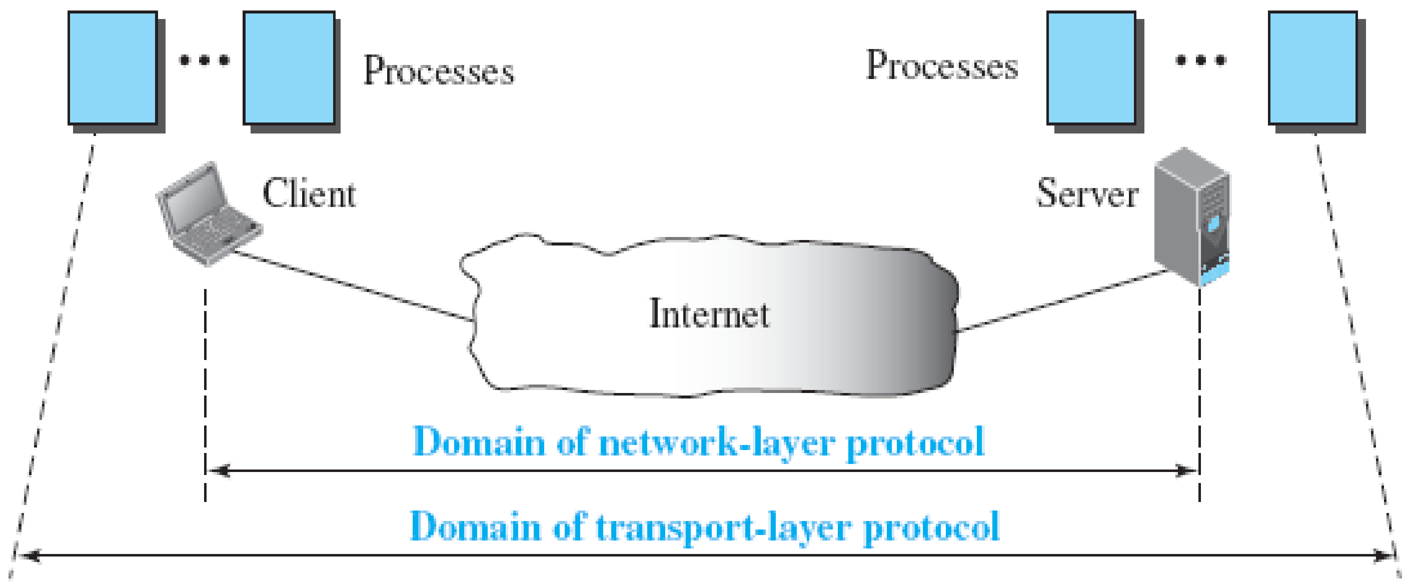
Transport Layer

Transport Layer Services

- ***Process-to-Process Communication***

Figure 23.2 *Network layer versus transport layer*

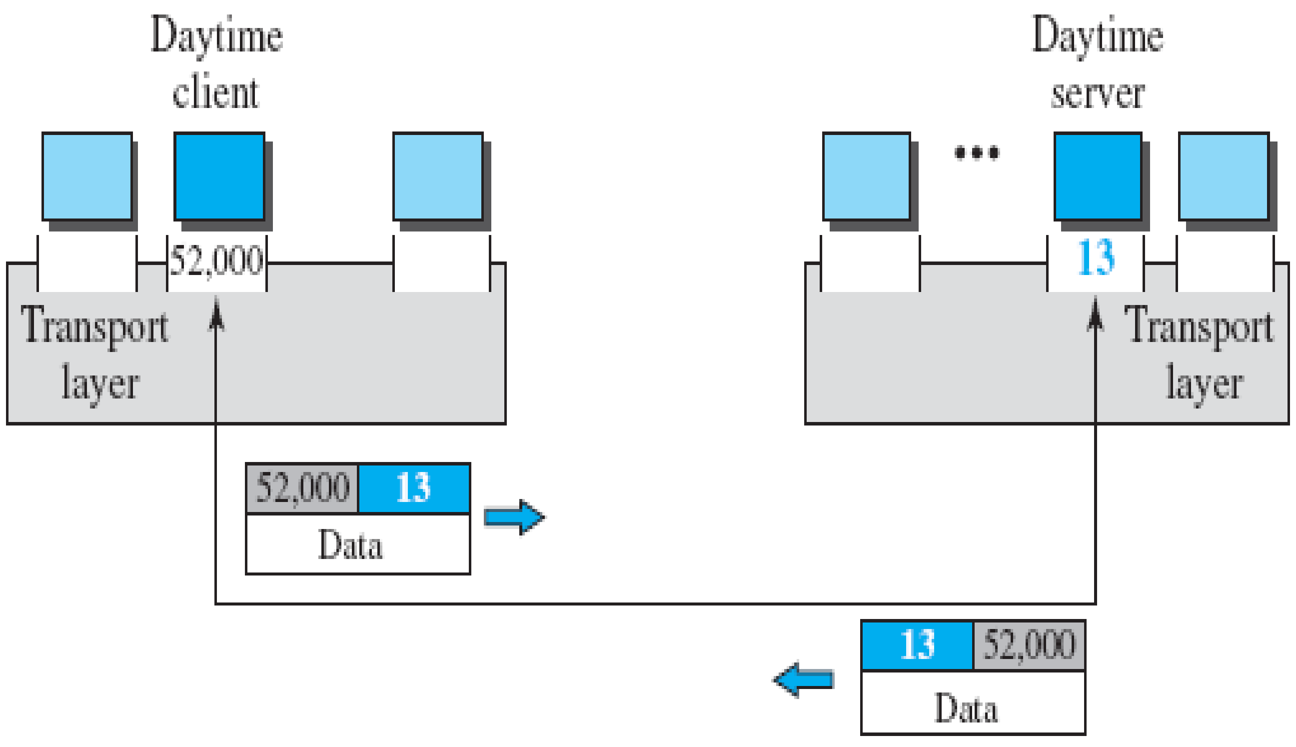


Transport Layer Services

- ***Addressing: Port Numbers***
- To define the processes, we need second identifiers, called **port numbers.**
- Port Address – 16 bit
- The client program defines itself with a port number, called the **ephemeral port number – short lived**
- TCP/IP has decided to use universal port numbers for servers; called **well-known port numbers.**

Transport Layer Services

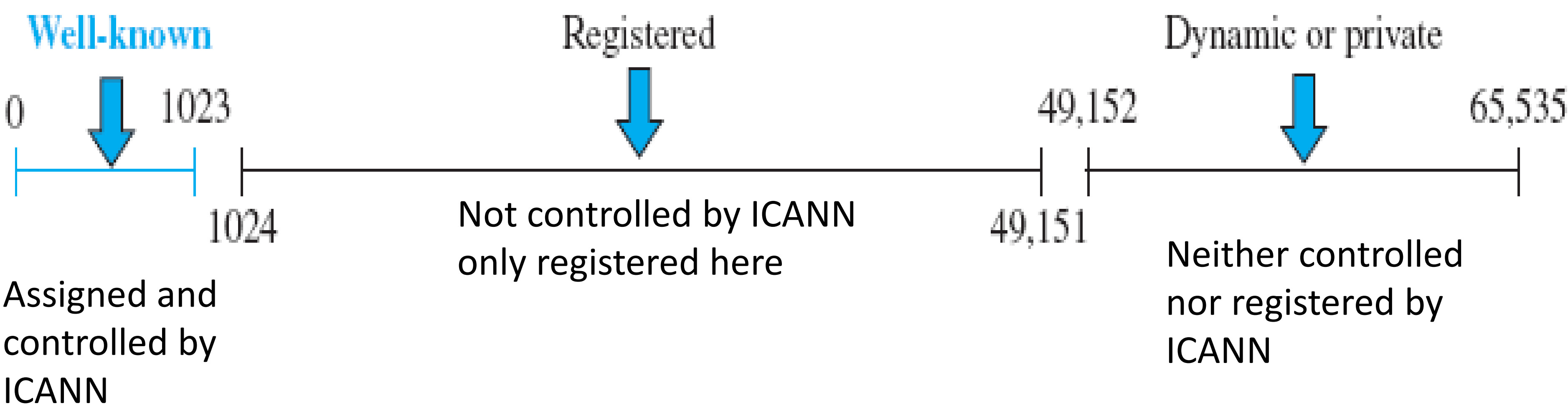
Figure 23.3 *Port numbers*



Transport Layer Services

- ***ICANN (Internet Corporation for Assigned Names and Numbers) Range***

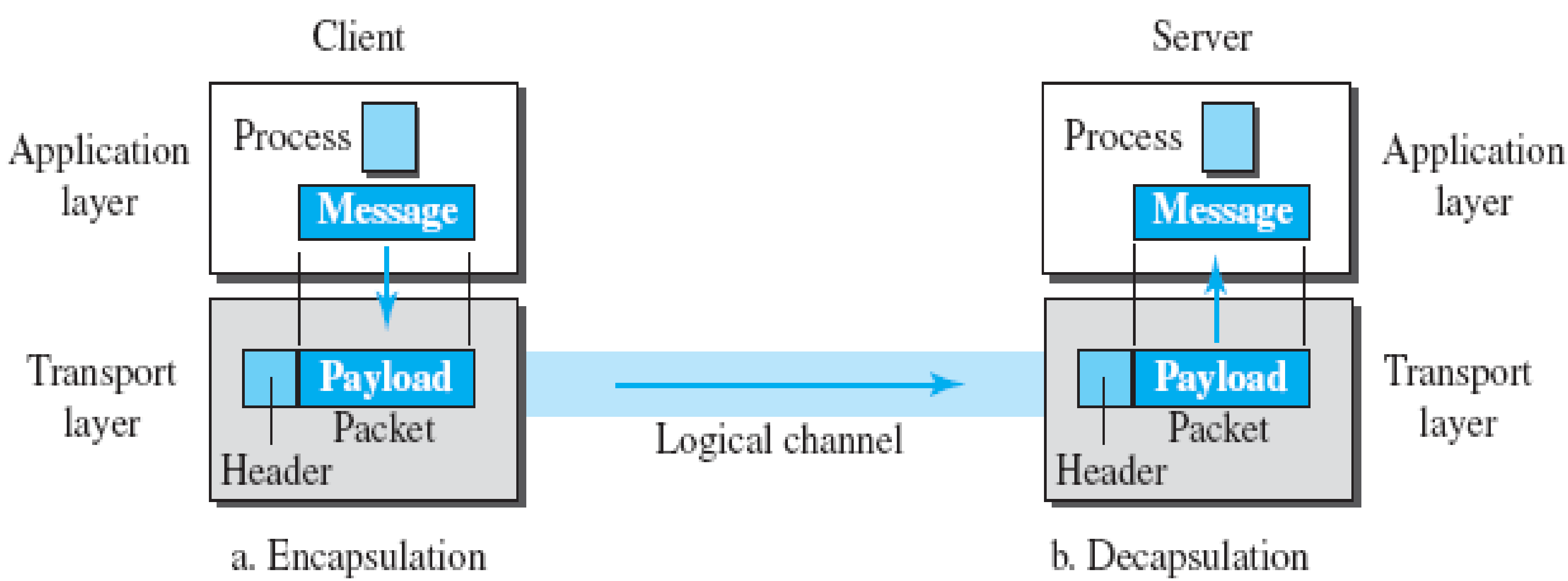
Figure 23.5 *ICANN ranges*



Transport Layer Services

- ***Encapsulation and Decapsulation***

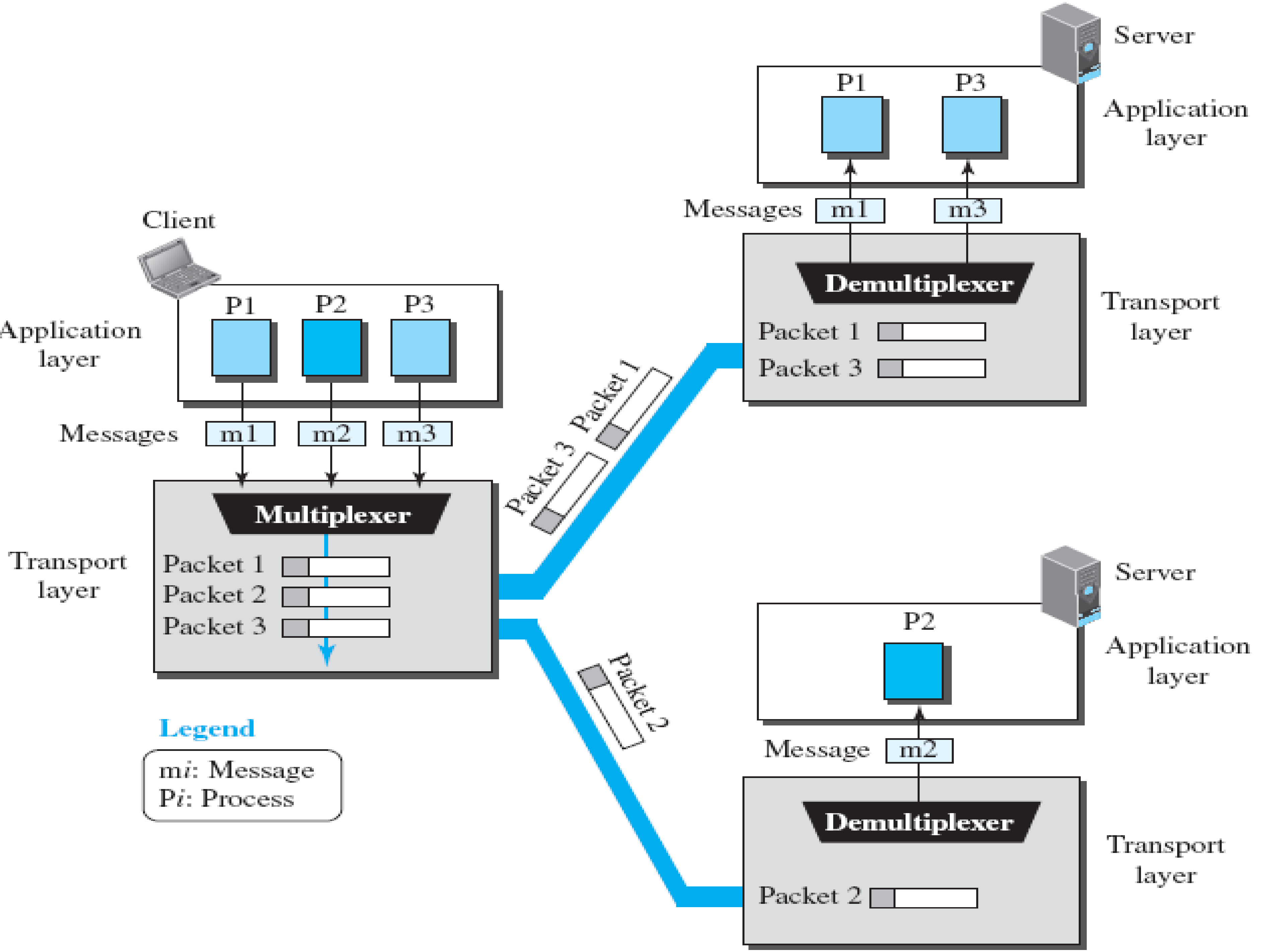
Figure 23.7 *Encapsulation and decapsulation*



Transport Layer Services

- ***Multiplexing and Demultiplexing***

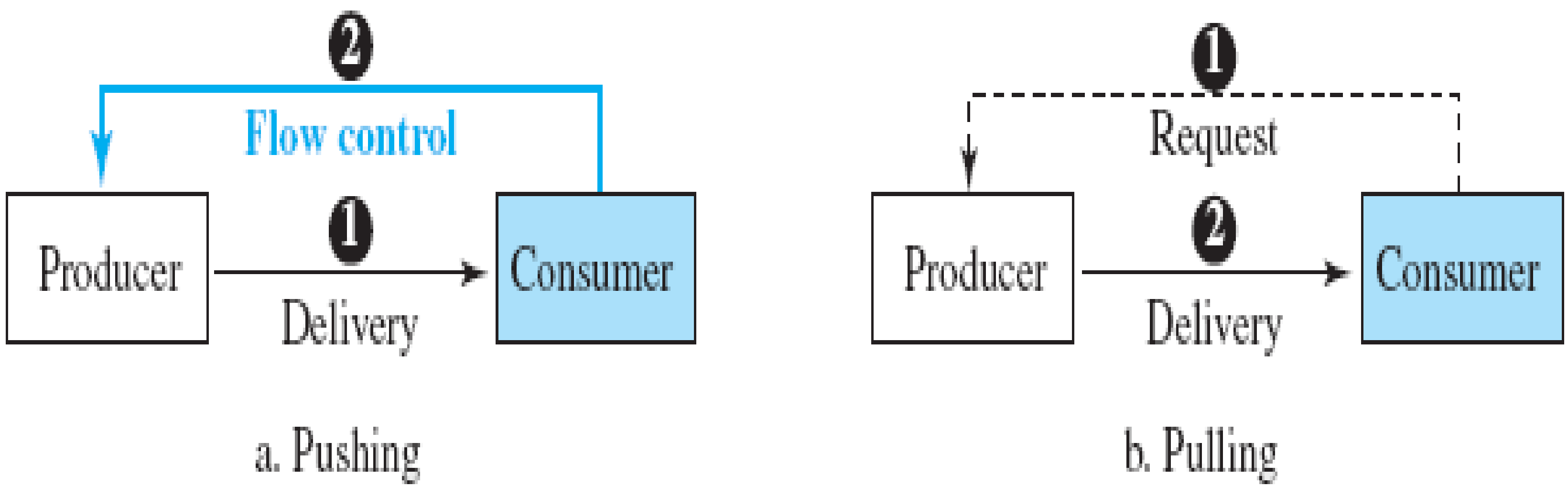
- The transport layer at the source performs multiplexing; the transport layer at the destination performs demultiplexing


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

Transport Layer Services

- ***Flow Control***
- balance between production and consumption rates.
- ***Pushing or Pulling***
- ***Pushing -*** If the sender delivers items whenever they are produced – without a prior request from the consumer
- ***Pulling -*** If the producer delivers the items after the consumer has requested them

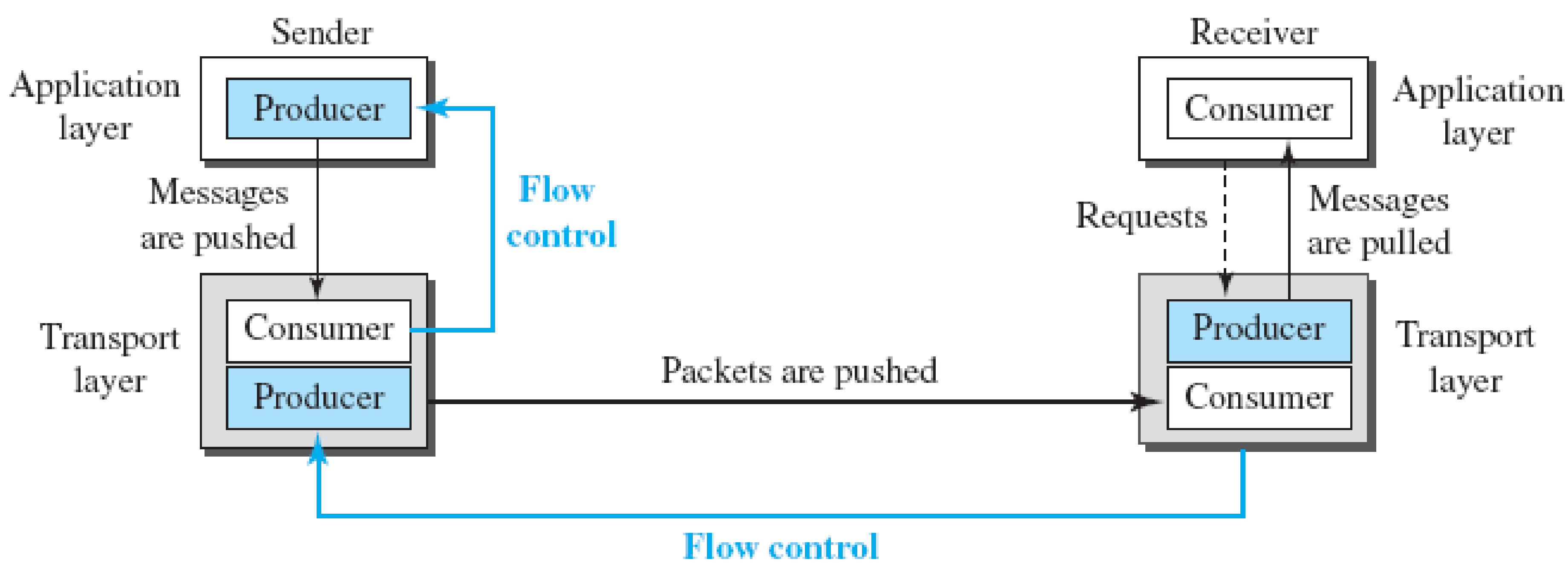
Figure 23.9 *Pushing or pulling*



Transport Layer Services

- ***Flow Control at Transport Layer***

Figure 23.10 *Flow control at the transport layer*



Transport Layer Services

- ***Buffers***
- use two buffers: one at the sending transport layer and the other at the receiving transport layer.

- When the buffer of the sending transport layer is full, it informs the application layer to stop passing chunks of messages.

- When the buffer of the receiving transport layer is full, it informs the sending transport layer to stop sending packets.

Transport Layer Services

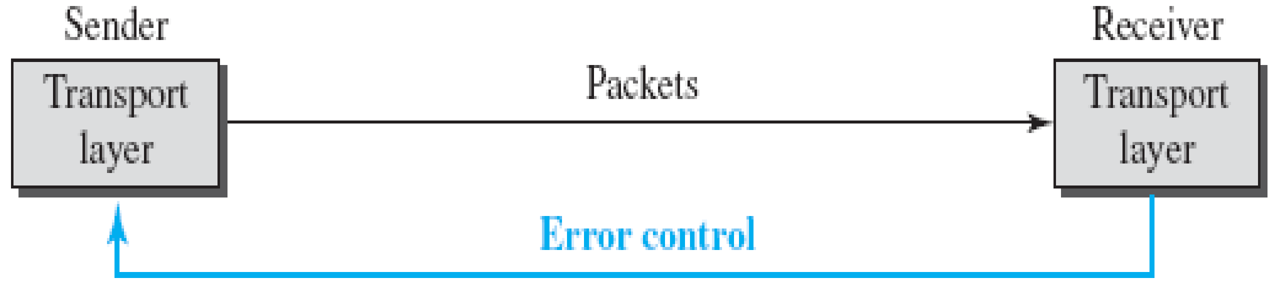
- ***Error Control***
- Error control at the transport layer is responsible for

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

Transport Layer Services

- Receiving transport layer manages error control, most of the time, by informing the sending transport layer about the problems.

Figure 23.11 *Error control at the transport layer*



Transport Layer Services

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

- ***Acknowledgment***
- both positive and negative ack signals as error control

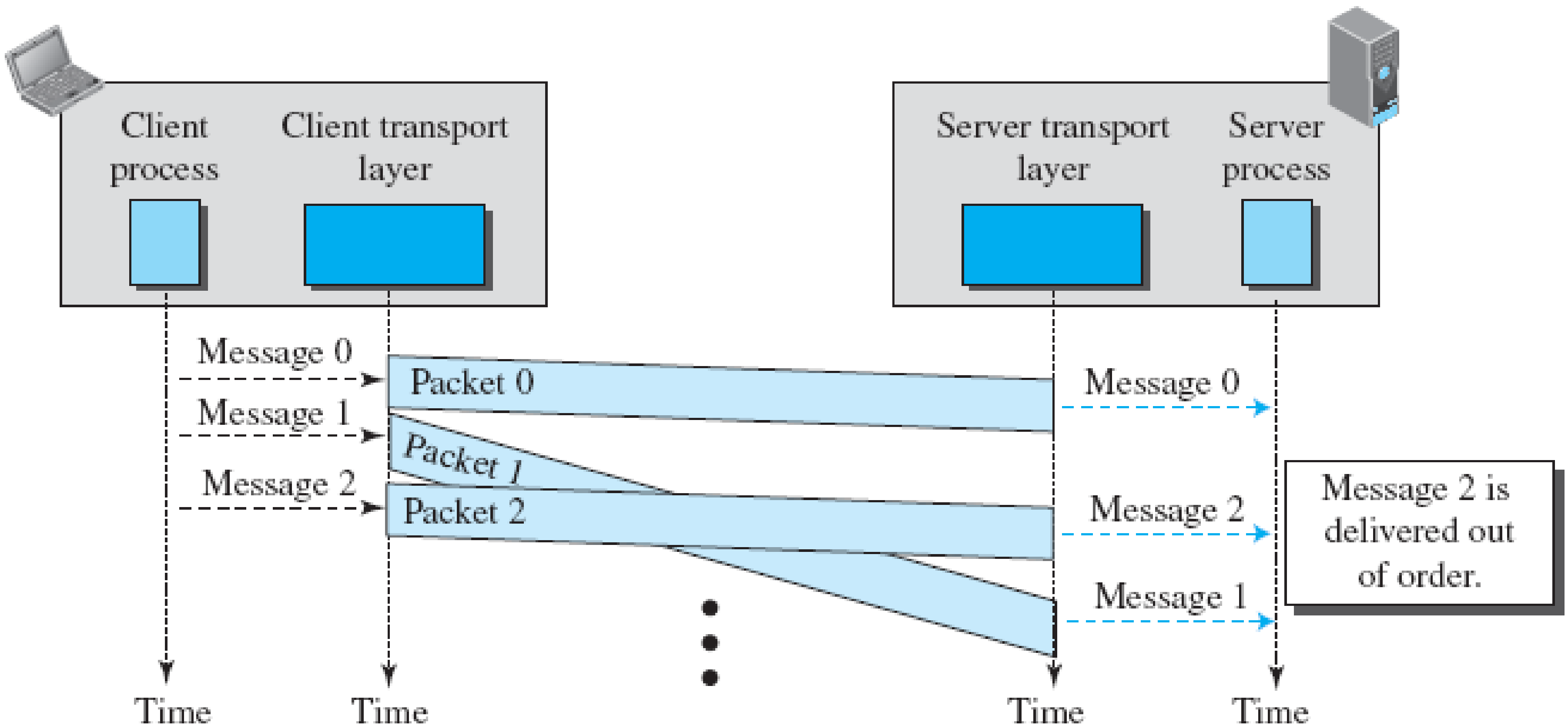
Transport Layer Services

- ***Congestion Control***
- Congestion control refers to the mechanisms and techniques that control the congestion and keep the load below the capacity.

- Congestion at the transport layer is actually the result of congestion at the network layer, which manifests itself at the transport layer.

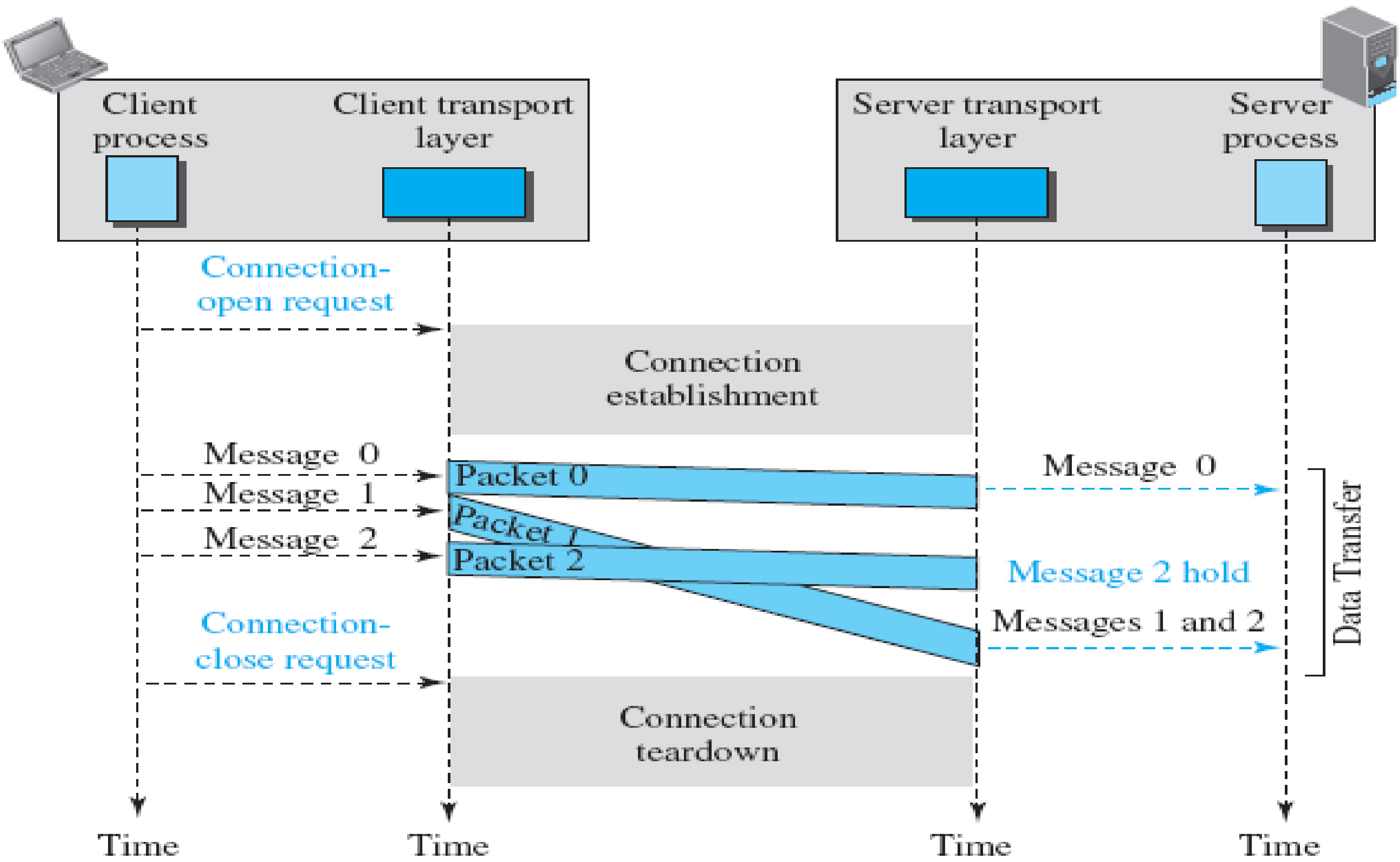
Connectionless service

Figure 23.14 *Connectionless service*



Connection-Oriented Service

Figure 23.15 *Connection-oriented service*

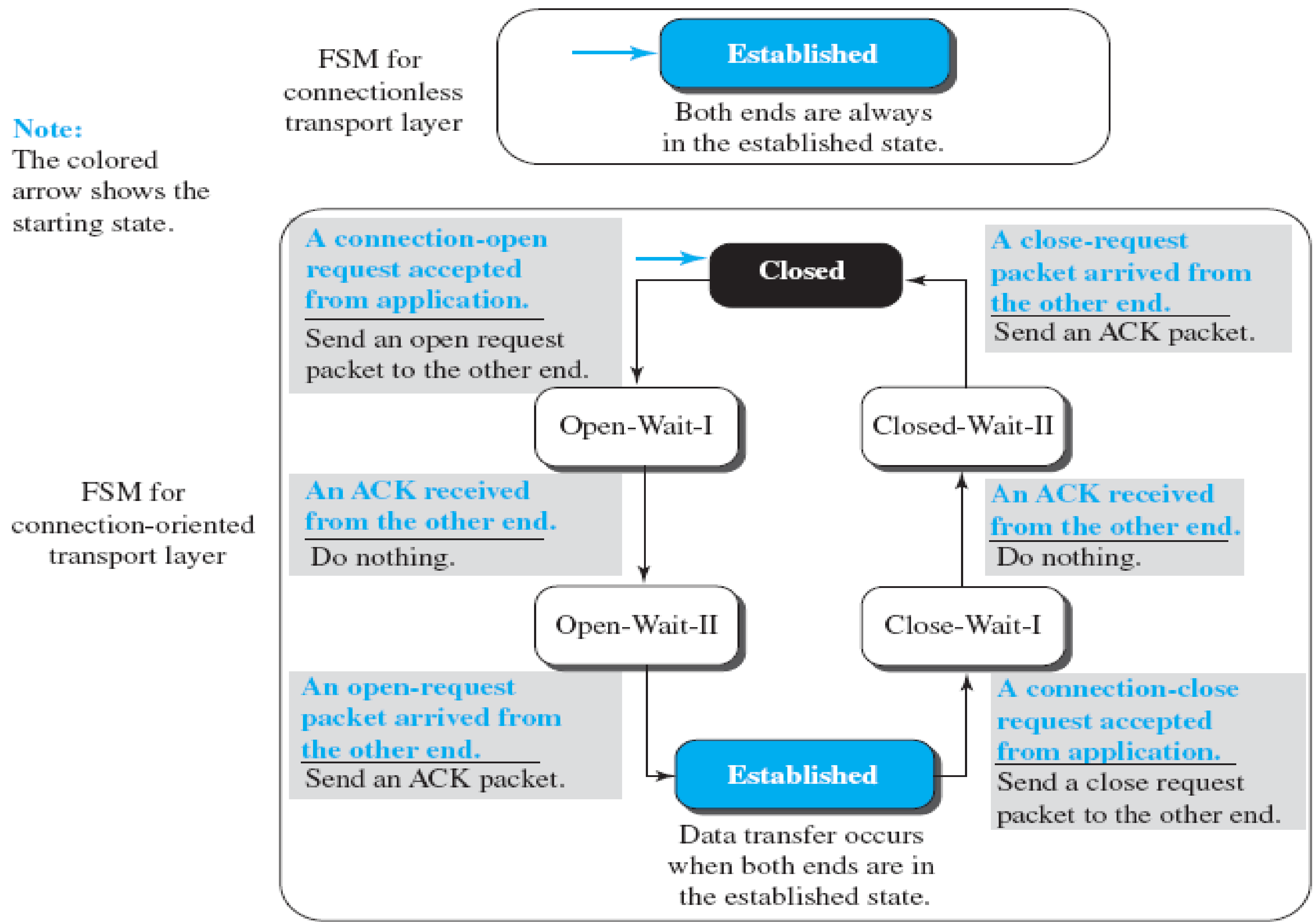


Finite State Machine

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

- each transport layer (sender or receiver) is taught as a machine with a finite number of states.

Figure 23.16 *Connectionless and connection-oriented service represented as FSMs*



USER DATAGRAM PROTOCOL

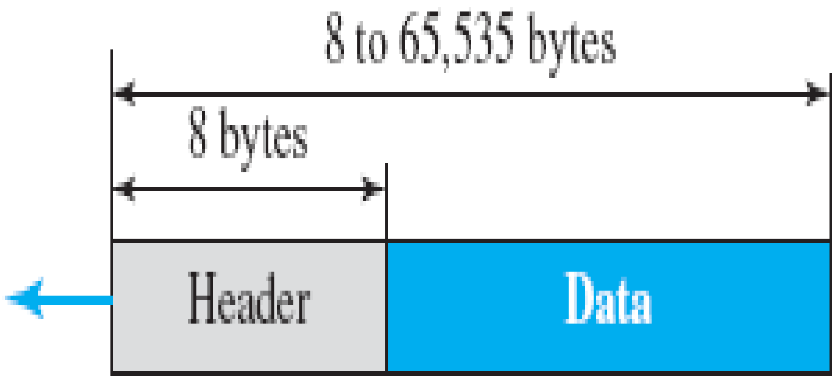
- UDP is an unreliable connectionless transport-layer protocol used for its simplicity and efficiency in applications where error control can be provided by the application-layer process.

Why use UDP?

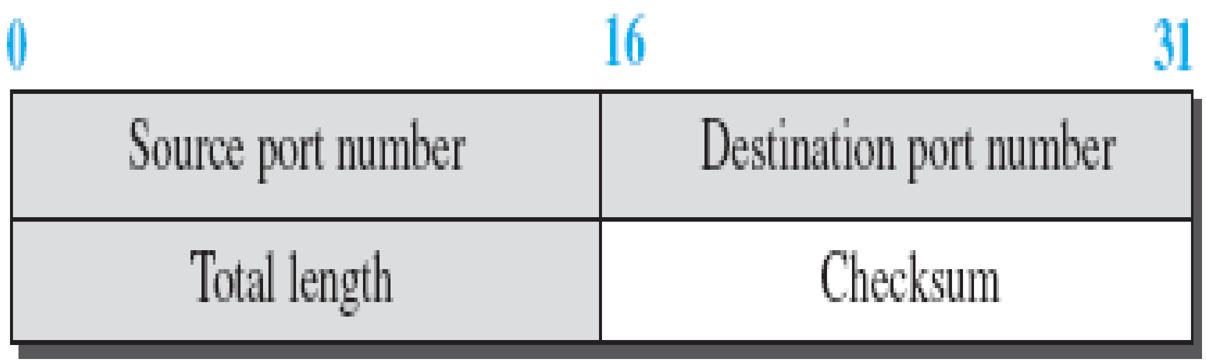
- UDP is a very simple protocol using a minimum of overhead.

- If a process wants to send a small message and does not care much about reliability, it can use UDP.

Figure 24.2 *User datagram packet format*



a. UDP user datagram



b. Header format

Example 14.1

The following is a dump of a UDP header in hexadecimal format.

CB84000D001C001C

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

Solution

a. The source port number is the first four hexadecimal digits ($CB84_{16}$), which means that the source port number is 52100.
b. The destination port number is the second four hexadecimal digits ($000D_{16}$), which means that the destination port number is 13.
c. The third four hexadecimal digits ($001C_{16}$) define the length of the whole UDP packet as 28 bytes.
d. The length of the data is the length of the whole packet minus the length of the header, or $28 - 8 = 20$ bytes.
e. Since the destination port number is 13 (well-known port), the packet is from the client to the server.
f. The client process is the Daytime (see Table 14.1).

UDP Services

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

Table 14.1 *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	Domain	Domain Name Service (DNS)
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)

UDP Services

- ***Checksum***
- What is Pseudo header?
- In Simple words, Pseudo header is one type of demo header that basically helps in calculating the CheckSum of TCP/UDP Packets. From the TCP or UDP point of view, the TCP packet does not contain IP addresses. Thus, to do a proper checksum, a "pseudo-header" is included.

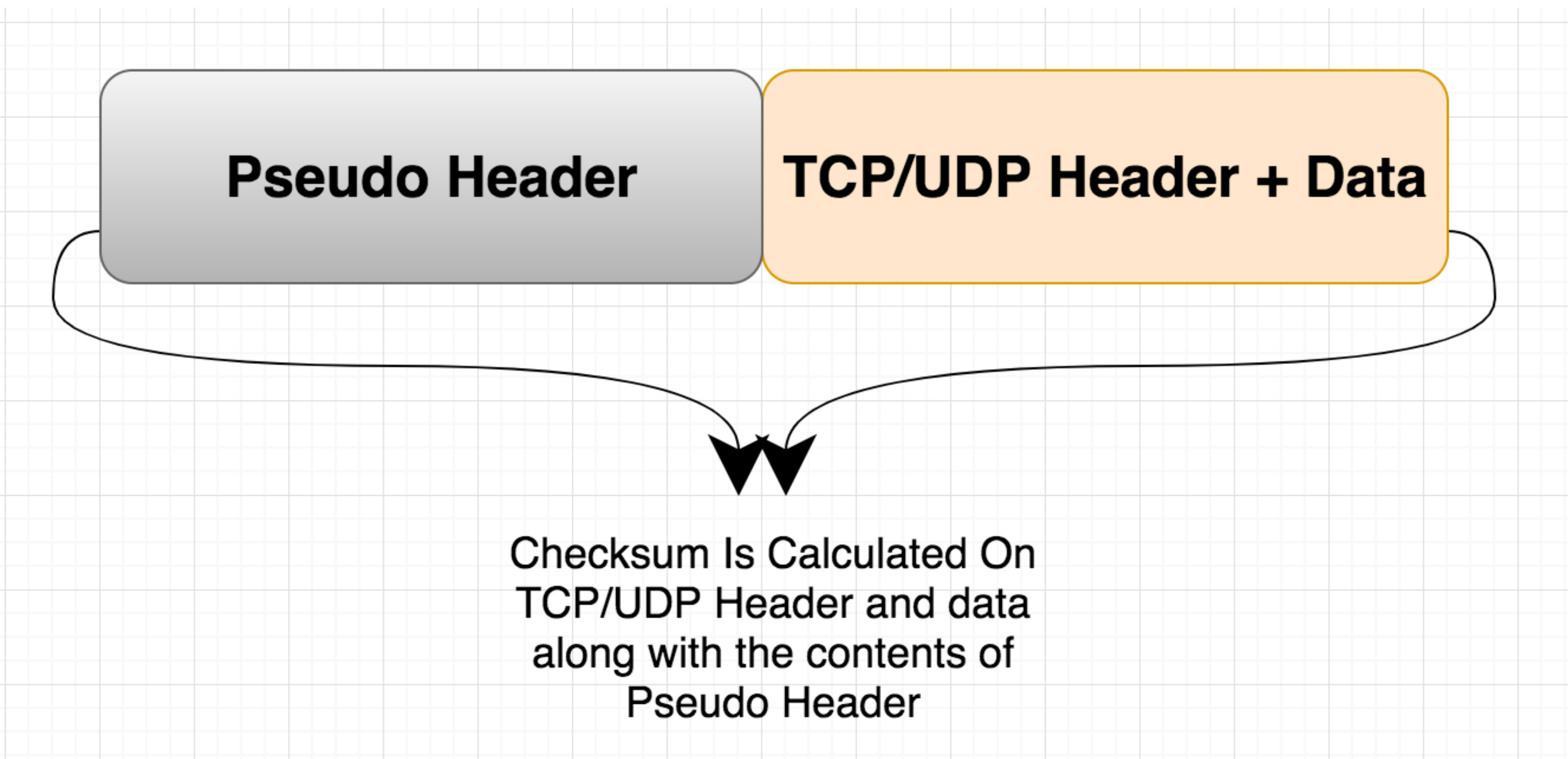

Pseudo Header
TCP/UDP Header + Data
Checksum Is Calculated On
TCP/UDP Header and data
along with the contents of
Pseudo Header

Figure 24.3 *Pseudoheader for checksum calculation*

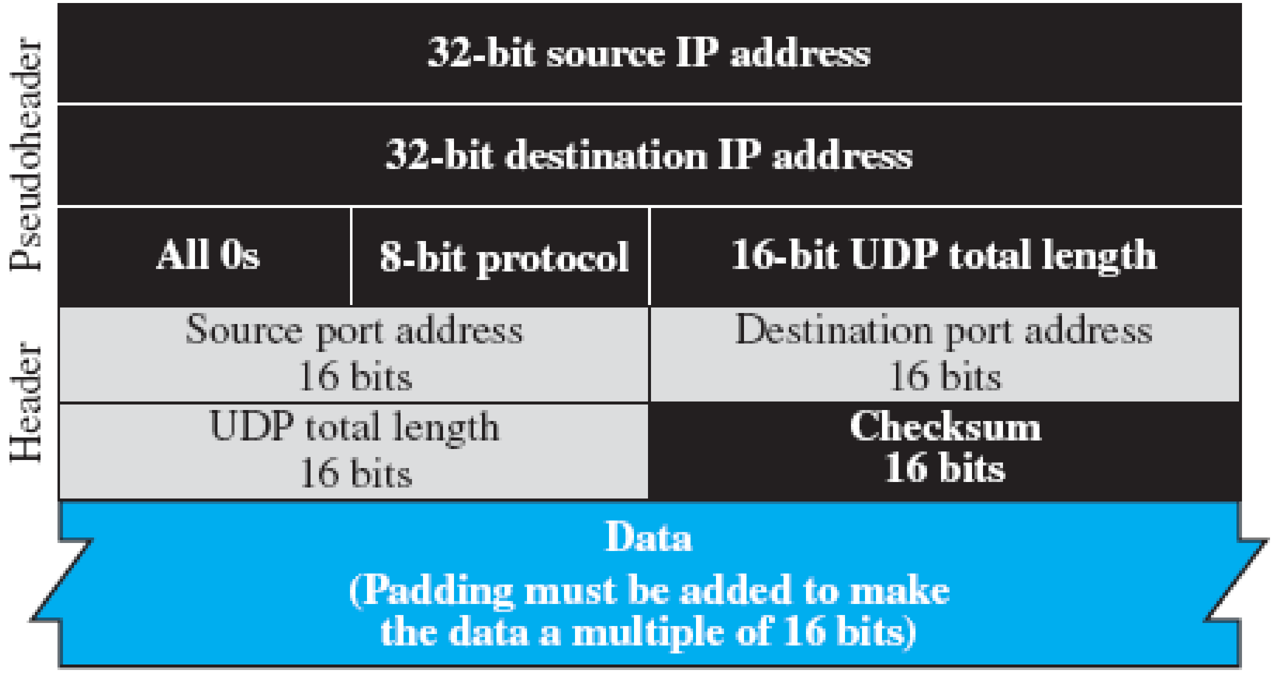
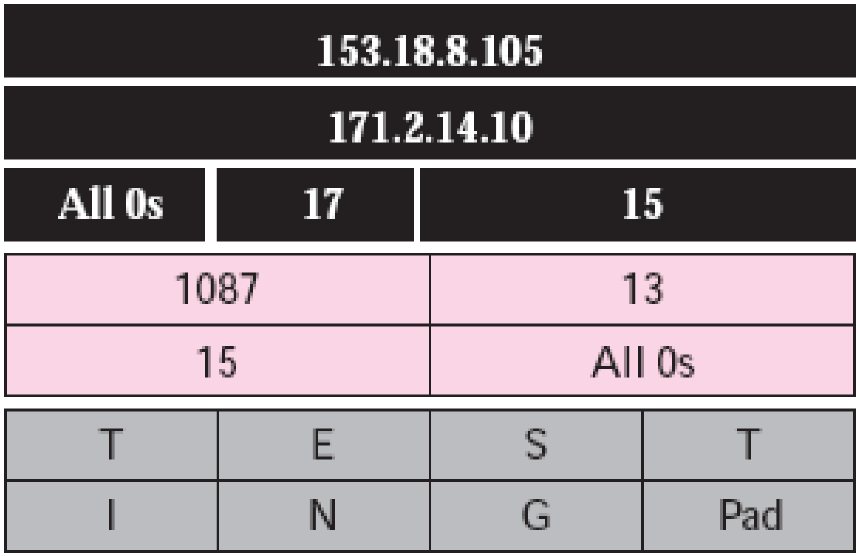


Figure 14.4 *Checksum calculation of a simple UDP user datagram*



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

UDP Services

- **Connectionless Services**
- UDP provides a *connectionless service.*

- Each user datagram sent by UDP is an independent datagram.

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

UDP Services

- **Flow Control**
- UDP is a very simple protocol. There is no *flow control*, and hence no window mechanism.

- **Error Control**
- There is no error control mechanism in UDP except for the checksum.

- Sender does not know if a message has been lost or duplicated.

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

- The lack of error control means that the process using UDP should provide for this service, if needed.

UDP Services

- **Congestion Control**
- UDP is a connectionless protocol, it does not provide congestion control.

- ***Encapsulation and Decapsulation***

Figure 14.5 *Encapsulation and decapsulation*

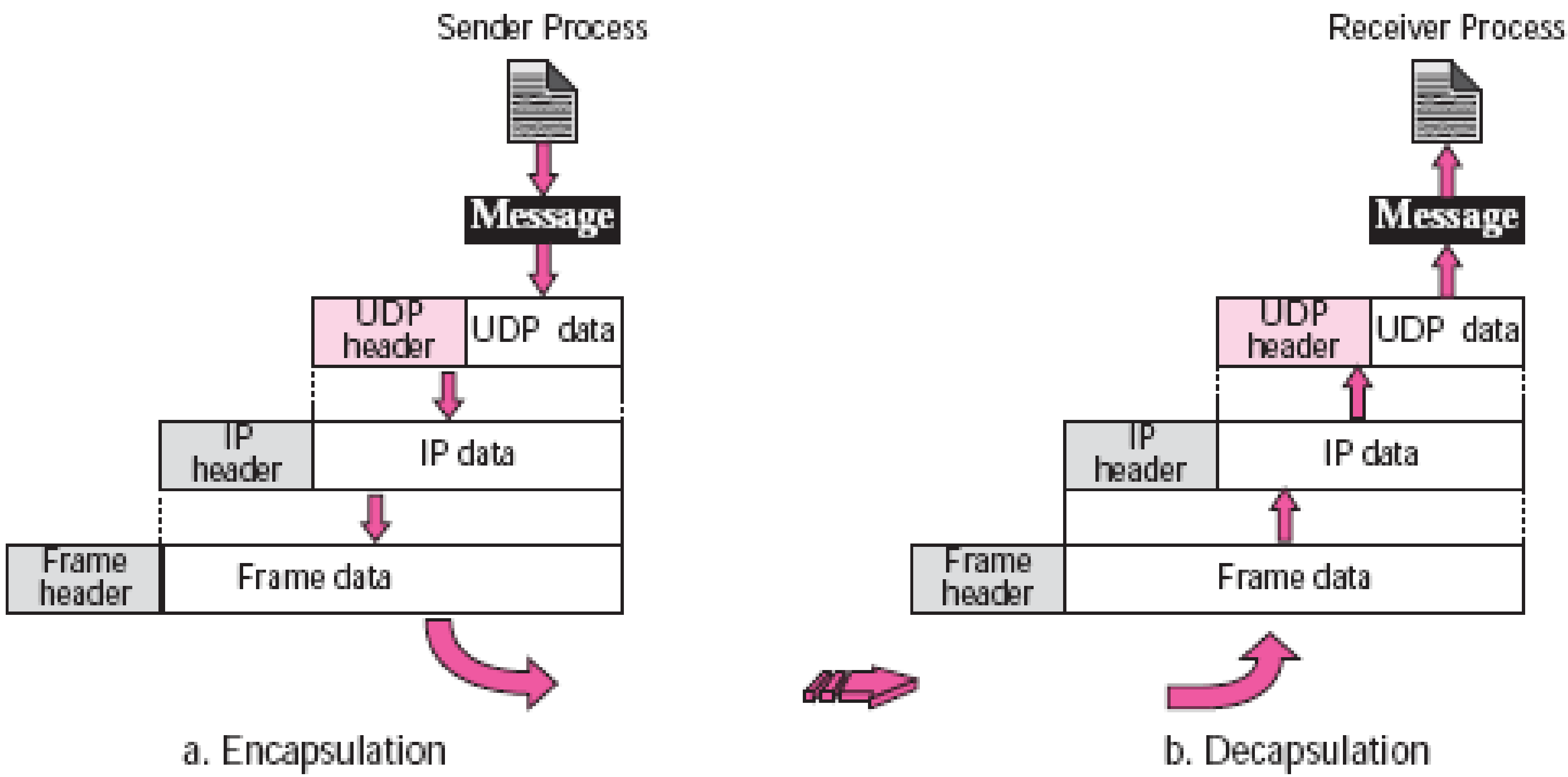
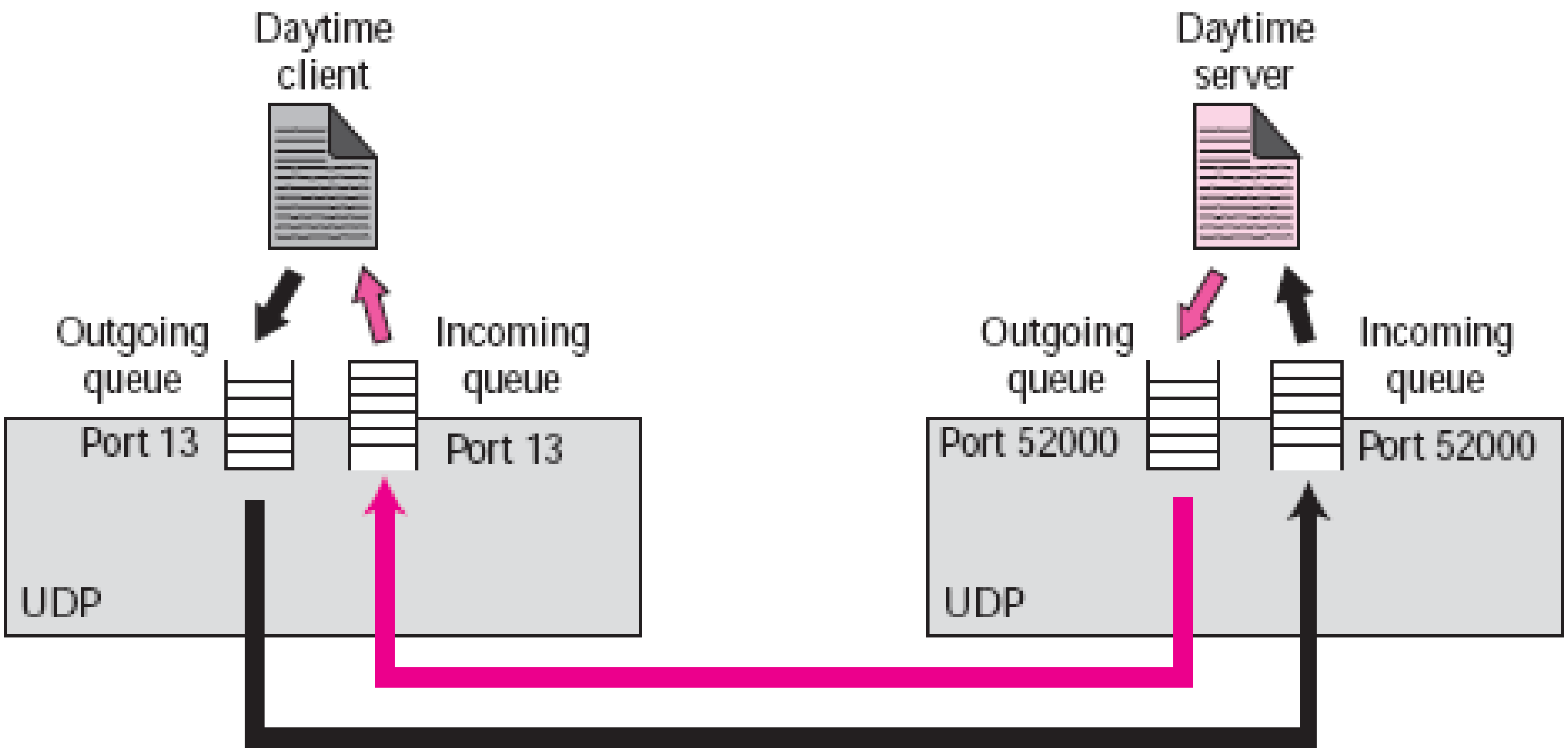


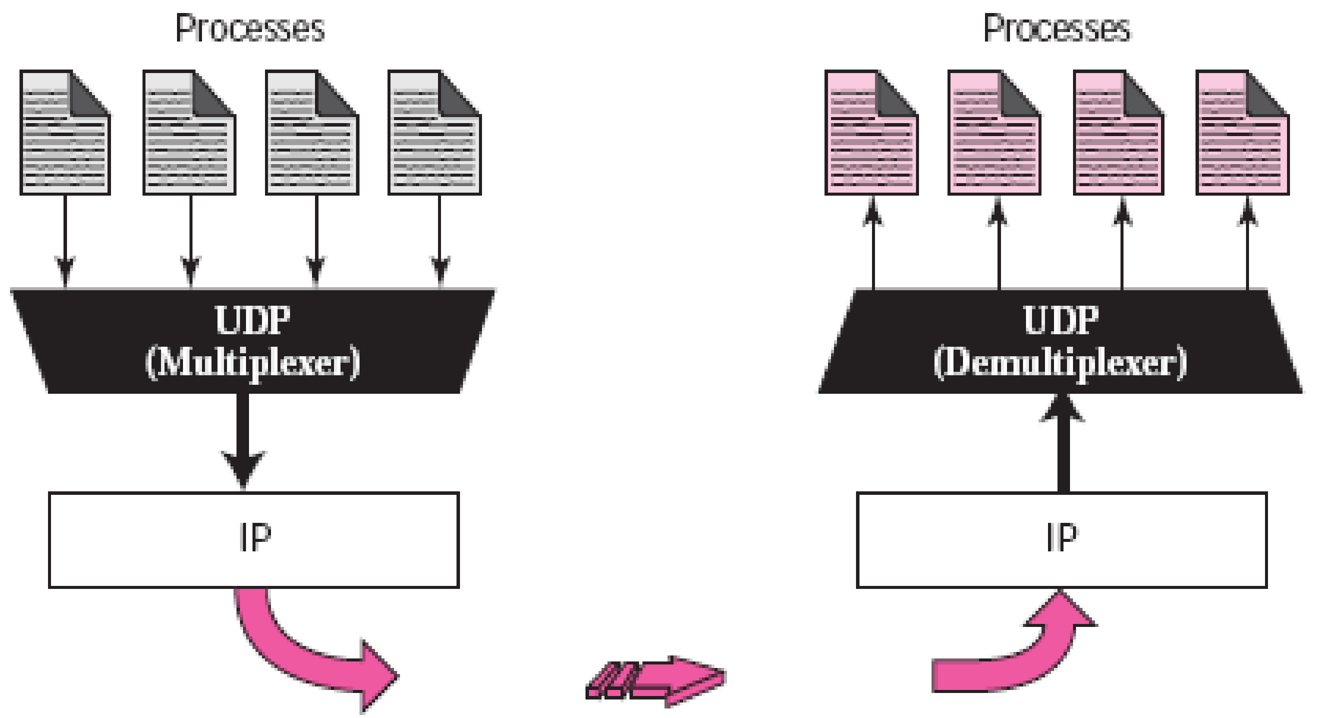
Figure 14.6 *Queues in UDP*



UDP Services

- **Multiplexing and Demultiplexing**

Figure 14.7 *Multiplexing and demultiplexing*



UDP Applications

- Trivial File Transfer Protocol (TFTP) process includes flow and error control. It can easily use UDP.
- UDP is a suitable transport protocol for multicasting.
- UDP is used for management processes such as SNMP
- UDP is used for some route updating protocols such as Routing Information Protocol (RIP)
- UDP is normally used for interactive real-time applications that cannot tolerate uneven delay between sections of a received message.

TRANSMISSION CONTROL PROTOCOL

- Transmission Control Protocol (TCP) is a **connection-oriented, reliable protocol.**

- TCP explicitly defines **connection establishment, data transfer, and connection termination phases** to provide a connection-oriented service.

- TCP uses a combination of **GBN and SR protocols** to provide reliability.

- TCP uses **checksum (for error detection), retransmission of lost or corrupted packets**, cumulative and selective acknowledgments, and timers.

TCP Services

- ***Process-to-Process Communication***

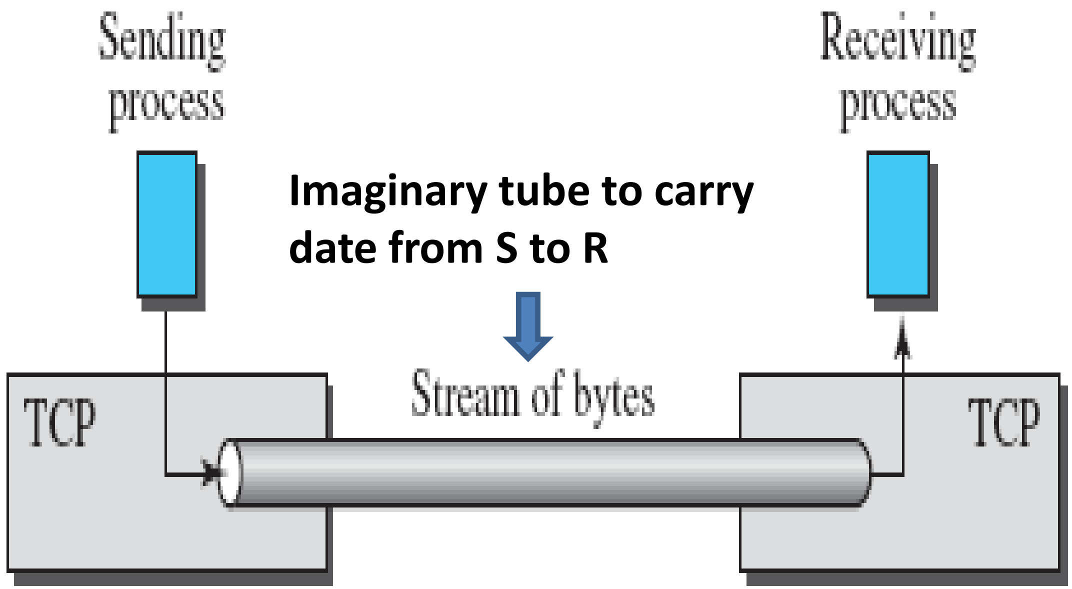
TCP provides process-to-process communication using port numbers.

- ***Stream Delivery Service***

The sending process produces (writes to) the stream and the receiving process consumes (reads from) it.

TCP Services

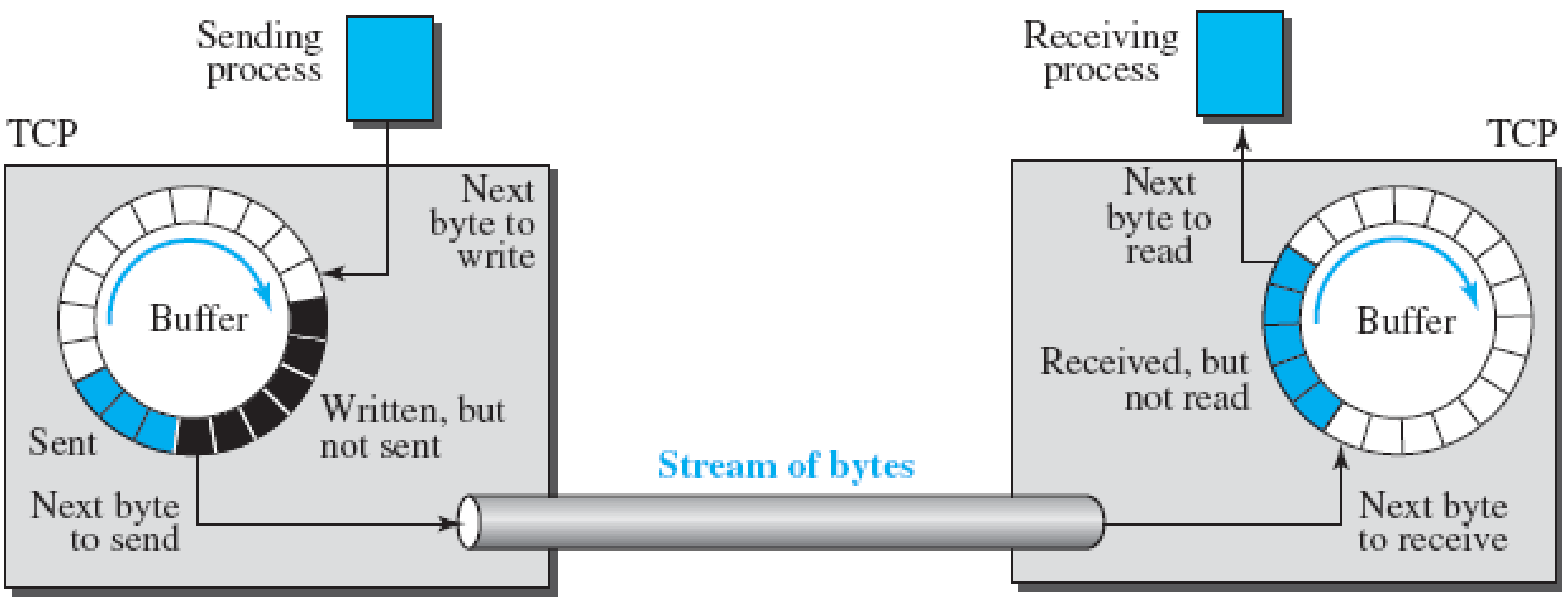
Figure 24.4 *Stream delivery*



TCP Services

- ***Sending and Receiving Buffers***

Figure 24.5 *Sending and receiving buffers*



TCP Services

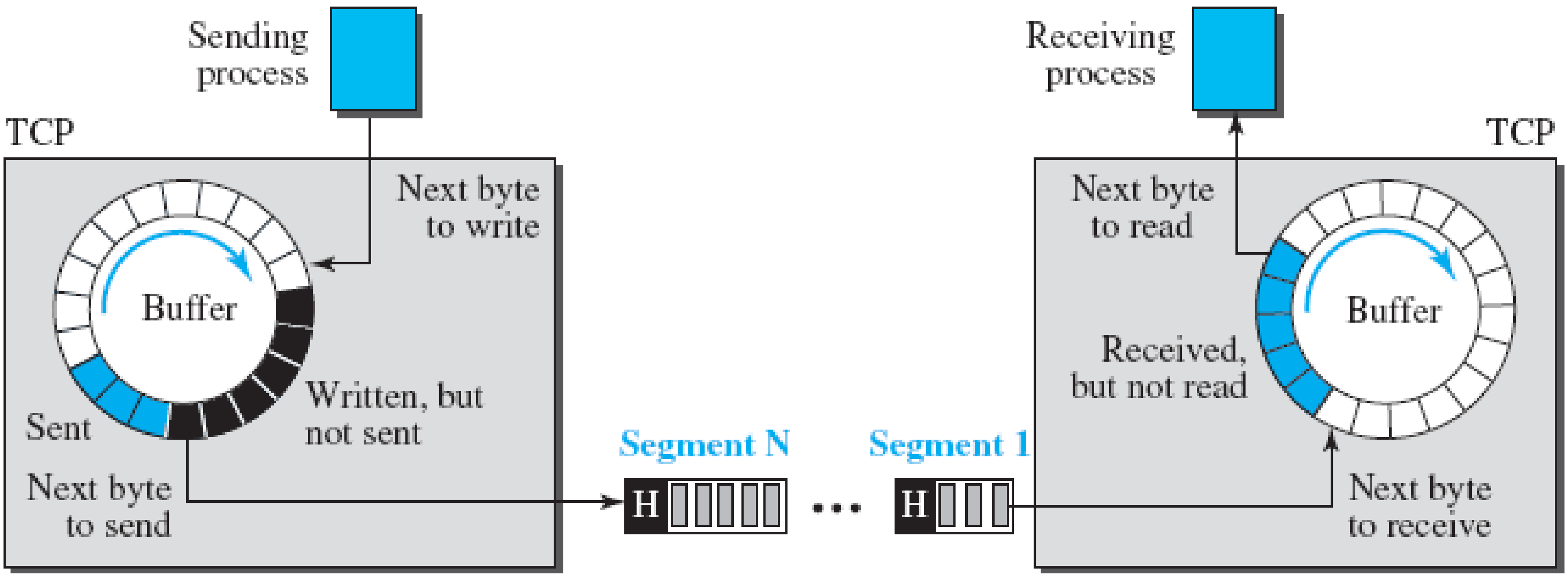
- ***Segments***
- Buffering - disparity between the speed of the producing and consuming processes.

- The network layer, as a service provider for TCP, needs to send data in packets, not as a stream of bytes.

- TCP groups a number of bytes together into a packet called a segment.

- Header added to every segment

Figure 24.6 *TCP segments*



TCP Services

- ***Full-Duplex Communication***
- TCP offers full-duplex services

- ***Multiplexing and Demultiplexing***
- Like UDP, TCP performs multiplexing at the sender and demultiplexing at the receiver.

TCP Services

- ***Connection-Oriented Service***
- TCP is connection oriented

1. The two TCP’s establish a logical connection between them.
2. Data are exchanged in both directions.
3. The connection is terminated.

- logical connection, not a physical connection.

TCP Services

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

TCP Features

- ***Numbering System***
- sequence number and the acknowledgment number.
- These two fields refer to a byte number and not a segment number.

- ***Byte Number***
- TCP numbers all data bytes (octets) that are transmitted in a connection.
- Numbering is arbitrary

TCP Features

- ***Sequence Number***

1. The sequence number of the first segment is the ISN (initial sequence number),

- which is a random number.

2. The sequence number of any other segment is the sequence number of the previous segment plus the number of bytes (real or imaginary) carried by the previous segment.

Example 24.7

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 are sent in five segments, each carrying 1000 bytes?

Solution

The following shows the sequence number for each segment:

Segment 1	→	Sequence Number:	10001	Range:	10001	to	11000
Segment 2	→	Sequence Number:	11001	Range:	11001	to	12000
Segment 3	→	Sequence Number:	12001	Range:	12001	to	13000
Segment 4	→	Sequence Number:	13001	Range:	13001	to	14000
Segment 5	→	Sequence Number:	14001	Range:	14001	to	15000

TCP Features

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

TCP Features

- **Segment**

Figure 24.7 *TCP segment format*

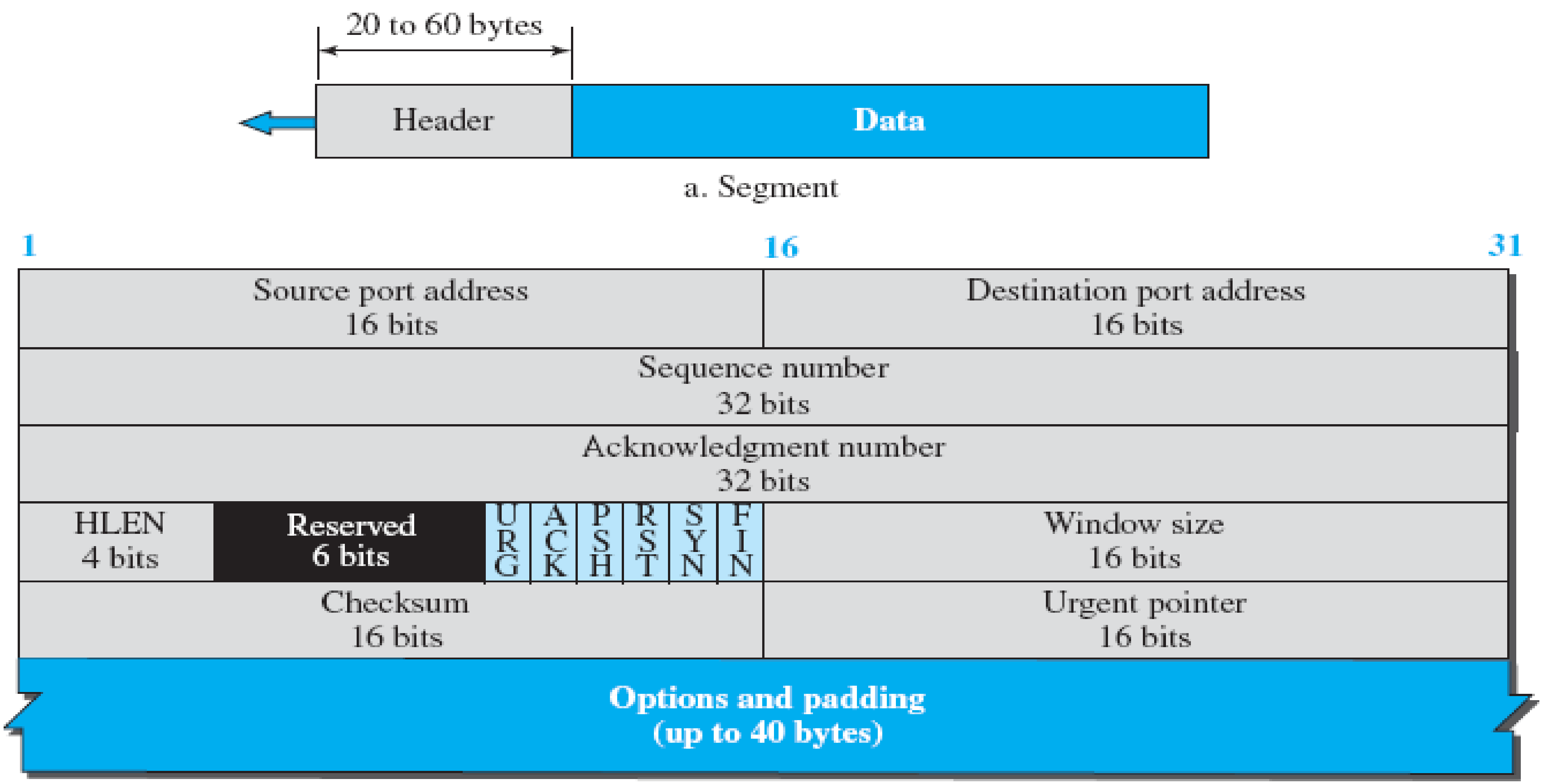
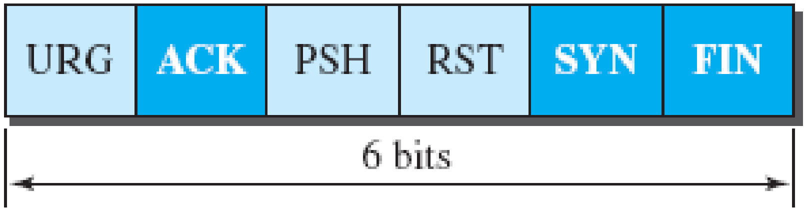
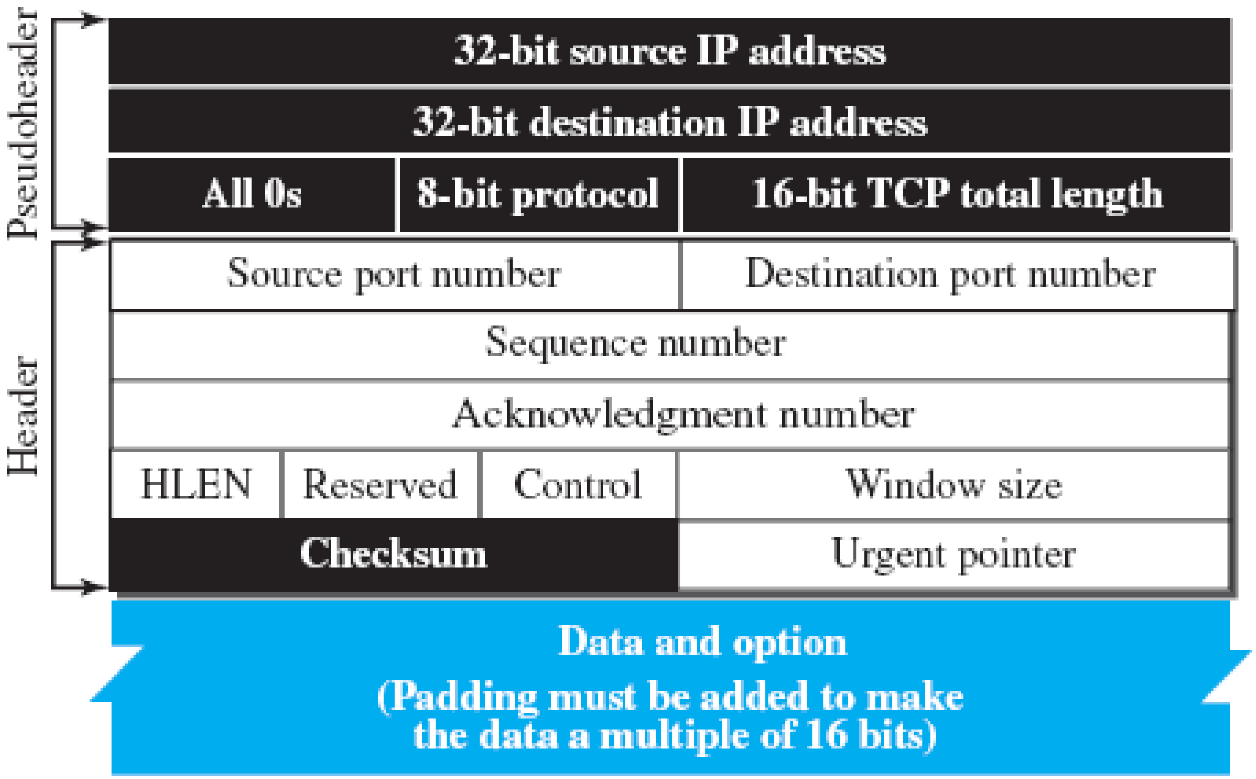


Figure 24.8 *Control field*



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

Figure 24.9 *Pseudoheader added to the TCP datagram*



The use of the checksum in TCP is mandatory.

TCP Features

- ***Encapsulation***
- A TCP segment encapsulates the data received from the application layer. The TCP segment is encapsulated in an IP datagram, which in turn is encapsulated in a frame at the data-link layer.

A TCP Connection

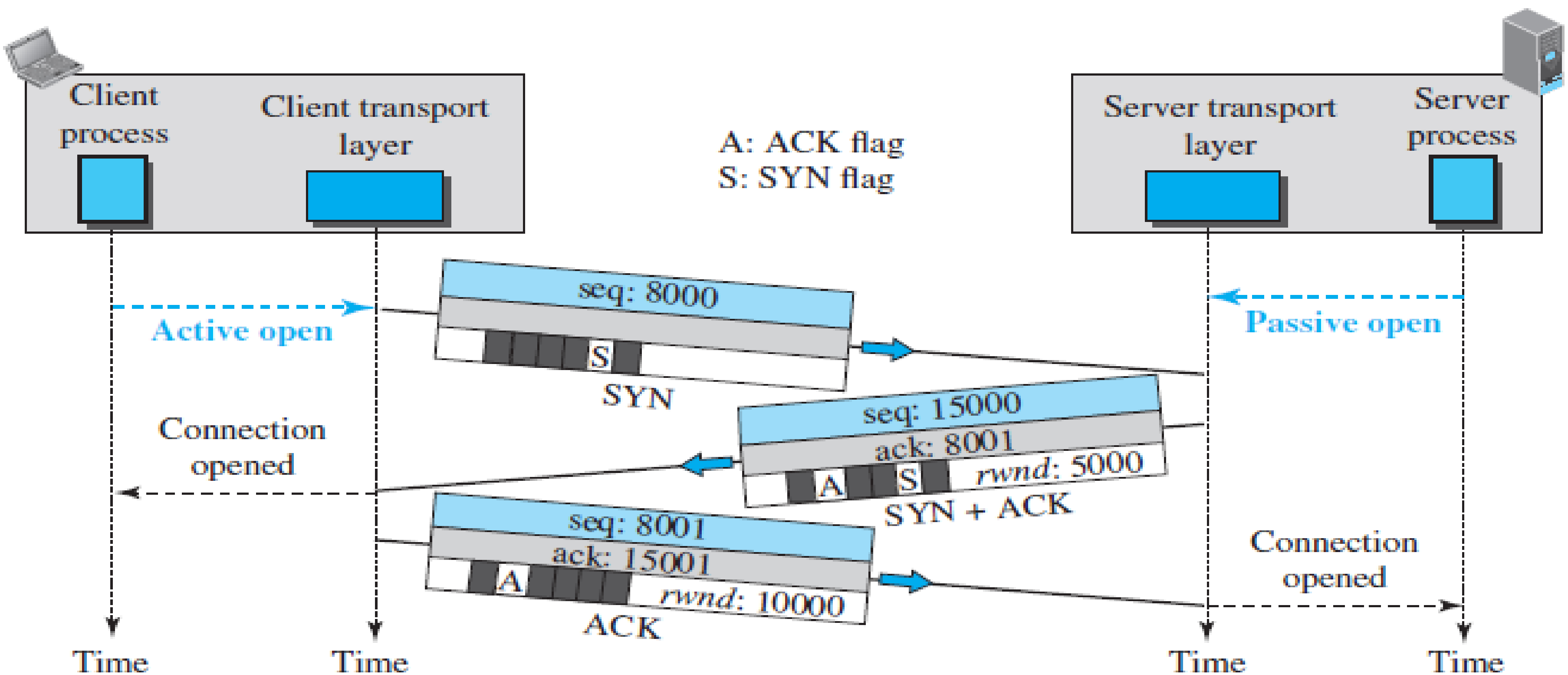
- Connection Establishment
- Data Transfer
- Connection Termination

Connection Establishment

- TCP transmits data in full duplex mode.
- Three-Way Handshake
  - Server program tells its TCP that it is ready to accept a connection. This request is called a *passive open*.
  - The client program issues a request for an *active open*.

Connection Establishment

Figure 24.10 *Connection establishment using three-way handshaking*



Connection Establishment

- The client sends the first segment, a SYN segment
  - Only the SYN flag is set.
  - This segment is for synchronization of sequence numbers
  - segment does not contain an acknowledgment number
  - Does not define window size.
  - **SYN segment cannot carry data, but it consumes one sequence number** (for one imaginary byte)**.**

Connection Establishment

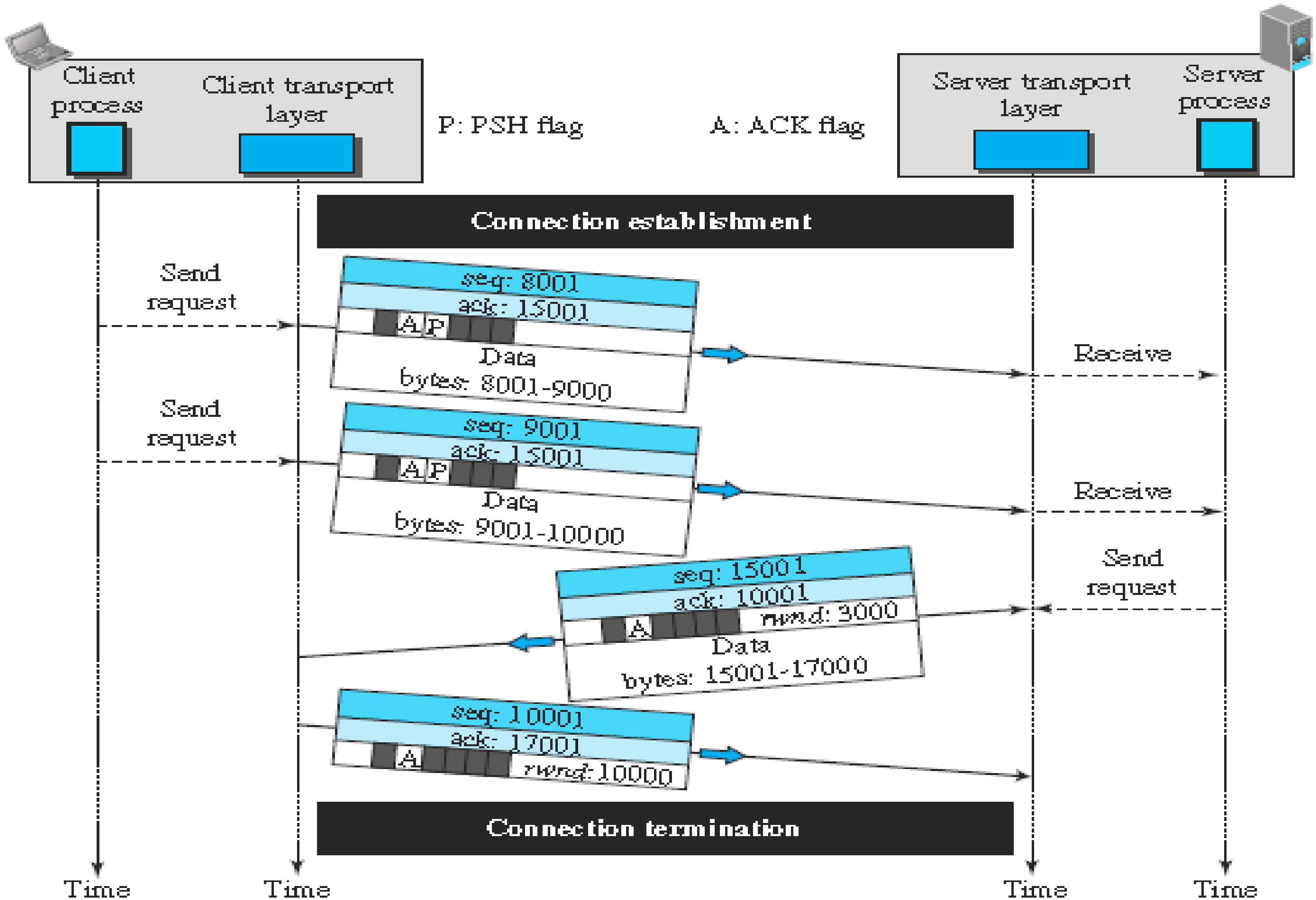
- Server sends the second segment, a SYN + ACK.
- This segment has a dual purpose:
  - SYN segment for communication in the other direction(initialize a sequence number for numbering the bytes sent from the server to the client).
  - The server also acknowledges the receipt of the SYN segment from the client by setting ACK flag.
- Because segment contains an ACK, it also need to define window size, *rwnd*(to be used by client).
- **A SYN + ACK segment cannot carry data, but it does consume one sequence number.**

Connection Establishment

- The client sends the third segment – an ACK segment.
- **An ACK segment, if carrying no data, consumes no sequence number.**

Data Transfer

Figure 24.11 *Data transfer*



Data Transfer

- After connection establishment, bidirectional data transfer can take place.
- acknowledgment can be piggybacked with the data.
- Pushing Data:
  - Delayed transmission and delayed delivery of data may not be acceptable by the application program.
  - application program at the sender can request a *push* operation
  - TCP can choose whether or not to use this feature.

Data Transfer

- After connection establishment, bidirectional data transfer can take place
- Urgent Data :
  - Each byte of data has a position in the stream. However, there are occasions in which an application program needs to send *urgent* bytes
  - send a segment with the URG bit set.
  - Sending TCP send a segment with the URG bit set.
  - Sending TCP creates a segment and inserts the urgent data at the beginning of the segment, rest segment can carry normal data.
  - The urgent pointer field in the header defines the end of the urgent data (the last byte of urgent data).

Urgent Flag/Pointer

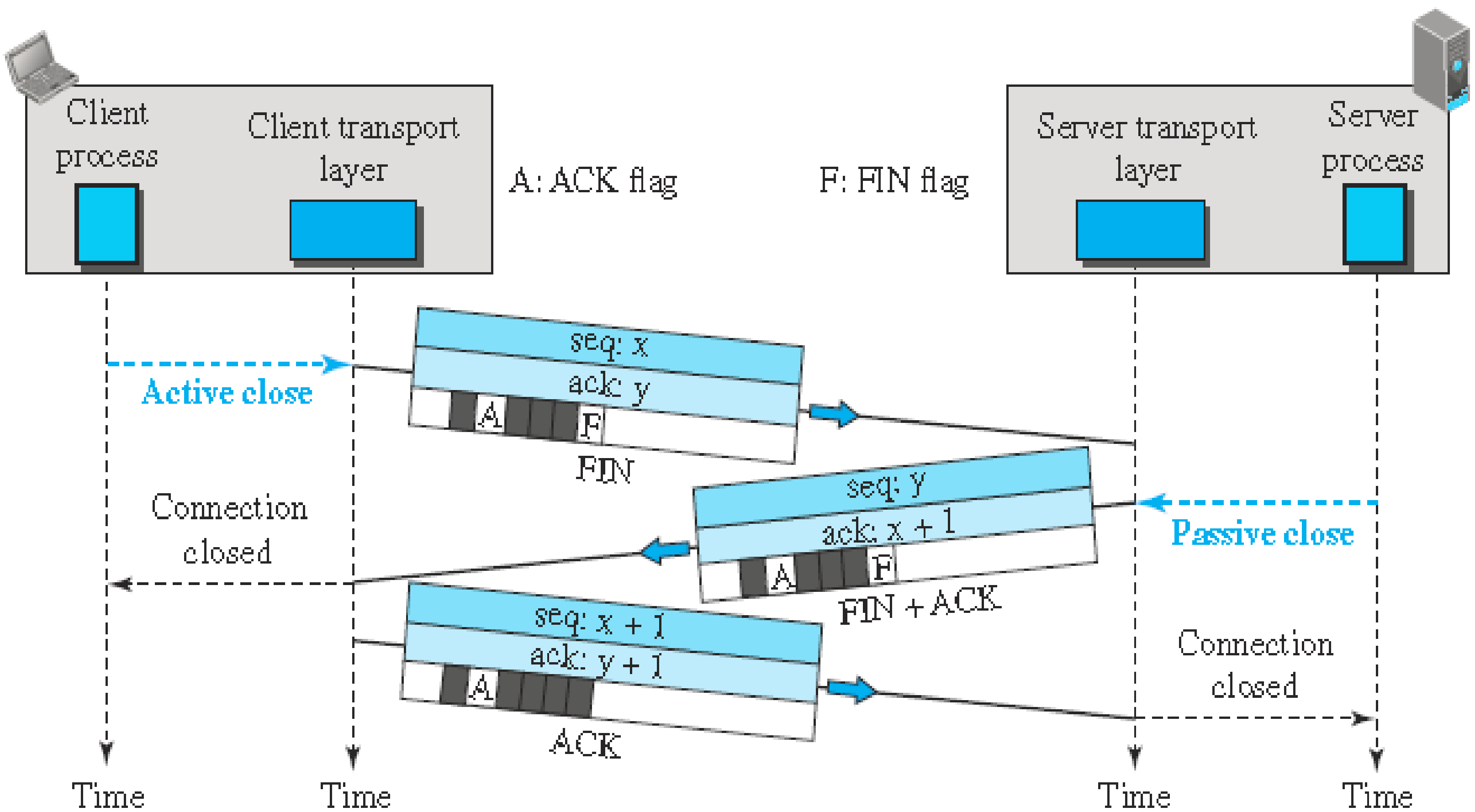
- For example, if
- the segment sequence number is 15000 and the value of the urgent pointer is 200, the first byte of urgent data is the byte 15000 and the last byte is the byte 15200. The rest of the bytes in the segment (if present) are non-urgent.

Connection Termination

- Either of client or server can close the connection, usually initiated by the client.
- Most implementation allow two options:
  - Three way handshaking
  - Four way handshaking with Half-close option.

Connection Termination

Figure 24.12 *Connection termination using three-way handshaking*



Connection Termination

I. Three Way Handshake :

- Client TCP sends a FIN segment in which FIN flag is set.
- FIN segment can include the last chunk of data sent by the client or it can be just a control segment.
- **The FIN segment consumes one sequence number if it does not carry data.**

Connection Termination

II. Three Way Handshake :

- Server TCP, after receiving FIN segment sends a FIN+ACK segment to confirm receipt of FIN segment and also to announce closing of connection in other direction.
- This segment can also carry last chunk of data from server.
- If it does not carry data, it consumes only one sequence number because it needs to be acknowledged.

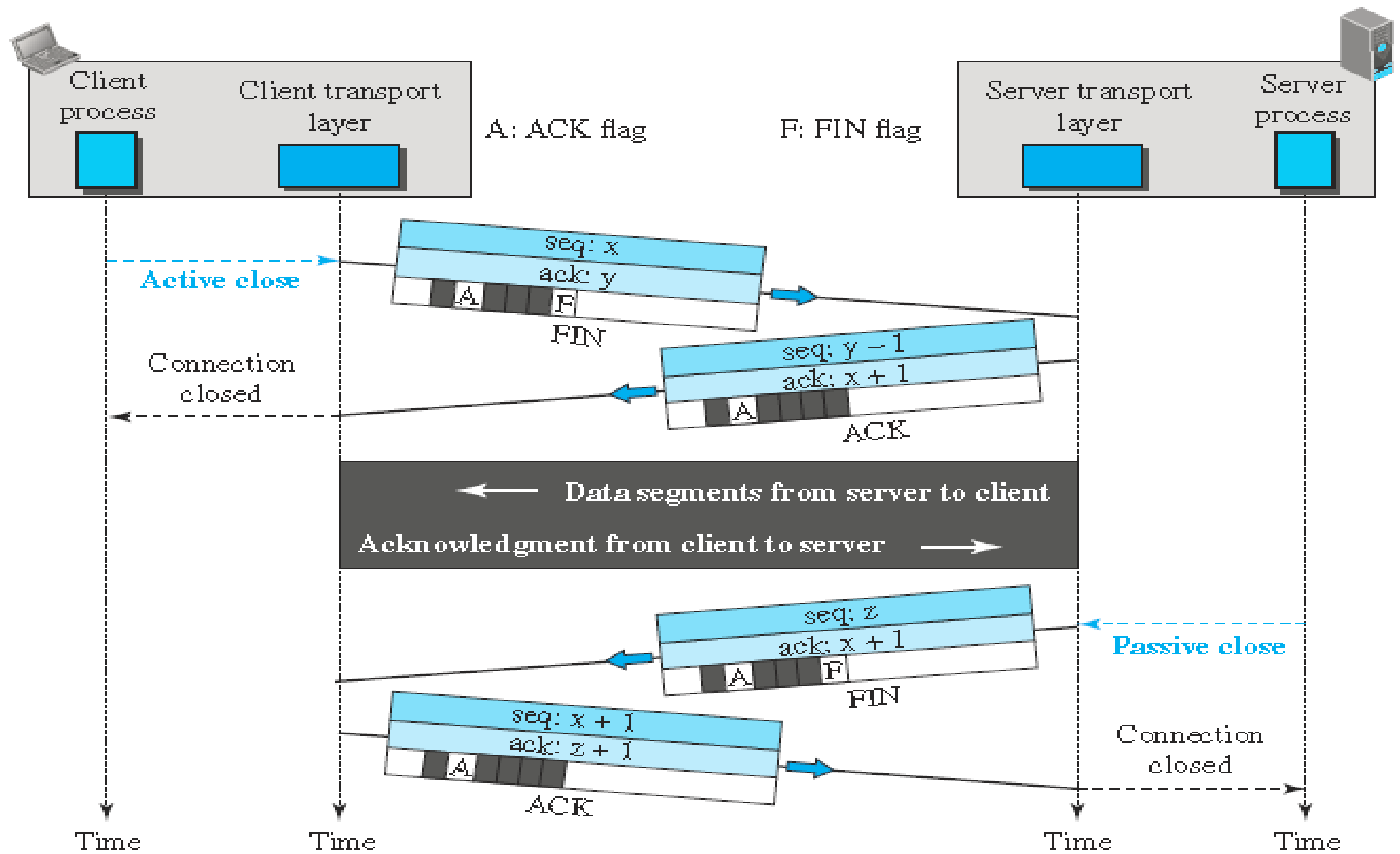
Connection Termination

III. Three Way Handshake :

- Client TCP sends the last segment, an ACK segment, to confirm the receipt of the FIN segment from the TCP server.
- This segment cannot carry data and consumes no sequence numbers

Half Close

Figure 24.13 *Half-close*



Half Close

- one end can stop sending data while still receiving data.

Connection Reset

- TCP at one end,
  - may deny a connection request
  - may abort an existing connection
  - may terminate an idle connection
- All above is done using RST flag.

State Transition Diagram

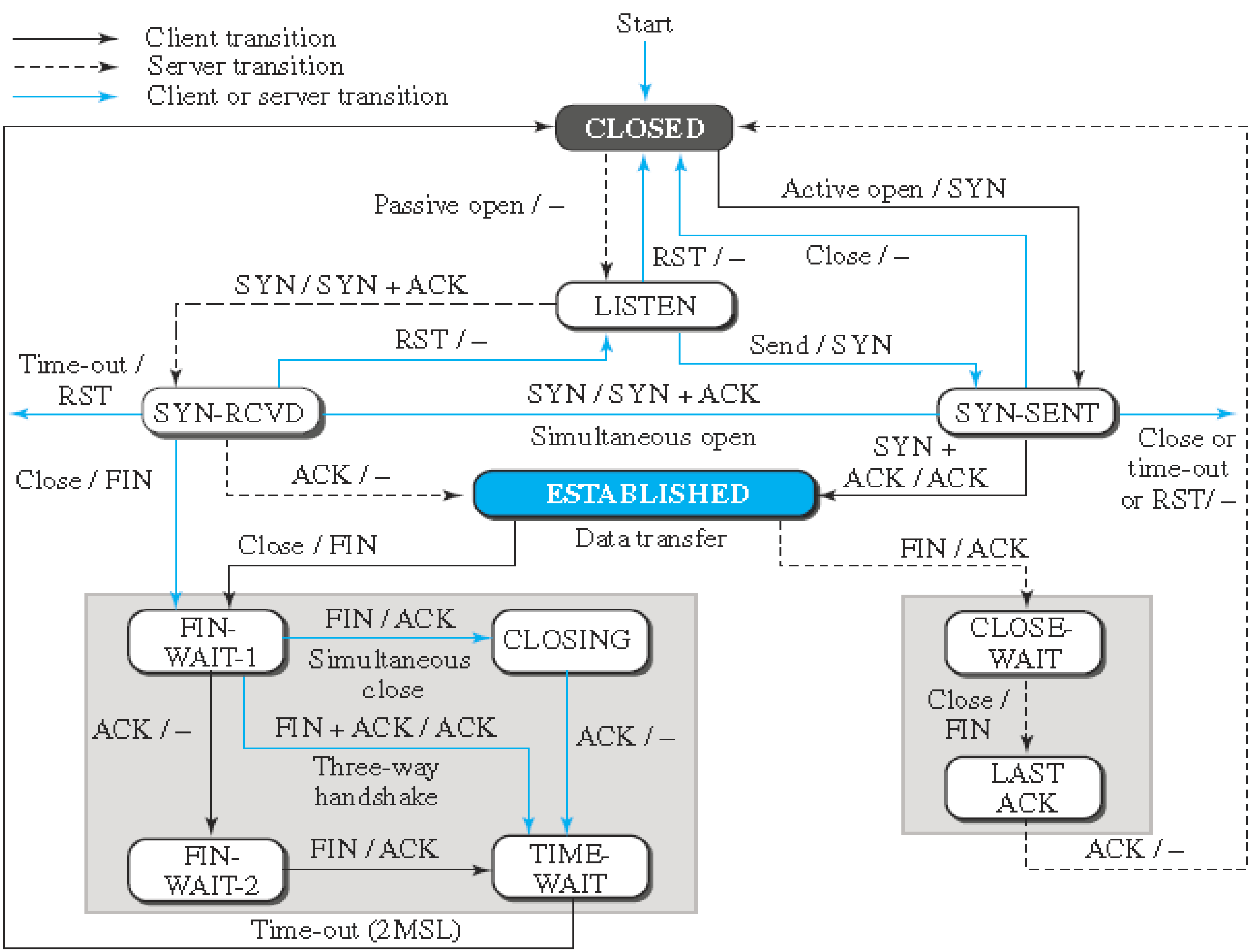
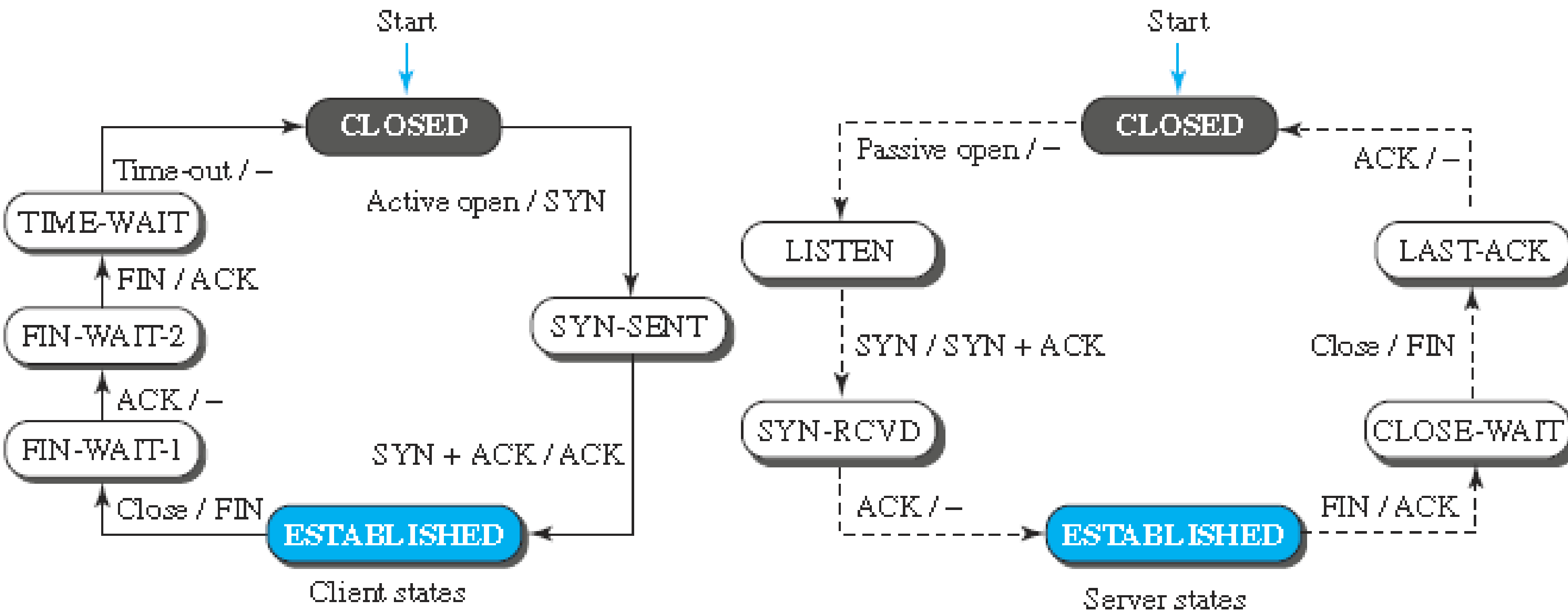


Table 24.2 *States for TCP*

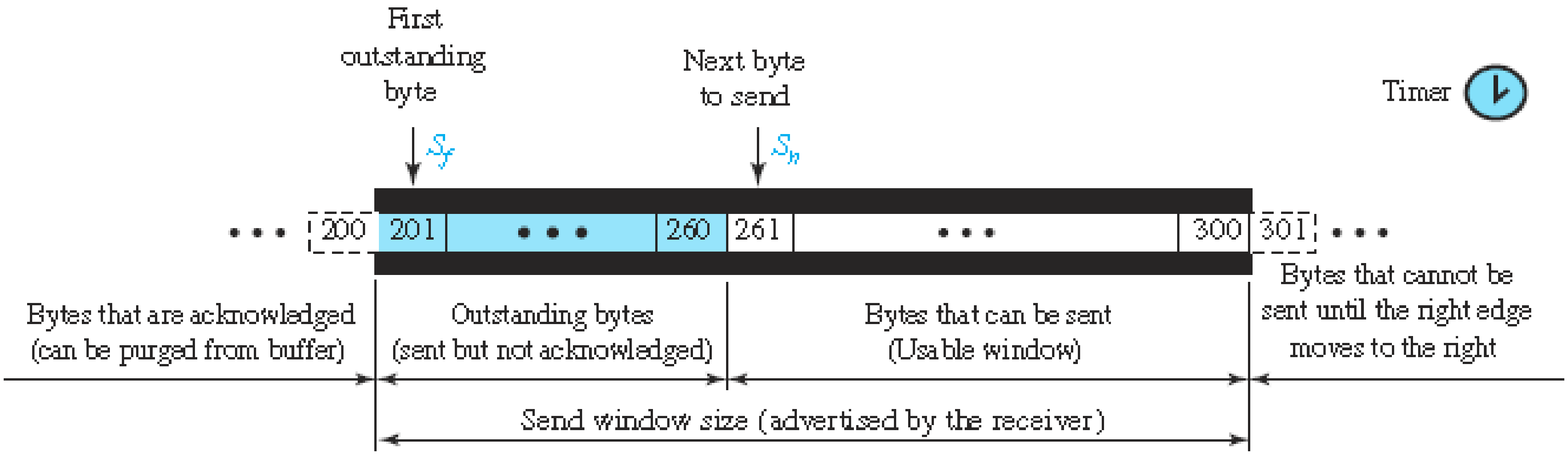
State	*Description*
CLOSED	No connection exists
LISTEN	Passive open received; waiting for SYN
SYN-SENT	SYN sent; waiting for ACK
SYN-RCVD	SYN + ACK sent; waiting for ACK
ESTABLISHED	Connection established; data transfer in progress
FIN-WAIT-1	First FIN sent; waiting for ACK
FIN-WAIT-2	ACK to first FIN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

Half Close Scenario



Windows in TCP

Figure 24.17 *Send window in TCP*



a. Send window

b. Opening, closing, and shrinking send window

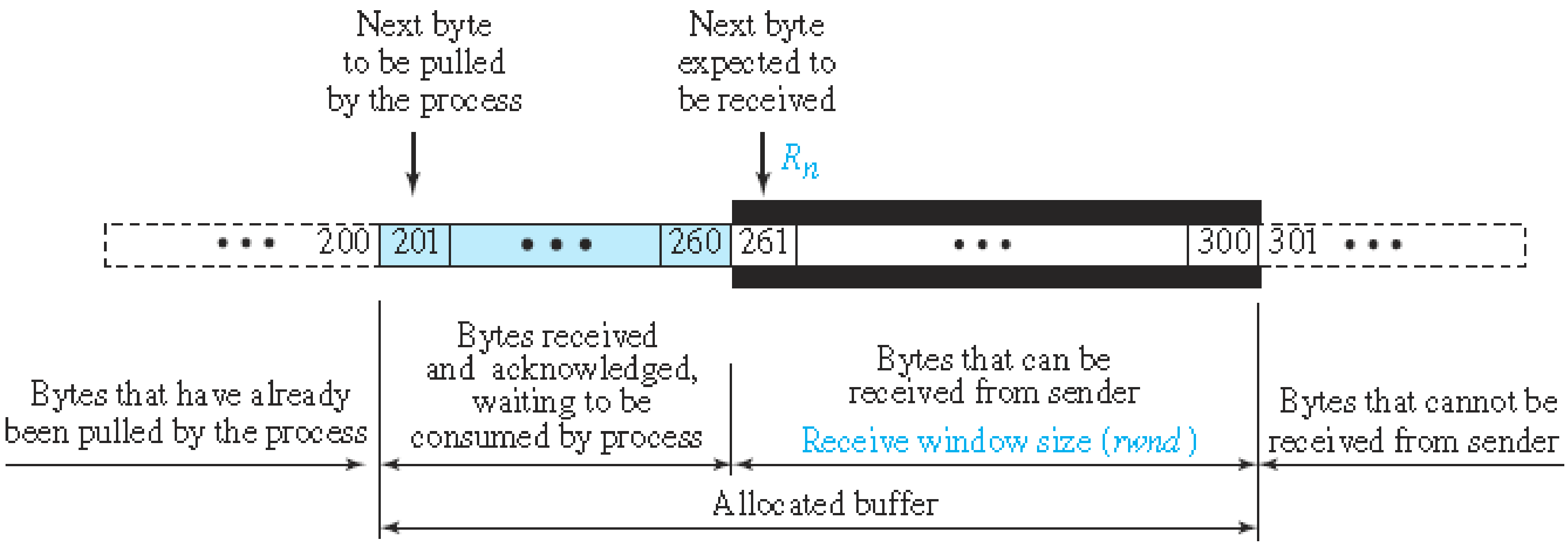
Send Window

- Send window size is dictated by the receiver (flow control) and the congestion in the underlying network (congestion control).
- The figure shows how a send window *opens*, *closes*, or *shrinks*.

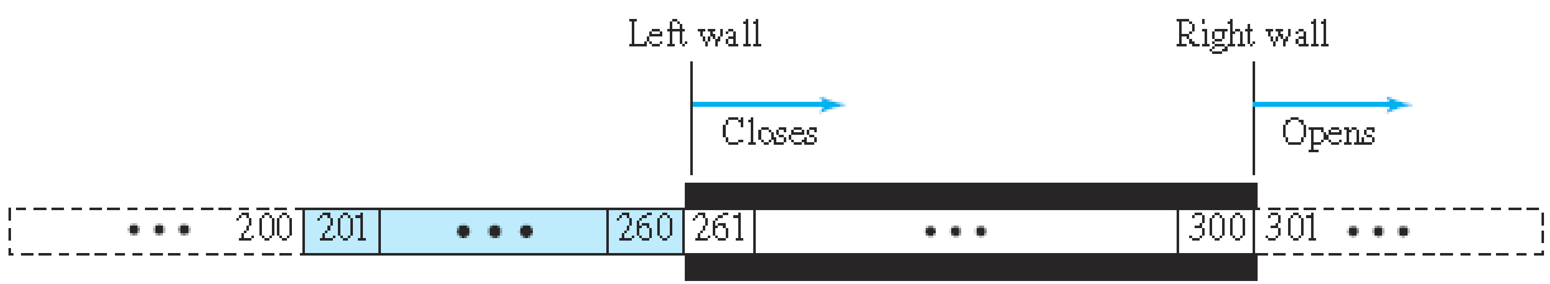
Difference between TCP and SR send window

- window size in SR is the number of packets, but the window size in TCP is the number of bytes.
- TCP can store data received from the process and send them later, but sending TCP is capable of sending segments of data as soon as it receives them from its process.
- number of timers

Receive Window



a. Receive window and allocated buffer



b. Opening and closing of receive window

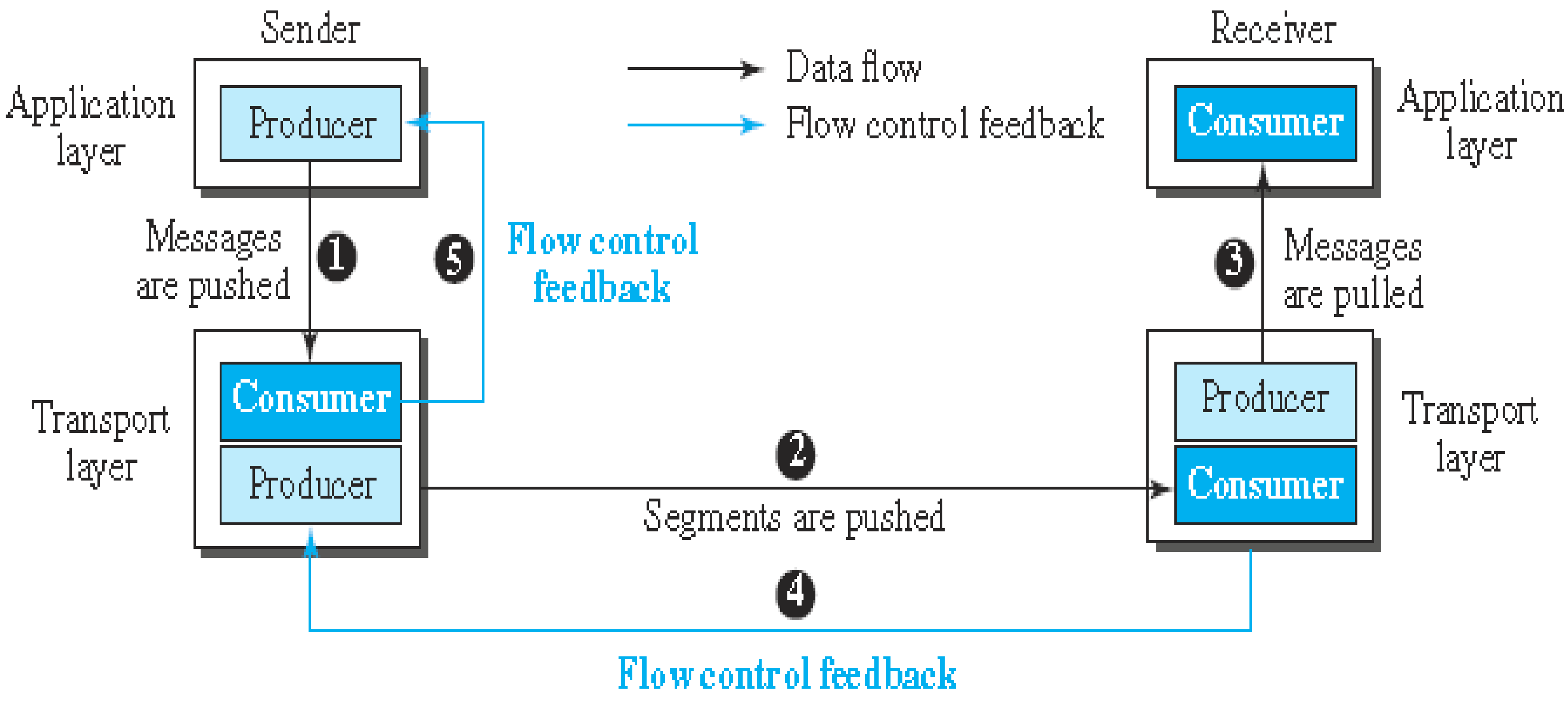
Difference between TCP and SR Receive window

- TCP allows the receiving process to pull data at its own pace.
- The receive window size determines the number of bytes that the receive window can accept from the sender before being overwhelmed (flow control)

  rwnd = **buffer size** - **number of waiting bytes to be pulled**

- Acknowledgments : New version of uses both cumulative and selective acknowledgments

Flow Control



Flow Control

- To achieve flow control, TCP forces the sender and the receiver to adjust their window sizes,
- The size of the buffer for both parties is fixed when the connection is established.
- The opening, closing, and shrinking of the send window is controlled by the receiver.
- The send window closes (moves its left wall to the right) when a new acknowledgment allows it to do so.
- The send window opens (its right wall moves to the right) when the receive window size (*rwnd*) advertised by the receiver allows it to do so.

- The receive window closes (moves its left wall to the right) when more bytes arrive from the sender;
- It opens (moves its right wall to the right) when more bytes are pulled by the process.

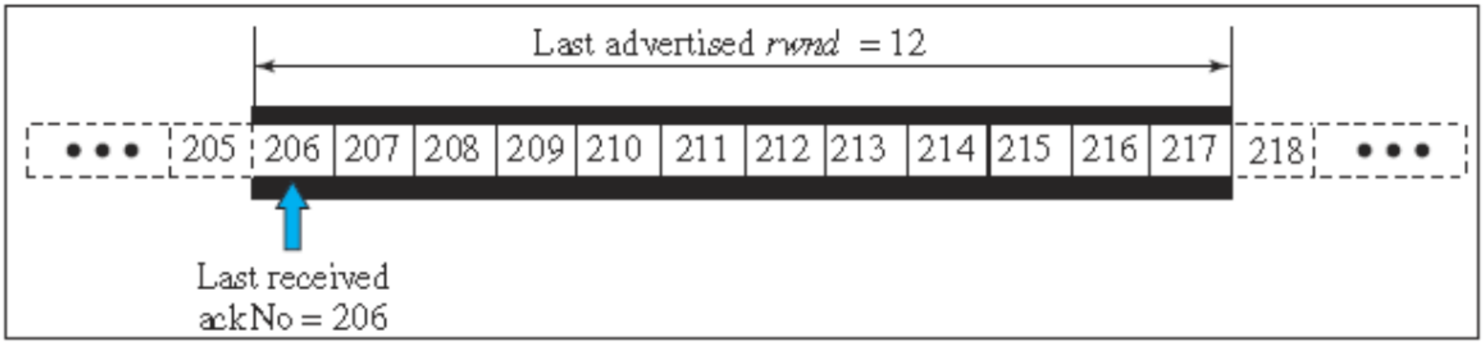
Shrinking of Windows

- Receive window cannot shrink.
- Send window, can shrink if the receiver defines a value for *rwnd* that results in shrinking the window
- Some implementation doesn't allow shrinking of sender window
- the receiver needs to keep the following relationship between the last and new acknowledgment and the last and new *rwnd* values to prevent shrinking of the send window.

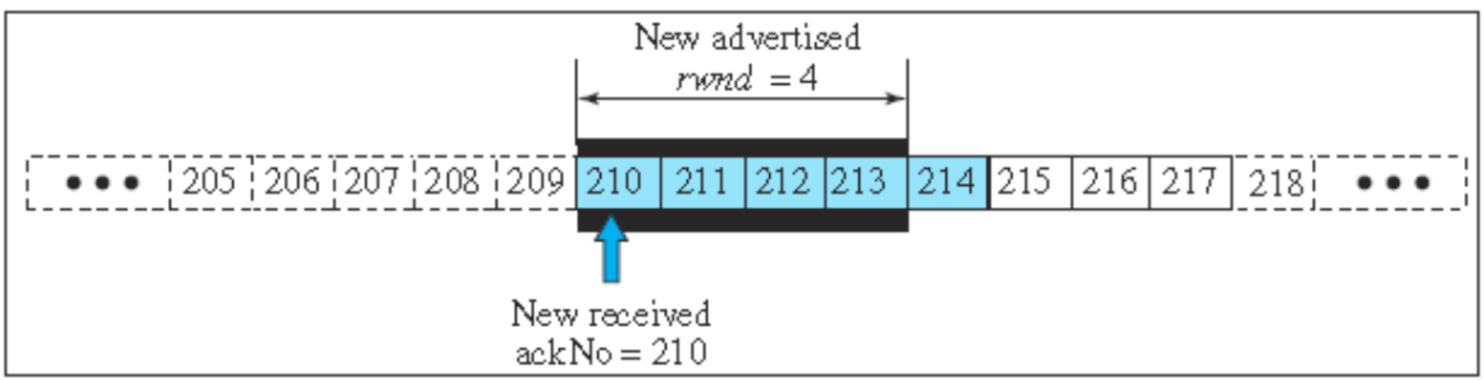
new ackNo + new *rwnd* ≥ last ackNo + last *rwnd*

(210) + (4) < (206) + (12)

Shrinking of WIndows



a. The window after the last advertisement



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