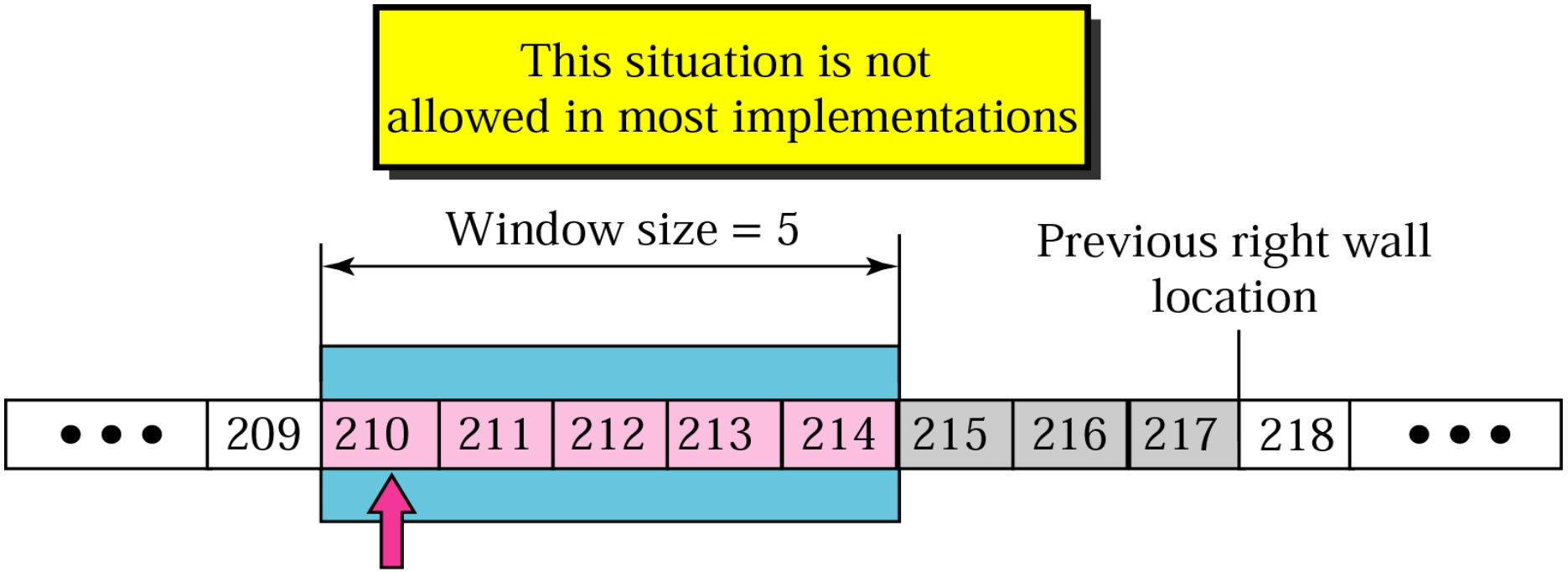


EXAMPLE 8

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

Solution

The value of rwnd is less than cwnd, so the size of the window is 5. Figure 12.24 shows the situation. Note that this is a case not allowed by most implementations. Although the sender has not sent bytes 215 to 217, the receiver does not know this.


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

The receiver needs to keep track of the last acknowledgment number and the last rwnd. If we add the acknowledgment number to rwnd we get the byte number following the right wall. If we want to prevent the right wall from moving to the left (shrinking), we must always have the following relationship.

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

or

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

Note:

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

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.***

12.6 ERROR CONTROL

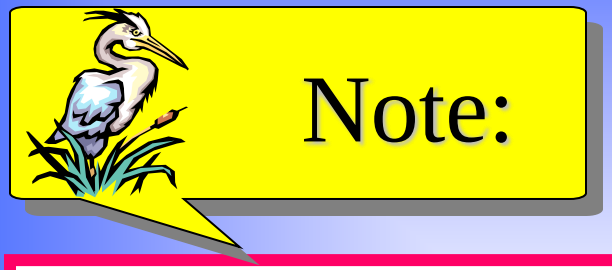
TCP provides reliability using error control, which detects corrupted, lost, out-of-order, and duplicated segments. Error control in TCP is achieved through the use of the checksum, acknowledgment, and time-out.

The topics discussed in this section include:

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

Note:

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



In modern implementations, a retransmission occurs if the retransmission timer expires or three duplicate ACK segments have arrived.

No retransmission timer is set for an ACK segment.

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.

Figure 12.25 *Normal operation*

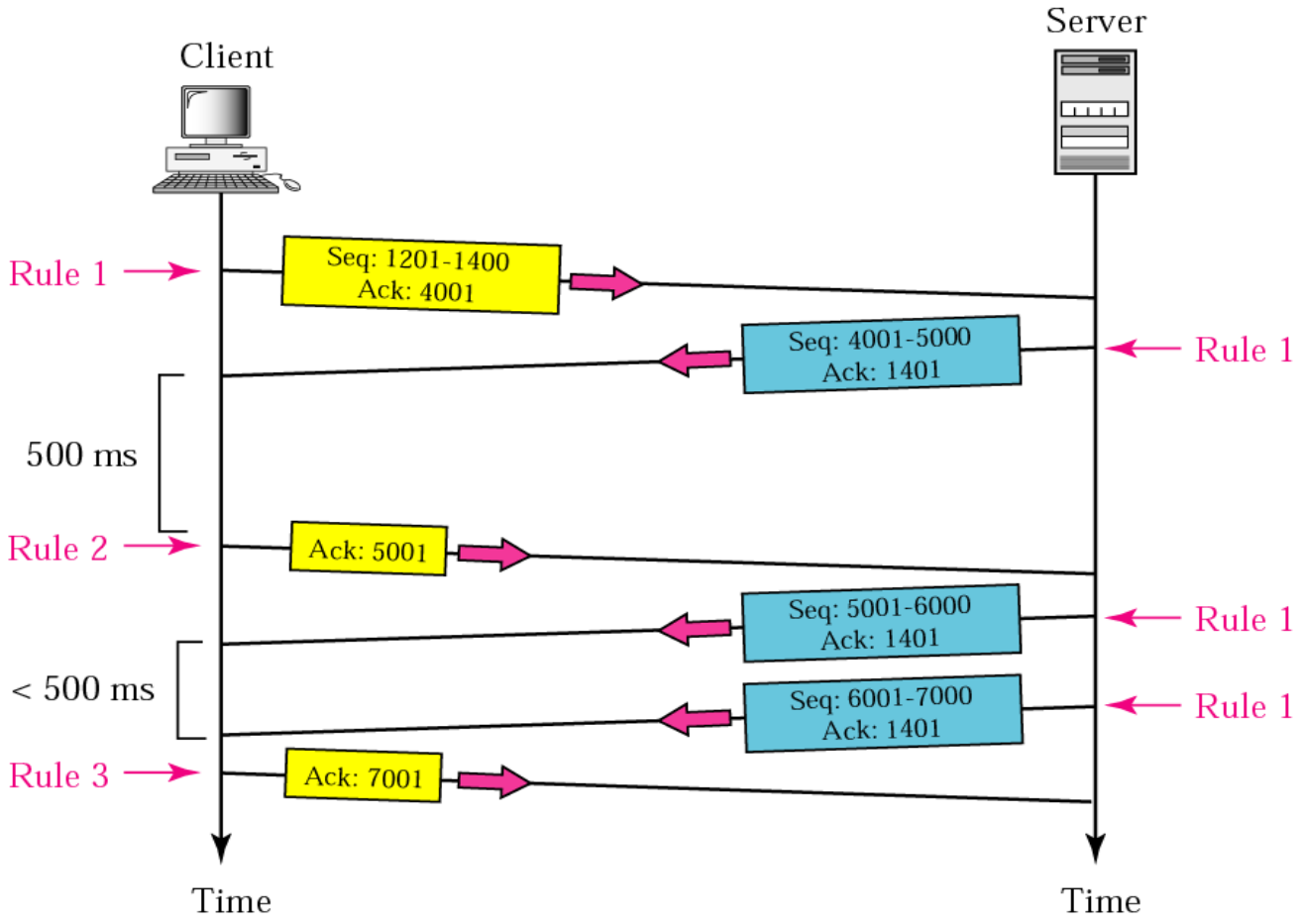
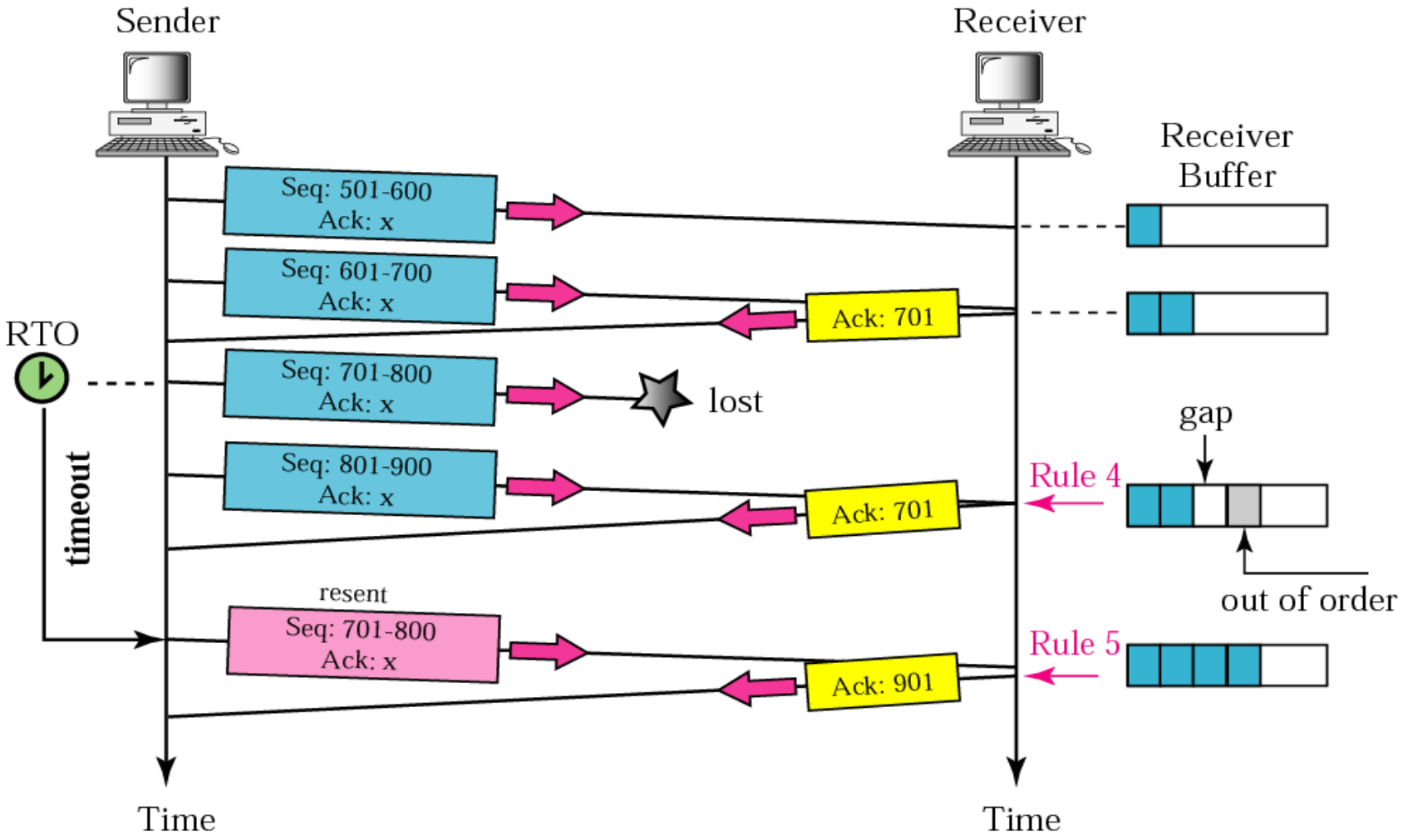


Figure 12.26 *Lost segment*



Note:

The receiver TCP delivers only ordered data to the process.

Figure 12.27 *Fast retransmission*

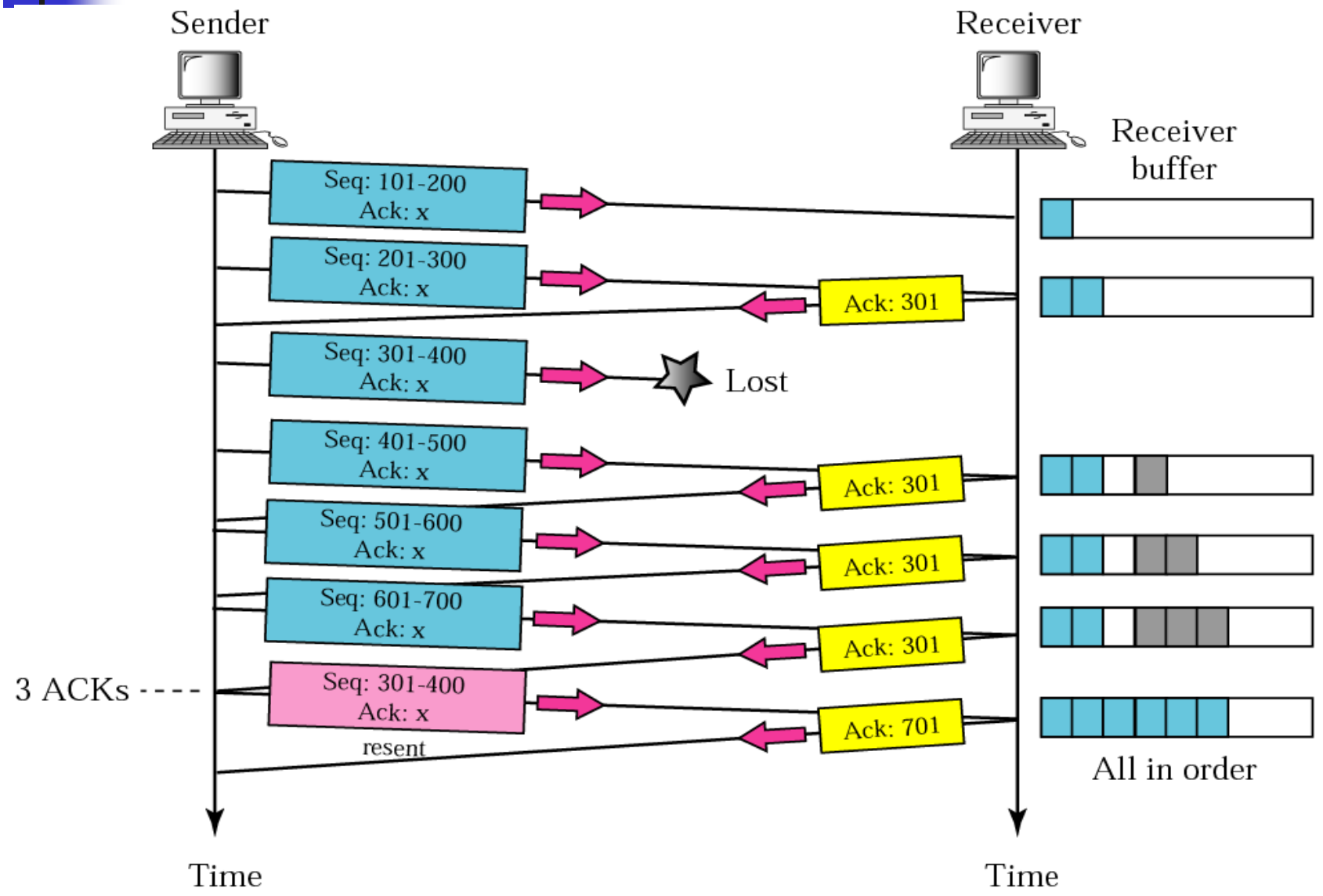


Figure 12.28 *Lost acknowledgment*

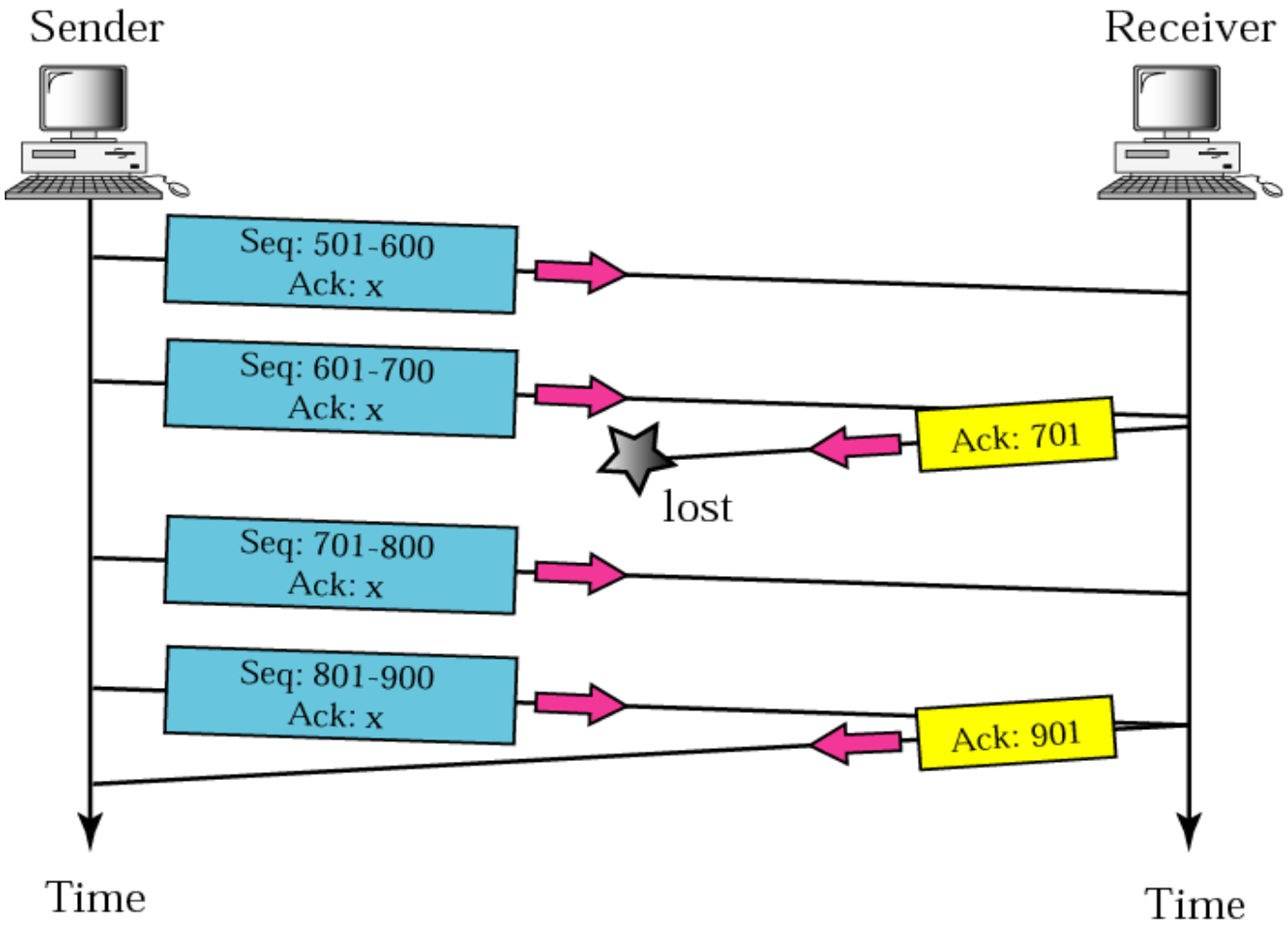
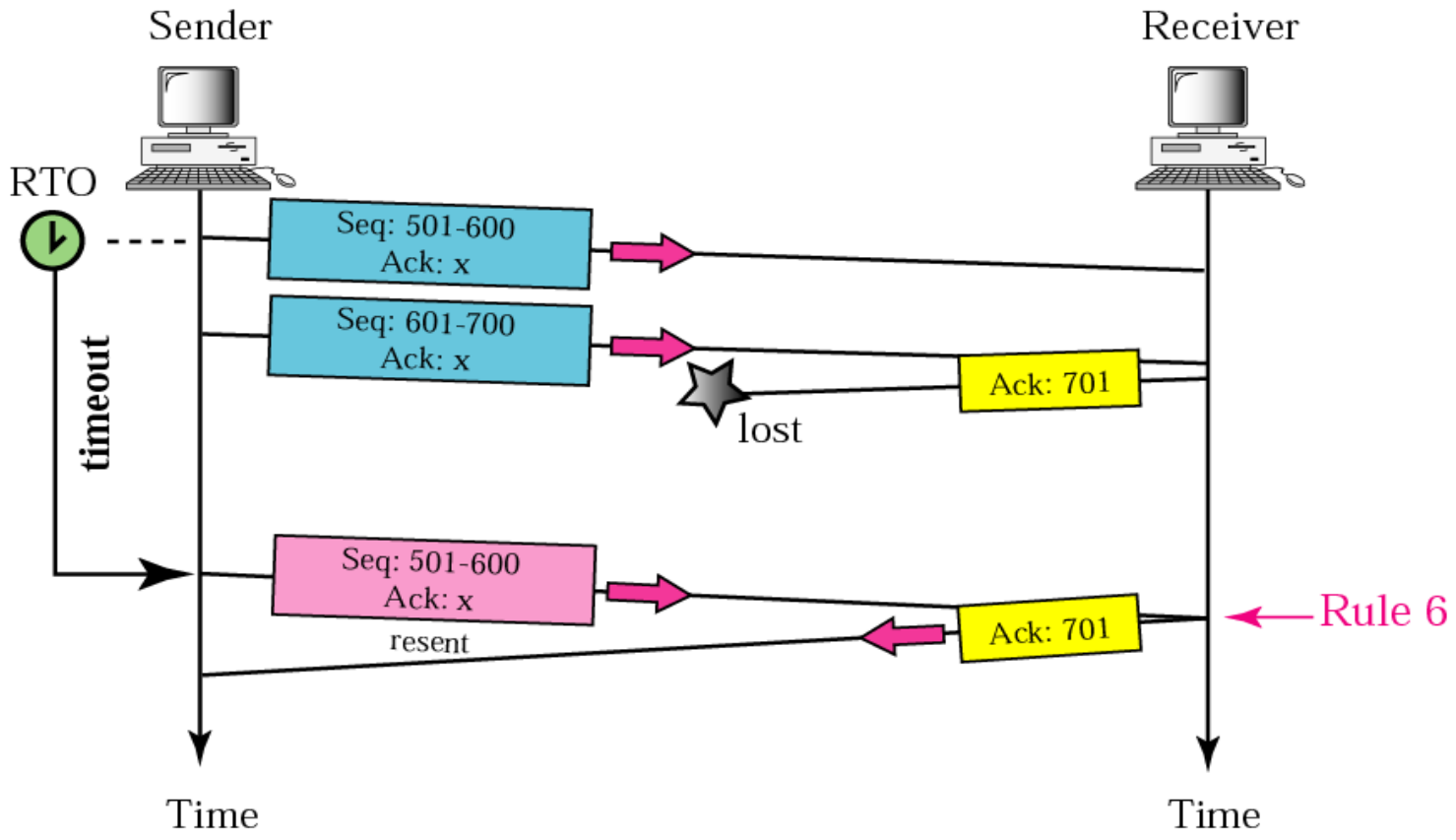


Figure 12.29 *Lost acknowledgment corrected by resending a segment*



Lost acknowledgments may create deadlock if they are not properly handled.

12.7 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

Figure 12.30 *Router queues*

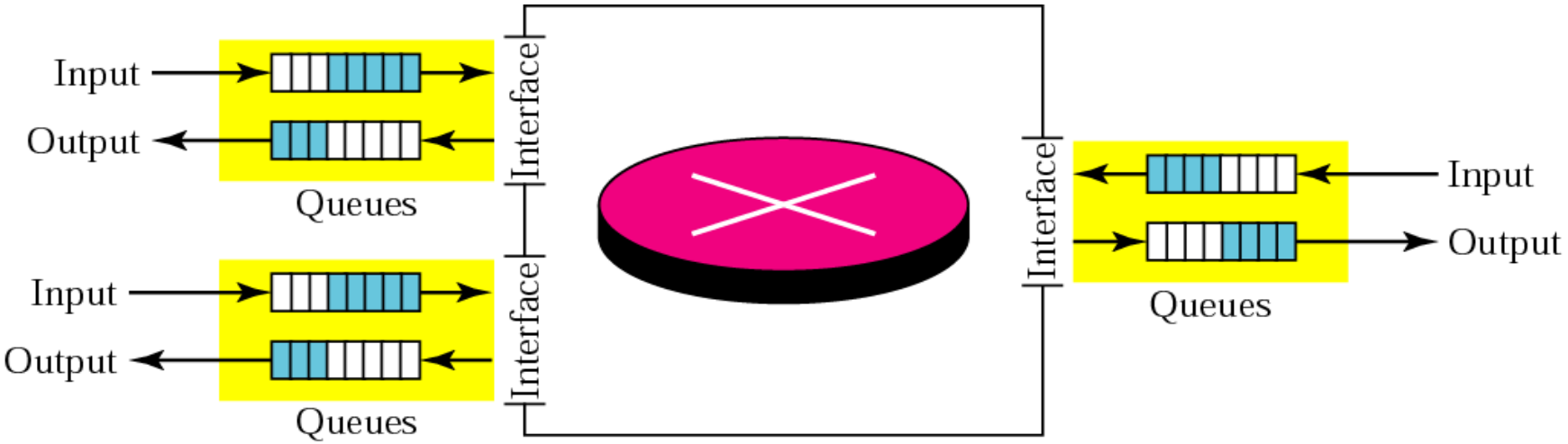


Figure 12.31 *Packet delay and network load*

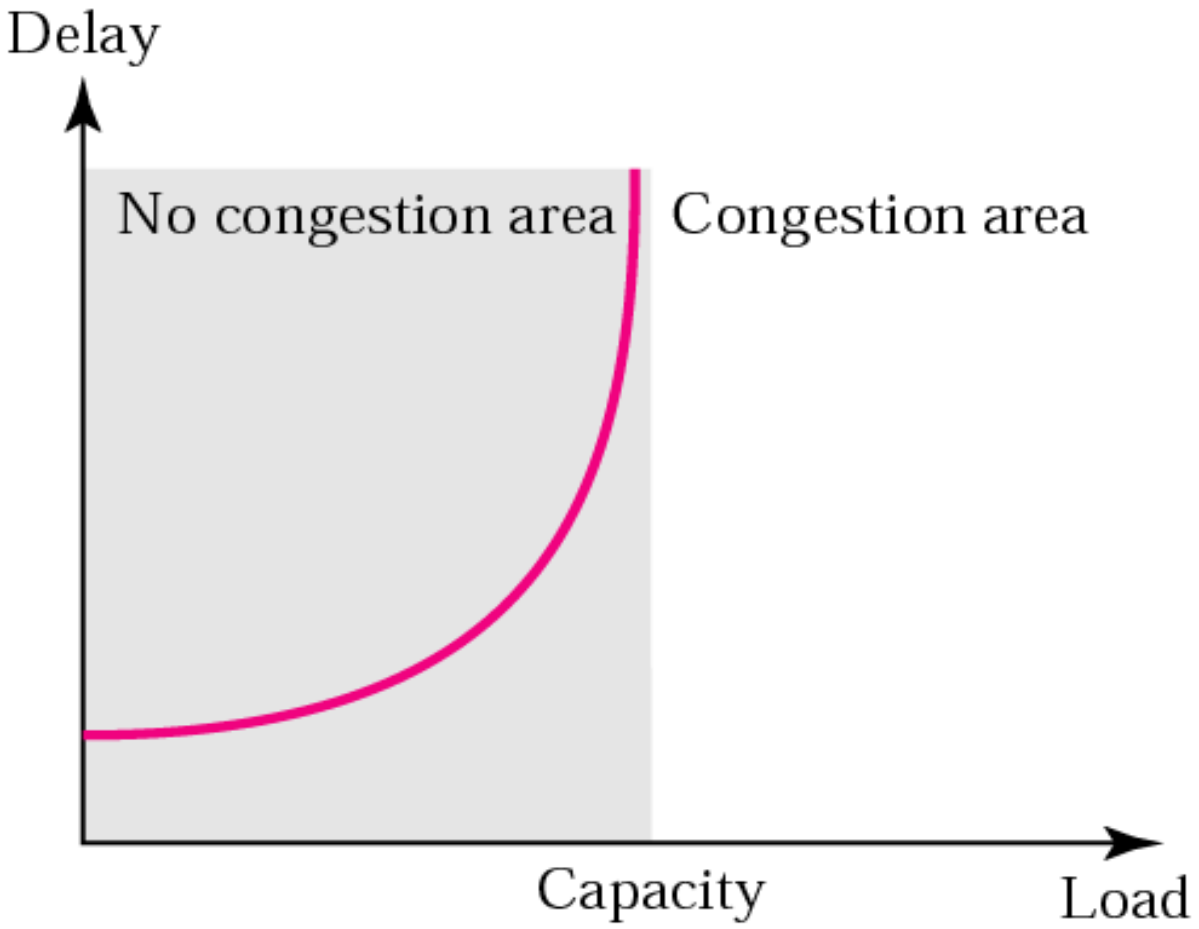


Figure 12.32 *Throughput versus network load*

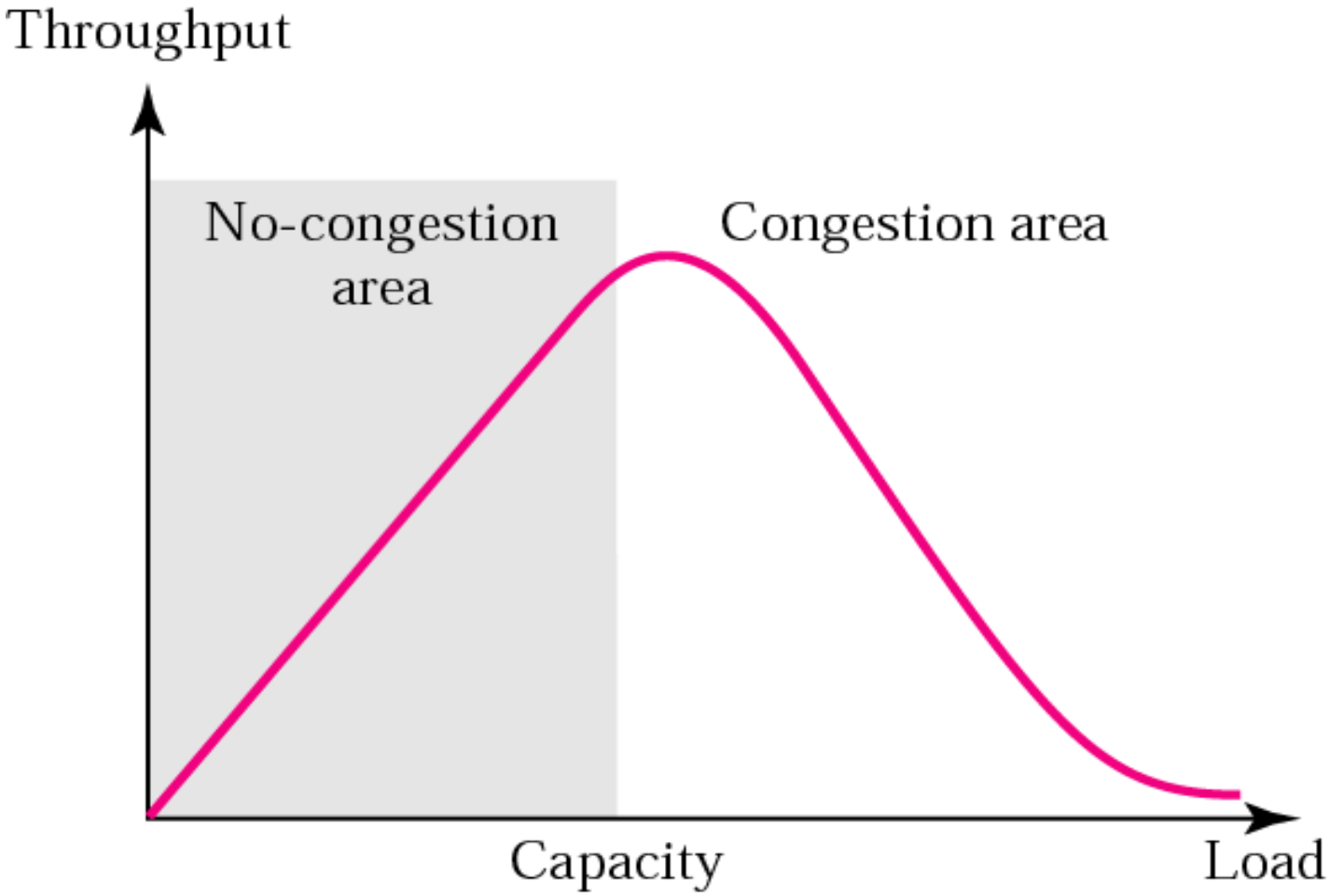
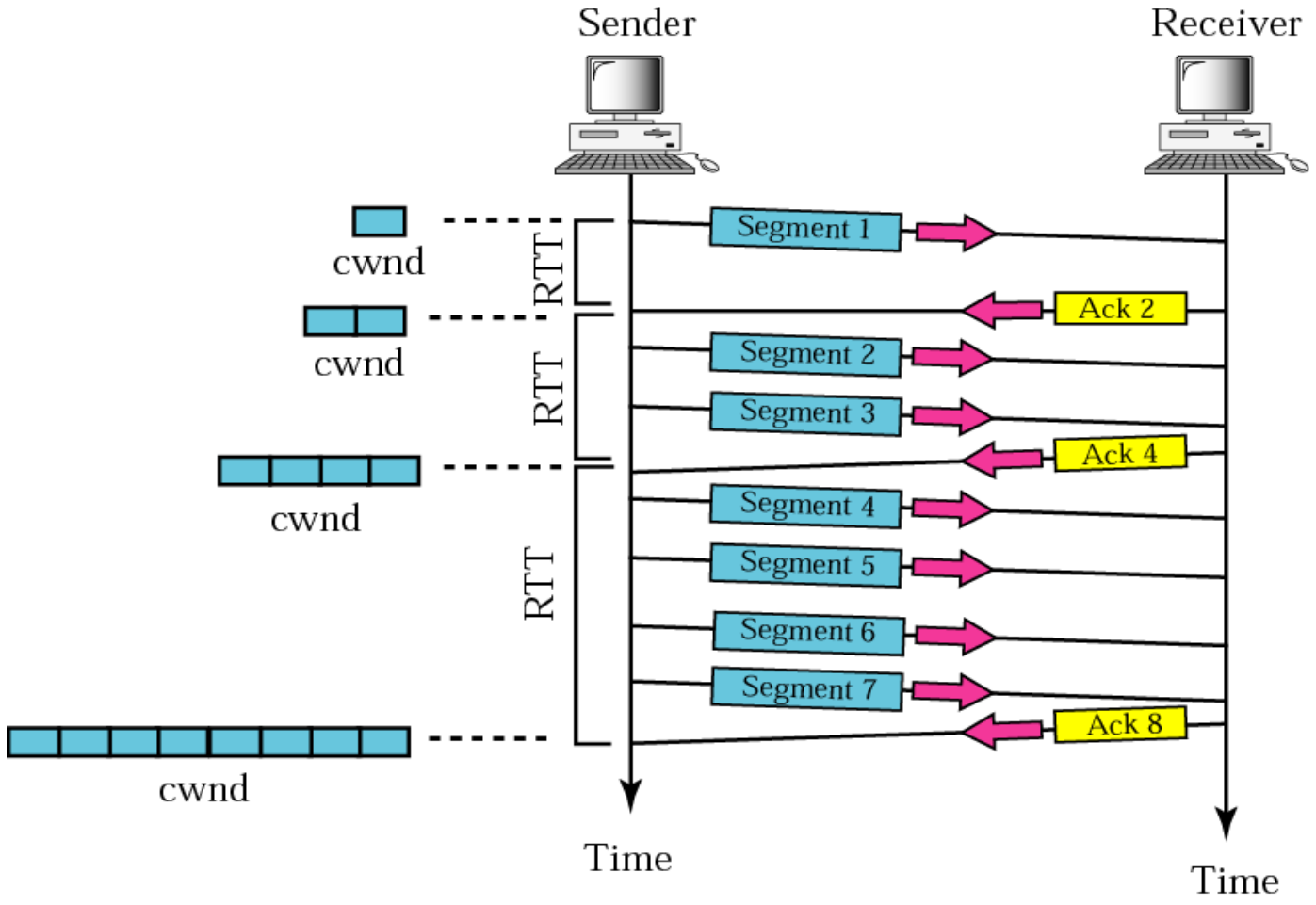
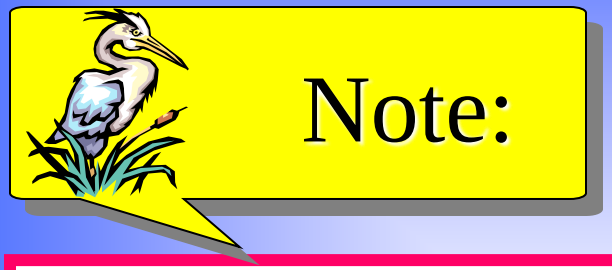


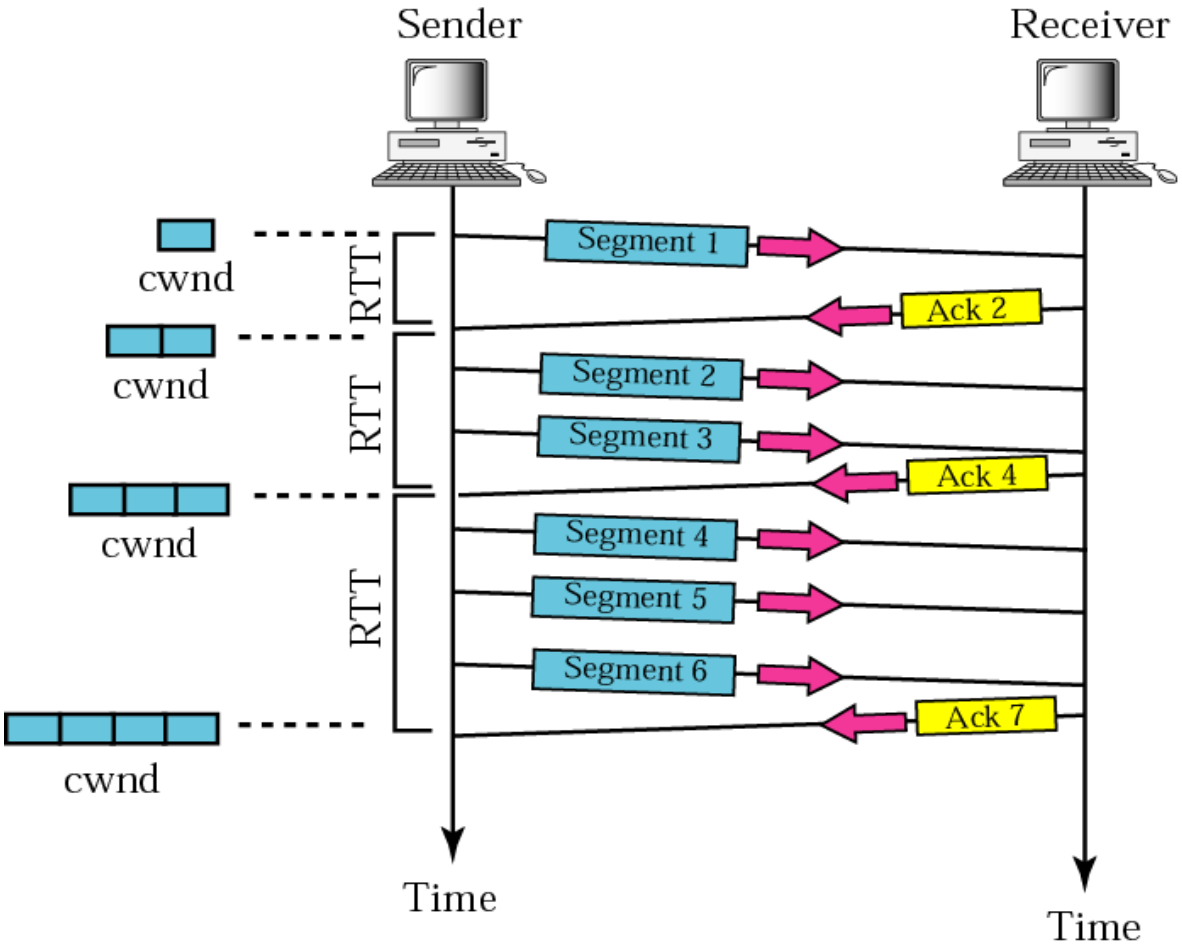
Figure 12.33 *Slow start, exponential increase*





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

Figure 12.34 *Congestion avoidance, additive increase*



Note:

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

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.***

Figure 12.35 *TCP congestion policy summary*

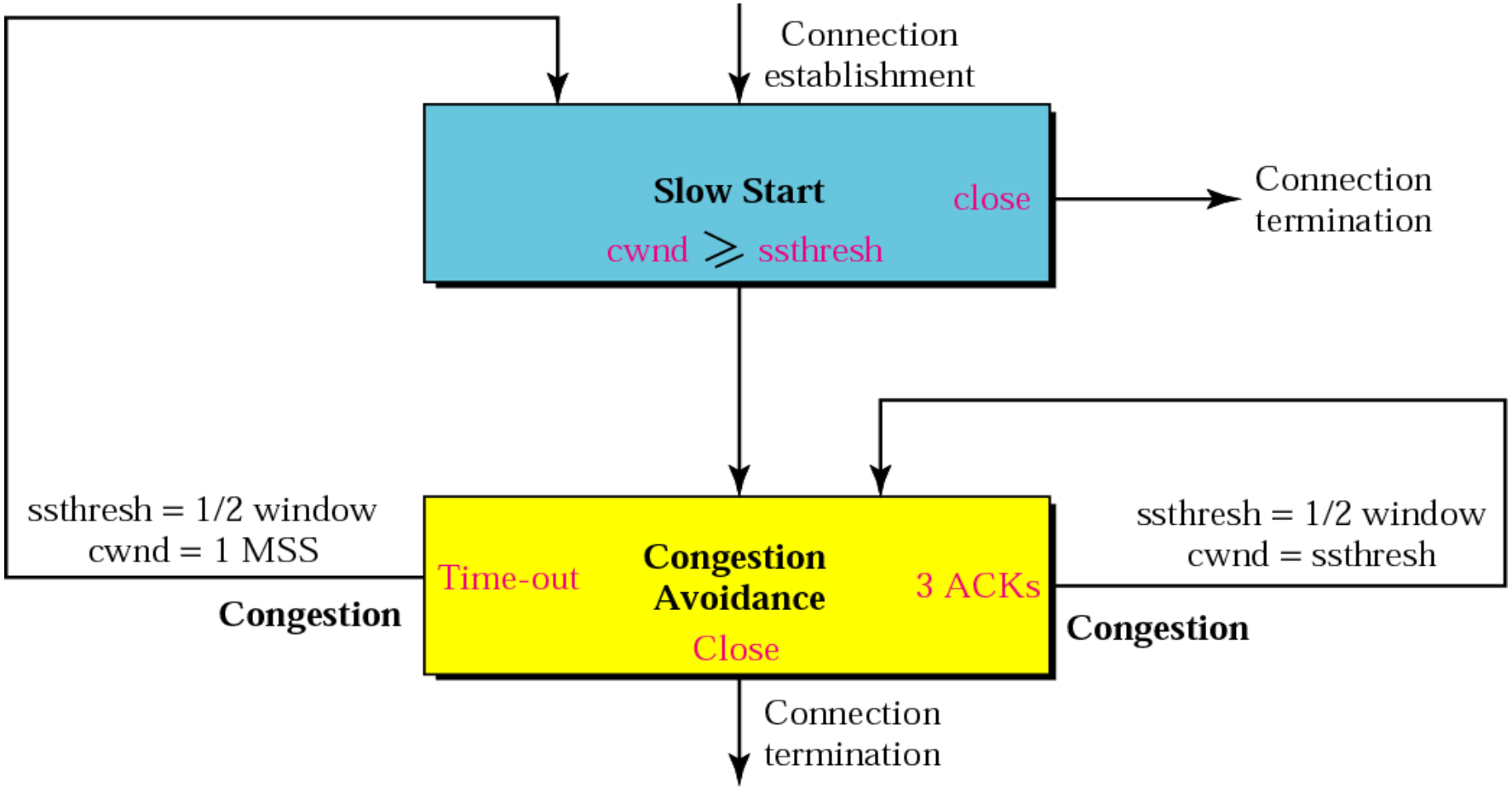
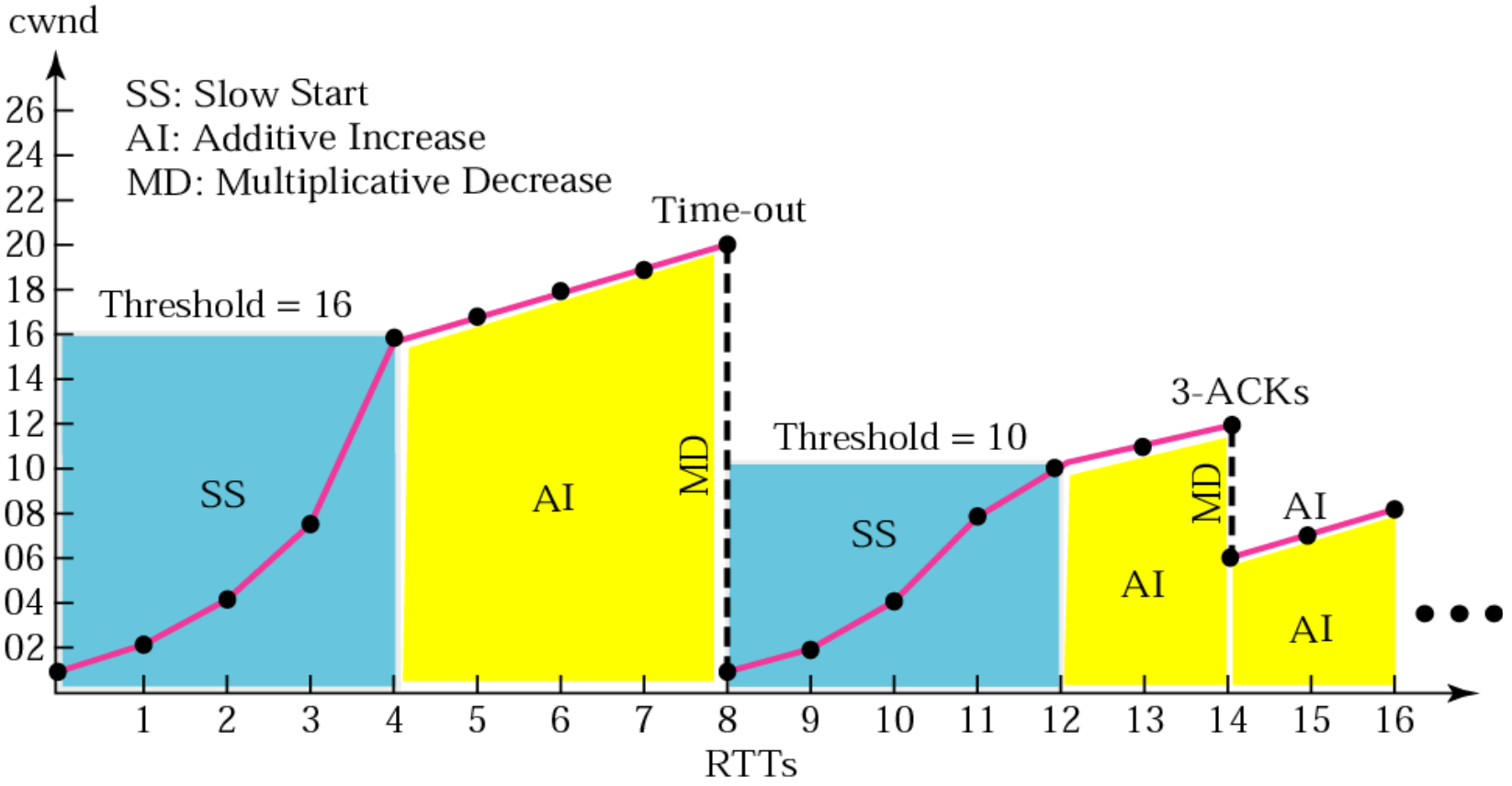


Figure 12.36 *Congestion example*



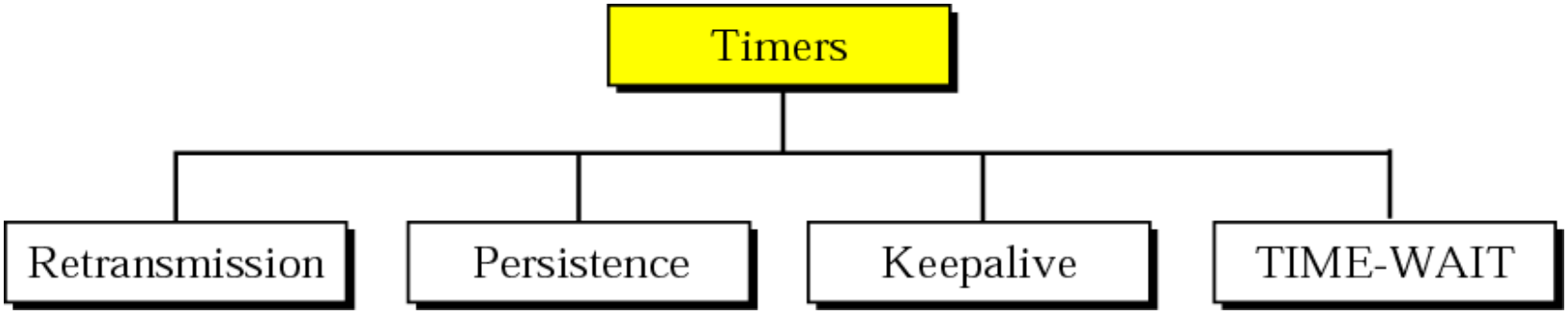
12.8 TCP TIMERS

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

Figure 12.37 *TCP timers*



In TCP, there can be only be one RTT measurement in progress at any time.

Measured RTT (RTT_M)

- Time to send a segment and receive an acknowledgment for it
- Remember that the segments and their acknowledgments do not have a one-to-one relationship
- Only one RTT measurement can be in progress at any time.

Smoothed RTT (RTT_S)

- Since RTT_M of various packets might vary considerably, RTT_S is used

Initially	$\rightarrow$	No value
After first measurement	$\rightarrow$	$RTT_S = RTT_M$
After each measurement	$\rightarrow$	$RTT_S = (1 - \alpha)\, RTT_S + \alpha \times RTT_M$

- The value of α is normally set to 1/8

RTT Deviation (RTT_D)

- Since RTT_M of various packets might vary considerably, RTT_S is used

Initially	→	No value
After first measurement	→	$RTT_D = RTT_M / 2$
After each measurement	→	$RTT_D = (1 - \beta)\, RTT_D + \beta \times \lvert RTT_S - RTT_M \rvert$

- The value of β is normally set to ¼

Retransmission Time-out (RTO)

- Since RTT_M of various packets might vary considerably, RTT_S is used

Original	→	Initial value
After any measurement	→	$RTO = RTT_S + 4 \times RTT$

EXAMPLE 10

Let us give a hypothetical example. 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. The following shows the value of these variables at this moment:

$RTT_M = 1.5$	$RTT_S = 1.5$
$RTT_D = 1.5 / 2 = 0.75$	$RTO = 1.5 + 4 \,.\, 0.75 = 4.5$

2. When the SYN+ACK segment arrives, RTT_M is measured and is equal to 1.5 seconds. The next slide shows the values of these variables:

EXAMPLE 10 (CONTINUED)

$RTT_M = 1.5$ $\quad$ $RTT_S = 1.5$

$RTT_D = 1.5 / 2 = 0.75$ $\quad$ $RTO = 1.5 + 4 \,.\, 0.75 = 4.5$

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.*

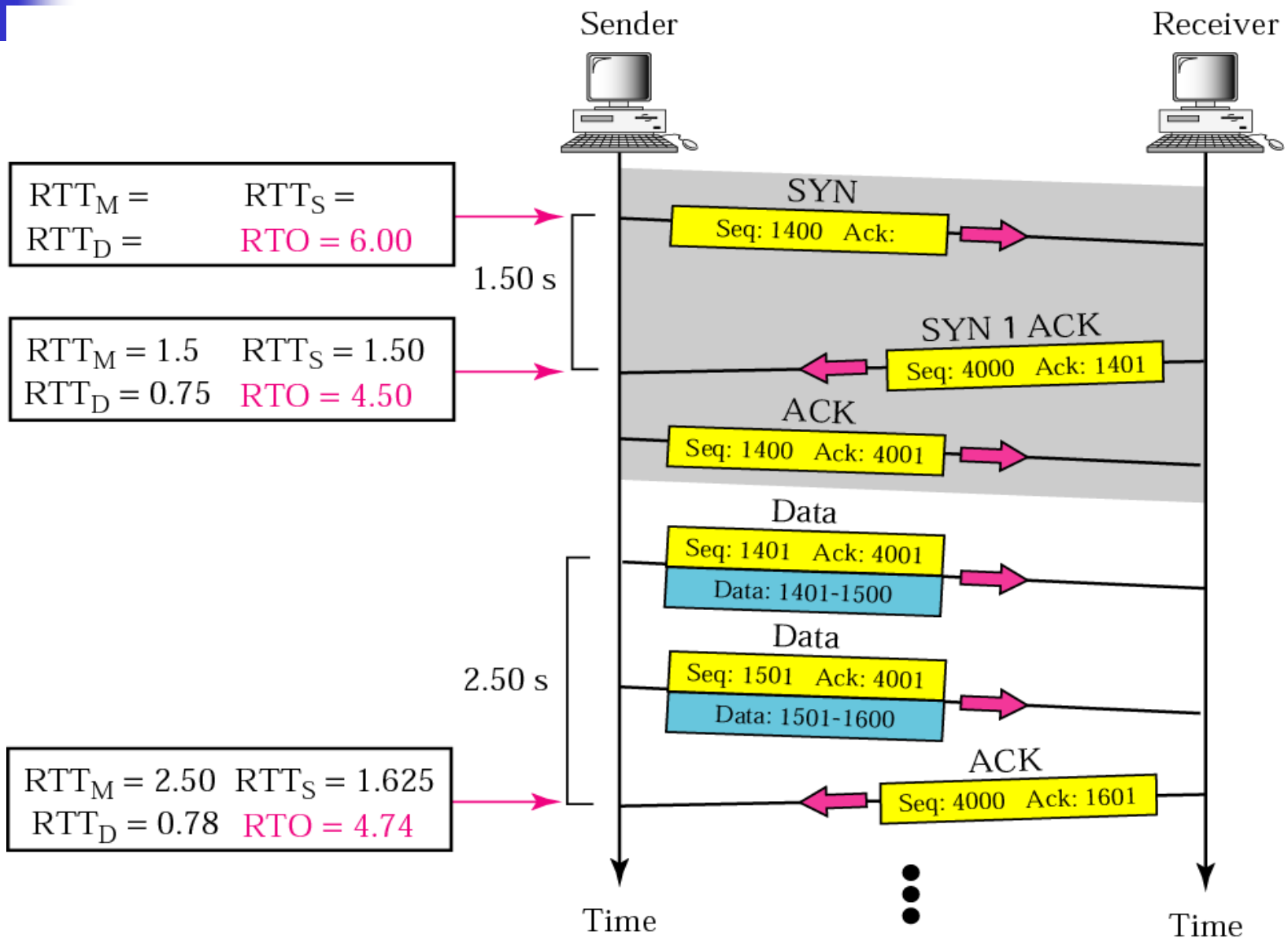
$RTT_M = 2.5$

$RTT_S = 7/8\ (1.5) + 1/8\ (2.5) = 1.625$

$RTT_D = 3/4\ (7.5) + 1/4\ \ |1.625 - 2.5| \ = 0.78$

$RTO = 1.625 + 4\ (0.78) = 4.74$

Figure 12.38 *Example 10*



• If a segment is not acknowledged during transmission, and retransmitted, the ack received can not be used for RTT measurement
• The sender will have to measure the RTT from the retransmitted segment in these cases
• Do not consider the round-trip time of a retransmitted segment in the calculation of RTTs
• Do not update the value of RTTs until you send a segment and receive an acknowledgment without the need for retransmission

Exponential Backoff

• In most TCP implementations, the value of RTO is doubled for each retransmission

TCP does not consider the RTT of a retransmitted segment in its calculation of a new RTO.

EXAMPLE 11

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. We wait until we send a new segment and receive the ACK for it before recalculating the RTO (Karn's algorithm).

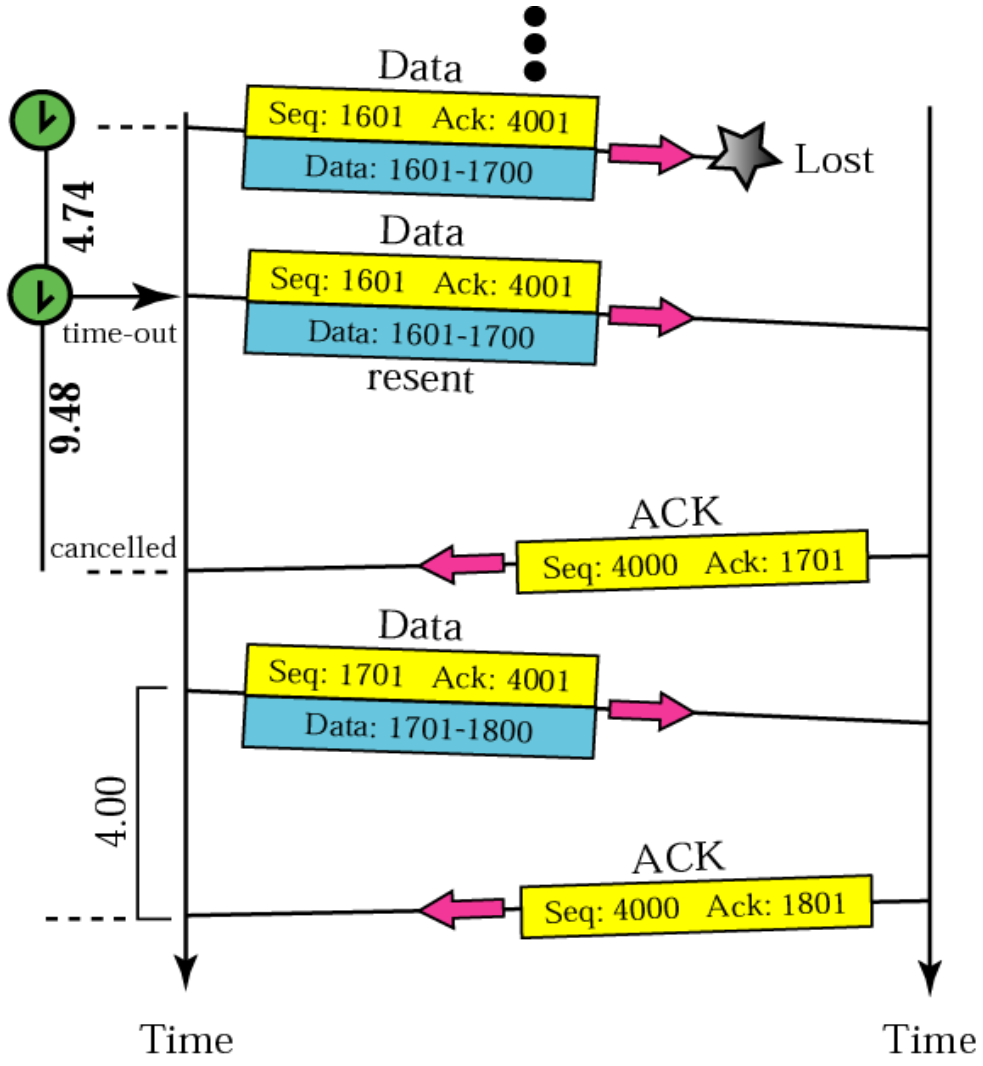
Figure 12.39 *Example 11*

$RTT_M = 2.50$ $RTT_S = 1.625$
$RTT_D = 0.78$ RTO = 4.74
Values from previous example

RTO = 2 x 4.74 = 9.48
Exponential Backoff of RTO

RTO = 2 x 4.74 = 9.48
No change, Karn's algorithm

$RTT_M = 4.00$ $RTT_S = 1.92$
$RTT_D = 1.105$ RTO = 6.34
New values based on new RTT_M



Persistent Timer

- Receiver TCP can send zero window advertisement
- The sender will freeze the sender window state and stop transmitting until the receiver sends ack announcing a nonzero window size
- This ack can be lost and acks are not acked
- Sender is waiting for ack and receiver is waiting for sender to send data segments
- It leads to deadlock
- To correct the deadlock, Persistent timer is used
- When the sending TCP receives an acknowledgment with a window size of zero, it starts a persistence timer
- On timeout the sending TCP sends a special segment called a probe
- The probe causes the receiving TCP to resend the acknowledgment

Thanks
and
Good Luck