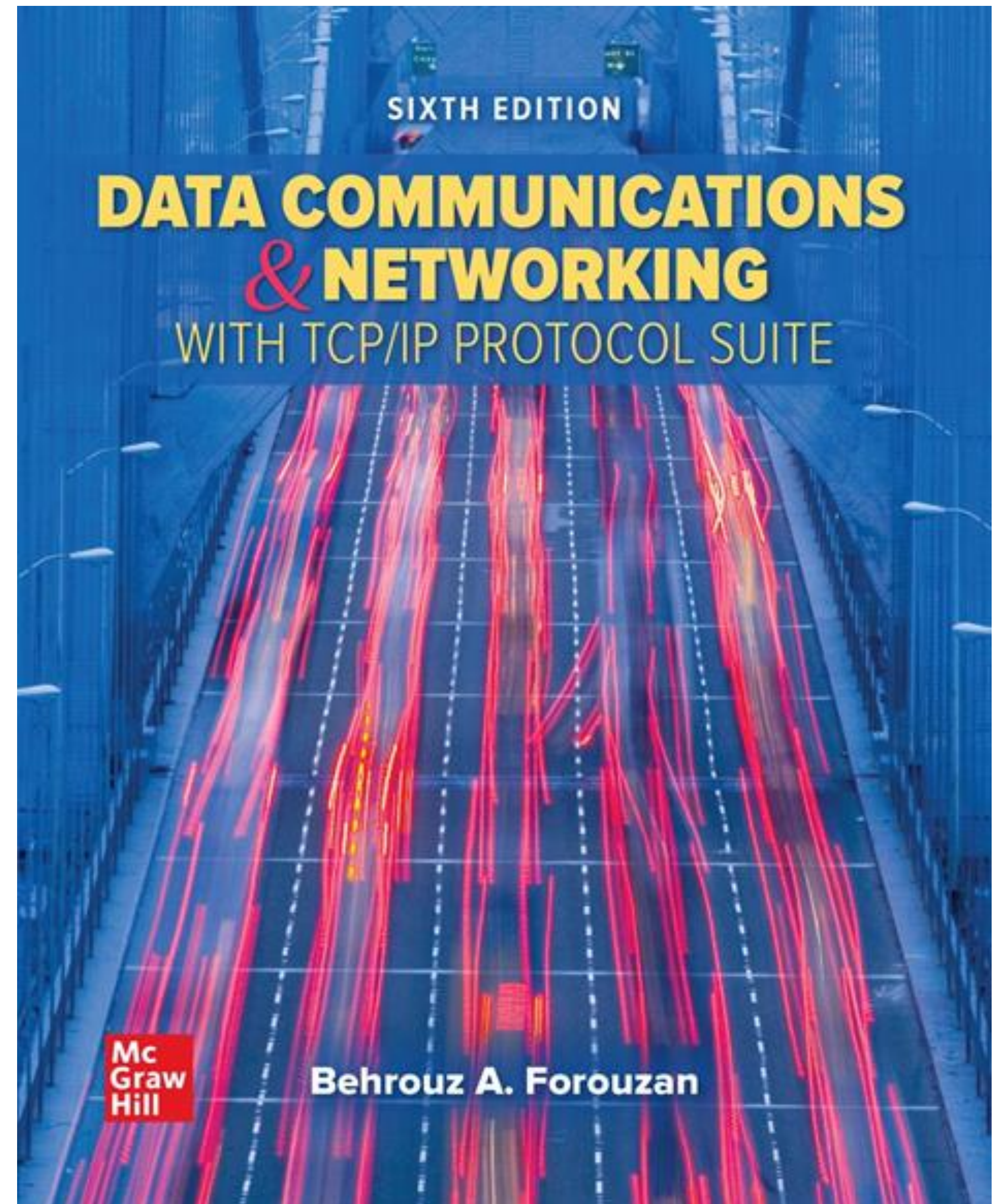
Chapter 09

Transport Layer

- Data Communications and Networking, With TCP/IP protocol suite Sixth Edition
- Behrouz A. Forouzan



- © 2022 McGraw Hill, LLC. All rights reserved. Authorized only for instructor use in the classroom.
- No reproduction or further distribution permitted without the prior written consent of McGraw Hill, LLC.

Chapter 9: Outline

- 9.1 Transport-Layer Services
- 9.2 Transport-Layer Protocols
- 9.2 User-Datagram Protocol (UDP)
- 9.3 Transmission Control Protocol (TCP)

9-1 TRANSPORT LAYER SERVICES

- The transport layer is located between the application layer and the network layer. It provides a **process-to-process communication between two application layers**, one at the **local host** and the other at **the remote host**. Communication is provided using **a logical connection**. Figure 9.1 shows the idea behind this logical connection.

Figure 9.1 Logical connection at the transport layer

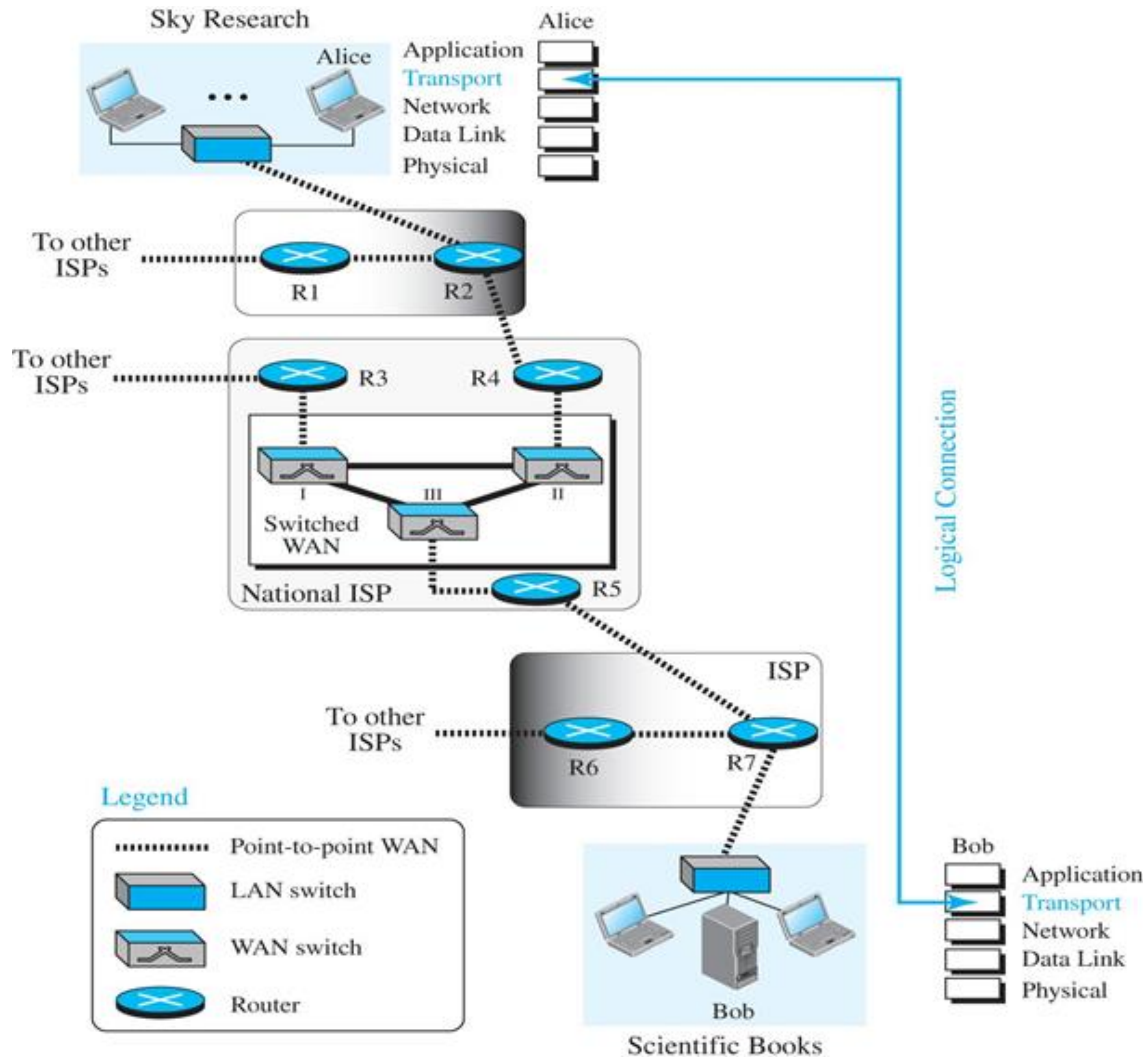
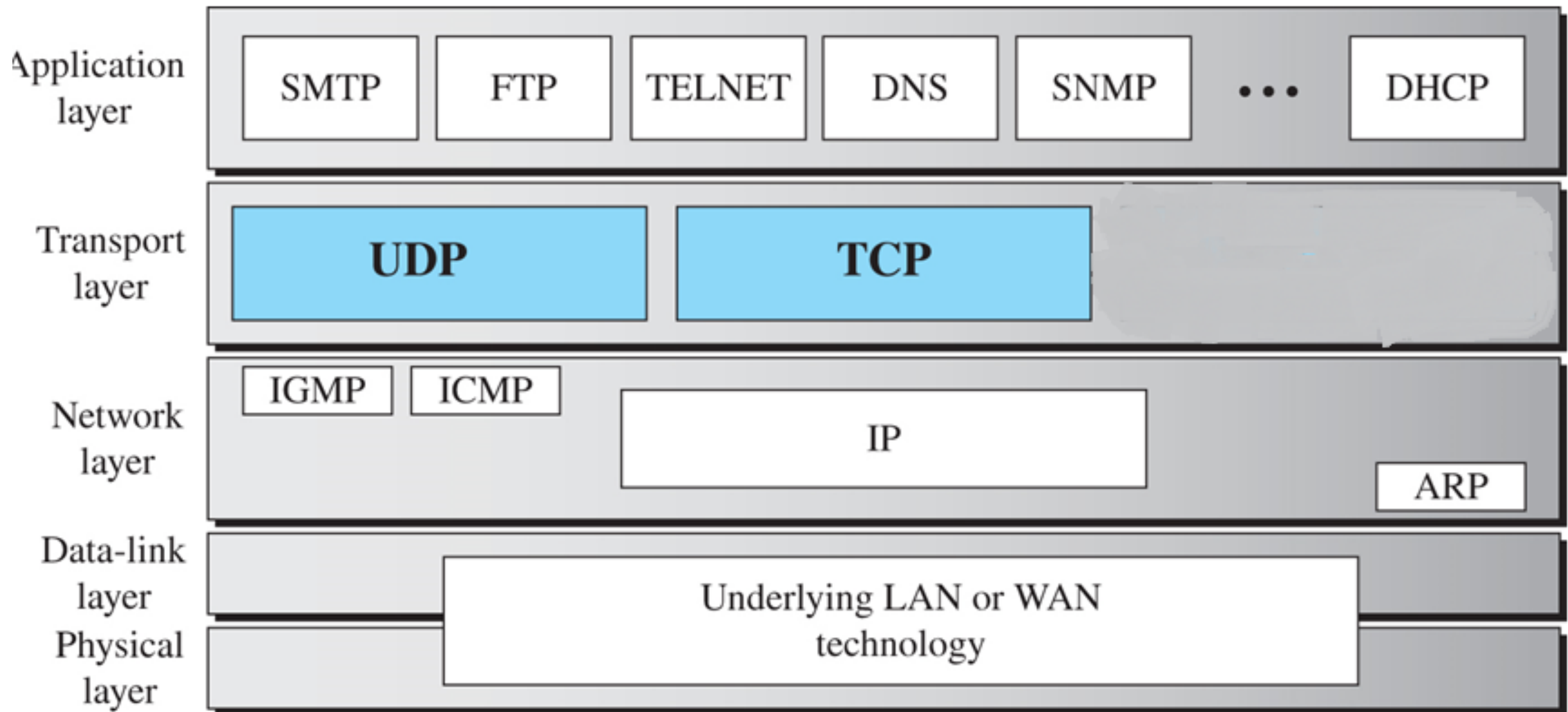


Figure 9.17 Position of transport-layer protocols in the TCP/IP protocol suite



Transport layer duties

1- Process to Process communication

2- Addressing

- Port numbers to identify which network application

3- Encapsulation and Decapsulation

4- Multiplexing and demultiplexing

5-Connection control

- Connection-oriented services
- Connectionless services

6- Reliability

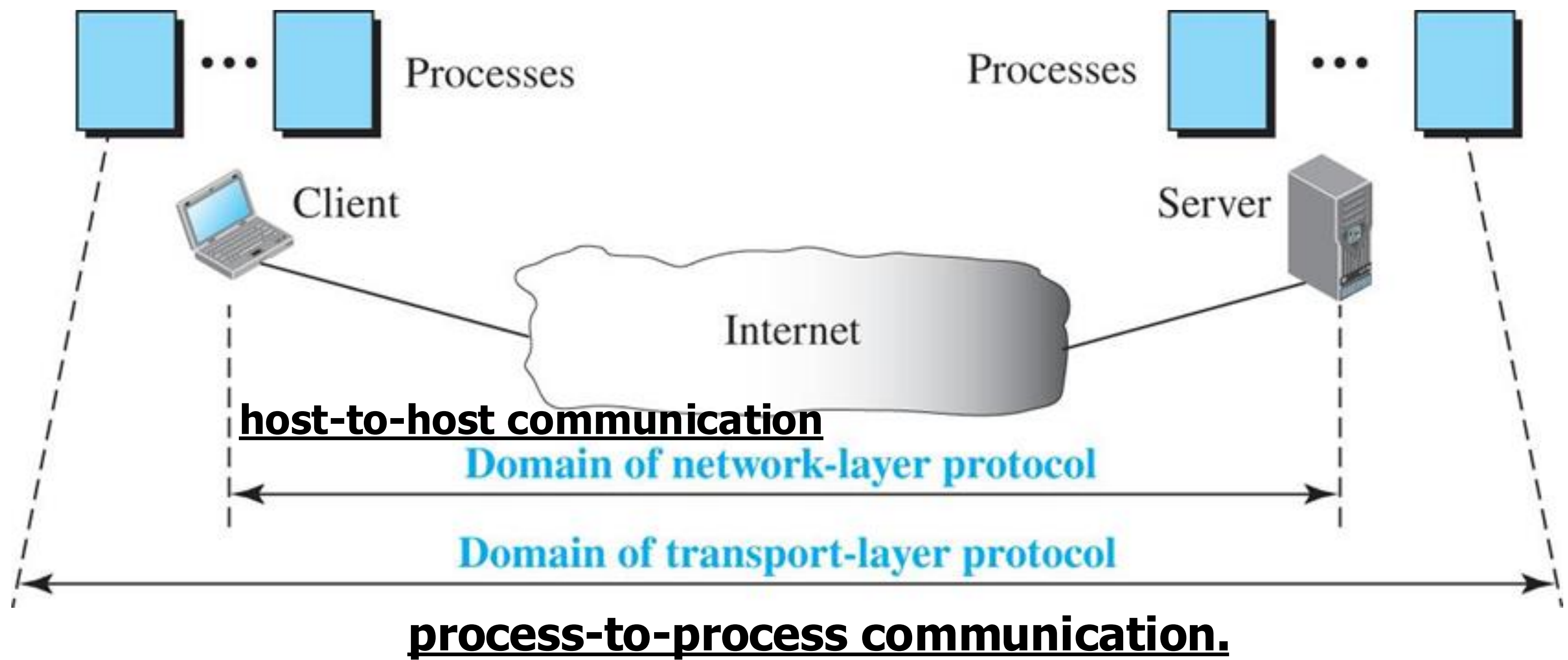
- Flow control
- Error Control

7- Congestion Control

9.1.1 Process-to-Process Communication

- The first duty of a transport-layer protocol is to provide **process-to-process communication**. A **process is an application-layer entity (running program) that uses the services of the transport layer**. Before we discuss how process-to-process communication can be accomplished, we need to understand the difference between host-to-host communication and process-to-process communication.

Figure 9.2 Network layer versus transport layer



PROCESS-TO-PROCESS Communication

- ***Network layer is responsible for host-to-host communication. Network layer delivers the packet to the destination computer but not to specific process.***
- ***The transport layer is responsible for process-to-process delivery—the delivery of a segment, part of a message, from one process to another.***
- **Process is an application layer entity (program in execution) that uses transport layer services**
- **Processes on two hosts communicate with each other by sending and receiving messages**

Addressing – Port numbers

- Each network process is assigned an address called port number that is unique to the host
- Port numbers are 16-bit integers between 0 – 65535
  - Well-known(0-1023): Assigned and controlled by Internet Assigned Numbers Authority IANA for example: FTP 20,21, TELNET 23, SMTP 25, HTTP 80
  - Only given to destination process

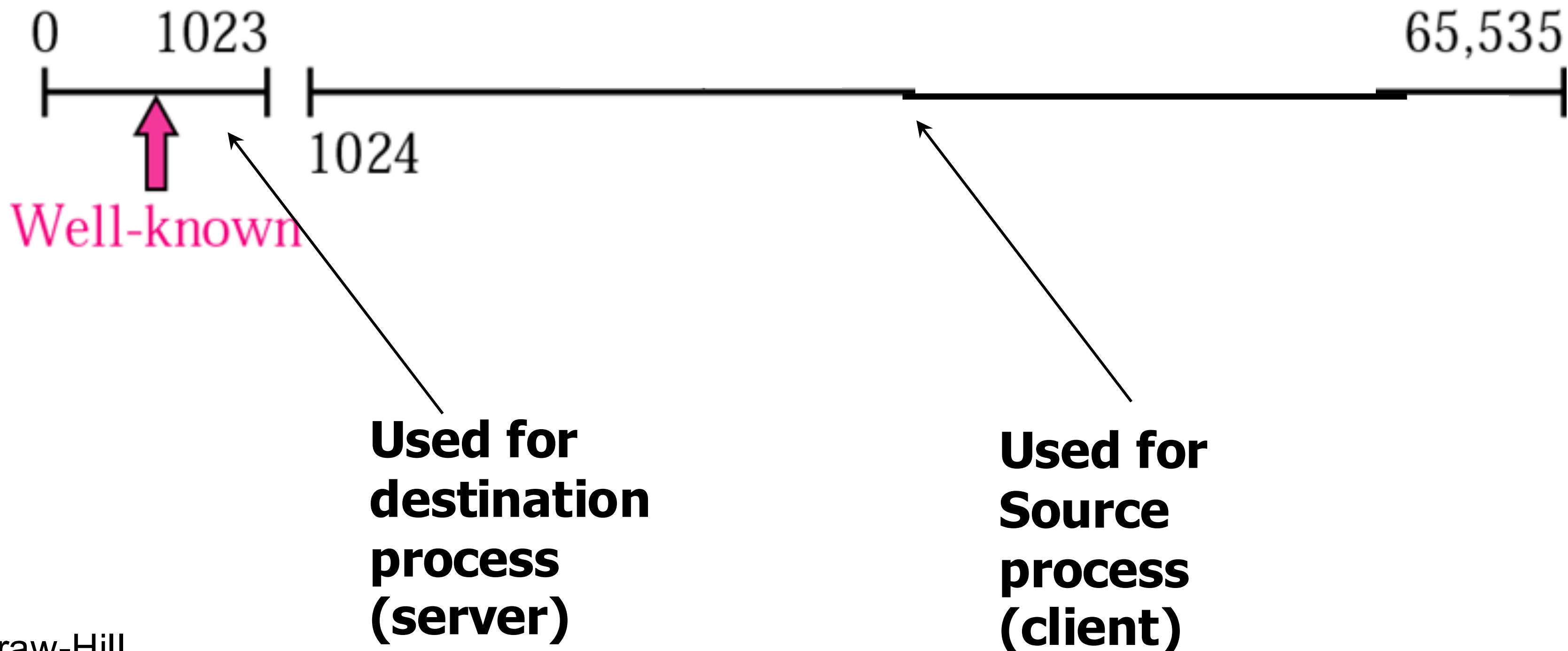
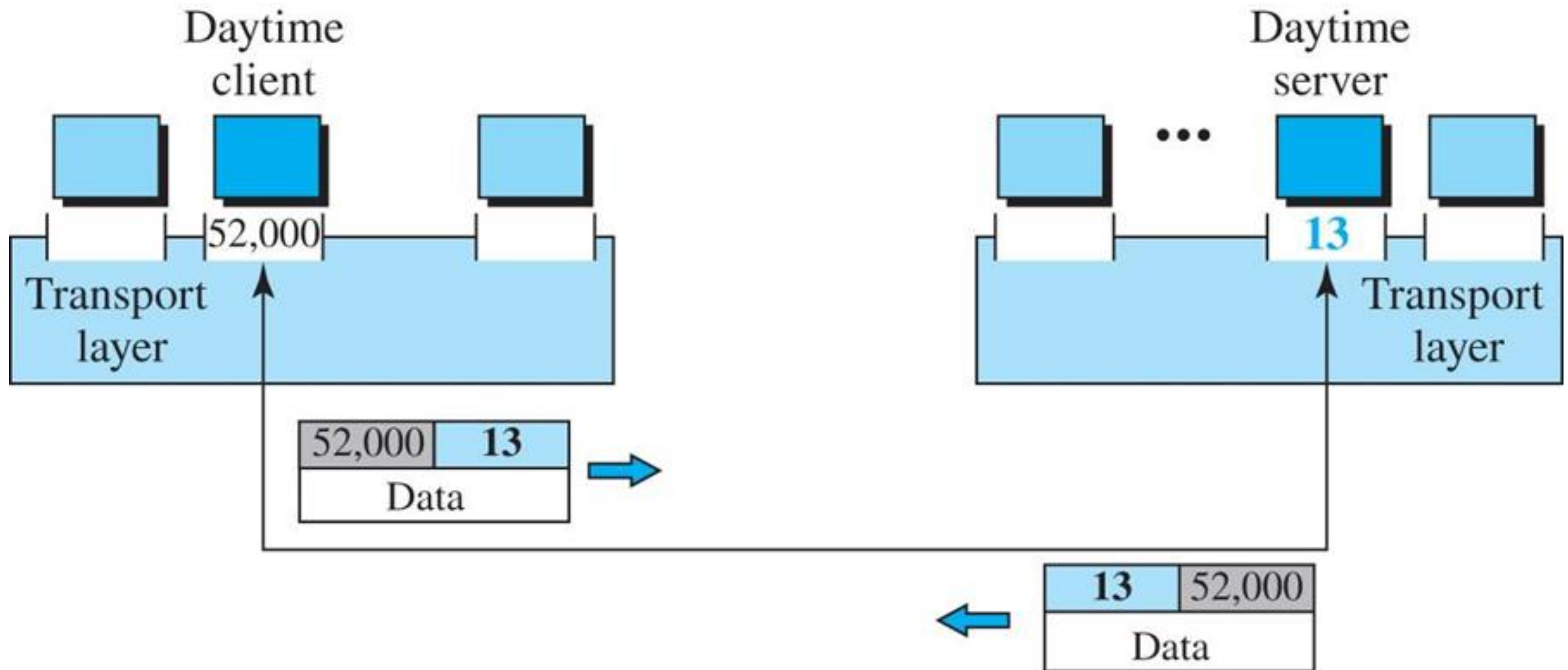


Table 9.1 Some well-known ports used with UDP and TCP

Port	*Protocol*	*UDP*	*TCP*	*SCTP*	*Description*
7	Echo	√	√	√	Echoes back a received datagram
9	Discard	√	√	√	Discards any datagram that is received
11	Users	√	√	√	Active users
13	Daytime	√	√	√	Returns the date and the time
17	Quote	√	√	√	Returns a quote of the day
19	Chargen	√	√	√	Returns a string of characters
20	FTP-data		√	√	File Transfer Protocol
21	FTP-21		√	√	File Transfer Protocol
23	TELNET		√	√	Terminal Network
25	SMTP		√	√	Simple Mail Transfer Protocol
53	DNS	√	√	√	Domain Name System
67	DHCP	√	√	√	Dynamic Host Configuration Protocol
69	TFTP	√	√	√	Trivial File Transfer Protocol
80	HTTP		√	√	Hypertext Transfer Protocol
111	RPC	√	√	√	Remote Procedure Call
123	NTP	√	√	√	Network Time Protocol
161	SNMP-server	√			Simple Network Management Protocol
162	SNMP-client	√			Simple Network Management Protocol

Figure 9.3 Port numbers



Addressing – Socket Address

- Two processes communicate in a **client/serve**r relationship,
- Local host process is called **client** and remote host process is called **server**
- In multiuser and multiprogramming **Client/Server** environments **four entities** must be defined:
- **Sending Node**
  - Local Host IP
  - Local Process Port number
- **Receiving Node**
  - Remote host IP
  - Remote Process ID Port number
- The **process** receives messages from, and sends messages into the network through its **socket**
- A **socket** is the **interface** between the **application layer** and the **transport layer** within a host.
- **Socket Address is IP and Port Number**

Figure 9.6 Socket address

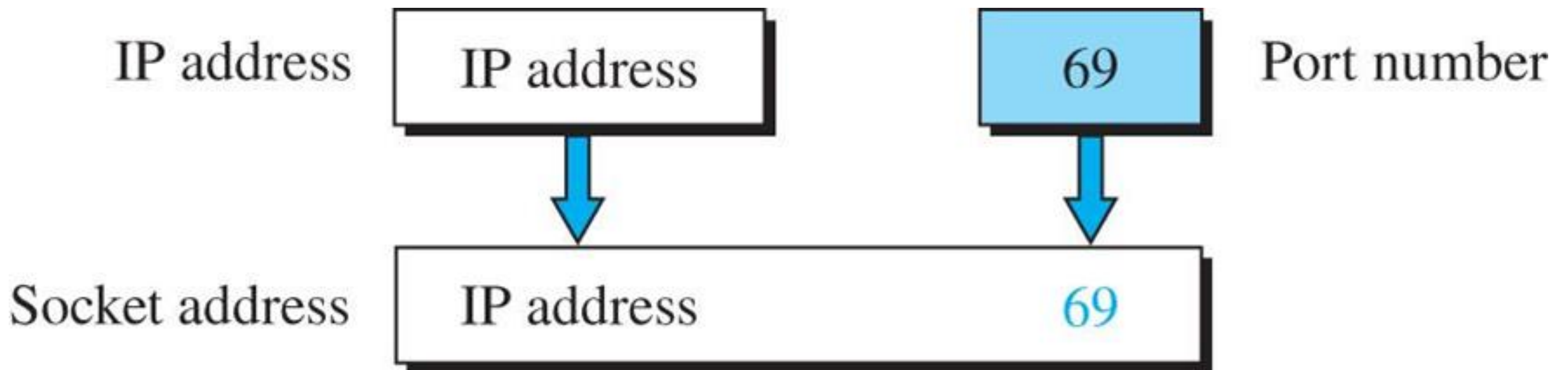
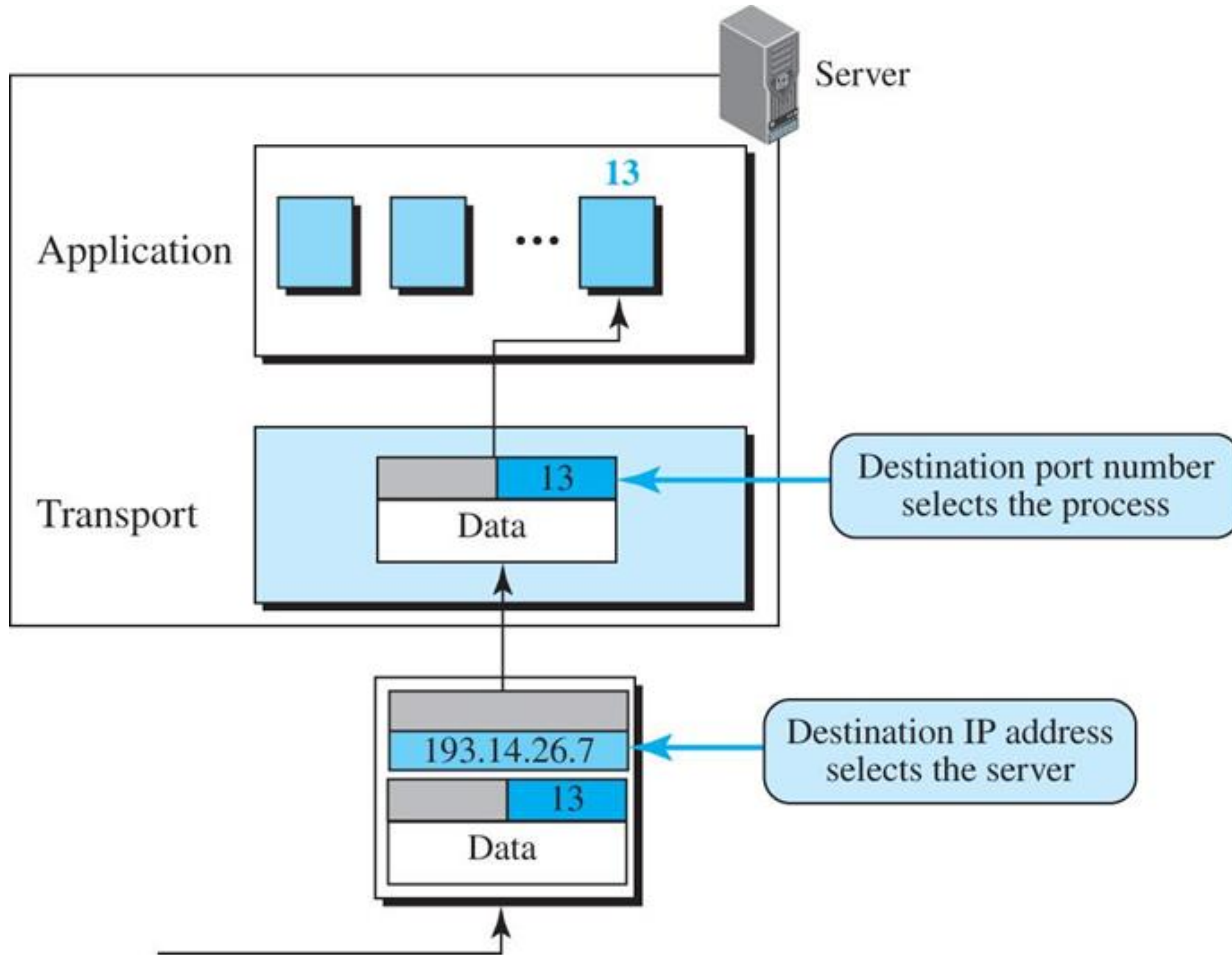
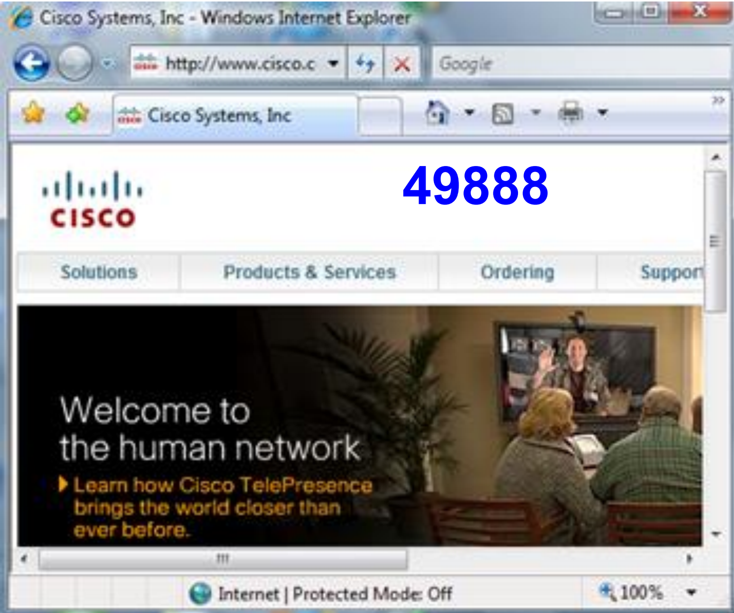
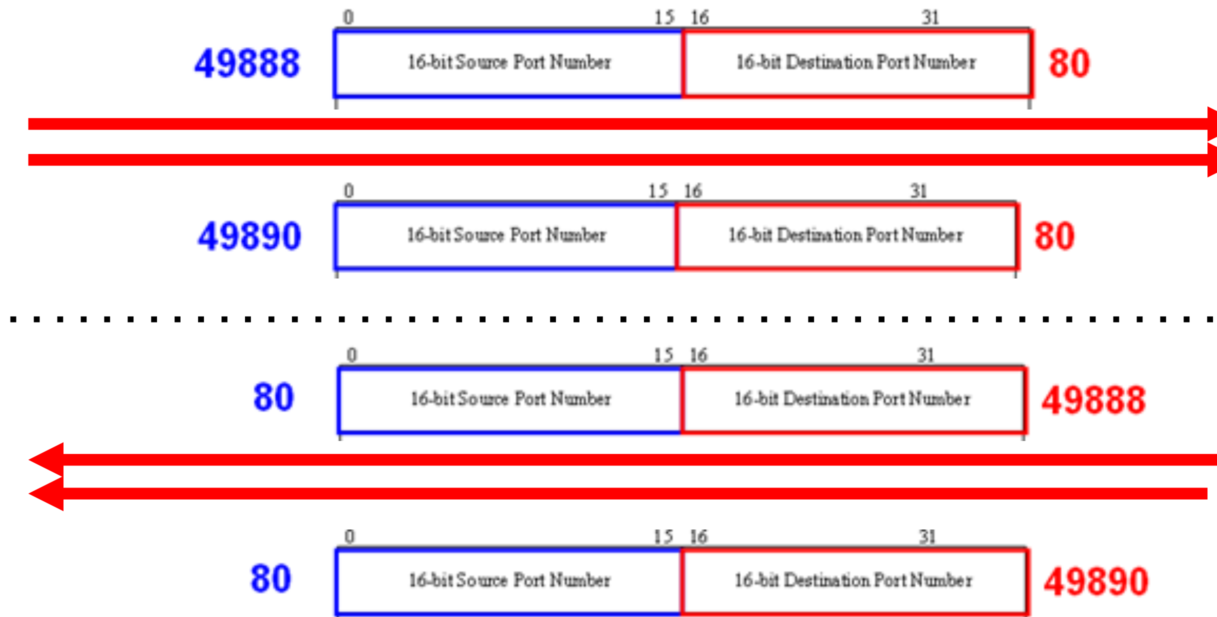


Figure 9.4 IP addresses versus port numbers





Same client to same server – Different source ports and same destination port



- Same client to same server - Two different HTTP sessions
- Client: Same destination port
- Client: Different source ports to uniquely identify this web session.

Two clients contacted same server with same source and destination port numbers

192.168.1.101

Source Port

49888

49890

Destination Port

198.133.219.25

80

80

80

www.cisco.com

172.16.5.5

Source Port

49888

What makes each connection unique? How does the server know which source port 49888 is who?

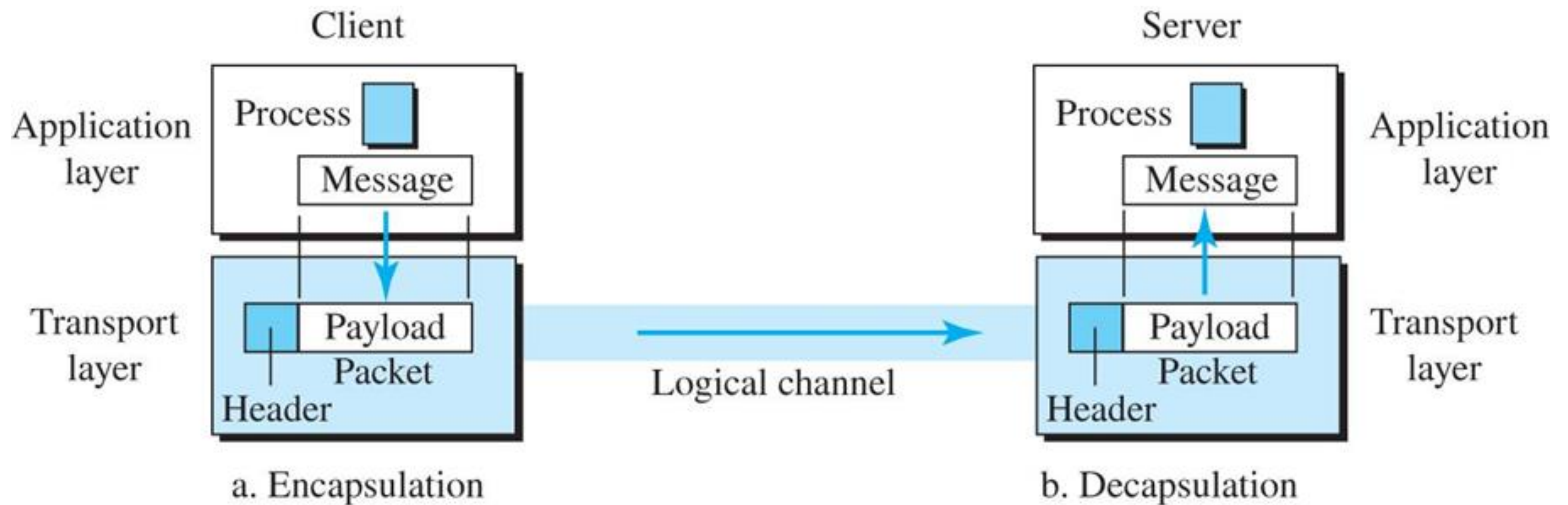
- Connection defined by the pair of numbers:
  - **Source IP address**, **Source port** (From Client to Server)
  - **Destination IP address**, **Destination port** (From Server to Client)
- Different connections can use the same destination port on server host as long as the source ports or source IPs are different.

McGraw-Hill

9.1.3 Encapsulation and Decapsulation

- To send a message from one process to another, the transport-layer protocol encapsulates and decapsulates messages.
- **Encapsulation** happens at the **sender site**.
- When a process has a message to send, it passes the message to the transport layer along with **a pair of socket addresses (Source Port No. and Destination Port No.)** and some other pieces of information, which depend on the transport-layer protocol.
- The transport layer receives the data and adds the transport-layer header.
- **Decapsulation** happens at the **receiver side**
- The transport layer receives the segment from the network layer , removes the headers, then passes the data to the process

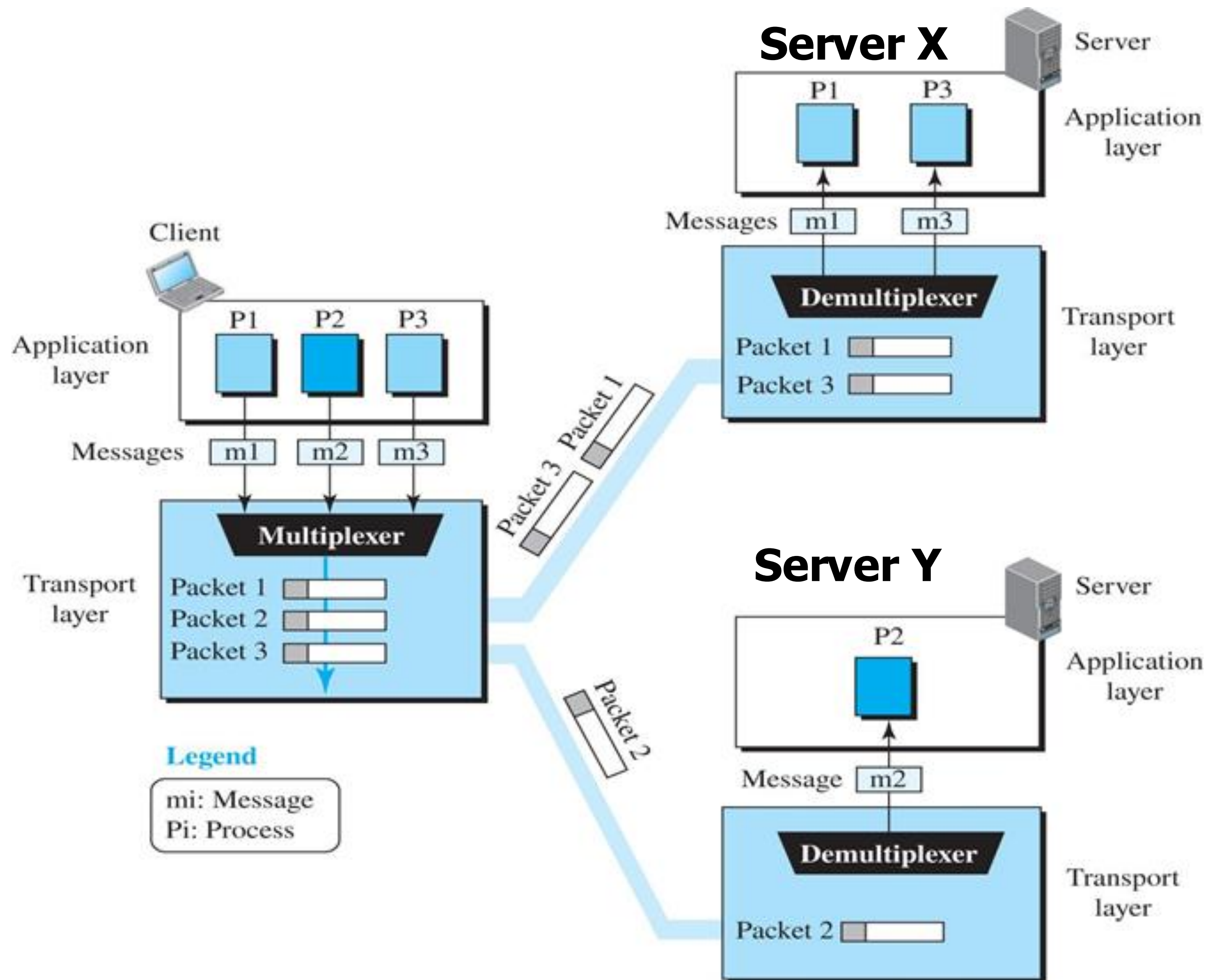
Figure 9.7 Encapsulation and decapsulation



9.1.4 Multiplexing and Demultiplexing

- Whenever an entity **accepts items from more than one source**, this is referred to as multiplexing (many to one);
- Whenever an entity **delivers items** t**o more than one source,** this is referred to as demultiplexing (one to many).
- The transport layer at **the source** performs **multiplexing** which means it accepts messages from more than one application layer processes (many to one)
- The transport layer at **the destination** performs **demultiplexing** which means it delivers segments to more than one application layer processes (one to many)

Figure 9.8 Multiplexing and demultiplexing



Congestion Control

- Congestion happens if the load on the network (number of segments sent to the network) **is greater** than the network Capacity (number of segments the network can handle).

- Congestion control are techniques that **control the load so that it stays below the capacity.**

Connection

- ***A transport-layer protocol, like a network-layer protocol, can provide two types of services: connectionless and connection-oriented. The nature of these services at the transport layer,***
- ***However, is different from the ones at the network layer.***
- ***At the network layer, a connectionless service may mean different paths for different segments belonging to the same message.***

Connection

- **Connectionless service (UDP)**
  - Means **independency** between segments
  - **No** flow and error control
  - **No** congestion control
- **connection-oriented (TCP)**
  - Means **dependency** between segments
  - Data can be transferred only after connection is established
  - Connection oriented means that a logical connection is established before any data is transferred.
  - **Logical connection** since Transport layer will make sure that segments are given to the receiver application in the same order as they were sent by the sender even if they travel through different physical paths
  - **Both sides** must initialize communication and **get approval from the other side before any data transfer,**
  - Flow and error control is **applied**
  - Congestion control is **applied**
  - Connection should be **terminated** after data exchange

Figure 9.14 Connectionless service

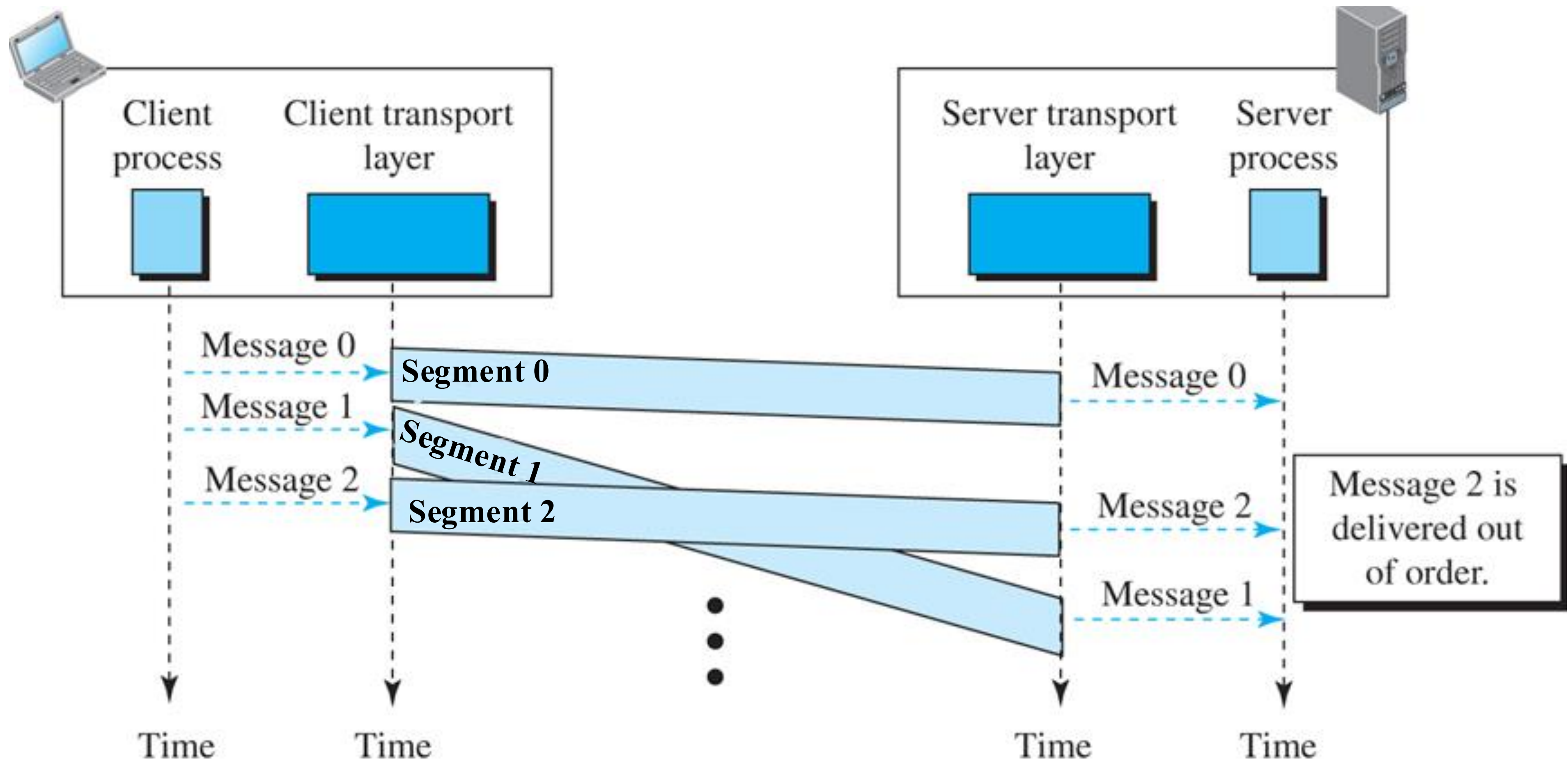
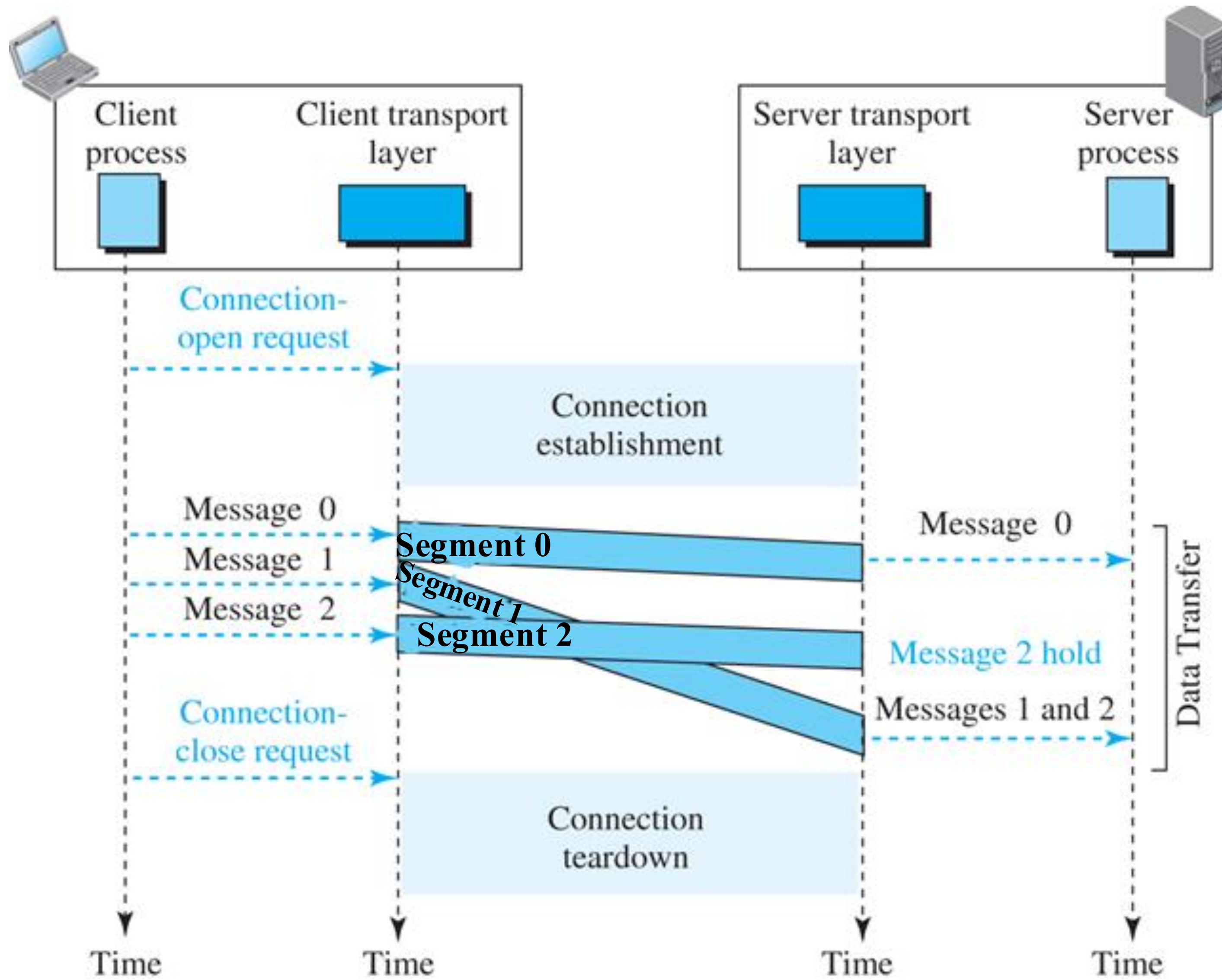


Figure 9.15 Connection-oriented service



Flow control and Error Protocols

- **Flow control** (process-to-process): Transport layer makes sure that the sender process **does not cause the receiver buffer to overflow**
- **Error control** (process-to-process): **entire message** arrives at the receiving transport layer
  - without error,
  - without loss,
  - without duplication,
  - and in the same order they were sent,

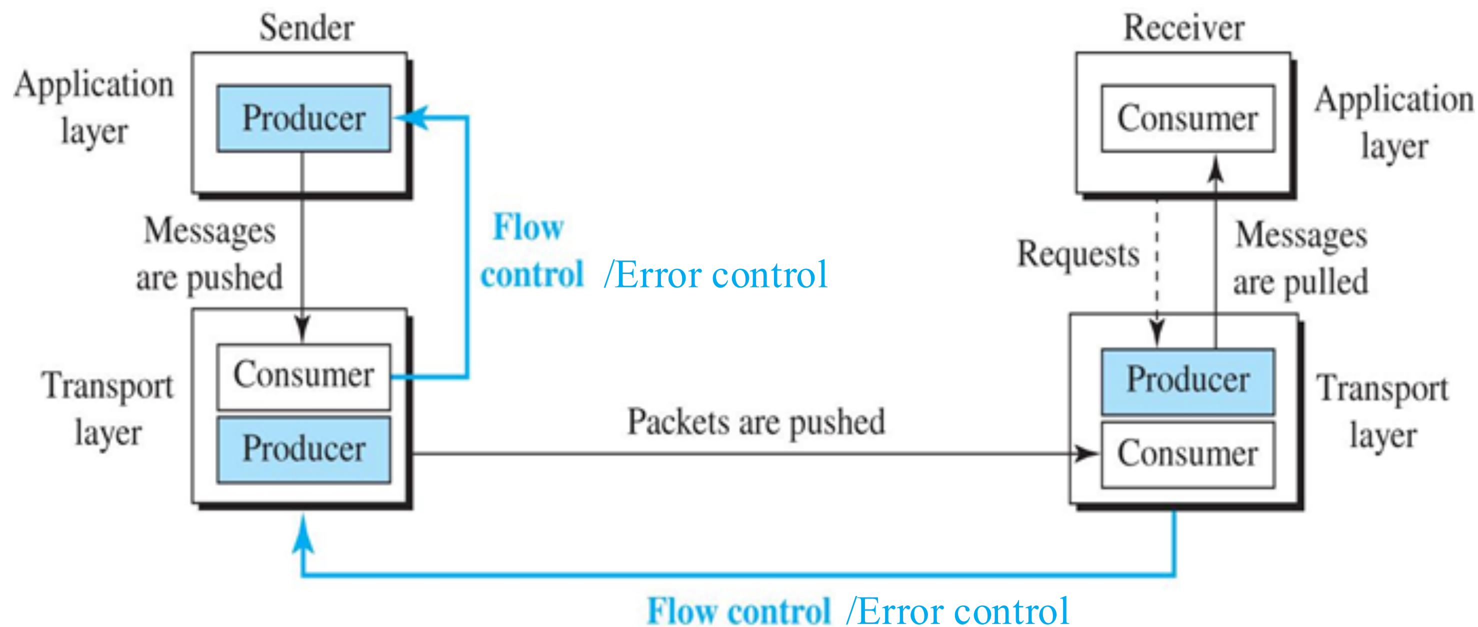
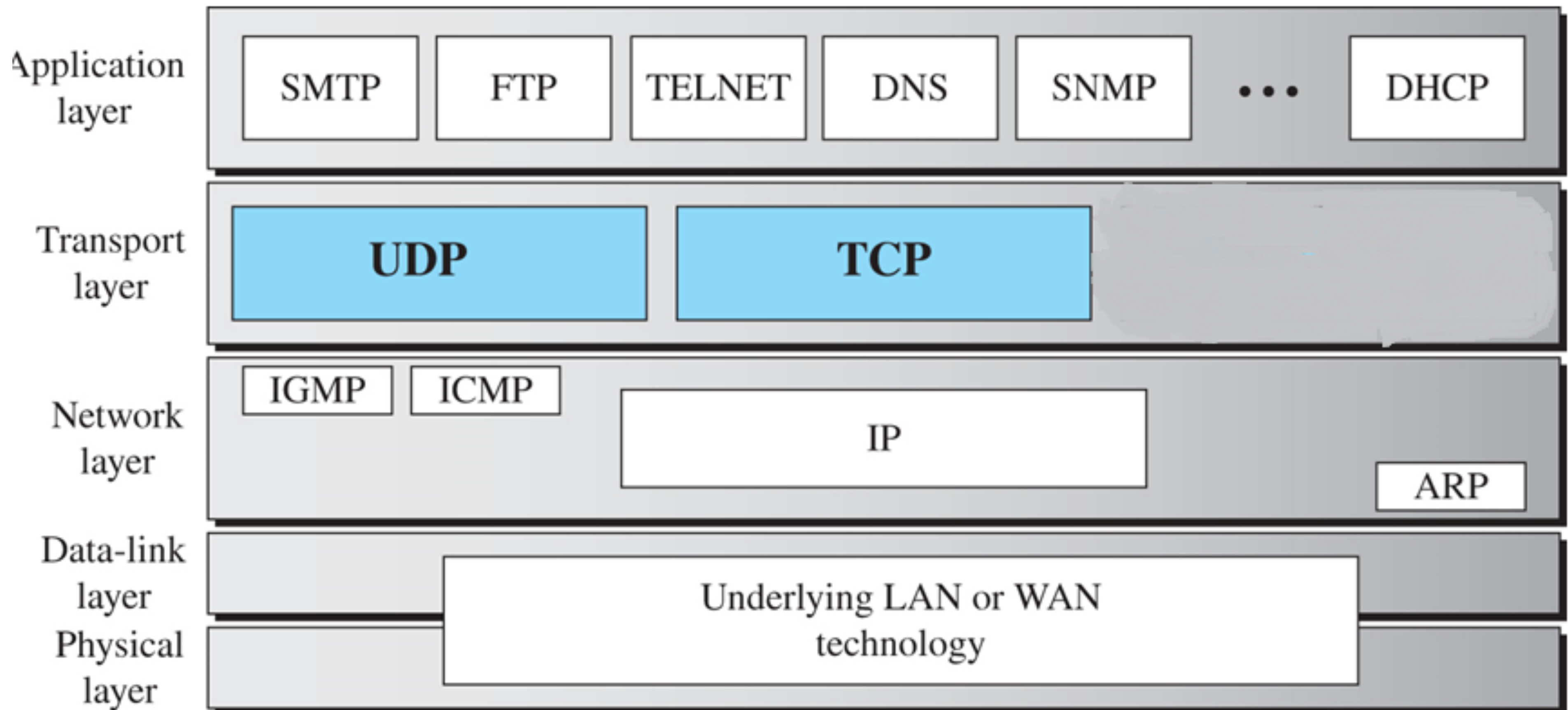


Figure 9.17 Position of transport-layer protocols in the TCP/IP protocol suite



Transmission Control Protocol (TCP)

- Transmission Control Protocol properties:
  - Connection-oriented (establishment & termination)
  - Reliable
    - Each segment belongs to a message is given a **unique sequence** number needed for error control
    - Performs **error control and flow control and congestion control**
  - Header is **20 bytes** by default and can be extended to **60 bytes** with options.

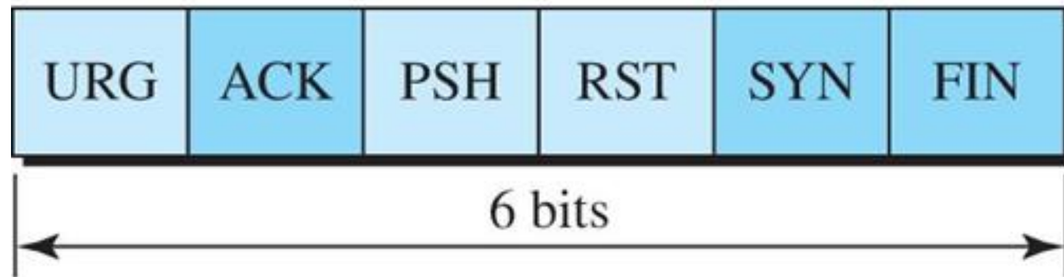
Figure 9.23 TCP segment format

a. Segment

1								16	31
Source port address 16 bits								Destination port address 16 bits	
Sequence number 32 bits									
Acknowledgment number 32 bits									
HLEN 4 bits	Reserved 6 bits	URG	ACK	PSH	RST	SYN	FIN	Window size 16 bits	
Checksum 16 bits								Urgent pointer 16 bits	
Options and padding (up to 40 bytes)									

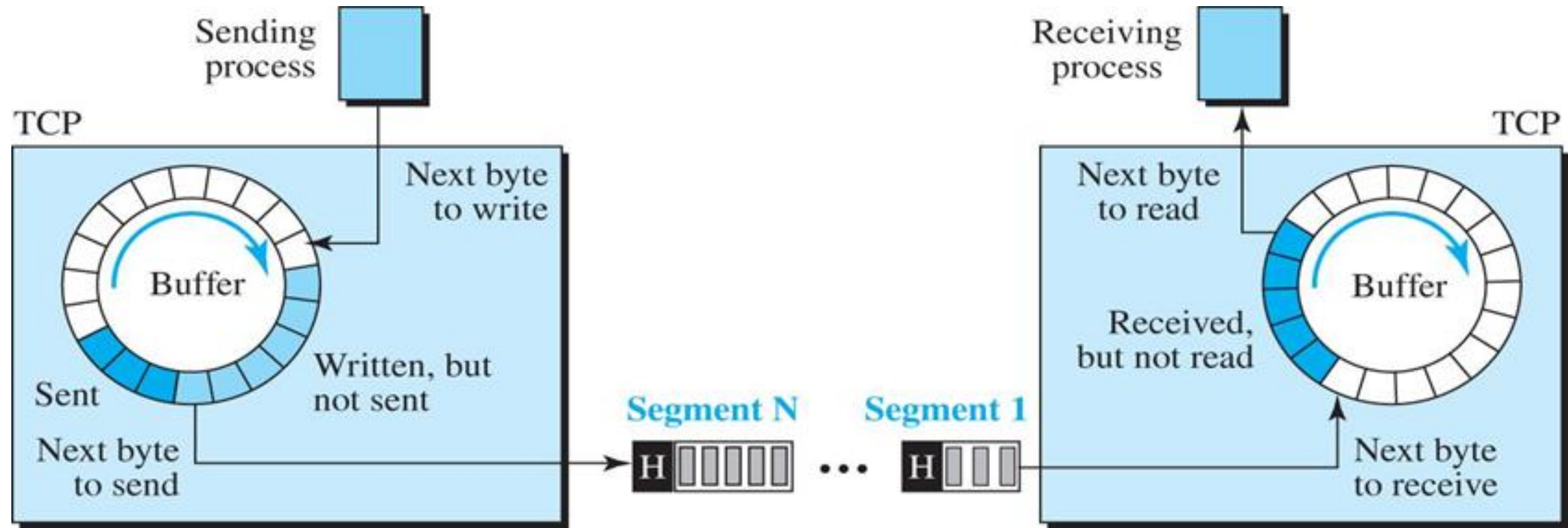
b. Header

Figure 9.24 Control flags



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 9.22 TCP segments



- Sender and receiver process operates at different **transfer rate**.
- To handle different rates, buffers (storage) are used
- There are **two buffers at each side**: sending buffer and receiving buffer
- Buffers stores **thousands of bytes**
- Transport layer **groups a number of bytes into segment**

TCP Sequence Number

- Sequence Numbers are given to individual bytes
- When connection is established each side announces the initial sequence number for its data. This is **a random number.**
- The sequence number of the **first segment is the initial sequence number**
- The sequence number of any other segment for the same message is:

  the sequence number of the previous segment + the number of bytes carried by the previous segment

- Sequence number is used in **flow and error control** and in **segmenting and reassembling of segments**.

Example 9.8

Suppose a TCP connection is transferring a file of 5,000 bytes. The first byte is numbered **10,001**. What are the sequence numbers for each segment if data are sent in **five** segments, each carrying **1,000** bytes?

Solution

The following shows the sequence number for each segment:

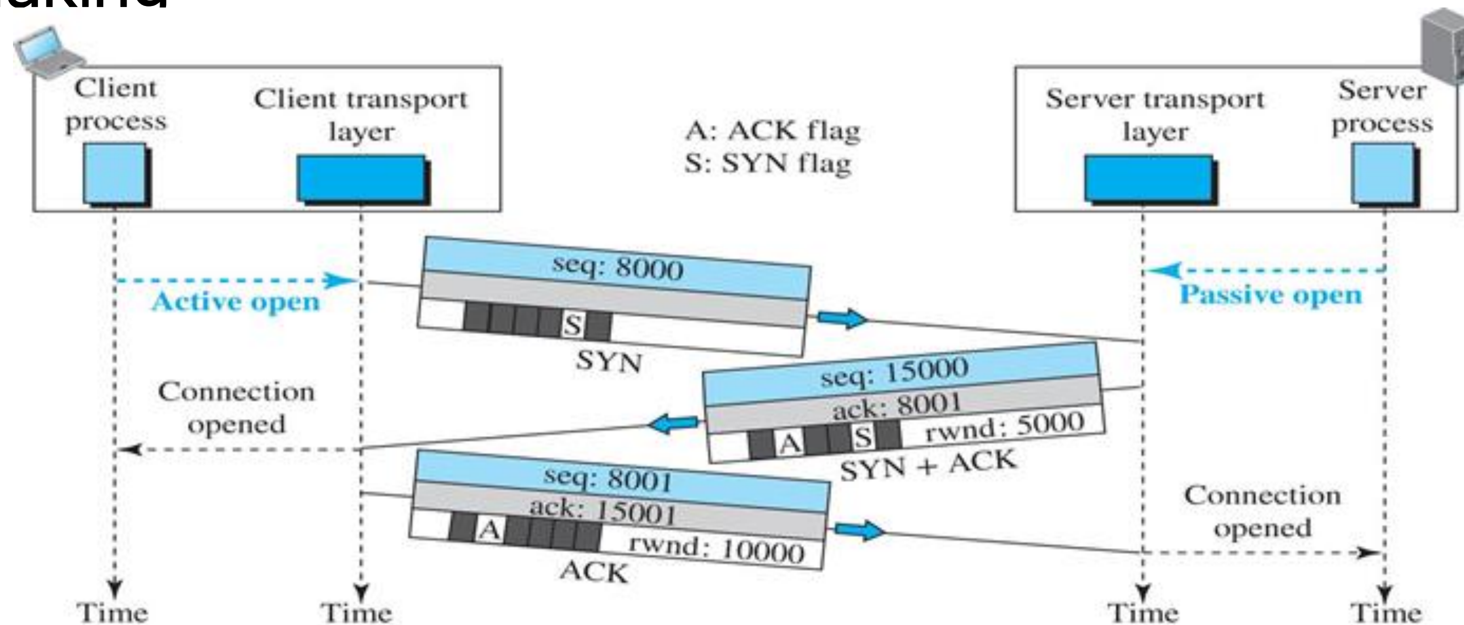
Segment 1	→	Sequence number:	10,001	Range:	10,001	to	11,000
Segment 2	→	Sequence number:	11,001	Range:	11,001	to	12,000
Segment 3	→	Sequence number:	12,001	Range:	12,001	to	13,000
Segment 4	→	Sequence number:	13,001	Range:	13,001	to	14,000
Segment 5	→	Sequence number:	14,001	Range:	14,001	to	15,000

© McGraw Hill, LLC

TCP Connection

- TCP is **reliable** because it has connection and session mechanisms.
- When a host wants to communicate with another host using TCP, a connection must be established before data can be exchanged.
- This is known as the **Three-way Handshake**
- After the communication is completed, the session is closed and the connection is terminated.
-

Figure 9.26 Connection establishment using three-way handshaking

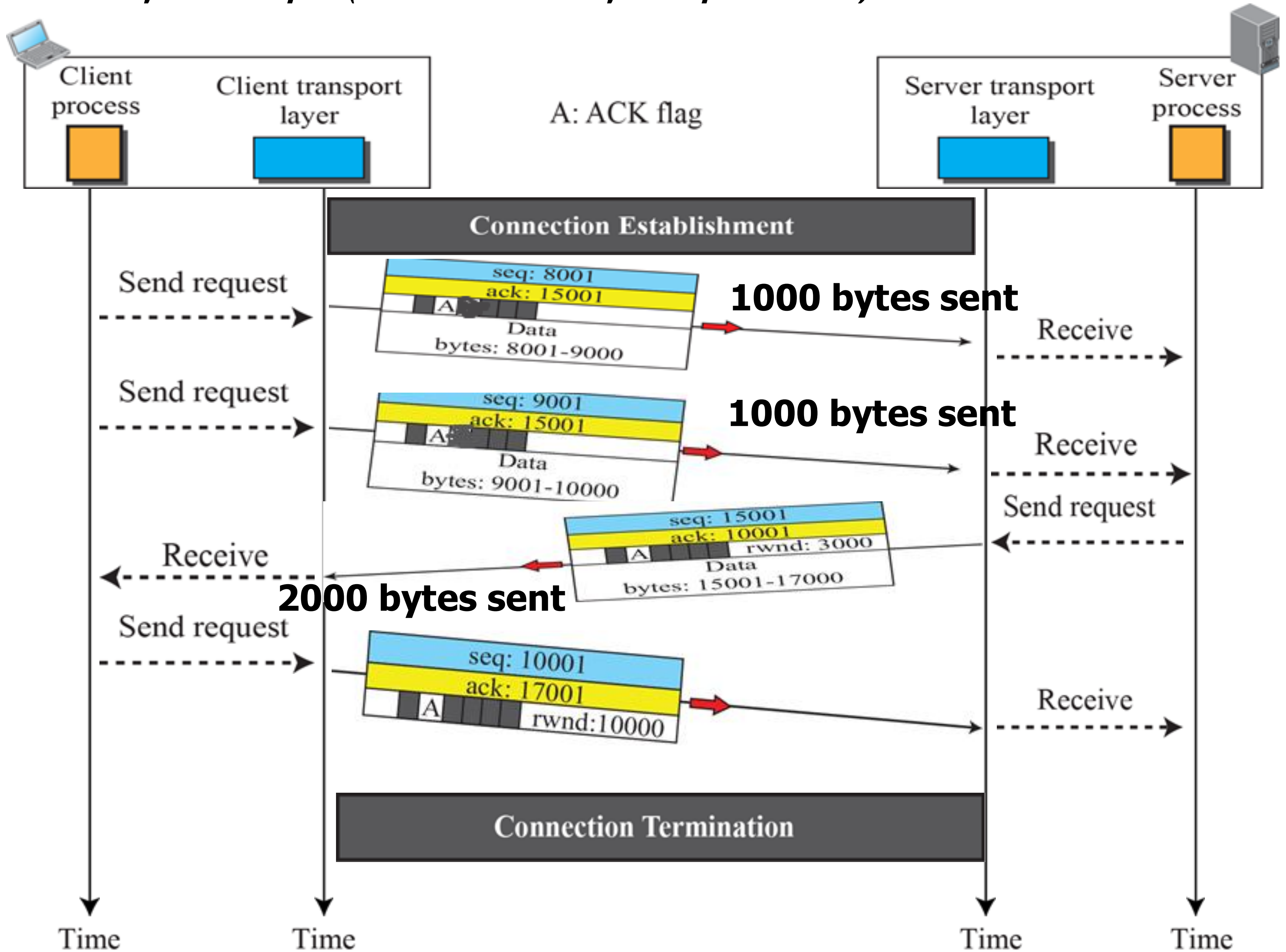


- **Step1:** The Client sends the initial segment which has SYN flag set to 1. This segments has the initial sequence number selected randomly.
- **Step 2:** The server responds by sending SYN+ACK segment which has the SYN and the ACK flags set to 1. This segment acknowledges the SYN segment sent by the client. Also, it has the initial sequence number for the segments sent by the server. This segment also defines the number of bytes that the **client** can send without waiting for acknowledgement from the server side. This is called the server receive window.
- **Step 3:** The client send a segment that acknowledges receipt of the previous segment sent by the server. Also, it carries the number of bytes that the **server** can send without waiting for acknowledgement from the client side. This is called the client receive window.

Reliable Delivery

- After the connection is established, the client and server can send data to each other.
- Each segment must be **acknowledged**. The acknowledgment number should be **the number of the last received bytes + 1.**
- Acknowledgment are cumulative. This means that if a device receives acknowledgment segment with acknowledgment number X, then this means that **all bytes up to byte number X-1** are received by the receiver and the receiver is ready for the next byte that has number **X**.
- The sender also starts a timer when it sends a segment, and it **retransmits** a segment if the timer expires before an acknowledgment arrives.

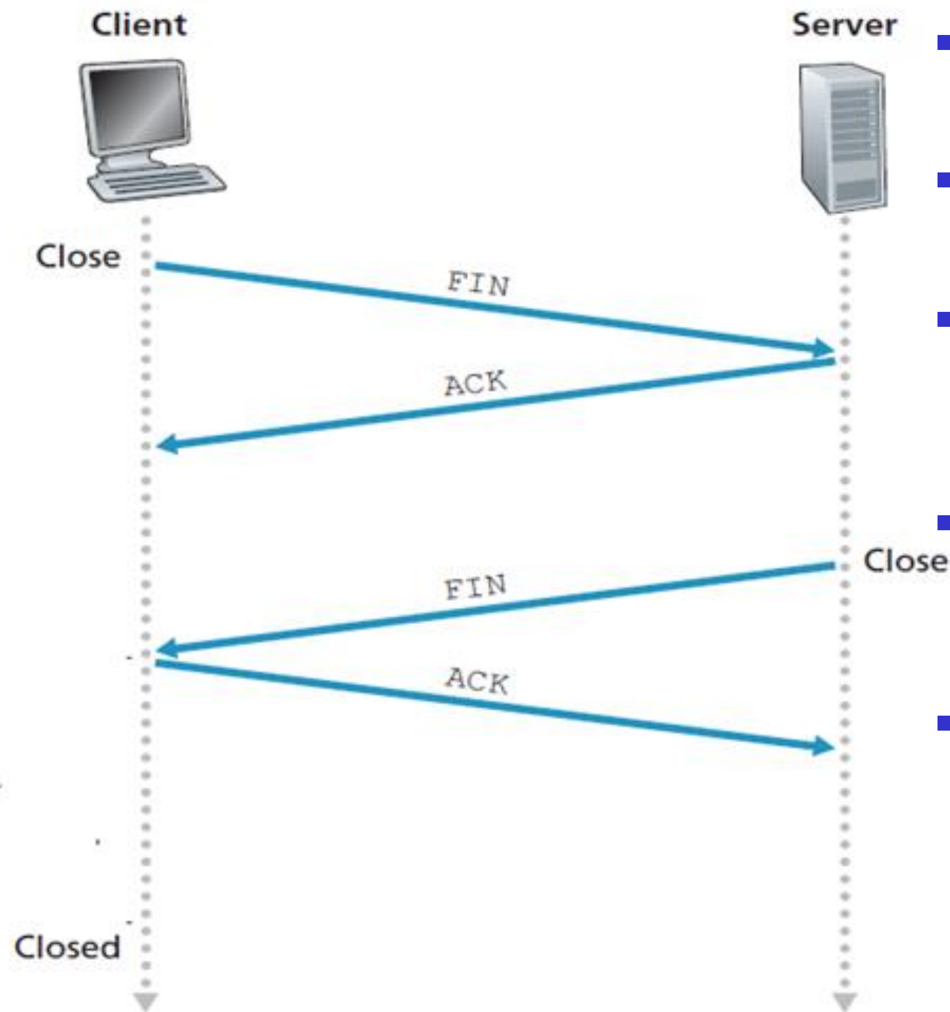
Data transfer example (See Next Slide for explanation)



Data transfer example

- After the connection was established, the client sends **two segments** each carries **1000 bytes**.
- The first segment has sequence number 8001 and carries bytes with sequence numbers 8001-9000.
- The second segment has sequence number 9001 carries bytes with sequence numbers 9001 - 10000
- The server sends a segment with sequence number 15001 that acknowledges the receipt of the 2000 bytes with acknowledgement number 10001 (*this means client received all bytes up to byte number 10000 and is ready for byte number 10001*).
- The same segment carries **2000 bytes** of data to the client with sequence numbers 15001- 17000.
- The client sends a segment that carries acknowledgement **only** with sequence number 10001 that acknowledges the receipt of the 2000 bytes with acknowledgement number 17001 (*this means client received all bytes up to byte number 17000 and is ready for byte number 17001*).

Time-line diagram for connection termination



- The 1st host sends a segment with the FIN bit set.
- The 2nd host replies with the ACK bit set.
- Second host can continue send data and the first host can only acknowledge but can't send data
- When second host wants to terminate the connection, it sends a segment with the FIN bit set.
- The 1st host replies with the ACK bit set:

Flow Control

- **TCP Windows**
  - **Flow control** uses windows to prevent a receiver from being overwhelmed by incoming data. Referred to a "**sliding windows**"
  - A window size specifies the **maximum number of segments the sender can forward without receiving an acknowledgment**.
  - Window size is included in every TCP segment starting with three-way handshake.
  - TCP implements flow control by **increasing and decreasing window sizes** as required.
  - TCP is a full duplex service. Both Client and server specify their own window sizes.
  - Each side has **two windows**: Send window and Receive Window
  - Window sizes are variable during the lifetime of a connection
  - Initial windows sizes are set at the connection establishment stage and then any side can change the size during the lifetime of a connection

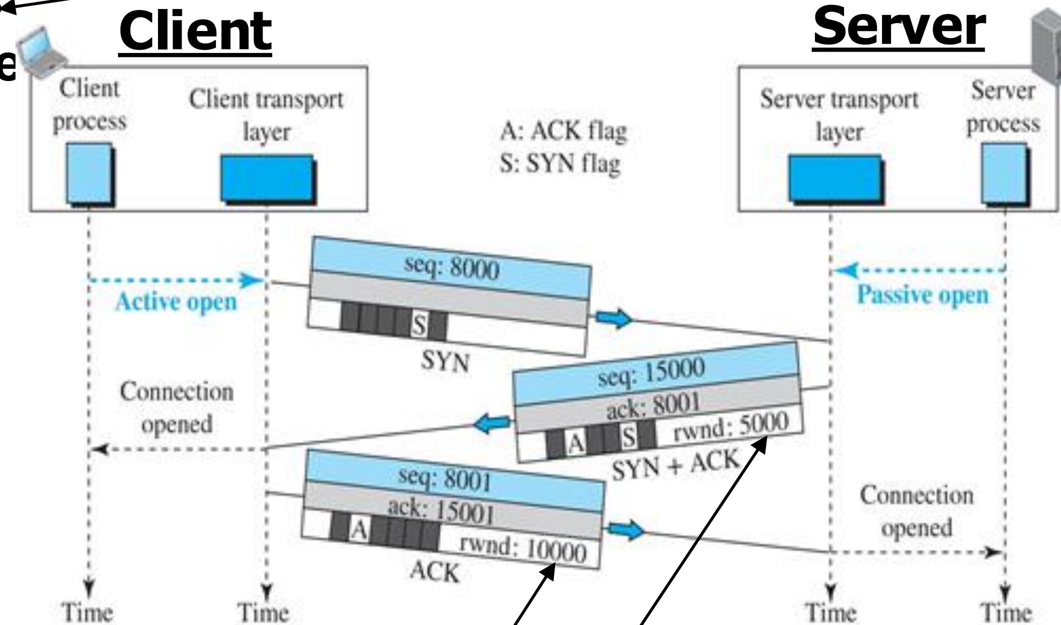
TCP Windows (Initial Windows sizes)

My Receive Window: 10,000

My send windows = Server's Receive Window: 5000

My Receive Window: 5000

My send windows = Client's Receive Window: 10,000



- **Client Receive Window Size=10,000 bytes** □ Client told the Server that the Server can only send 10,000 bytes before it has to stop and wait for an acknowledgement from the client. This is the Server Send Window Size.
- **Server Receive Window Size=5,000 bytes** □ Server told the client that the Client can only send 5,000 bytes before it has to stop and wait for an acknowledgement from Server. This is the Client Send Window Size.

TCP Applications

- Following applications require reliable data transfer through TCP:
  - WWW using HTTP
  - Electronic mail using SMTP
  - Telnet
  - File transfer using FTP

User Datagram Protocol (UDP)

- **Connectionless**
  - **No handshaking** between UDP sender, receiver
  - Each UDP segment handled **independently** of others
- A **server application** that uses UDP serves only **<u>ONE request</u>** at a time. All other requests are stored in a **queue** waiting for service.
- **<u>Unreliable protocol</u> has no flow and error control**
  - A UDP segment can be **lost**, **arrive out of order**, **duplicated**, or **corrupted**
  - **Checksum field checks error in <u>the entire UDP segment</u>. <u>It is Optional</u>**
  - UDP **doe not do anything to recover** from an error it simply **discard** the segment Application accepts full responsibility for errors
- It **uses port numbers** to multiplex/demultiplex data from/to the application layer.
- Advantages: Simple, **minimum overhead**, **no connection delay**
- **Services provided by UDP:**
  - Process-to-Process delivery
  - Error checking (however, if there is an error UDP does NOT do anything to recover from **error. It will just** discard the message

UDP Applications

- Used for applications that can tolerate small amount of packet loss:
  - Multimedia applications,
  - Internet telephony,
  - real-time-video conferencing
  - Domain Name System messages
  - Audio
  - Some Routing Protocols

UDP is suitable for applications that makes one request

- A client-server application such as DNS uses the services of UDP because a client needs to send a **short request** to a server and to receive a **quick response** from it.
- The request and response can each fit in **one user datagram.**
- Since only one message is exchanged in each direction, the connectionless feature is not an issue; the client or server does not worry that messages are delivered out of order.

UDP is NOT suitable for applications that do not tolerate loss or errors or out of order arrival

- A client-server application such as SMTP, which is used in electronic mail, cannot use the services of UDP because a user can send a long e-mail message, which may include multimedia (images, audio, or video). If the application uses UDP and the message does not fit in one single user datagram, the message must be split by the application into different user datagrams. Here the connectionless service may create problems. The user datagrams may arrive and be delivered to the receiver application **out of order**. The receiver application may not be able to reorder the pieces. This means the connectionless service has a disadvantage for an application program that **sends long messages.**

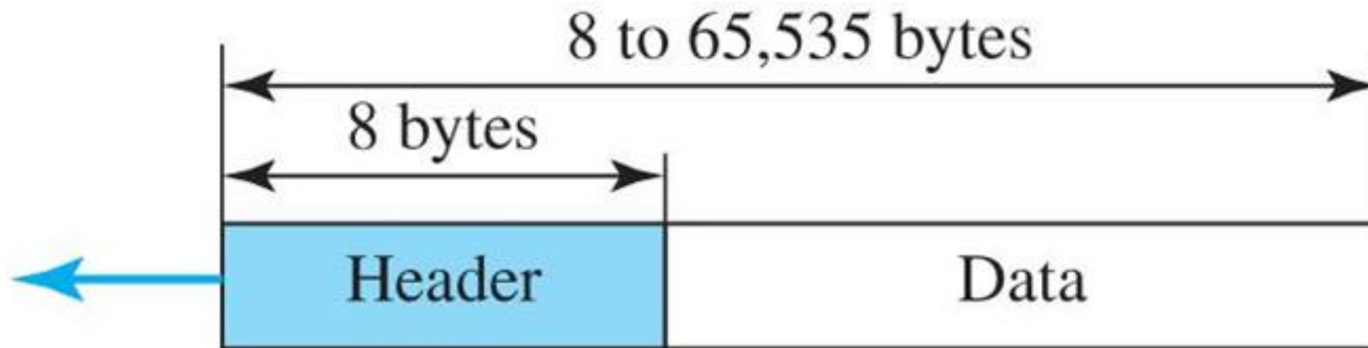
UDP is NOT suitable for applications that do not tolerate loss or errors out of order arrival

- Assume we are downloading a very large text file from the Internet. We definitely need to use a transport layer that provides reliable service. We don't want part of the file to be **missing or corrupted** when we open the file. The delay created between the deliveries of the parts is not a concern for us; we wait until the whole file is composed before looking at it. In this case, UDP is not a suitable transport layer.

UDP is suitable for real time applications

- Assume we are using a real-time interactive application, such as Skype. **Audio and video are divided into frames and sent one after another**. If the transport layer is supposed to resend a corrupted or lost frame, **the synchronizing of the whole transmission may be lost**. The viewer suddenly sees a blank screen and needs to wait until the second transmission arrives. **This is not tolerable.** However, if each small part of the screen is sent using one single user datagram, the receiving UDP can easily ignore the corrupted or lost packet and deliver the rest to the application program. That part of the screen is blank for a very short period of time, which most viewers do not even notice.

Figure 9.18 User datagram packet format



a. UDP user datagram

b. Header format