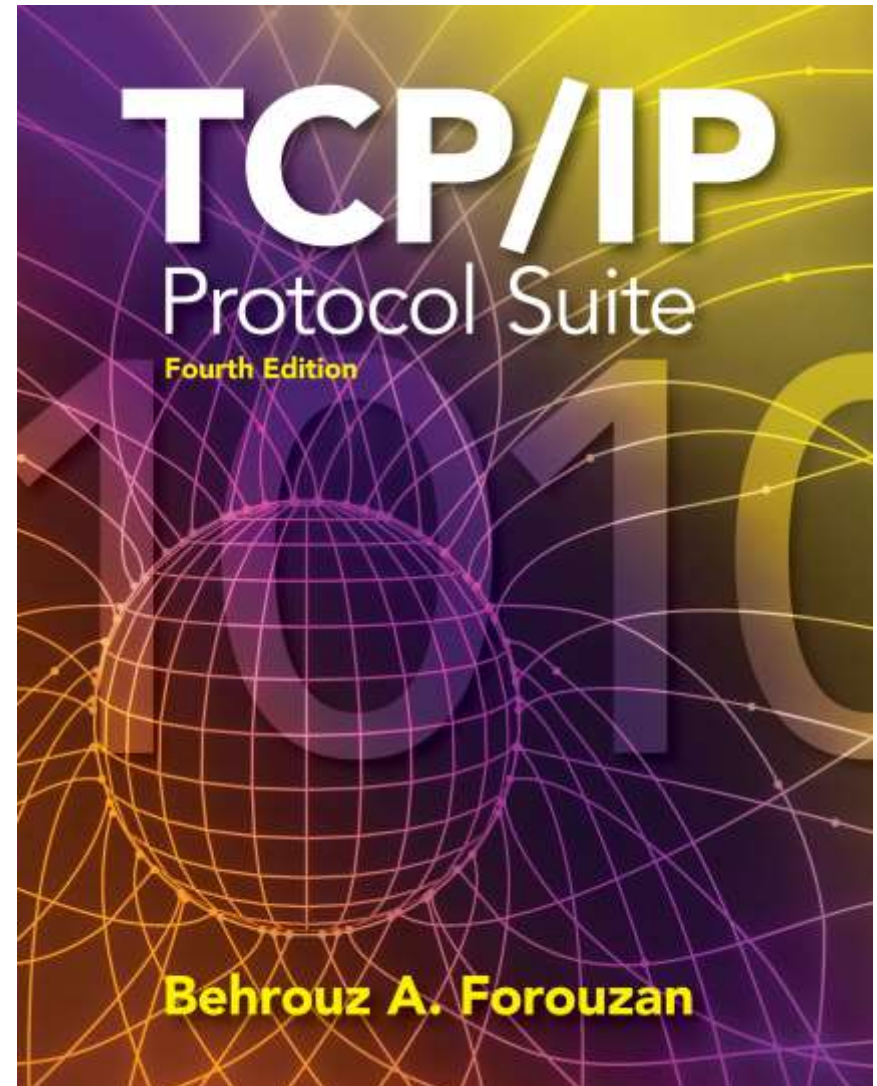


Chapter 2

The OSI Model and the TCP/IP Protocol Suite



OBJECTIVES:

- ❑ To discuss the idea of multiple layering in data communication and networking and the interrelationship between layers.**
- ❑ To discuss the OSI model and its layer architecture and to show the interface between the layers.**
- ❑ To briefly discuss the functions of each layer in the OSI model.**
- ❑ To introduce the TCP/IP protocol suite and compare its layers with the ones in the OSI model.**
- ❑ To show the functionality of each layer in the TCP/IP protocol with some examples.**
- ❑ To discuss the addressing mechanism used in some layers of the TCP/IP protocol suite for the delivery of a message from the source to the destination.**

Chapter Outline

2.1 Protocol Layers

2.2 The OSI Model

2.3 TCP/IP Protocol Suite

2.4 Addressing

2-1 PROTOCOL LAYERS

In Chapter 1, we discussed that a protocol is required when two entities need to communicate. When communication is not simple, we may divide the complex task of communication into several layers. In this case, we may need several protocols, one for each layer.

Let us use a scenario in communication in which the role of protocol layering may be better understood. We use two examples. In the first example, communication is so simple that it can occur in only one layer.

Topics Discussed in the Section

- ✓ **Hierarchy**
- ✓ **Services**

2-2 THE OSI MODEL

Established in 1947, the *International Standards Organization (ISO)* is a multinational body dedicated to worldwide agreement on international standards. Almost three-fourths of countries in the world are represented in the ISO. An ISO standard that covers all aspects of network communications is the *Open Systems Interconnection (OSI)* model. It was first introduced in the late 1970s.

Topics Discussed in the Section

- ✓ **Layered Architecture**
- ✓ **Layer-to-layer Communication**
- ✓ **Encapsulation**
- ✓ **Layers in the OSI Model**
- ✓ **Summary of OSI Layers**



Note

***ISO is the organization;
OSI is the model.***

Figure 2.3 *The OSI model*

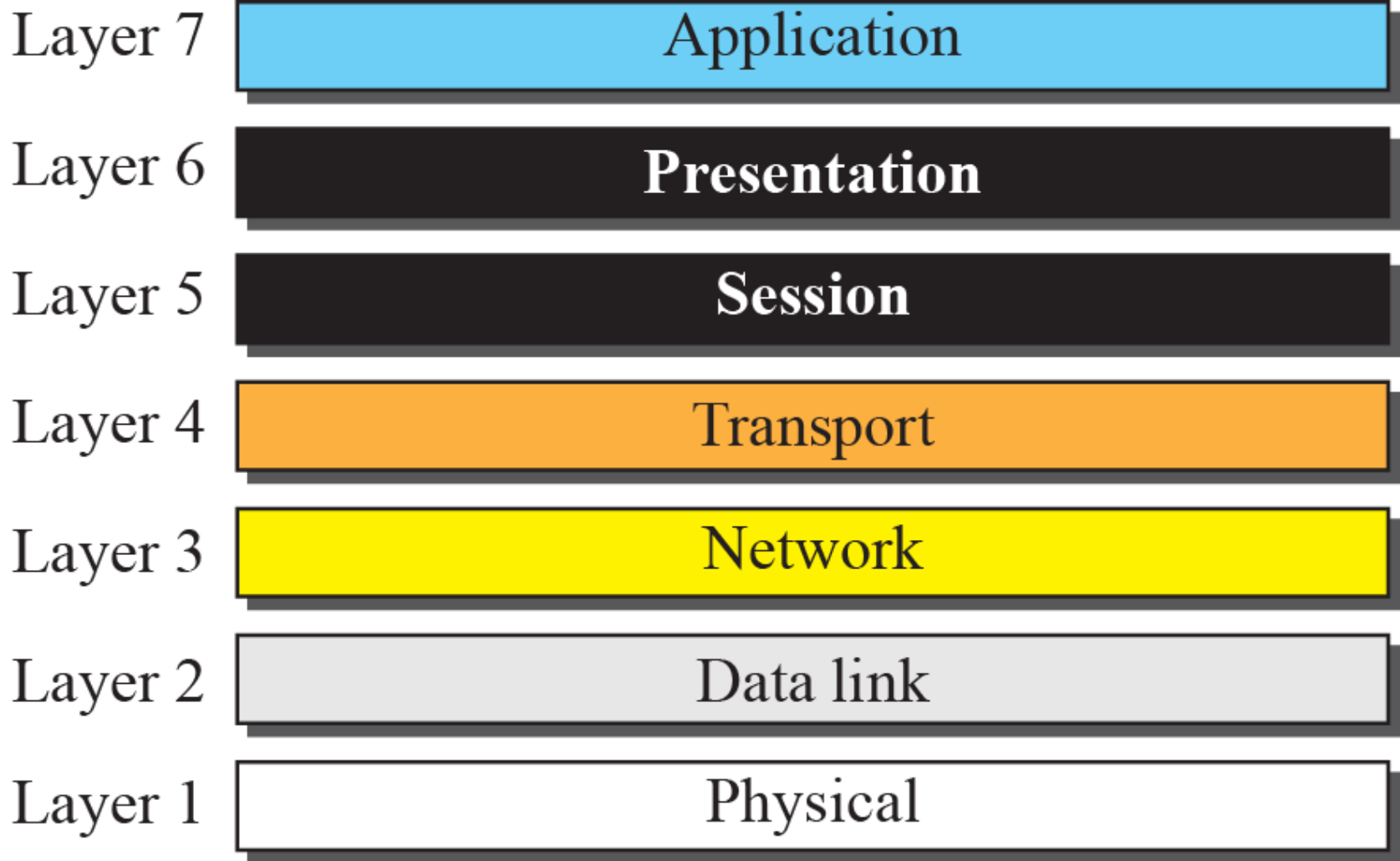


Figure 2.4 *OSI layers*

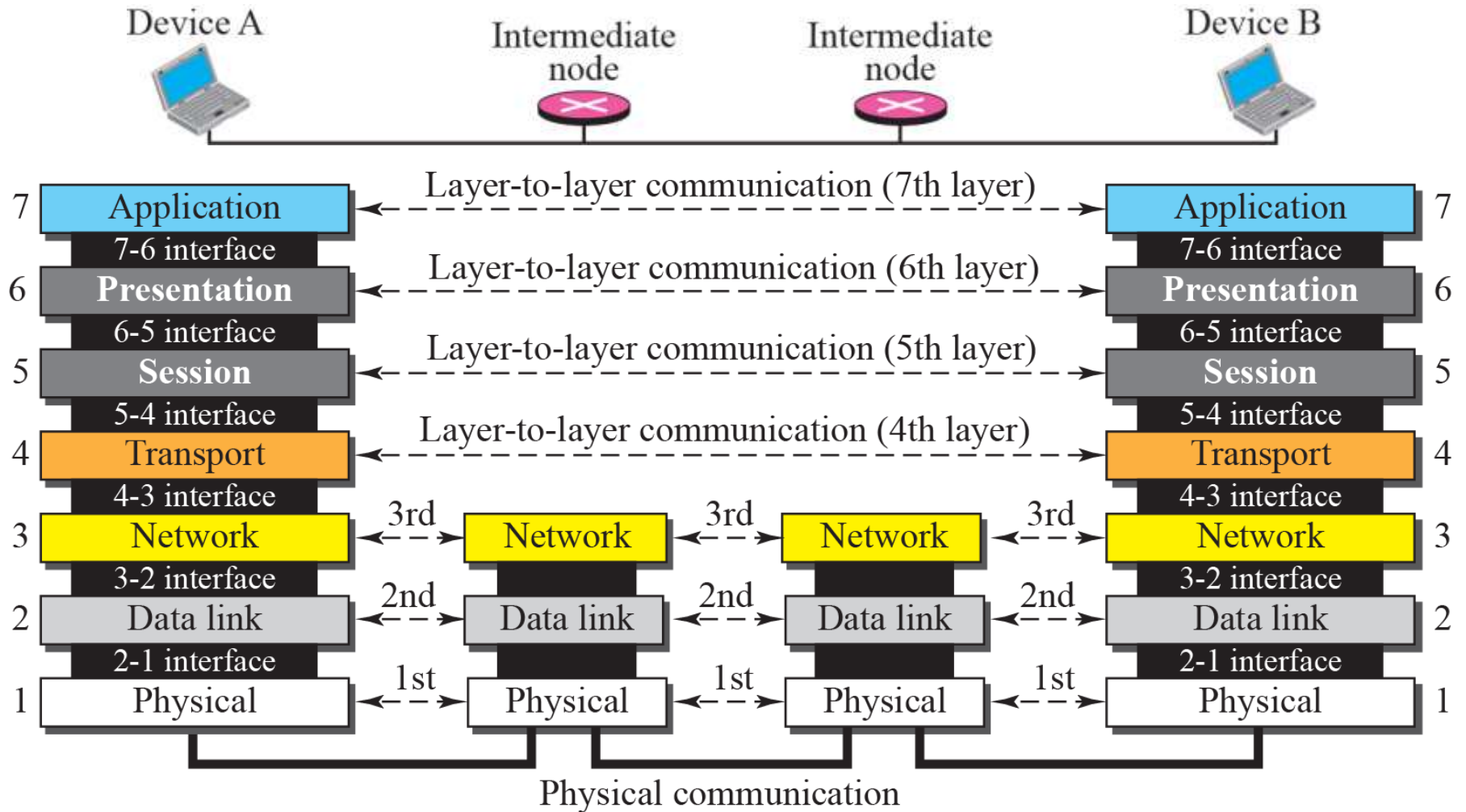
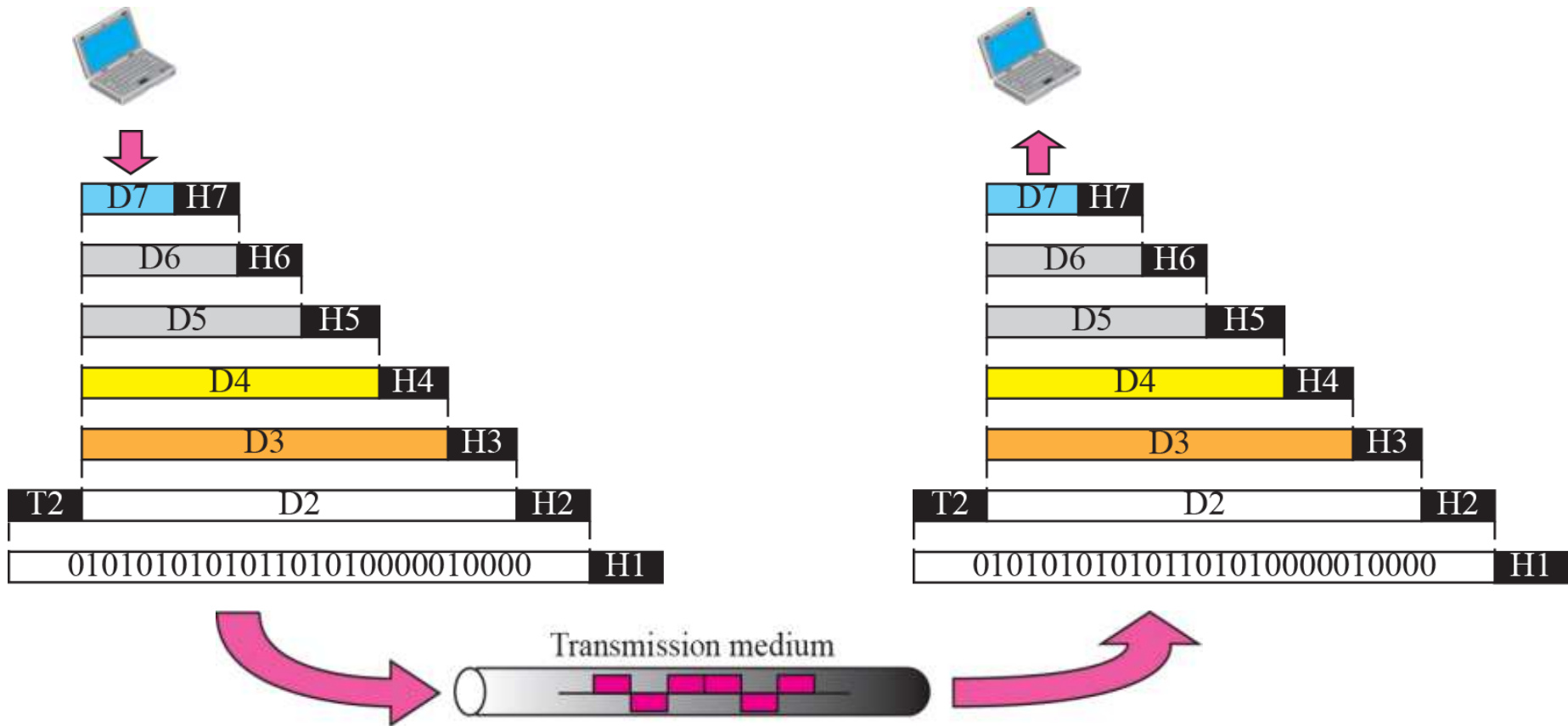


Figure 2.5 *An exchange using the OSI model*

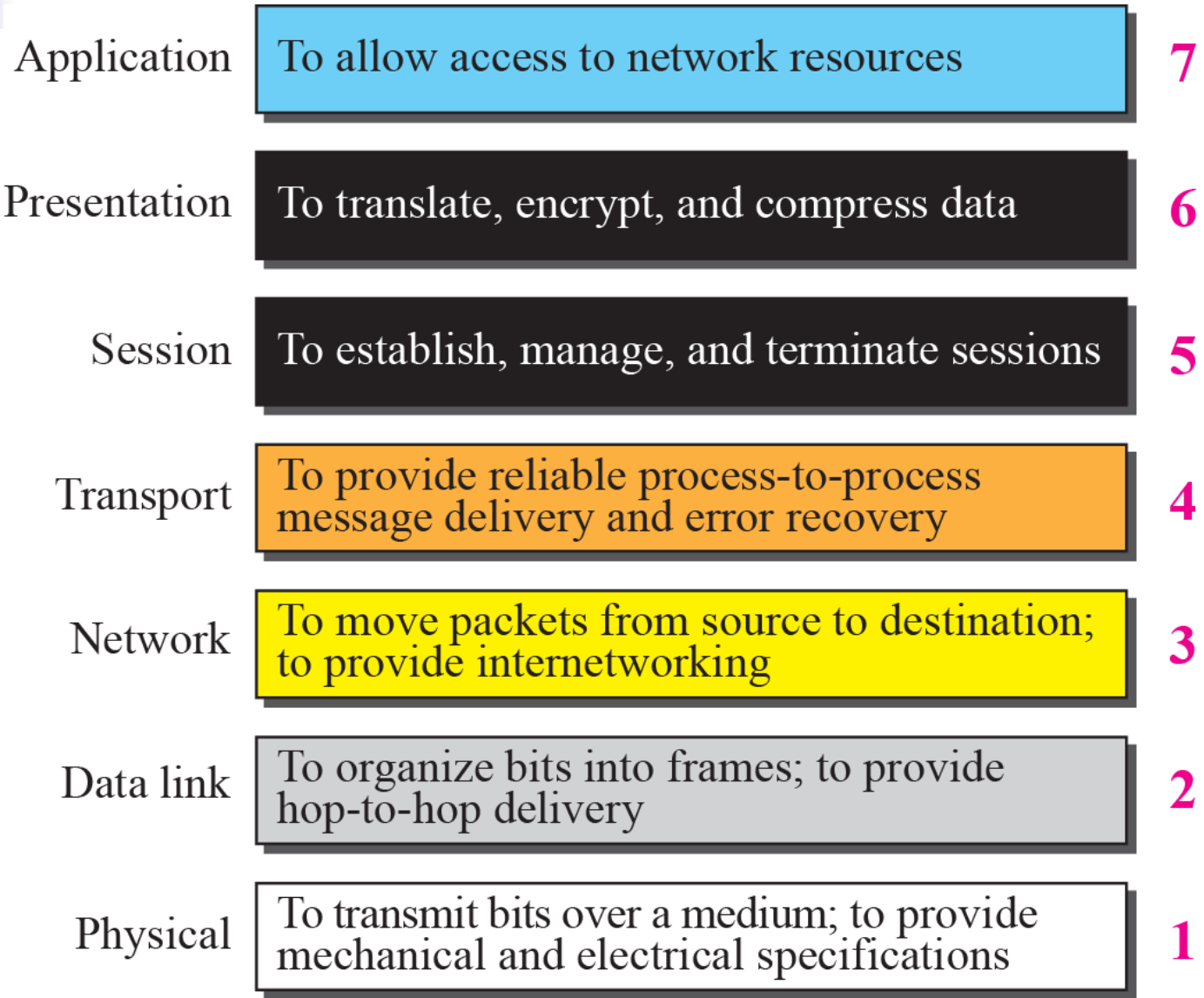




Note

The physical layer is responsible for moving individual bits from one (node) to the next.

Figure 2.6 *Summary of OSI Layers*



2-3 TCP/IP PROTOCOL SUITE

The TCP/IP protocol suite was developed prior to the OSI model. Therefore, the layers in the TCP/IP protocol suite do not match exactly with those in the OSI model. The original TCP/IP protocol suite was defined as four software layers built upon the hardware. Today, however, TCP/IP is thought of as a five-layer model with the layers named similarly to the ones in the OSI model. Figure 2.7 shows both configurations.

Topics Discussed in the Section

- ✓ **Comparison between OSI and TCP/IP**
- ✓ **Layers in the TCP/IP Suite**

Figure 2.7 *Layers in the TCP/IP Protocol Suite*

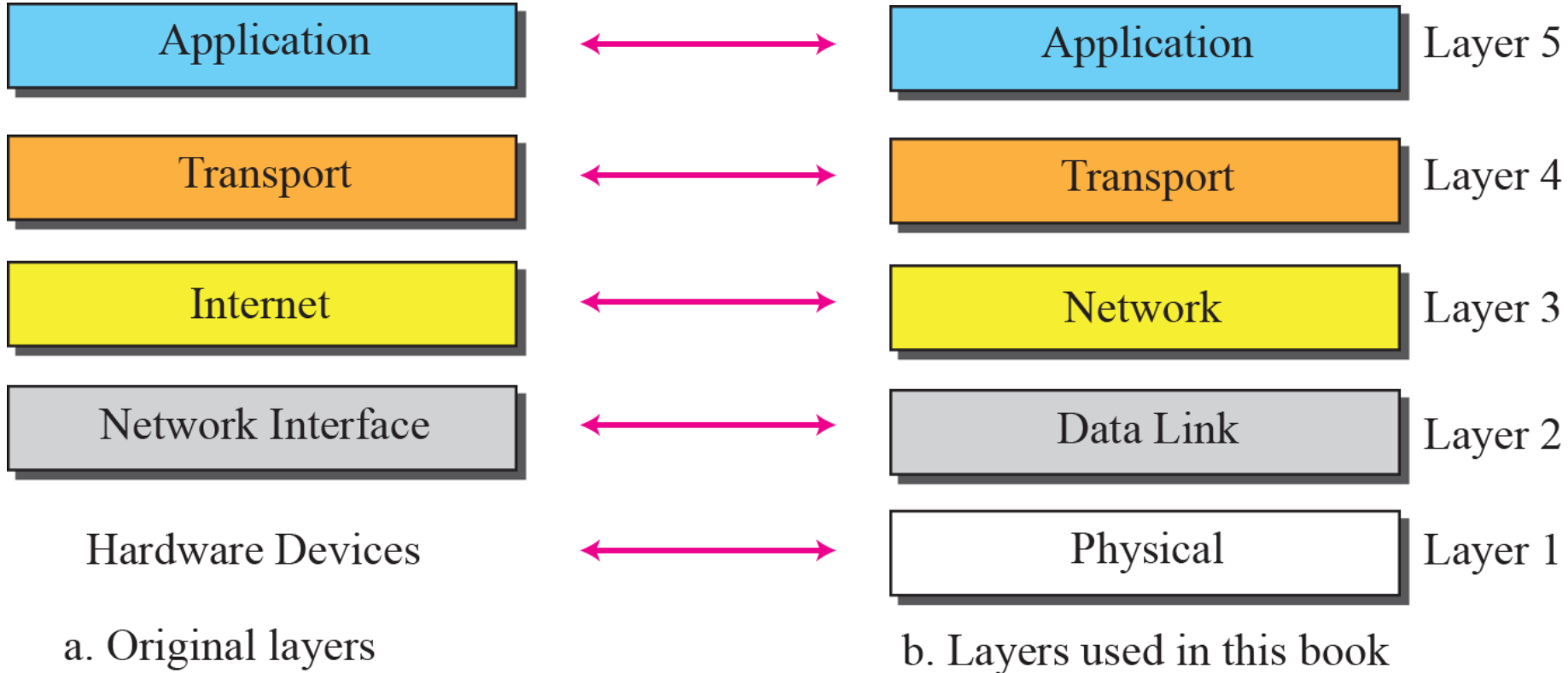


Figure 2.8 *TCP/IP and OSI model*

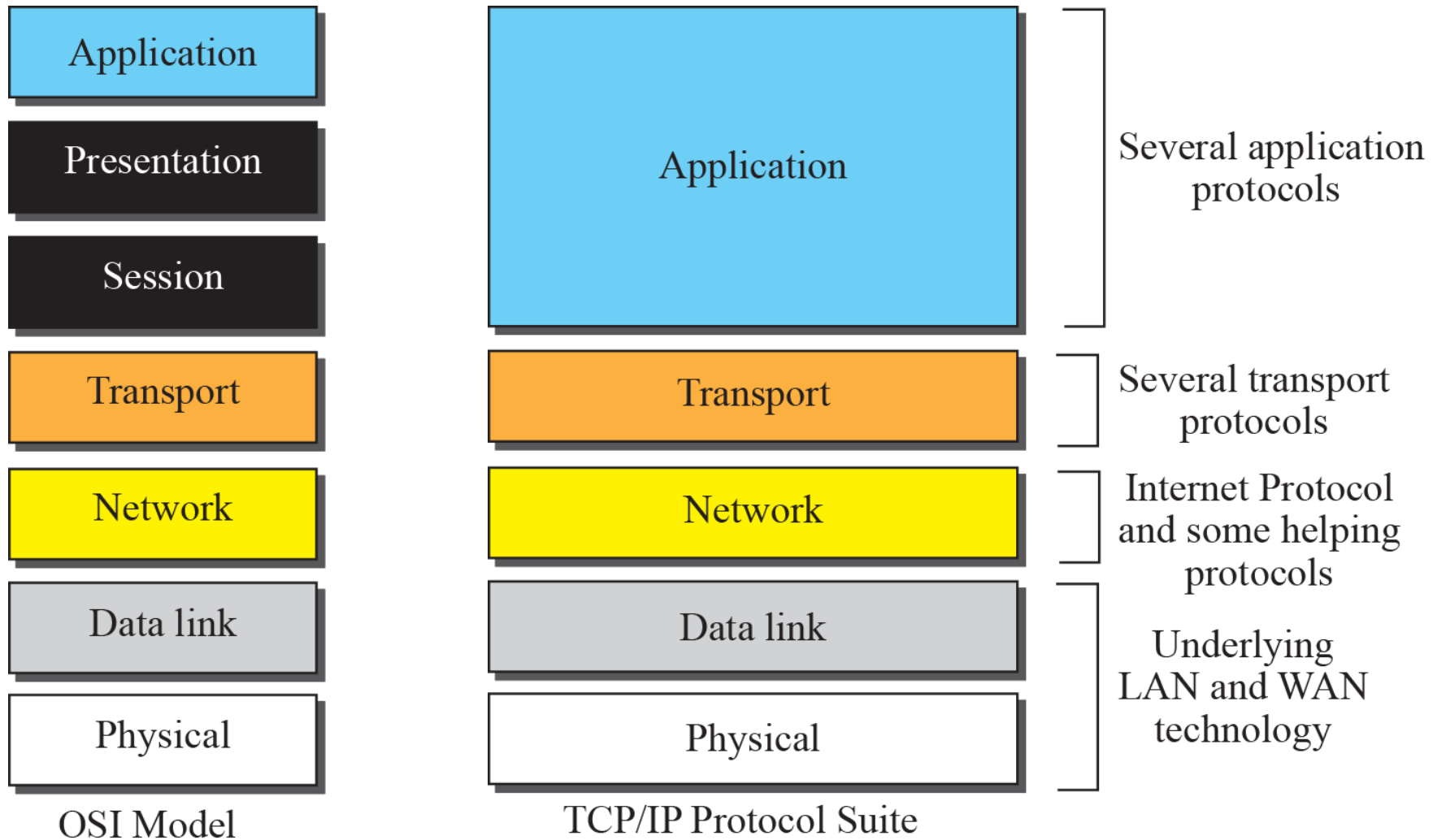


Figure 2.9 *A private internet*

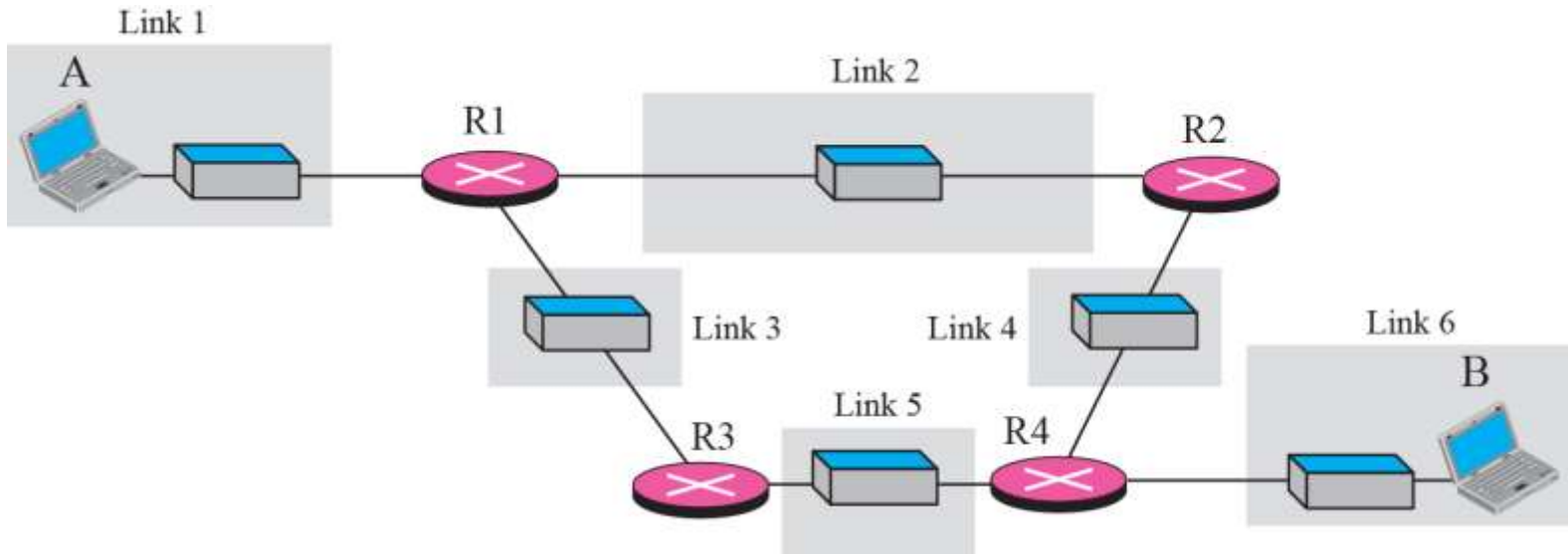
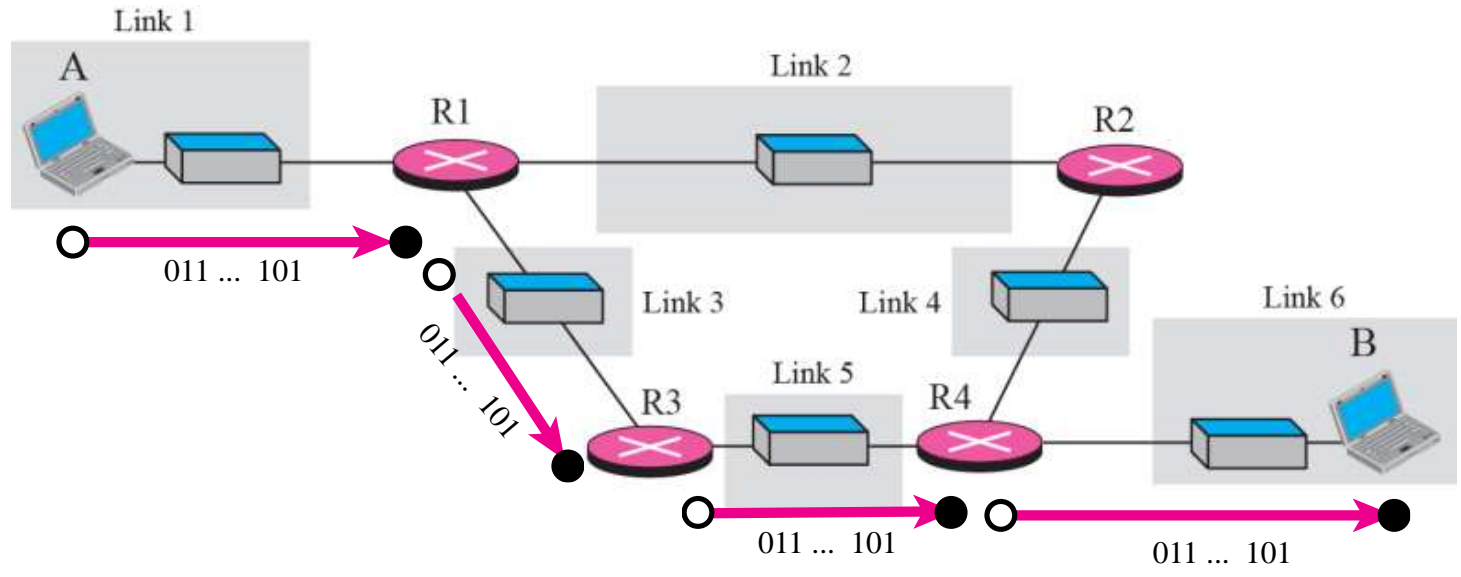
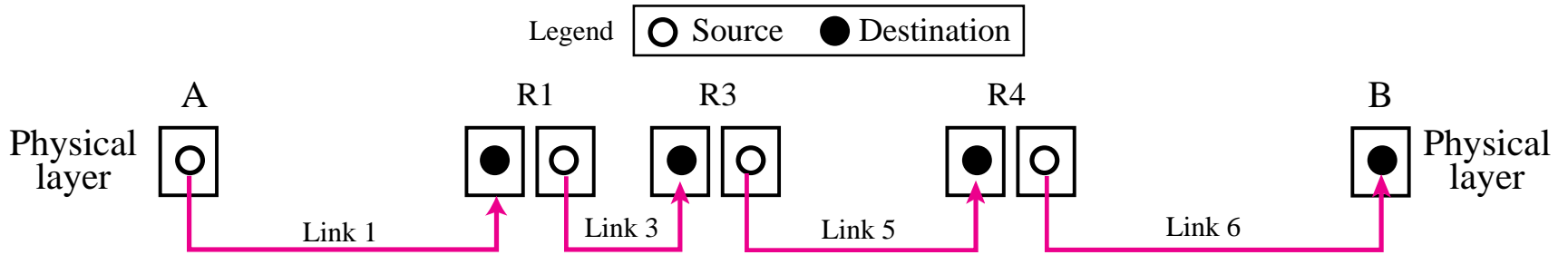


Figure 2.10 *Communication at the physical layer*

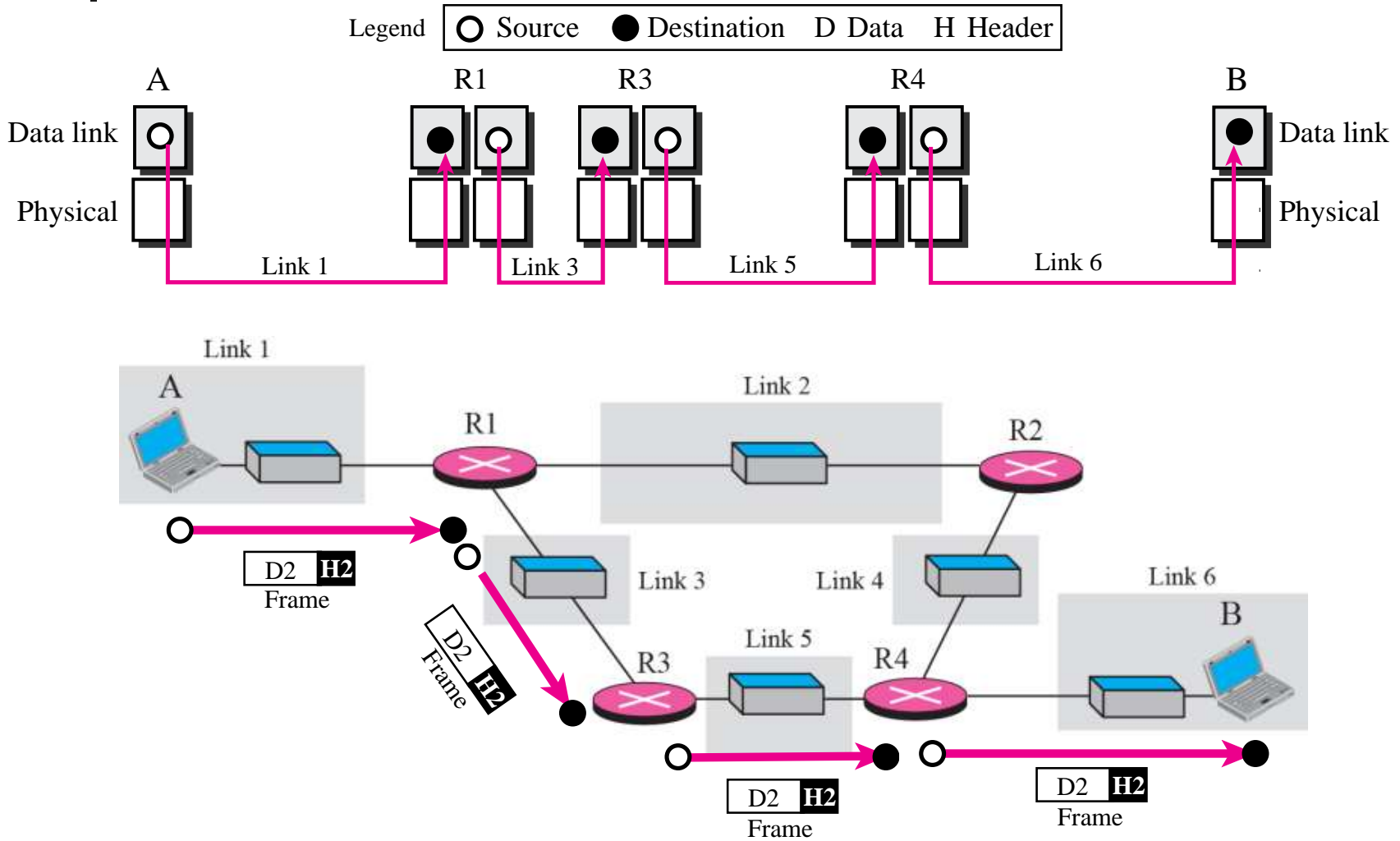




Note

The unit of communication at the physical layer is a bit.

Figure 2.11 *Communication at the data link layer*

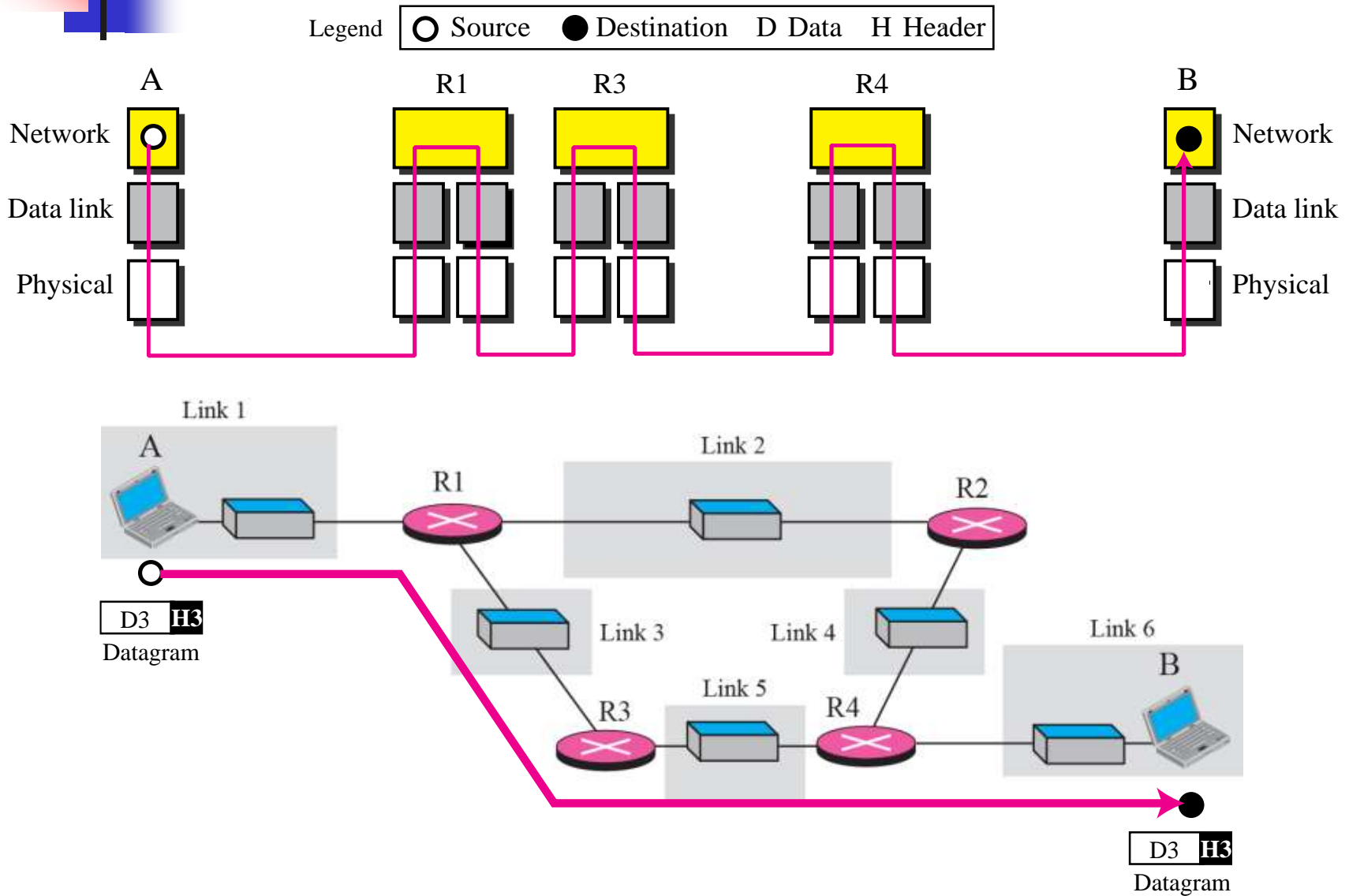




Note

The unit of communication at the data link layer is a frame.

Figure 2.12 *Communication at the network layer*

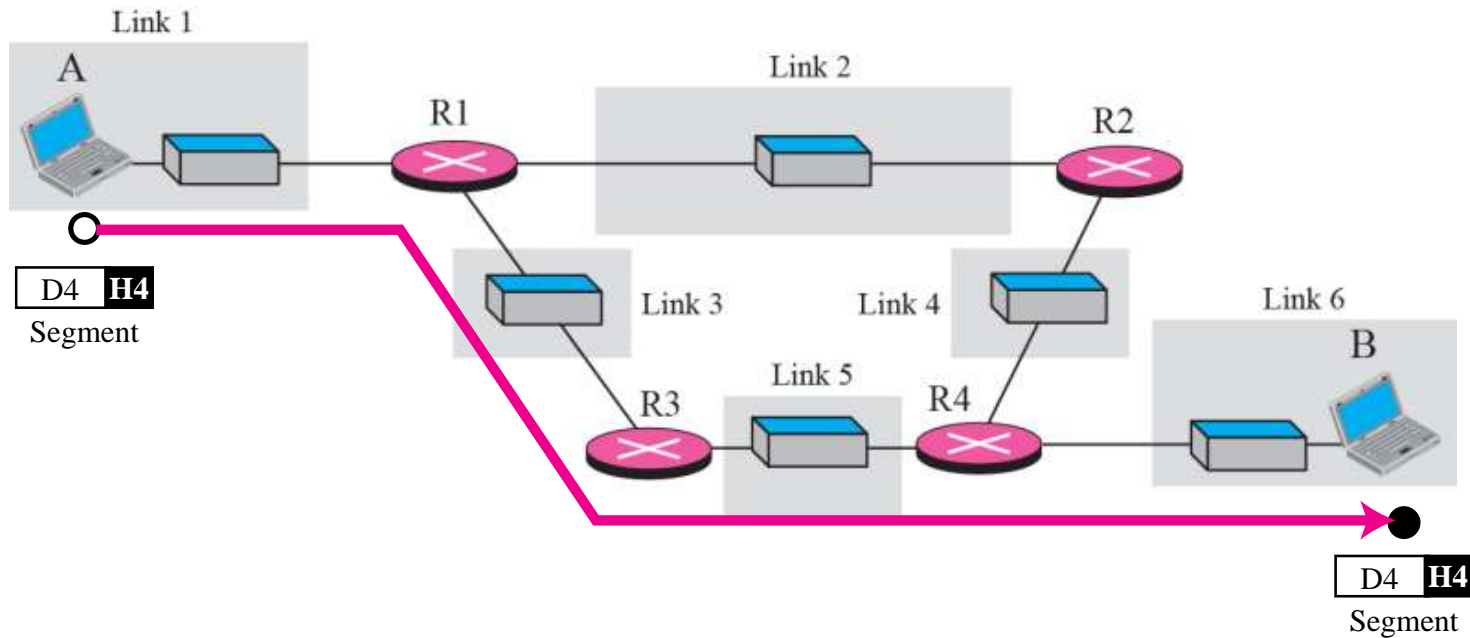
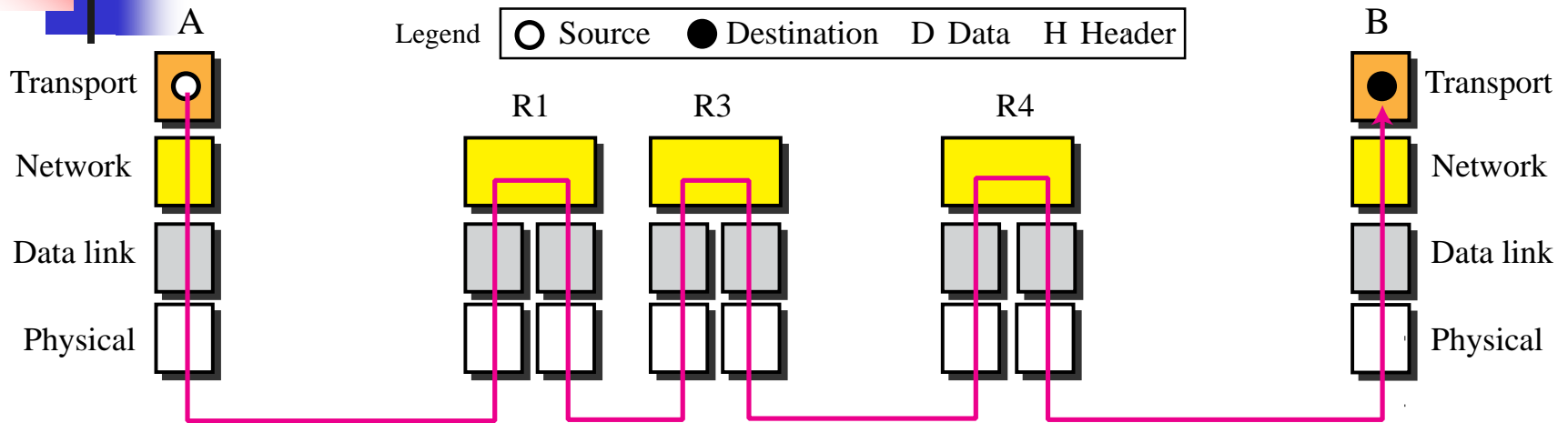




Note

The unit of communication at the network layer is a datagram.

Figure 2.13 *Communication at transport layer*

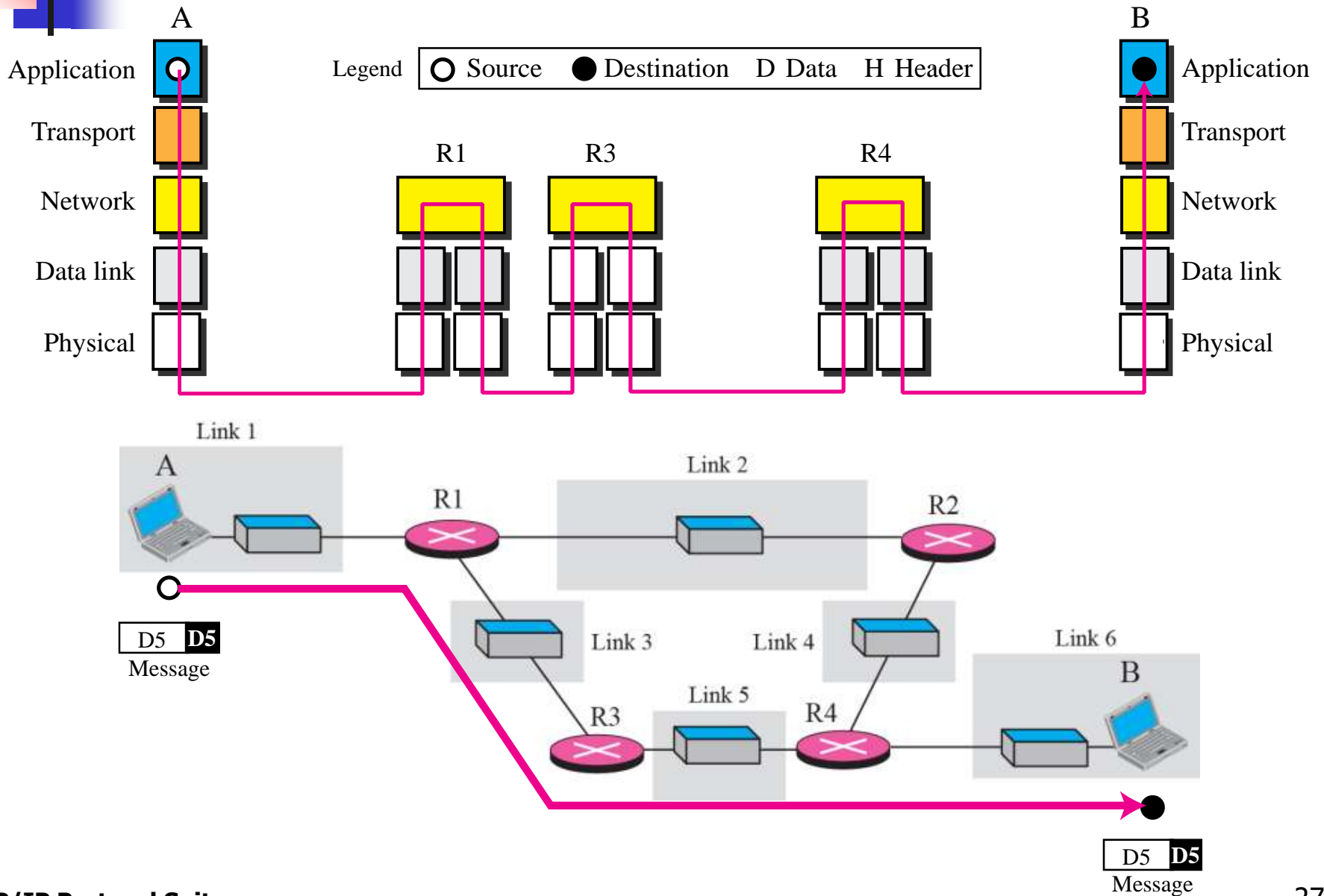




Note

The unit of communication at the transport layer is a segment, user datagram, or a packet, depending on the specific protocol used in this layer.

Figure 2.14 *Communication at application layer*





Note

The unit of communication at the application layer is a message.

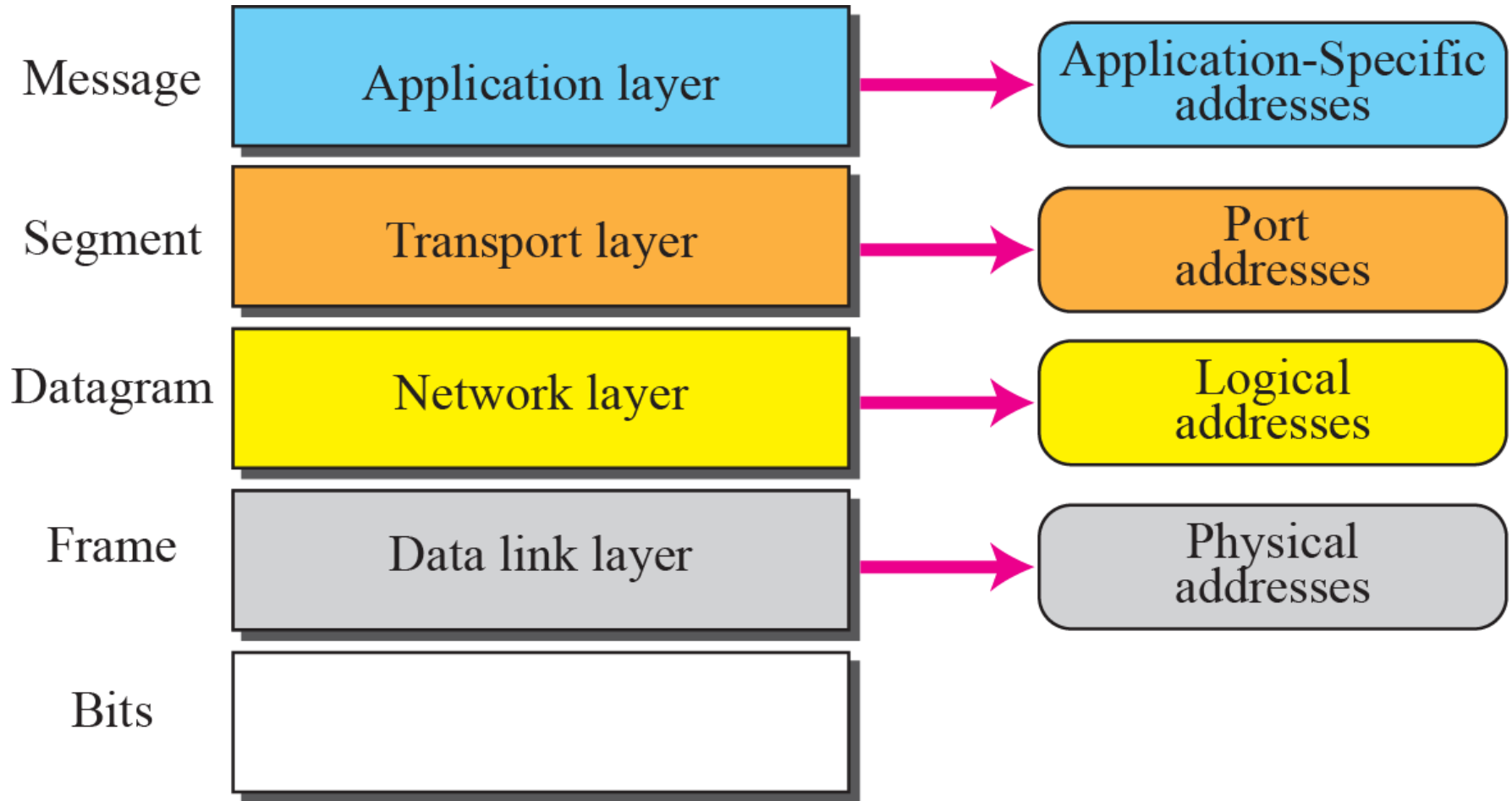
2-4 ADDRESSING

Four levels of addresses are used in an internet employing the TCP/IP protocols: physical address, logical address, port address, and application-specific address. Each address is related to a one layer in the TCP/IP architecture, as shown in Figure 2.15.

Topics Discussed in the Section

- ✓ **Physical Addresses**
- ✓ **Logical Addresses**
- ✓ **Port Addresses**
- ✓ **Application-Specific Addresses**

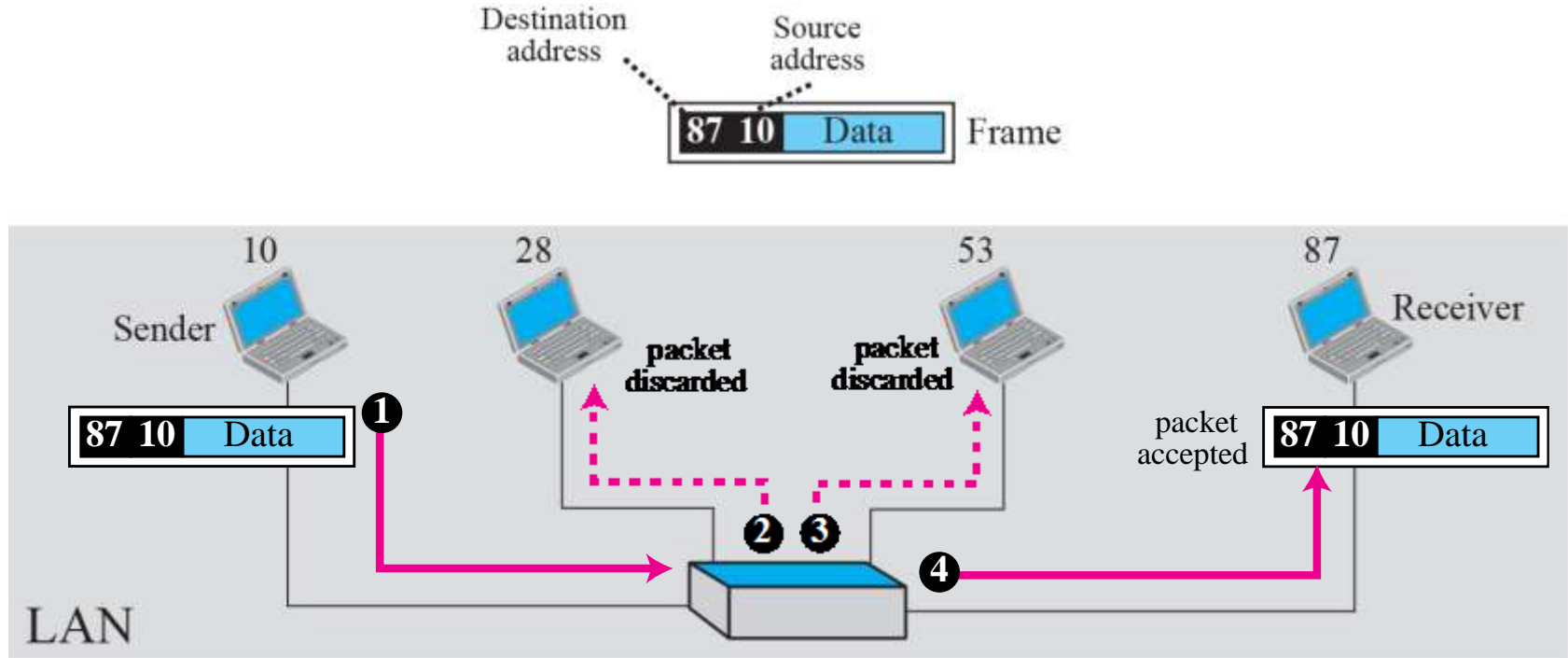
Figure 2.15 *Addresses in the TCP/IP protocol suite*



Example 2.3

In Figure 2.16 a node with physical address 10 sends a frame to a node with physical address 87. The two nodes are connected by a link (a LAN). At the data link layer, this frame contains physical (link) addresses in the header. These are the only addresses needed. The rest of the header contains other information needed at this level. As the figure shows, the computer with physical address 10 is the sender, and the computer with physical address 87 is the receiver. The data link layer at the sender receives data from an upper layer. It encapsulates the data in a frame. The frame is propagated through the LAN. Each station with a physical address other than 87 drops the frame because the destination address in the frame does not match its own physical address. The intended destination computer, however, finds a match between the destination address in the frame and its own physical address.

Figure 2.16 *Example 2.3: physical addresses*



Example 2.4

As we will see in Chapter 3, most local area networks use a 48-bit (6-byte) physical address written as 12 hexadecimal digits; every byte (2 hexadecimal digits) is separated by a colon, as shown below:

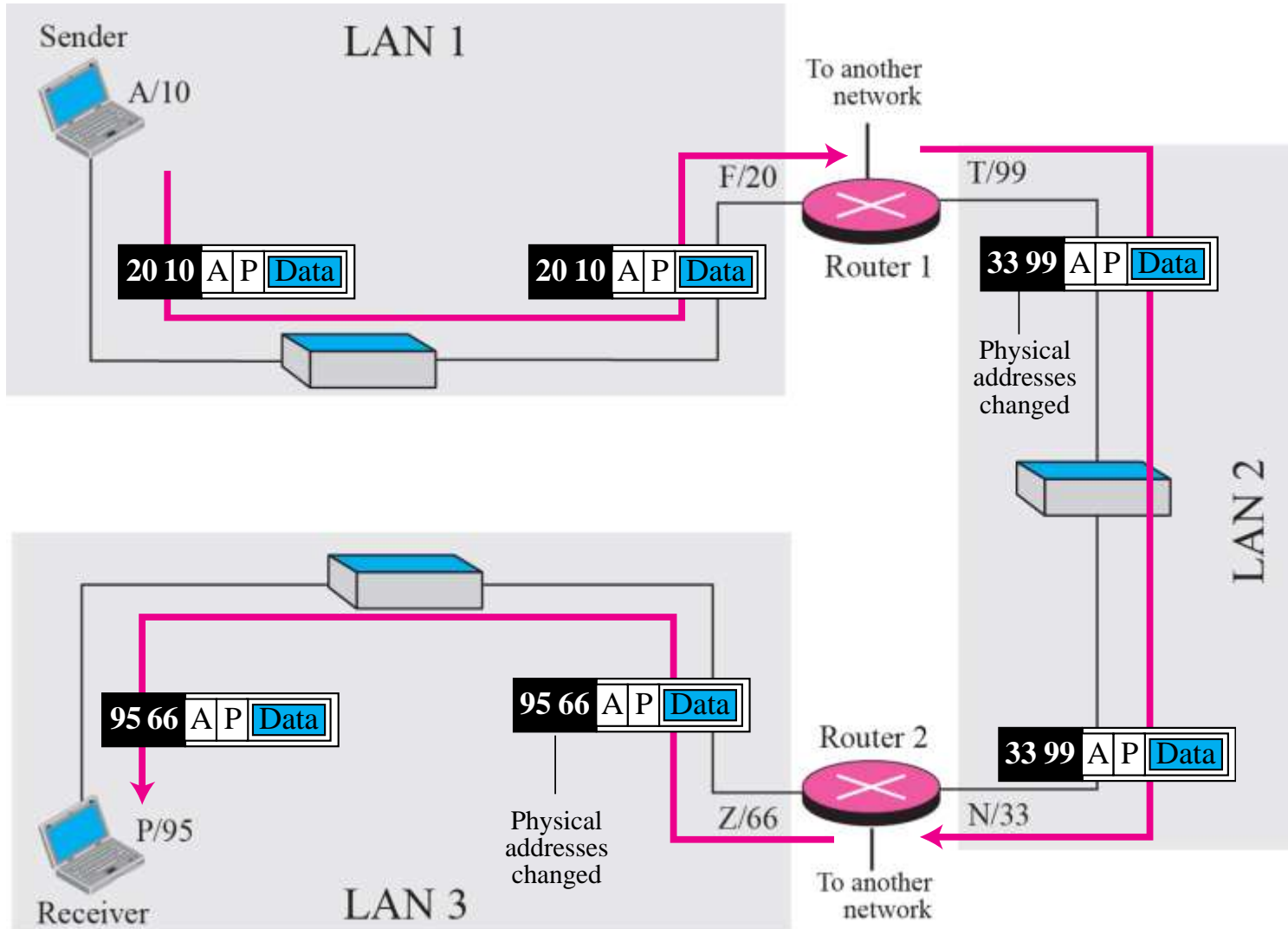
07:01:02:01:2C:4B

A 6-byte (12 hexadecimal digits) physical address

Example 2.5

Figure 2.17 shows a part of an internet with two routers connecting three LANs. Each device (computer or router) has a pair of addresses (logical and physical) for each connection. In this case, each computer is connected to only one link and therefore has only one pair of addresses. Each router, however, is connected to three networks. So each router has three pairs of addresses, one for each connection. Although it may be obvious that each router must have a separate physical address for each connection, it may not be obvious why it needs a logical address for each connection. We discuss these issues in Chapters 11 and 12 when we discuss routing. The computer with logical address A and physical address 10 needs to send a packet to the computer with logical address P and physical address 95. We use letters to show the logical addresses and numbers for physical addresses, but note that both are actually numbers, as we will see in later chapters.

Figure 2.17 *Example 2.5: logical addresses*





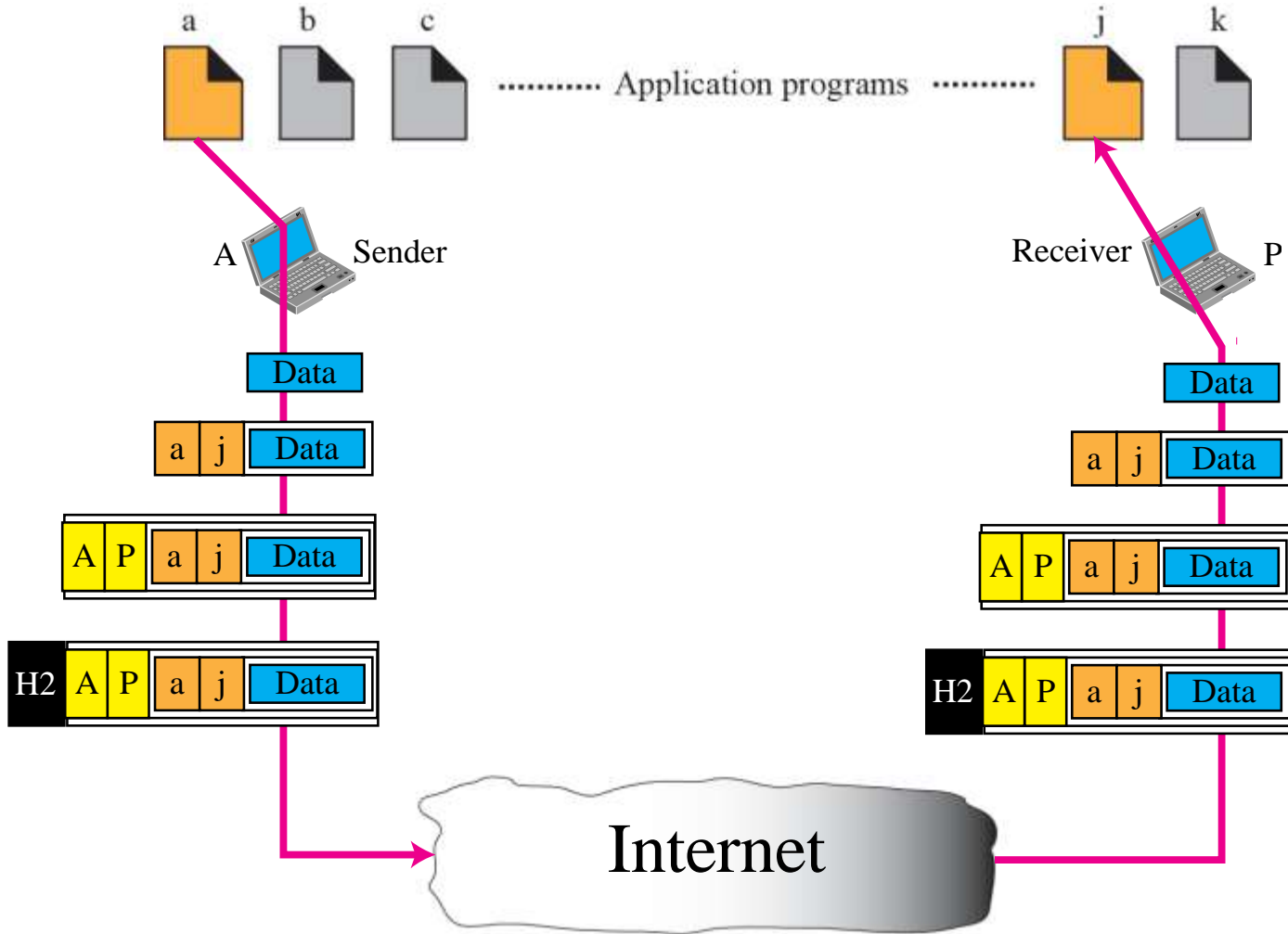
Note

The physical addresses will change from hop to hop, but the logical addresses remain the same.

Example 2.6

Figure 2.18 shows two computers communicating via the Internet. The sending computer is running three processes at this time with port addresses a, b, and c. The receiving computer is running two processes at this time with port addresses j and k. Process a in the sending computer needs to communicate with process j in the receiving computer. Note that although both computers are using the same application, FTP, for example, the port addresses are different because one is a client program and the other is a server program, as we will see in Chapter 17.

Figure 2.18 *Example 2.6: port numbers*





Note

The physical addresses change from hop to hop, but the logical and port addresses usually remain the same.

Example 2.7

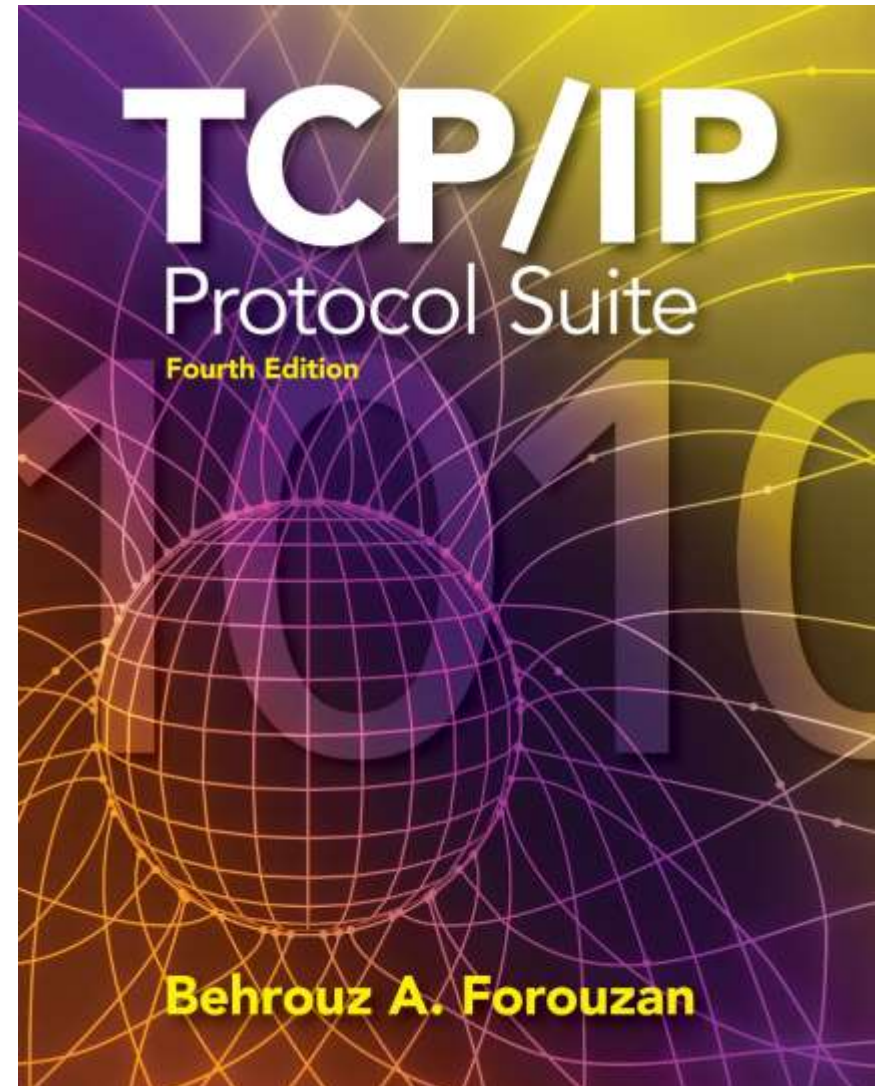
As we will see in future chapters, a port address is a 16-bit address represented by one decimal number as shown.

753

A 16-bit port address represented as one single number

Chapter 11

Unicast Routing Protocols



Chapter Outline

11.1 Introduction

*11.2 Intra- and Inter-Domain
Routing*

11.3 Distance Vector Routing

11.4 RIP

11.5 Link State Routing

11.6 OSPF

11.7 Path Vector Routing

11.8 BGP

11-1 INTRODUCTION

An internet is a combination of networks connected by routers. When a datagram goes from a source to a destination, it will probably pass through many routers until it reaches the router attached to the destination network.

Topics Discussed in the Section

- ✓ **Cost or Metric**
- ✓ **Static versus Dynamic Routing Table**
- ✓ **Routing Protocol**

11-2 INTER- AND INTRA-DOMAIN ROUTING

Today, an internet can be so large that one routing protocol cannot handle the task of updating the routing tables of all routers. For this reason, an internet is divided into autonomous systems. An autonomous system (AS) is a group of networks and routers under the authority of a single administration. Routing inside an autonomous system is called intra-domain routing. Routing between autonomous systems is called inter-domain routing

Figure 11.1 *Autonomous systems*

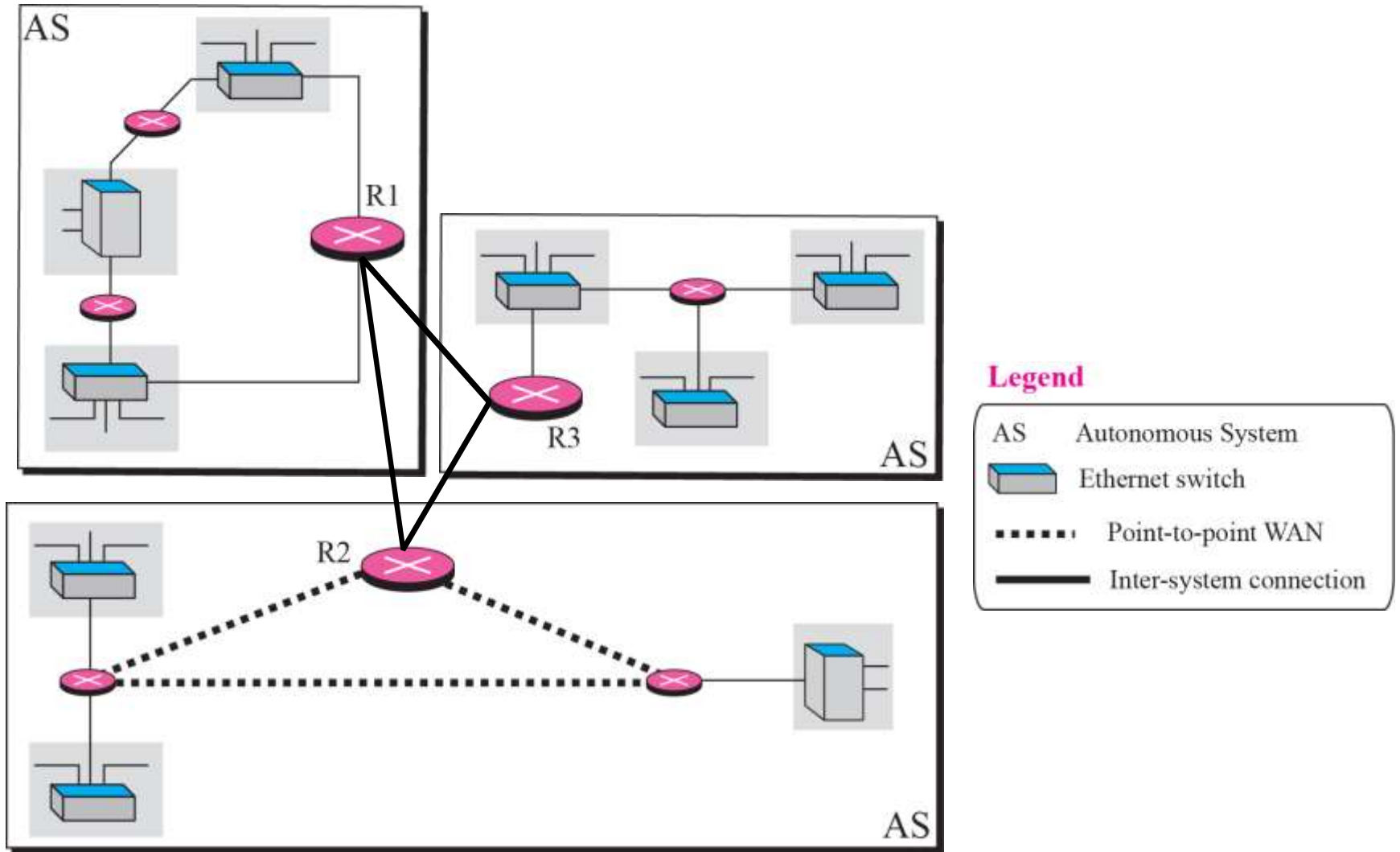
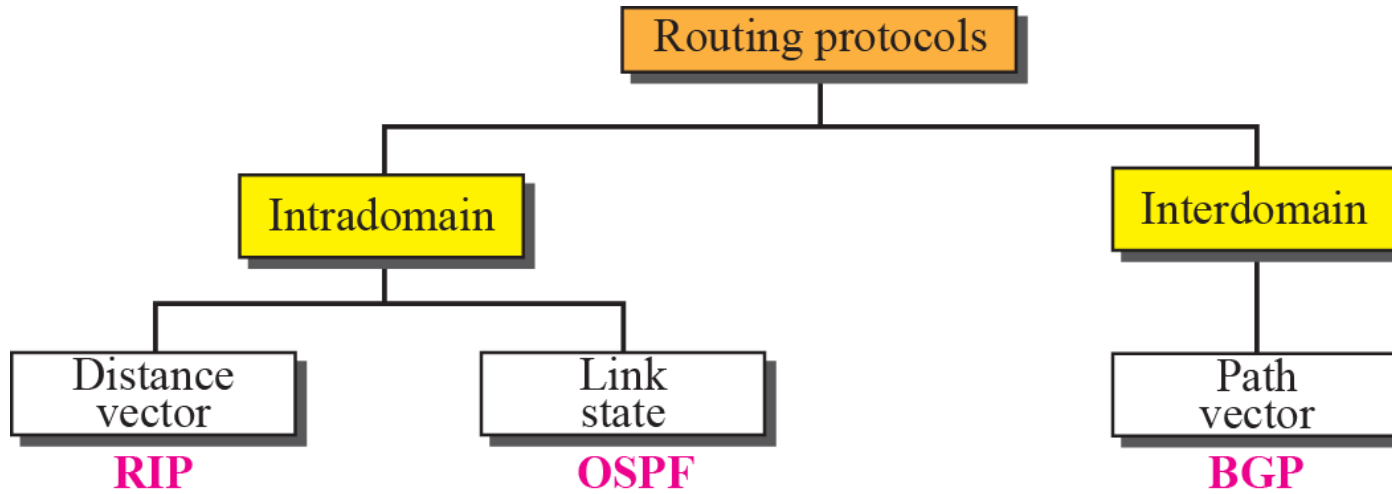


Figure 11.2 *Popular routing protocols*



11-3 DISTANCE VECTOR ROUTING

Today, an internet can be so large that one routing protocol cannot handle the task of updating the routing tables of all routers. For this reason, an internet is divided into autonomous systems. An autonomous system (AS) is a group of networks and routers under the authority of a single administration. Routing inside an autonomous system is called intra-domain routing. Routing between autonomous systems is called inter-domain routing

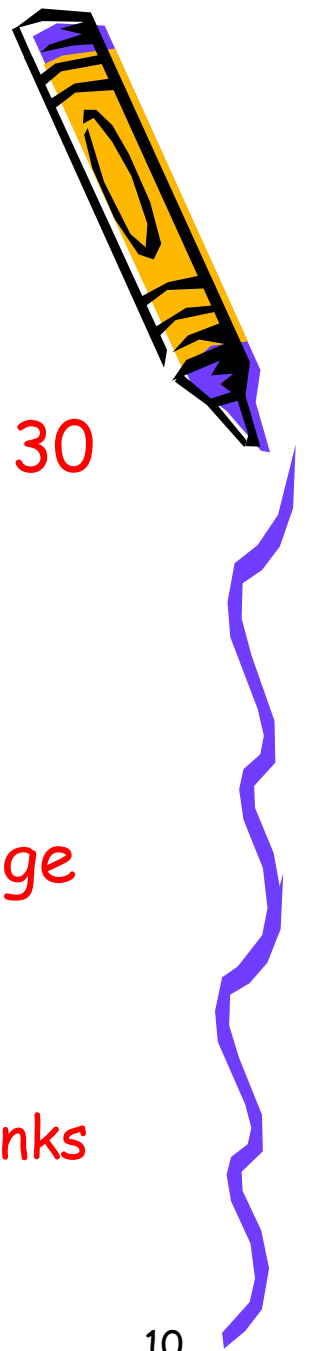
Updating Routing Table



- If the next-node entry is different
 - The receiving node chooses the row with the smaller cost
 - If there is a tie, the old one is kept
- If the next-node entry is the same
 - i.e. the sender of the new row is the provider of the old entry
 - The receiving node chooses the new row, even though the new value is infinity.



When to Share



- Periodic Update

- A node sends its routing table, normally 30 seconds, in a periodic update

- Triggered Update

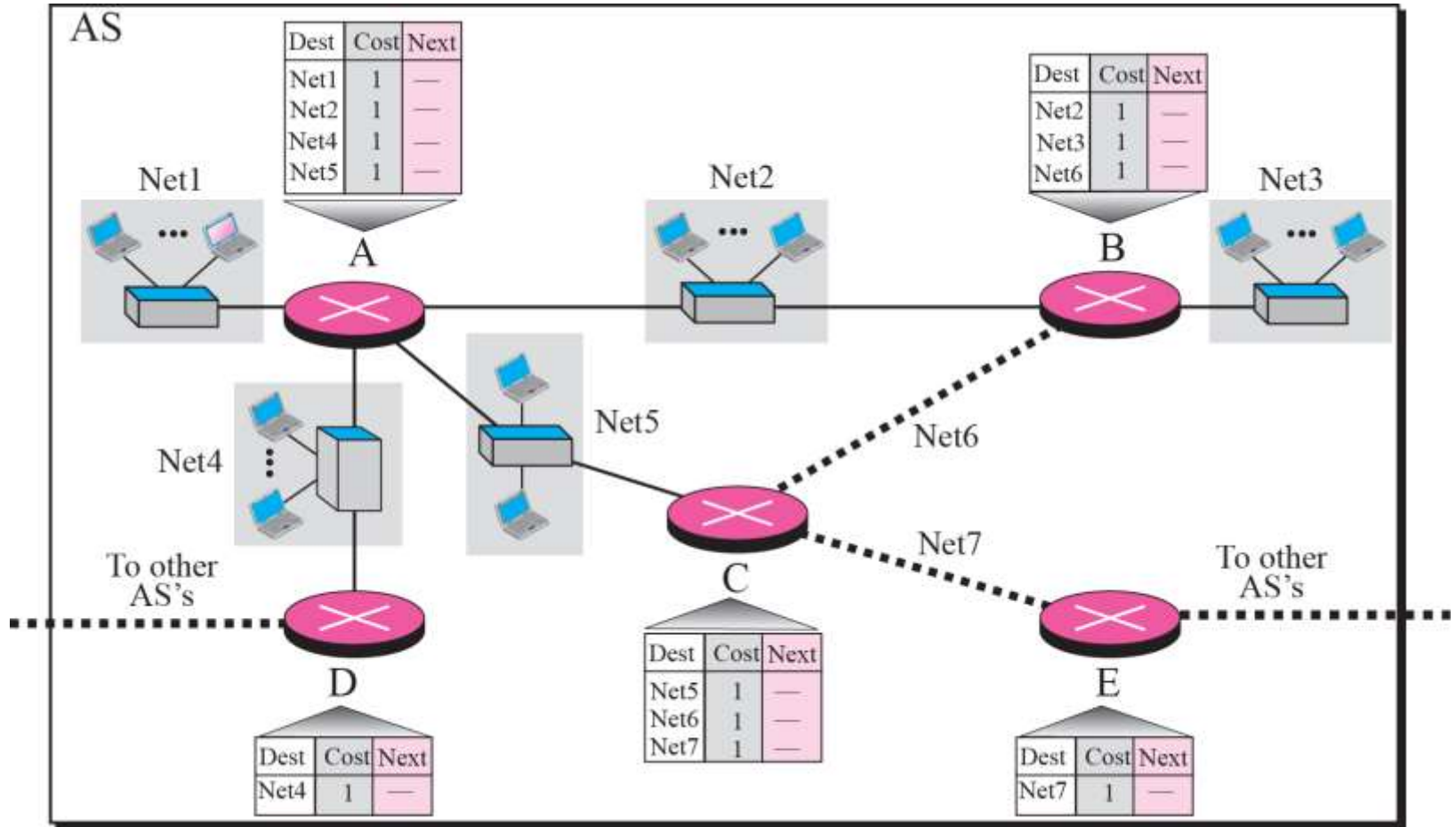
- A node sends its routing table to its neighbors any time when there is a change in its routing table
 - 1. After updating its routing table, or
 - 2. Detects some failure in the neighboring links



Example 11.1

Figure 11.5 shows the initial routing table for an AS. Note that the figure does not mean that all routing tables have been created at the same time; each router creates its own routing table when it is booted.

Figure 11.5 Example 11.1



Example 11.2

Now assume router A sends four records to its neighbors, routers B, D, and C. Figure 11.6 shows the changes in B's routing table when it receives these records. We leave the changes in the routing tables of other neighbors as exercise.

Figure 11.6 *Example 11.2*



Routing Table B

Dest	Cost	Next
Net1	2	A
Net2	1	—
Net3	1	—
Net6	1	—

After receiving record 1

Routing Table B

Dest	Cost	Next
Net1	2	A
Net2	1	—
Net3	1	—
Net6	1	—

After receiving record 2

Routing Table B

Dest	Cost	Next
Net1	2	A
Net2	1	—
Net3	1	—
Net4	2	A
Net6	1	—

After receiving record 3

Routing Table B

Dest	Cost	Next
Net1	2	A
Net2	1	—
Net3	1	—
Net4	2	A
Net5	2	A
Net6	1	—

After receiving record 4

Example 11.3

Figure 11.7 shows the final routing tables for routers in Figure 11.5.

Figure 11.7 *Example 11.3*

A

Dest	Cost	Next
Net1	1	—
Net2	1	—
Net3	2	B
Net4	1	—
Net5	1	—
Net6	2	C
Net7	2	C

B

Dest	Cost	Next
Net1	2	A
Net2	1	—
Net3	1	—
Net4	2	A
Net5	2	A
Net6	1	—
Net7	2	C

C

Dest	Cost	Next
Net1	2	A
Net2	2	A
Net3	2	B
Net4	2	A
Net5	1	—
Net6	1	—
Net7	1	—

D

Dest	Cost	Next
Net1	2	A
Net2	2	A
Net3	3	A
Net4	1	—
Net5	1	A
Net6	3	A
Net7	3	A

E

Dest	Cost	Next
Net1	3	C
Net2	3	C
Net3	3	C
Net4	3	C
Net5	2	C
Net6	2	C
Net7	1	—

Figure 11.8 *Two-node instability*

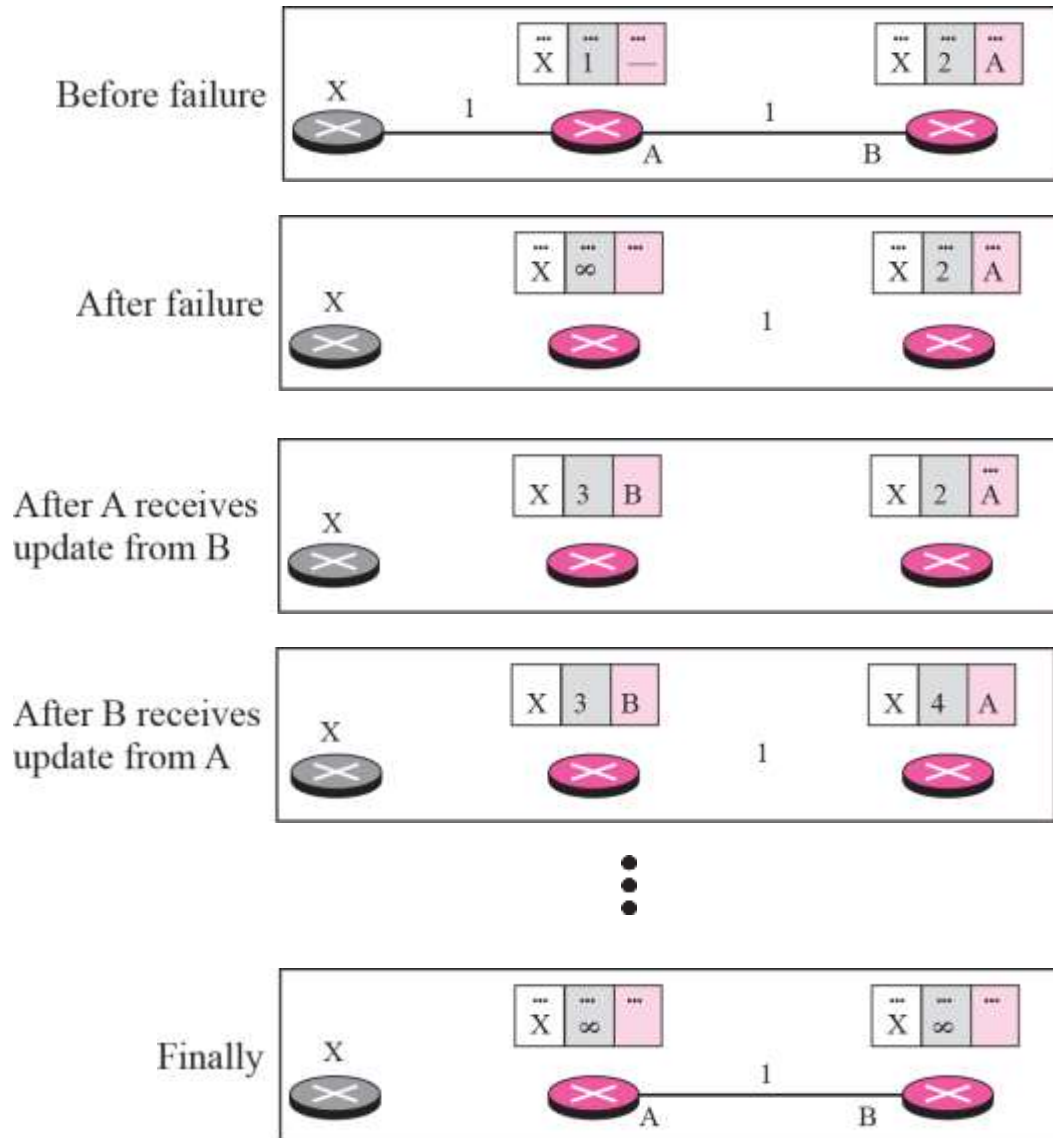




Figure 11.8 *Count to Infinity*

- A problem with distance vector routing is that any decrease in cost (good news) propagates quickly, but any increase in cost (bad news) propagates slowly.
- For a routing protocol to work properly, if a link is broken (cost becomes infinity), every other router should be aware of it immediately
- In distance vector routing, this takes some time. The problem is referred to as **count to infinity**.
- It takes several updates before the cost for a broken link is recorded as infinity by all routers.

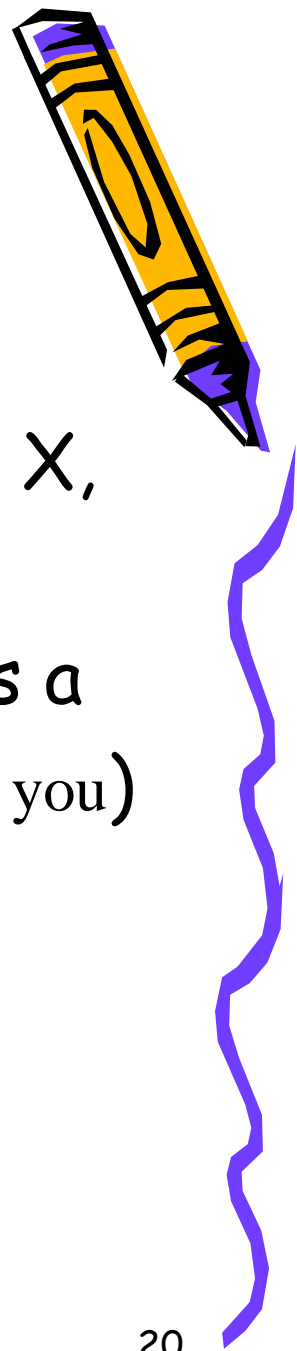
Two-Node Instability (1)



- Defining Infinity
 - Most implementations define 16 as infinity
- Split Horizon
 - Instead of flooding the table through each interface, each node sends only part of its table through each interface
 - E.g. node B thinks that the optimum route to reach X is via A, it does not need to advertise this piece of information to A



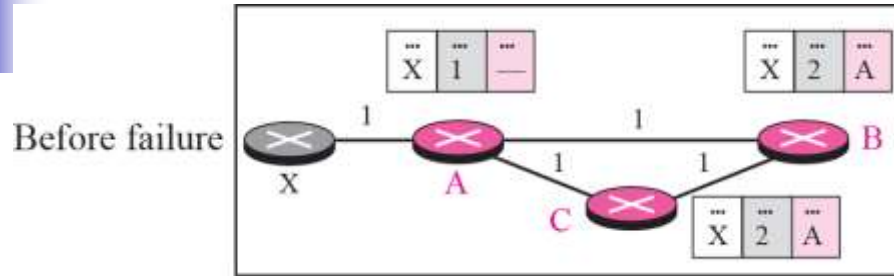
Two-Node Instability (2)



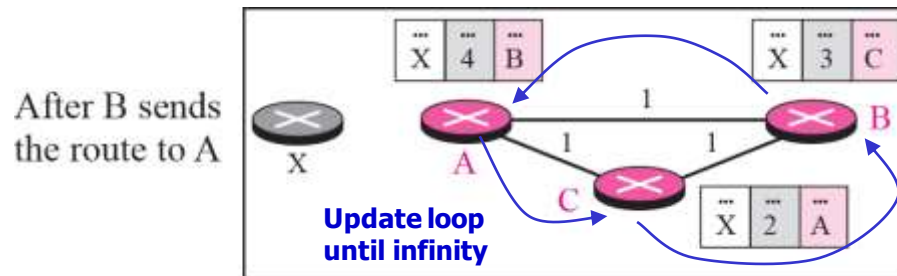
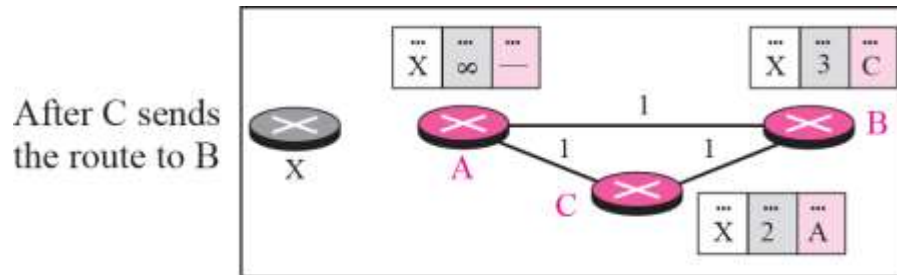
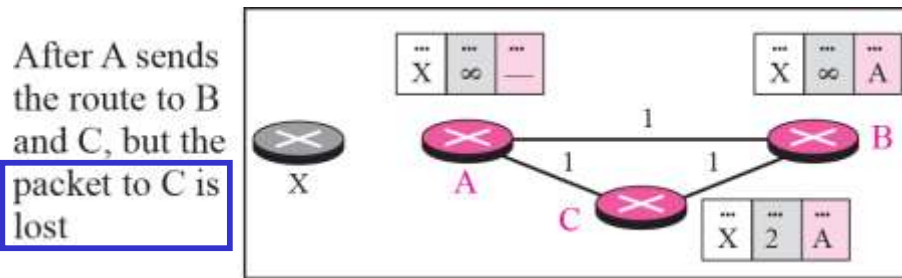
- **Poison Reverse**
- Node B can still advertise the value for X, but if the source of information is A, it can replace the distance with infinity as a warning (what I know about this route comes from you)



Figure 11.9 *Three-node instability*



If the instability is btw three nodes, stability cannot be guaranteed



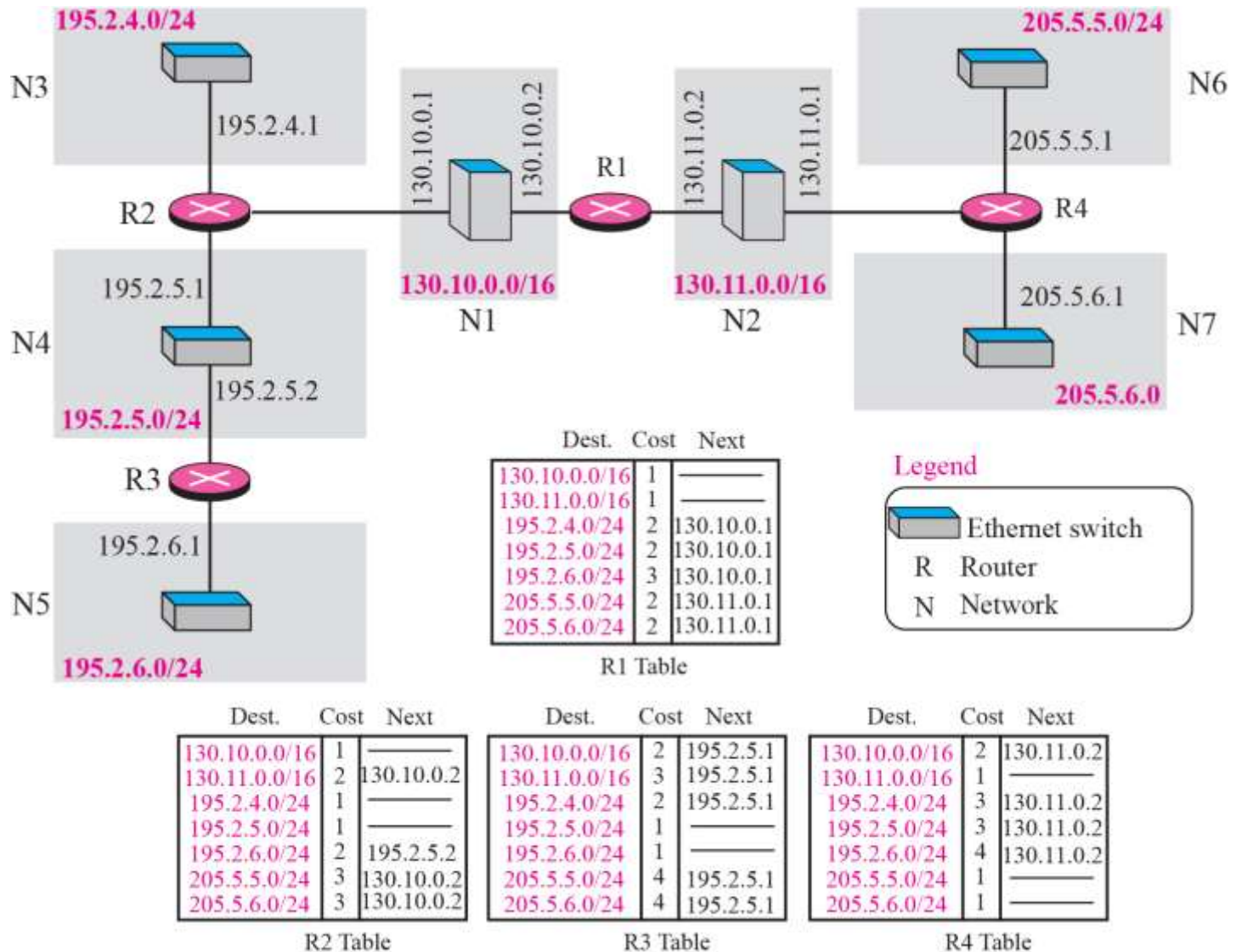
11-4 RIP

The Routing Information Protocol (RIP) is an intra-domain (interior) routing protocol used inside an autonomous system. It is a very simple protocol based on distance vector routing. RIP implements distance vector routing directly with some considerations.

RIP

- **RIP implements distance vector routing directly with some considerations:**
 - The destination in a routing table is a network, which means the first column defines a network address.
 - In RIP; the distance is defined as the number of links (networks) that have to be used to reach the destination. For this reason, the metric in RIP is called a **hop count**.
 - Infinity is defined as 16, which means that any route in an autonomous system using RIP cannot have more than 15 hops.
 - The next node column defines the address of the router to which the packet is to be sent to reach its destination.

Figure 11.10 *Example of a domain using RIP*



RIP messages

- Request

- A request message is sent by a router that has just come up or by a router that has some time-out entries
- A request can ask about specific entries or all entries

- Response

- A response can be either solicited (based on request) or unsolicited (30s or when there is a change in the routing table)



RIPv2 vs. RIPv1



- Classless Addressing
- Authentication
- Multicasting
 - RIPv1 uses broadcasting to send RIP messages to every neighbors. Routers as well as hosts receive the packets
 - RIPv2 uses the all-router multicast address to send the RIP messages only to RIP routers in the network

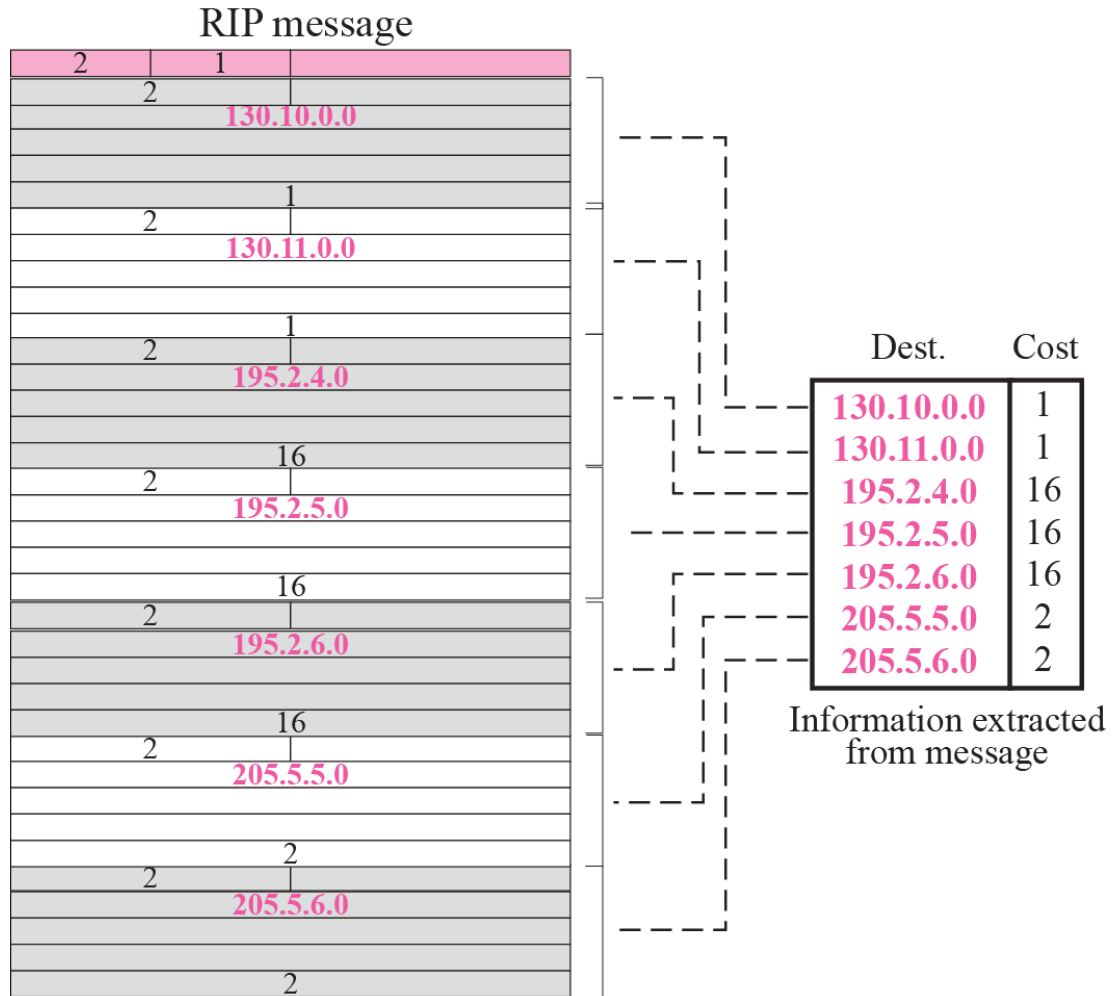


Example 11.4

Figure 11.13 shows the update message sent from router R1 to router R2 in Figure 11.10. The message is sent out of interface 130.10.0.2.

The message is prepared with the combination of split horizon and poison reverse strategy in mind. Router R1 has obtained information about networks 195.2.4.0, 195.2.5.0, and 195.2.6.0 from router R2. When R1 sends an update message to R2, it replaces the actual value of the hop counts for these three networks with 16 (infinity) to prevent any confusion for R2. The figure also shows the table extracted from the message. Router R2 uses the source address of the IP datagram carrying the RIP message from R1 (130.10.0.2) as the next hop address. Router R2 also increments each hop count by 1 because the values in the message are relative to R1, not R2.

Figure 11.13 *Solution to Example 11.4*



Example 11.5

A routing table has 20 entries. It does not receive information about five routes for 200 s. How many timers are running at this time?

Solution

The 21 timers are listed below:

Periodic timer: 1

Expiration timer: $20 - 5 = 15$

Garbage collection timer: 5



Note

RIP uses the services of UDP on well-known port 520.

11-5 LINK STATE ROUTING

Link state routing has a different philosophy from that of distance vector routing. In link state routing, if each node in the domain has the entire topology of the domain—the list of nodes and links, how they are connected including the type, cost (metric), and the condition of the links (up or down)—the node can use the Dijkstra algorithm to build a routing table.

Topics Discussed in the Section

- ✓ **Building Routing tables**

Figure 11.17 *Concept of Link state routing*

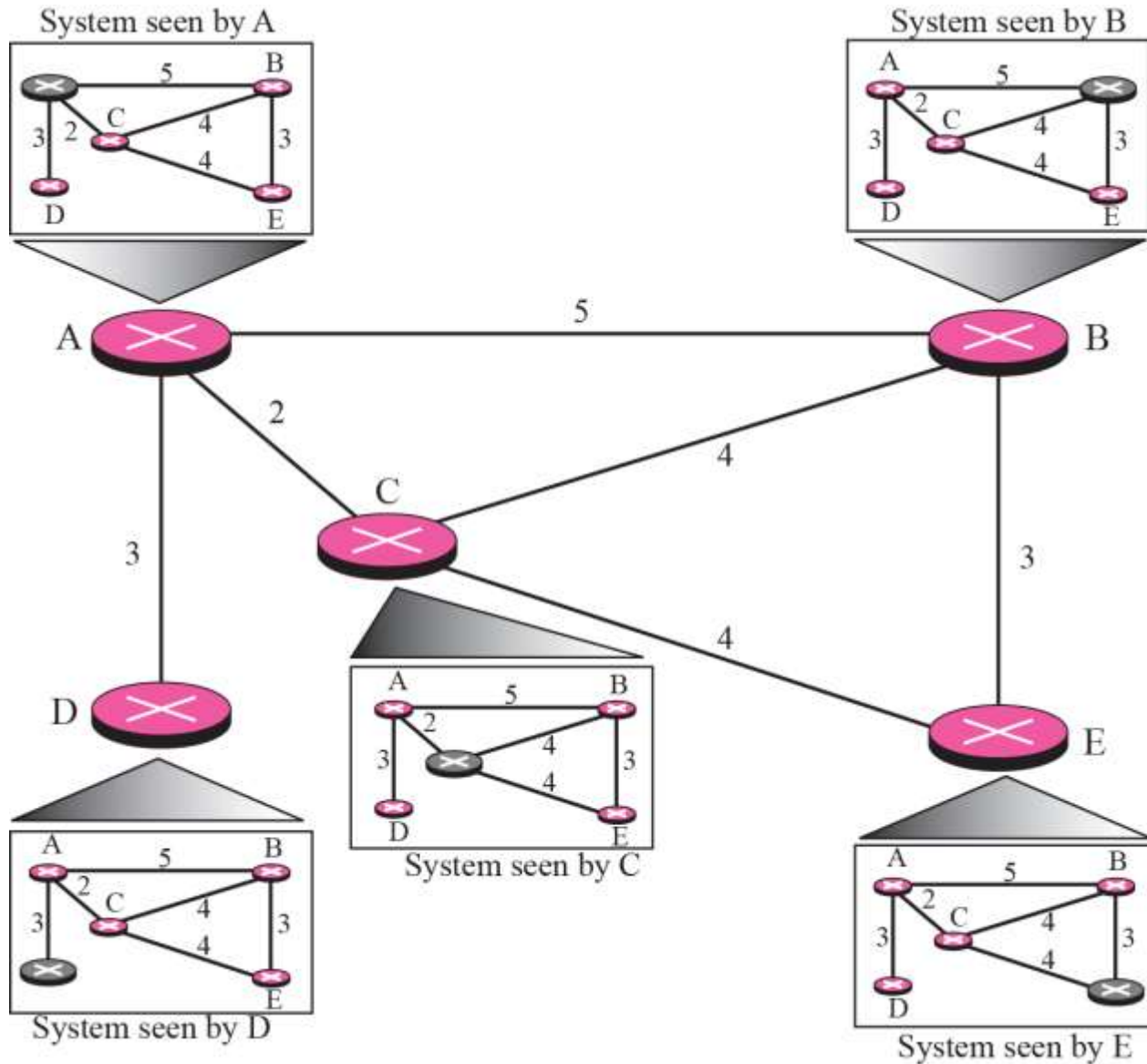
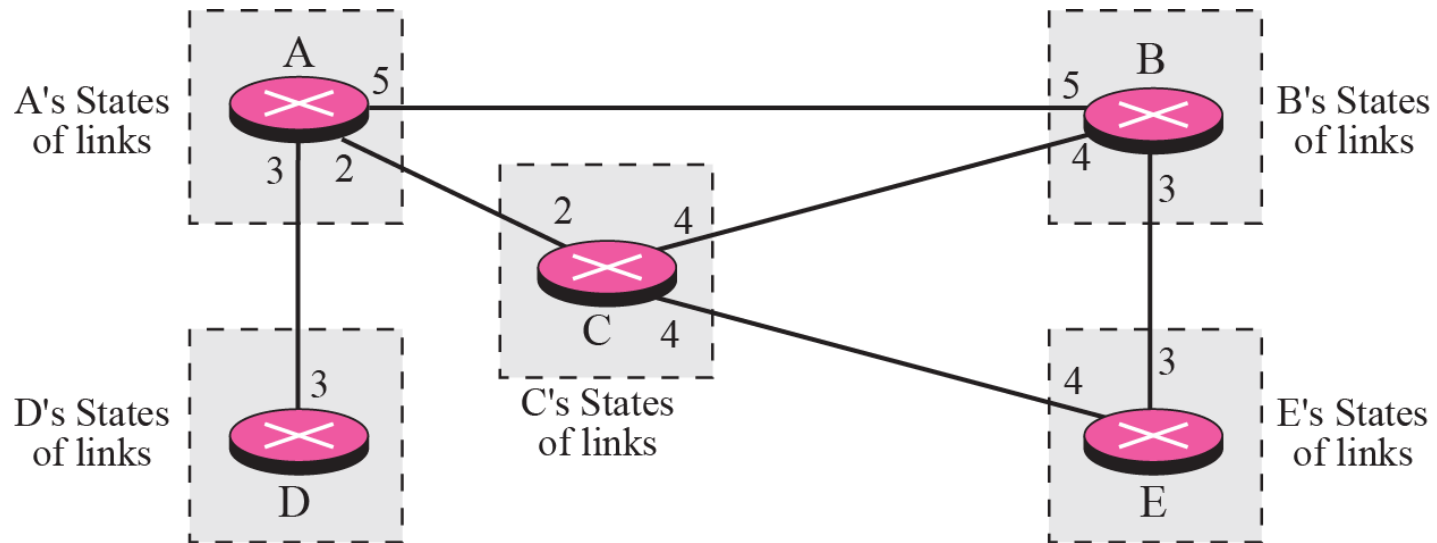
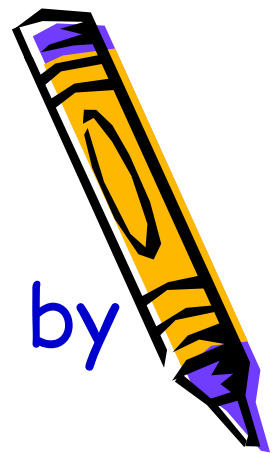


Figure 11.18 *Link state knowledge*



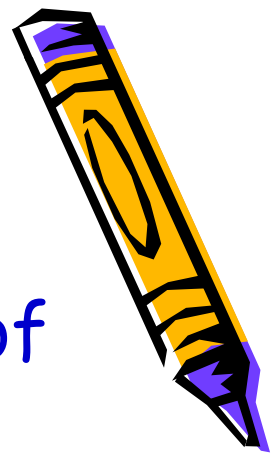
Building Routing Tables



- Creation of the states of the links by each node, called the **link state packets (LSP)**
- Dissemination of LSPs to every other routers, called **flooding** (efficiently)
- Formation of a **shortest path tree** for each node
- Calculation of a **routing table** based on the shortest path tree



Creation of LSP



- **LSP data:** E.g. the node ID, the list of links, a sequence number, and age.
- **LSP Generation**
 - When there is a change in the topology of the domain
 - On a periodic basis
 - There is no actual need for this type of LSP, normally 60 minutes or 2 hours



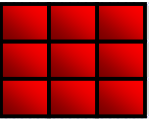
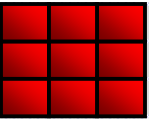


Table 11.3 *Dijkstra's Algorithm*

```
1 Dijkstra ( )
2 {
3     // Initialization
4     Path = {s}           // s means self
5     for (i = 1 to N)
6     {
7         if (i is a neighbor of s and i ≠ s)     $D_i = c_{si}$ 
8         if (i is not a neighbor of s)          $D_i = \infty$ 
9     }
10     $D_s = 0$ 
11
12 } // Dijkstra
```



Continued

```
13 // Iteration
14 Repeat
15 {
16     // Finding the next node to be added
17     Path = Path  $\cup$   $i$    if  $D_i$  is minimum among all remaining nodes
18
19     // Update the shortest distance for the rest
20     for ( $j = 1$  to  $M$ )    //  $M$  number of remaining nodes
21     {
22          $D_j = \text{minimum} (D_j , D_j + c_{ij})$ 
23     }
24 } until (all nodes included in the path,  $M = 0$ )
25
```

Figure 11.19 *Forming shortest path three for router A in a graph*

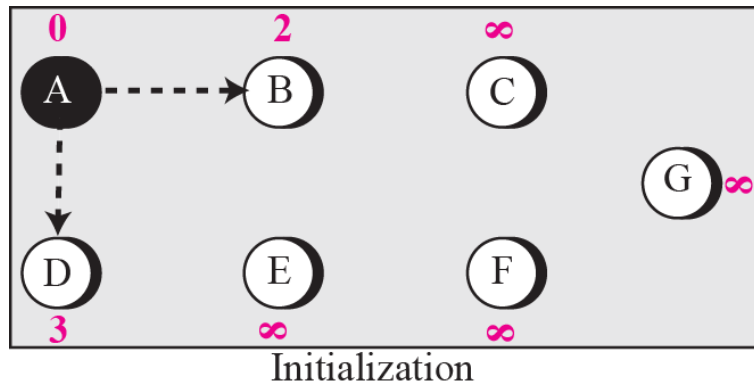
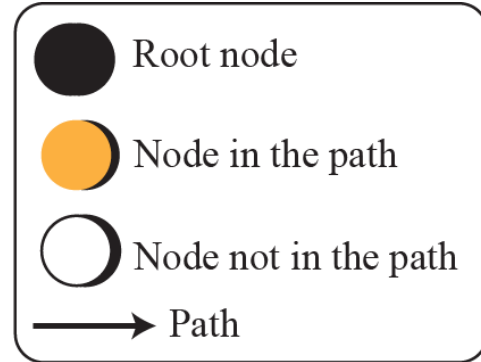
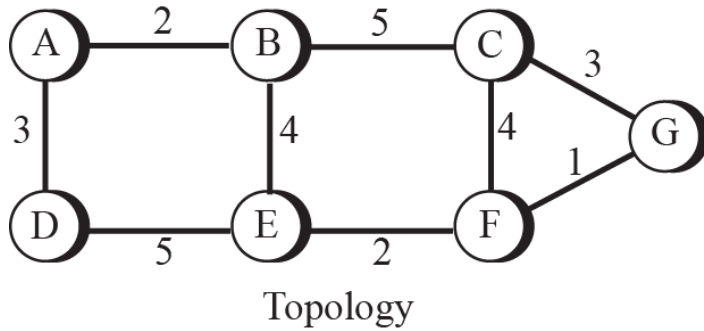


Figure 11.19 *Continued*

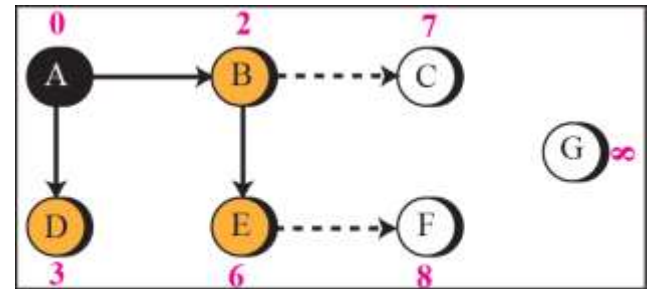
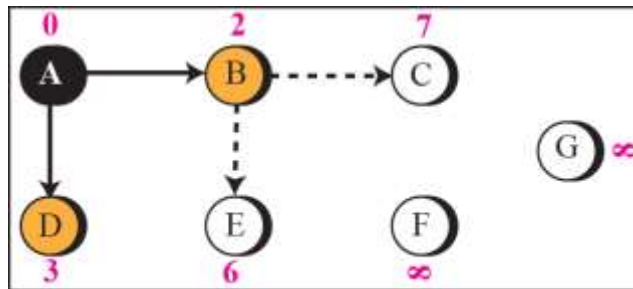
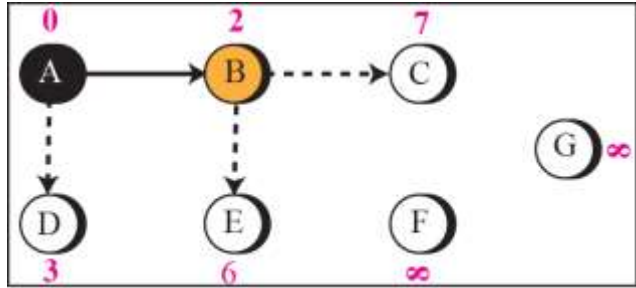
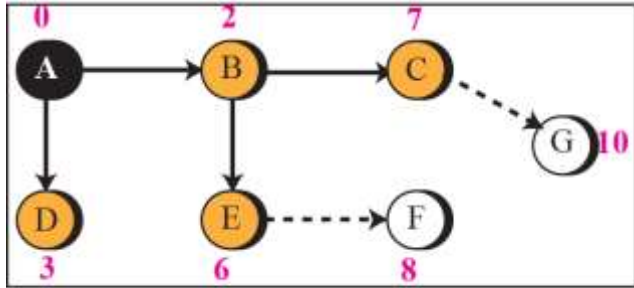
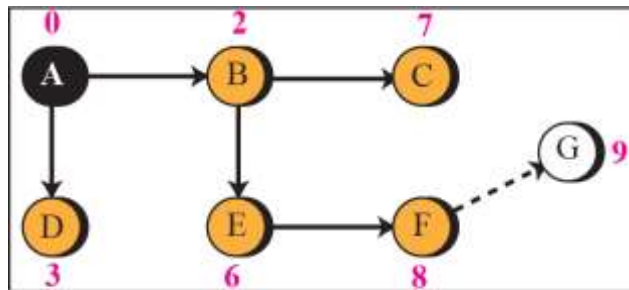


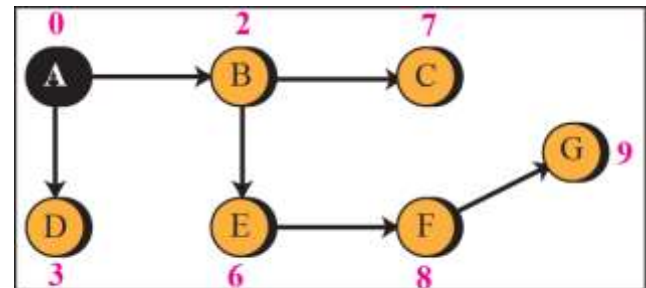
Figure 11.19 *Continued*



Iteration 4



Iteration 5

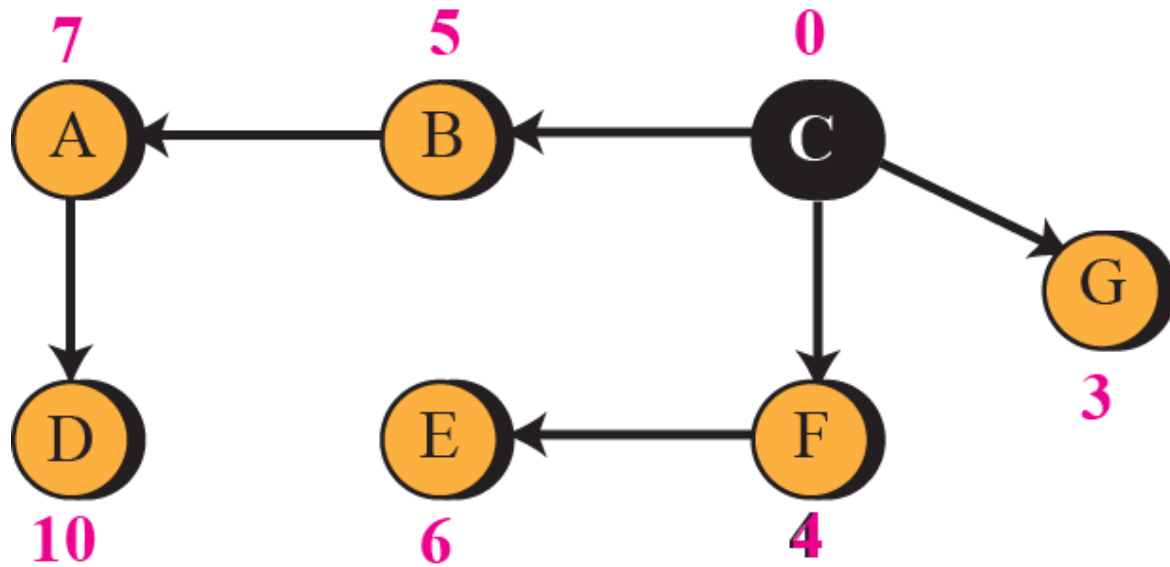


Iteration 6

Example 11.6

To show that the shortest path tree for each node is different, we found the shortest path tree as seen by node C (Figure 11.20). We leave the detail as an exercise.

Figure 11.20 Example 11.6



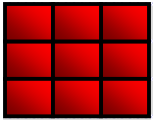


Table 11.4 *Routing Table for Node A*

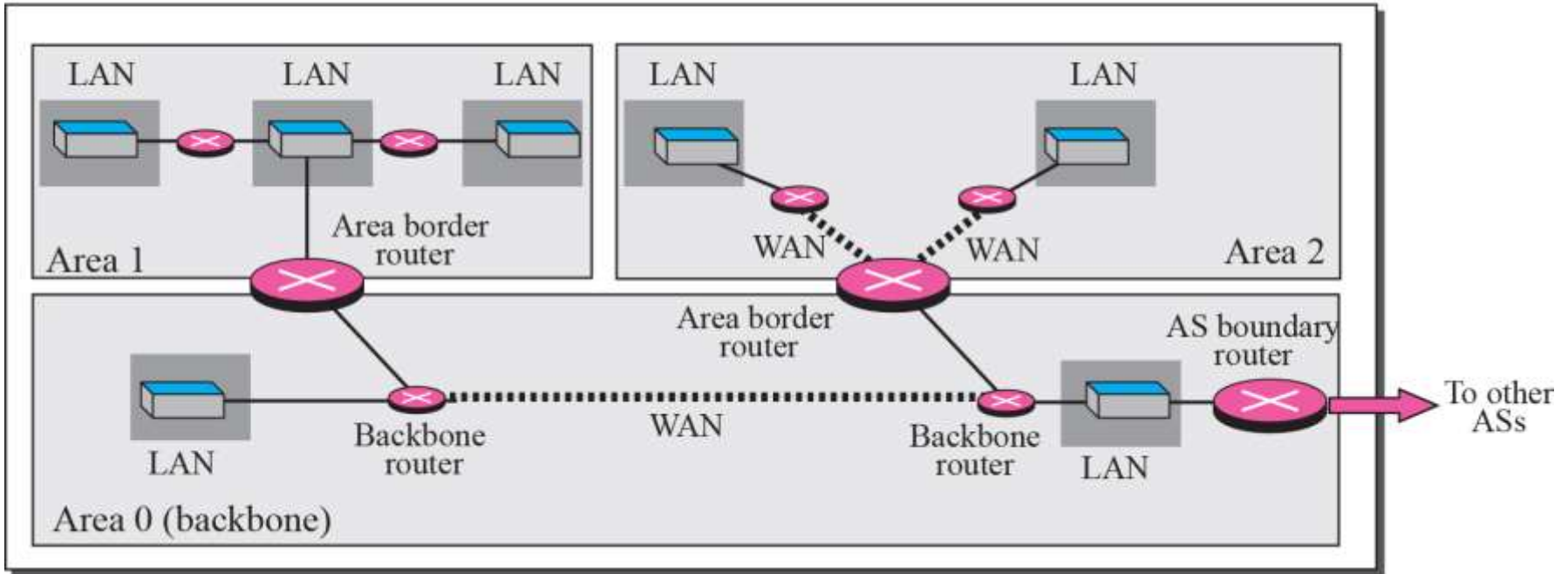
<i>Destination</i>	<i>Cost</i>	<i>Next Router</i>
A	0	—
B	2	—
C	7	B
D	3	—
E	6	B
F	8	B
G	9	B

11-6 OSPF

The Open Shortest Path First (OSPF) protocol is an intra-domain routing protocol based on link state routing. Its domain is also an autonomous system.

Figure 11.21 *Areas in an autonomous system*

Autonomous System (AS)



Area in OSPF (1)



- A collection of networks with **area ID**
- Routers inside an area **flood** the area with routing information
- **Area border routers** summarize the information about the area and send it to other areas
- **Backbone area and backbone routers**
 - All of the area inside an AS must be connected to the backbone



Area in OSPF (2)



- **Virtual link**

- If, because of some problem, the connectivity between a backbone and an area is broken, a **virtual link** between routers must be created by the administration to allow continuity of the functions of the backbone as the primary area



LSA General Header (3)



- Advertising router
 - The IP address of the router advertising this message
- Link state sequence number
 - A sequence number assigned to each link state update message



11-7 PATH VECTOR ROUTING

Distance vector and link state routing are both interior routing protocols. They can be used inside an autonomous system. Both of these routing protocols become intractable when the domain of operation becomes large. Distance vector routing is subject to instability if there is more than a few hops in the domain of operation. Link state routing needs a huge amount of resources to calculate routing tables. It also creates heavy traffic because of flooding. There is a need for a third routing protocol which we call path vector routing.

Topics Discussed in the Section

- ✓ **Reachability**
- ✓ **Routing Table**

Example 11.10

The difference between the distance vector routing and path vector routing can be compared to the difference between a national map and an international map. A national map can tell us the road to each city and the distance to be traveled if we choose a particular route; an international map can tell us which cities exist in each country and which countries should be passed before reaching that city.

Figure 11.50 *Reachability*

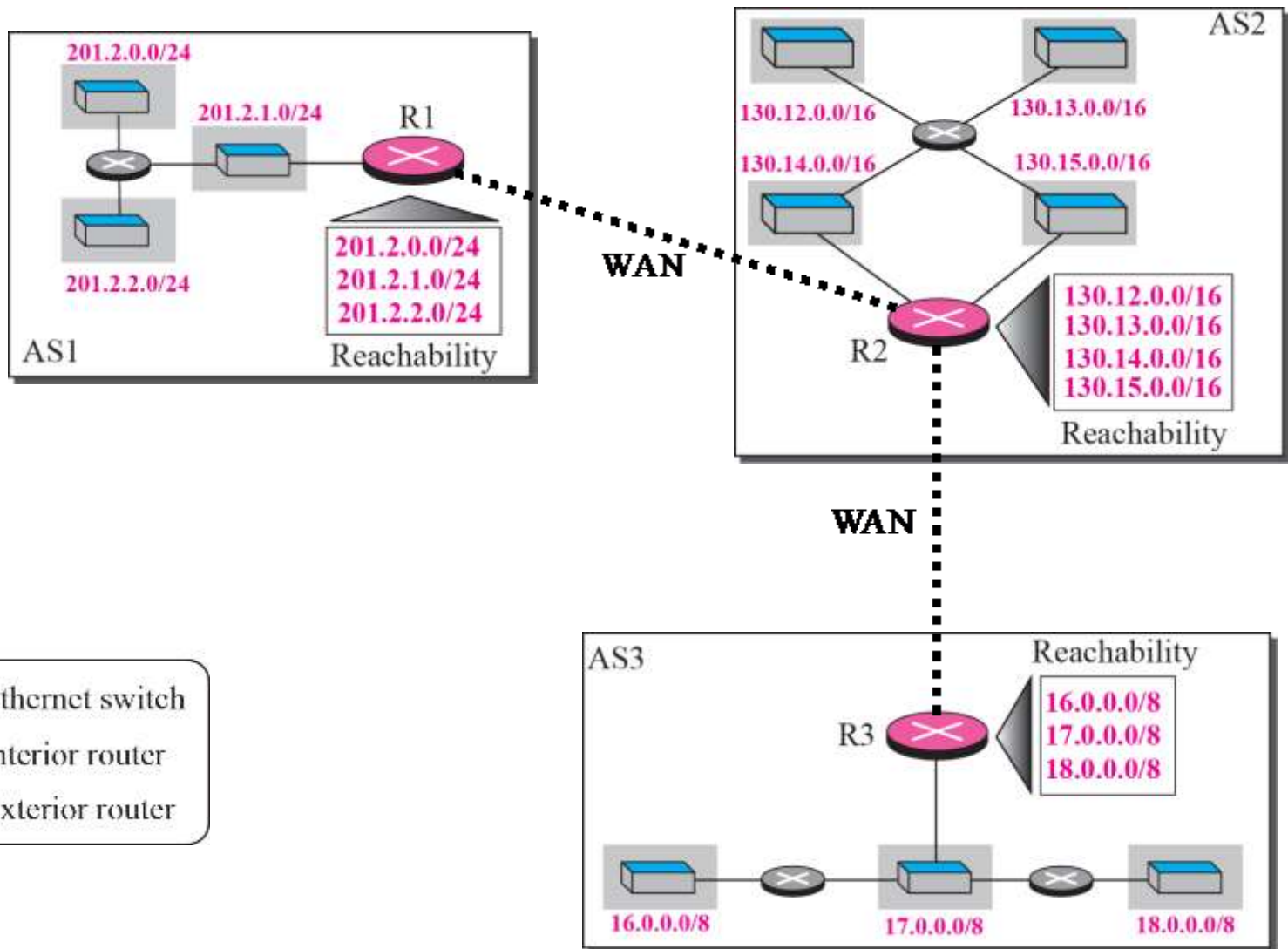


Figure 11.51 *Stabilized table for three autonomous system*



Network	Path
201.2.0.0/24	AS1 (This AS)
201.2.1.0/24	AS1 (This AS)
201.2.2.0/24	AS1 (This AS)
130.12.0.0/16	AS1, AS2
130.13.0.0/16	AS1, AS2
130.14.0.0/16	AS1, AS2
130.15.0.0/16	AS1, AS2
16.0.0.0/8	AS1, AS2, AS3
17.0.0.0/8	AS1, AS2, AS3
18.0.0.0/8	AS1, AS2, AS3

Path-Vector Routing Table



Network	Path
201.2.0.0/24	AS2, AS1
201.2.1.0/24	AS2, AS1
201.2.2.0/24	AS2, AS1
130.12.0.0/16	AS2 (This AS)
130.13.0.0/16	AS2 (This AS)
130.14.0.0/16	AS2 (This AS)
130.15.0.0/16	AS2 (This AS)
16.0.0.0/8	AS2, AS3
17.0.0.0/8	AS2, AS3
18.0.0.0/8	AS2, AS3

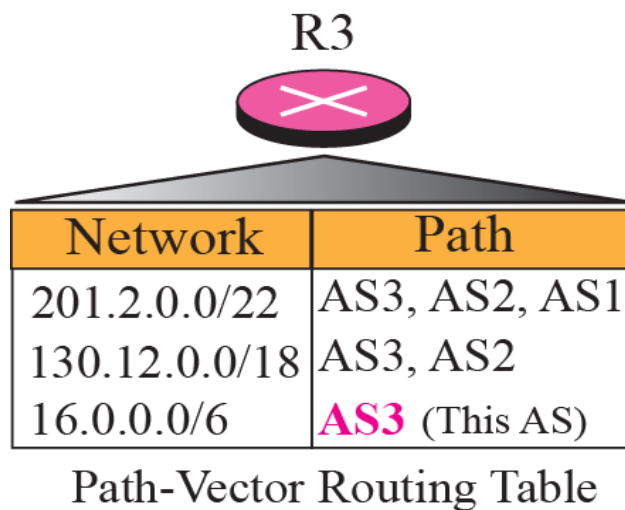
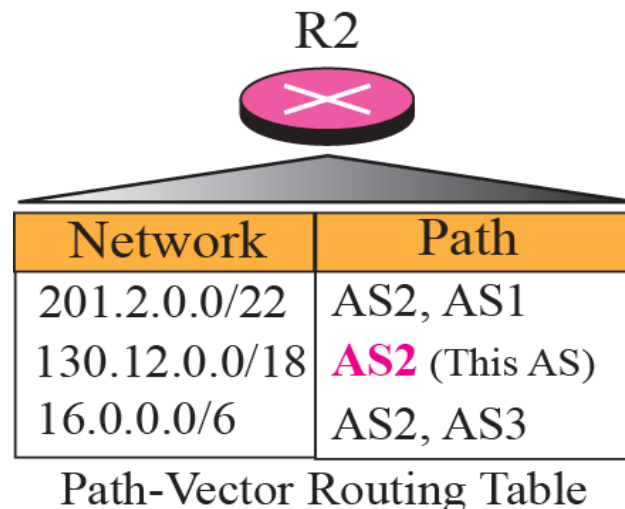
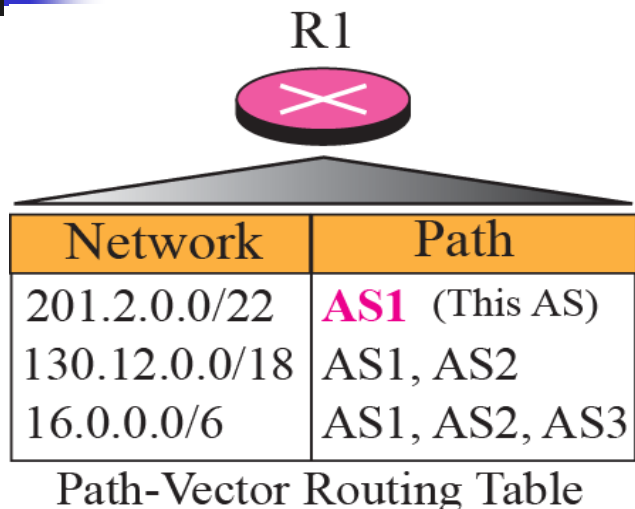
Path-Vector Routing Table



Network	Path
201.2.0.0/24	AS3, AS2, AS1
201.2.1.0/24	AS3, AS2, AS1
201.2.2.0/24	AS3, AS2, AS1
130.12.0.0/16	AS3, AS2
130.13.0.0/16	AS3, AS2
130.14.0.0/16	AS3, AS2
130.15.0.0/16	AS3, AS2
16.0.0.0/8	AS3 (This AS)
17.0.0.0/8	AS3 (This AS)
18.0.0.0/8	AS3 (This AS)

Path-Vector Routing Table

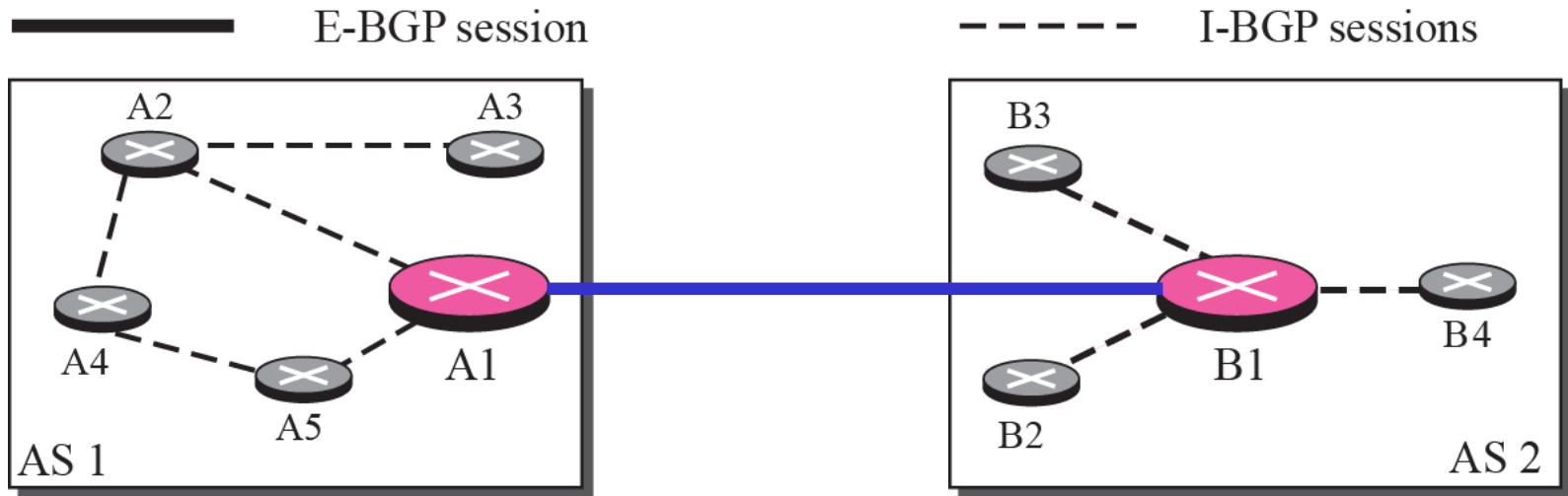
Figure 11.52 *Routing tables after aggregation*



11-8 BGP

Border Gateway Protocol (BGP) is an interdomain routing protocol using path vector routing. It first appeared in 1989 and has gone through four versions.

Figure 11.53 *Internal and external BGP sessions*



A speaker node advertises the path, not the metric of the nodes, in its AS or other ASs.

Path Vector Routing (1)



- **Sharing**
 - A speaker in an AS shares its table with immediate neighbors
- **Updating**
 - Adding the nodes that are not in its routing table and adding its own AS and the AS that sent the table
 - The routing table shows the path completely



Path Vector Routing (2)



- Loop prevention

- A route checks to see if its AS is in the path list to the destination

- Policy routing

- If one of the ASs listed in the path is against its policy, it can ignore that path and that destination
- It does not update its routing table with the path, and it does not send this message to its neighbors



Chapter 12

Multicasting And Multicast Routing Protocols

Presented by:

Dr. Mohammad Alhammouri

OBJECTIVES:

- ❑ To compare and contrast unicasting, multicasting, and broadcasting communication.**
- ❑ To define multicast addressing space in IPv4 and show the division of the space into several blocks.**
- ❑ To discuss the IGMP protocol, which is responsible for collecting group membership information in a network.**
- ❑ To discuss the general idea behind multicast routing protocols and their division into two categories based on the creation of the shortest path trees.**
- ❑ To discuss multicast link state routing in general and its implementation in the Internet: a protocol named MOSPF.**

Chapter Outline

12.1 Introduction

12.2 Multicast Addresses

12.3 IGMP

12.4 Multicast Routing

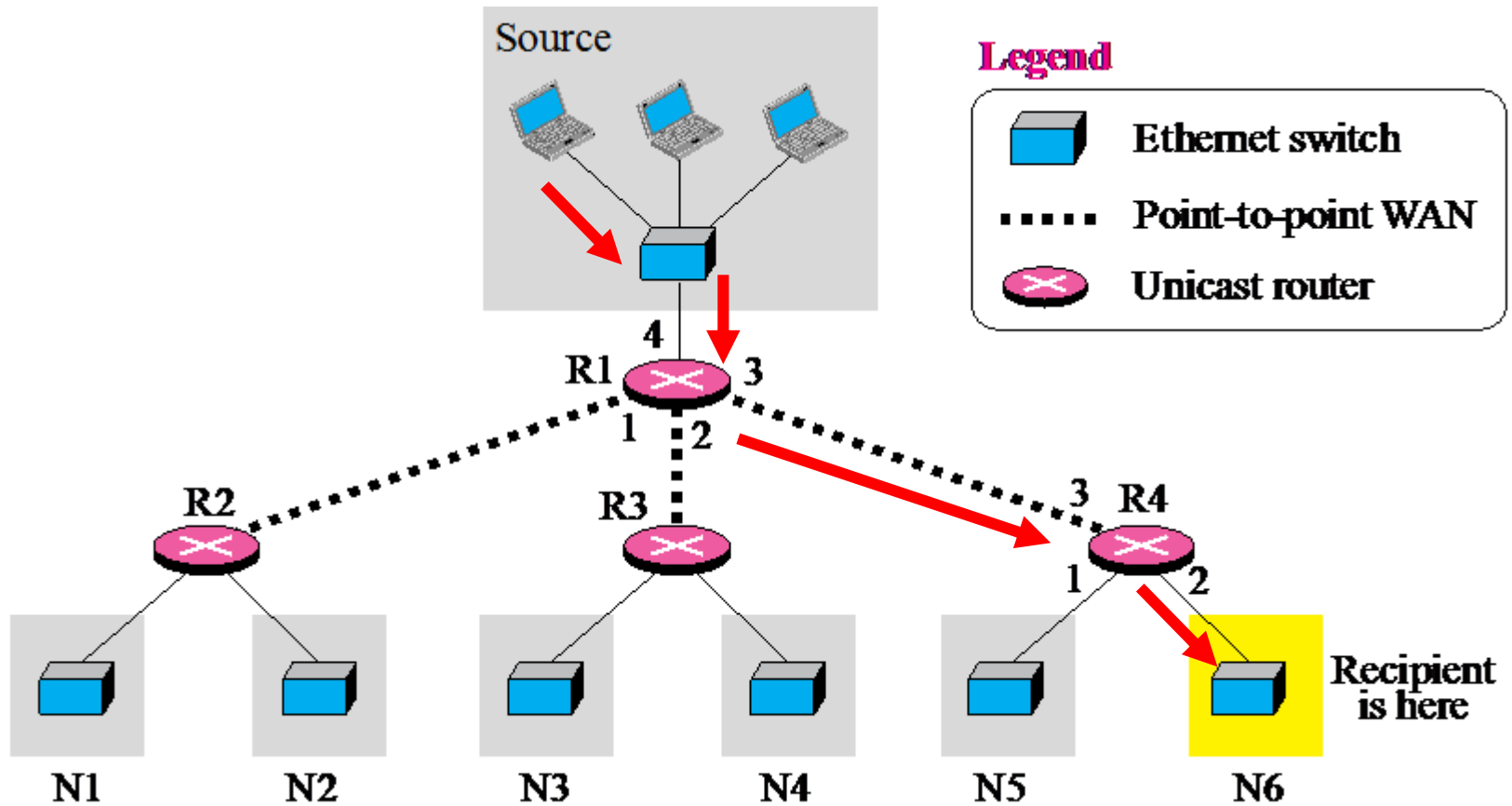
12.5 Routing Protocols

12.6 MBONE

Topics Discussed in the Section

- ✓ **Unicasting**
- ✓ **Multicasting**
- ✓ **Broadcasting**

Figure 12.1 *Unicasting*

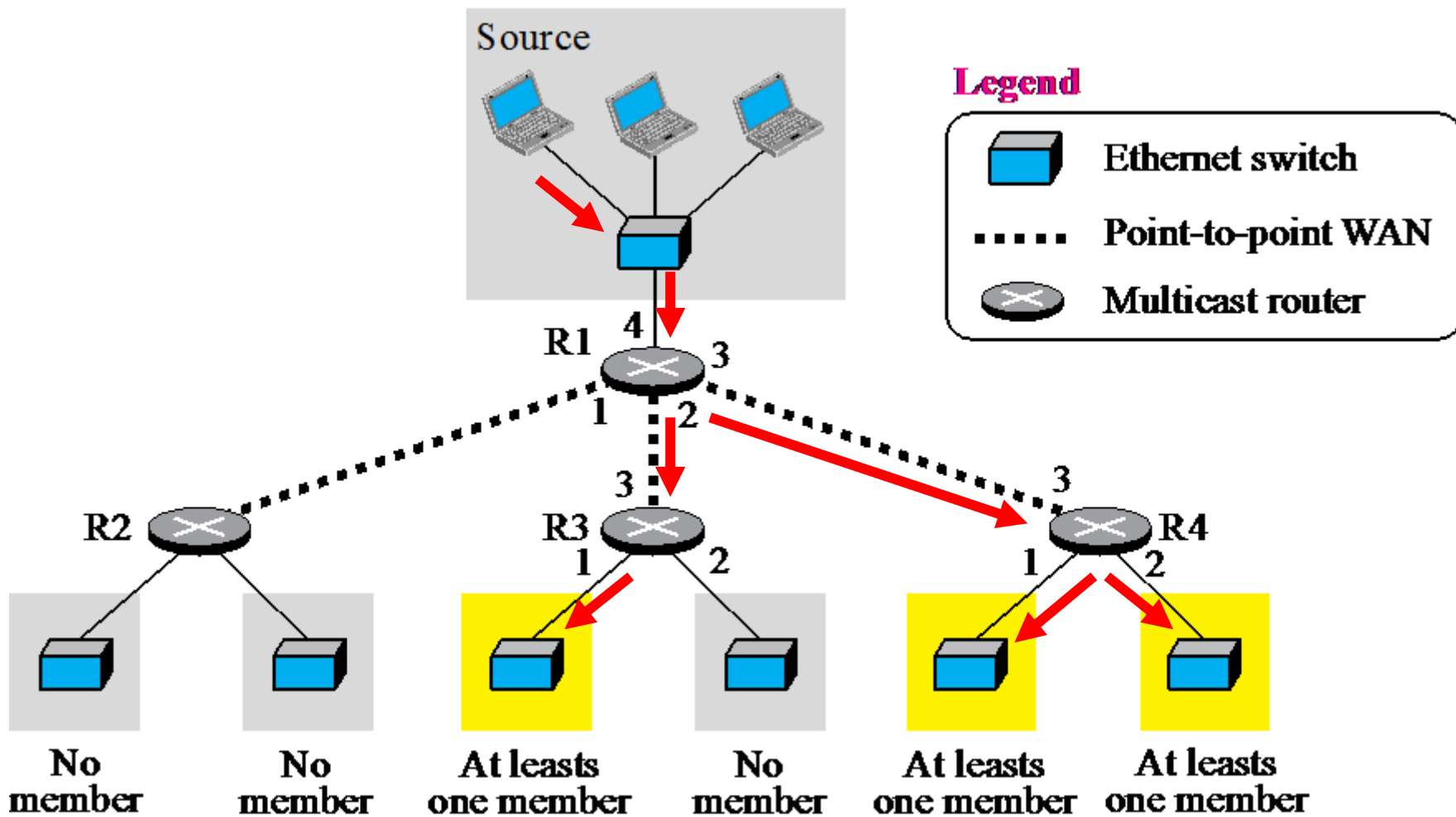




Note

In unicasting, the router forwards the received datagram through only one of its interfaces.

Figure 12.2 Multicasting

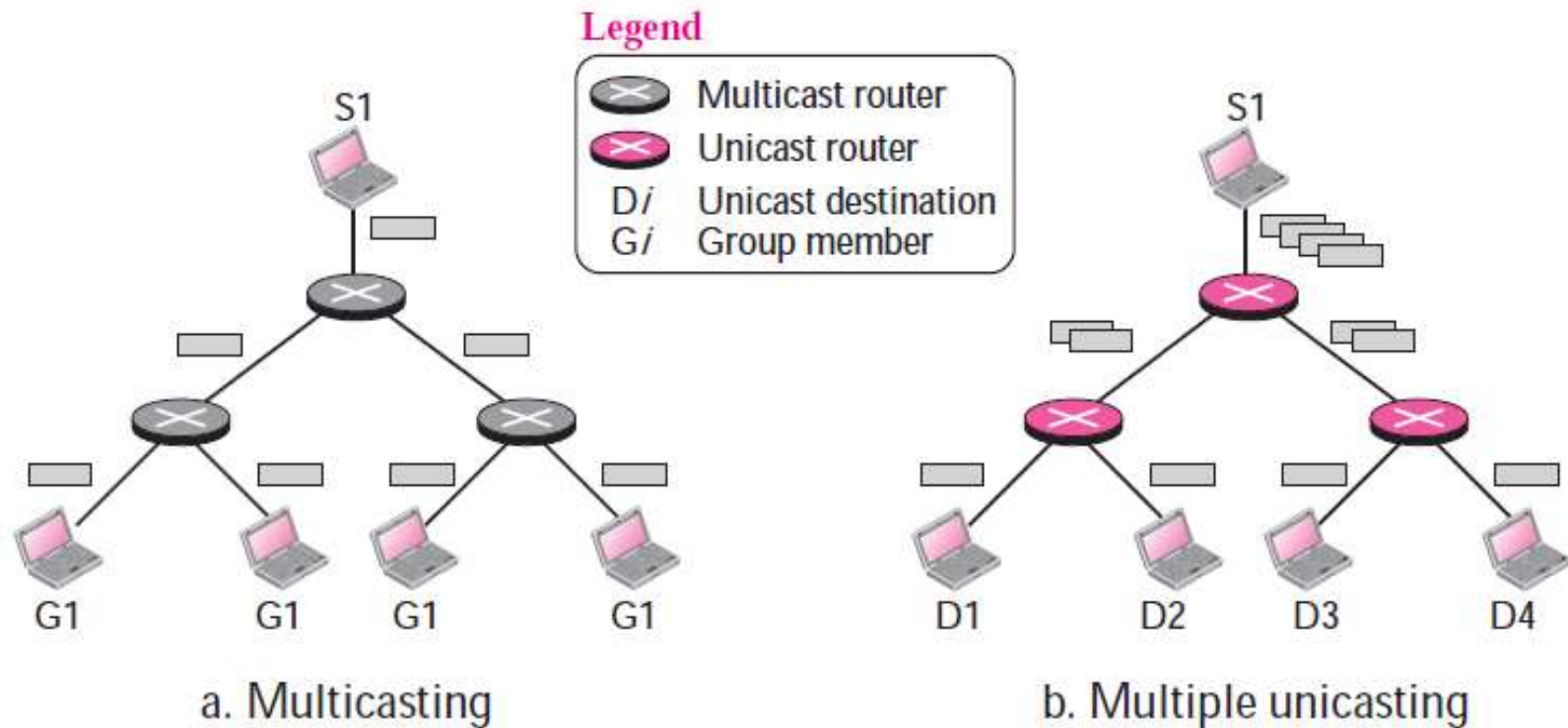




Note

In multicasting, the router may forward the received datagram through several of its interfaces.

Figure 12.3 *Multicasting versus multiple unicasting*



Multicasting starts with one single packet from the source that is duplicated by the routers. The **destination address** in each packet is the same for all duplicates. Only **one single copy** of the packet travels between any two routers



Note

Emulation of multicasting through multiple unicasting is not efficient and may create long delays, particularly with a large group.

12-2 MULTICAST ADDRESSES

A multicast address is a destination address for a group of hosts that have joined a multicast group. A packet that uses a multicast address as a destination can reach all members of the group unless there are some filtering restriction by the receiver.

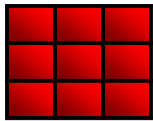


Table 12.1 *Multicast Address Ranges*

<i>CIDR</i>	<i>Range</i>	<i>Assignment</i>
224.0.0.0/24	224.0.0.0 → 224.0.0.255	Local Network Control Block
224.0.1.0/24	224.0.1.0 → 224.0.1.255	Internetwork Control Block
	224.0.2.0 → 224.0.255.255	AD HOC Block
224.1.0.0/16	224.1.0.0 → 224.1.255.255	ST Multicast Group Block
224.2.0.0/16	224.2.0.0 → 224.2.255.255	SDP/SAP Block
	224.3.0.0 → 231.255.255.255	Reserved
232.0.0.0/8	232.0.0.0 → 224.255.255.255	Source Specific Multicast (SSM)
233.0.0.0/8	233.0.0.0 → 233.255.255.255	GLOP Block
	234.0.0.0 → 238.255.255.255	Reserved
239.0.0.0/8	239.0.0.0 → 239.255.255.255	Administratively Scoped Block

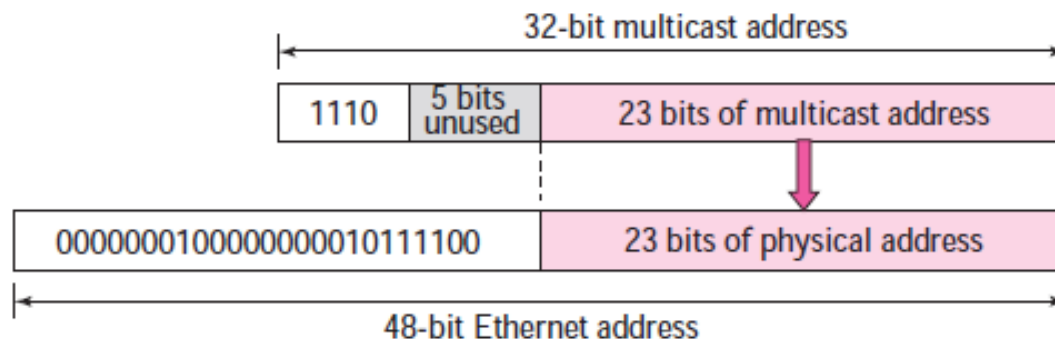
Delivery of Multicast Packets at Data Link Layer

- ** ARP** protocol cannot find the corresponding **MAC (physical)** address to forward the packet at the data link layer (because multicast IPs)
- ** Data link layer** supports physical multicast addresses
- ** LANs** support physical multicast addressing, **Ethernet** is one of them

Delivery of Multicast Packets at Data Link Layer

****** If the first 25 bits in an Ethernet address are **0000 0001 0000 0000 0101 1110 0**, this identifies a physical multicast address. The remaining 23 bits can be used to define a group

Figure 12.4 *Mapping class D to Ethernet physical address*



Example 12.2

Change the multicast IP address 232.43.14.7 to an Ethernet multicast physical address.

Solution

a. We write the rightmost 23 bits of the IP address in hexadecimal. **(43.14.7) → 2B:0E:07**

then subtracting 8 from the leftmost digit if it is greater than or equal to 8 ($2 < 8$). In our example the result is 2B:0E:07.

b. We add the result of part a to the starting Ethernet multicast address, which is **01:00:5E:00:00:00**. The result is

01:00:5E:2B:0E:07

Example 12.3

Change the multicast IP address 238.212.24.9 to an Ethernet multicast address.

Solution

- a. The rightmost 3 bytes in hexadecimal are (212.24.9) → **D4:18:09**.

- b. We need to subtract 8 from the leftmost digit ($D - 8 = 5$), resulting in 54:18:09.
- b. We add the result of part a to the Ethernet multicast starting address. The result is

01:00:5E:54:18:09

Figure 12.5 *Tunneling*

When network does not support multicast, multicast packet encapsulated in unicast packet.

The destination router which support multicast processes the packet as multicast packet

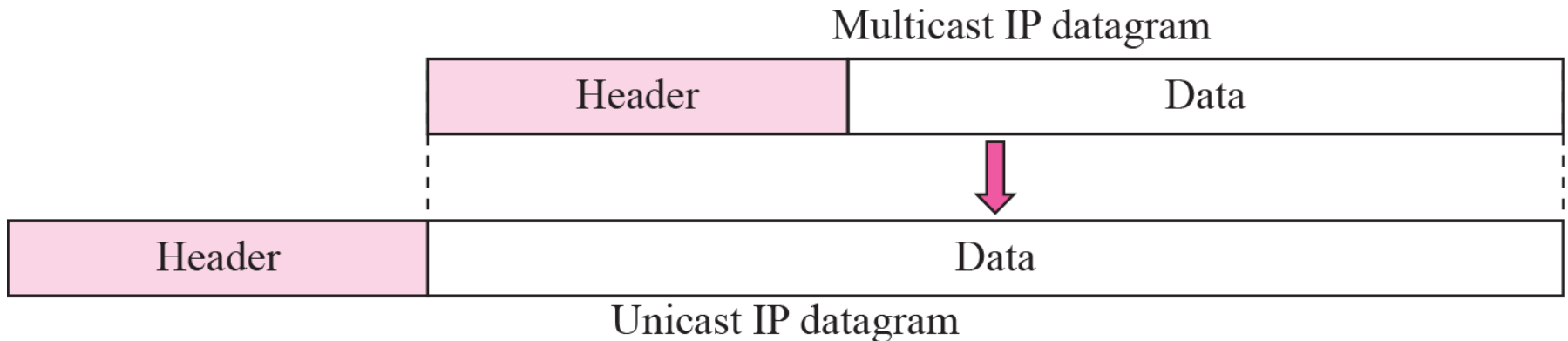
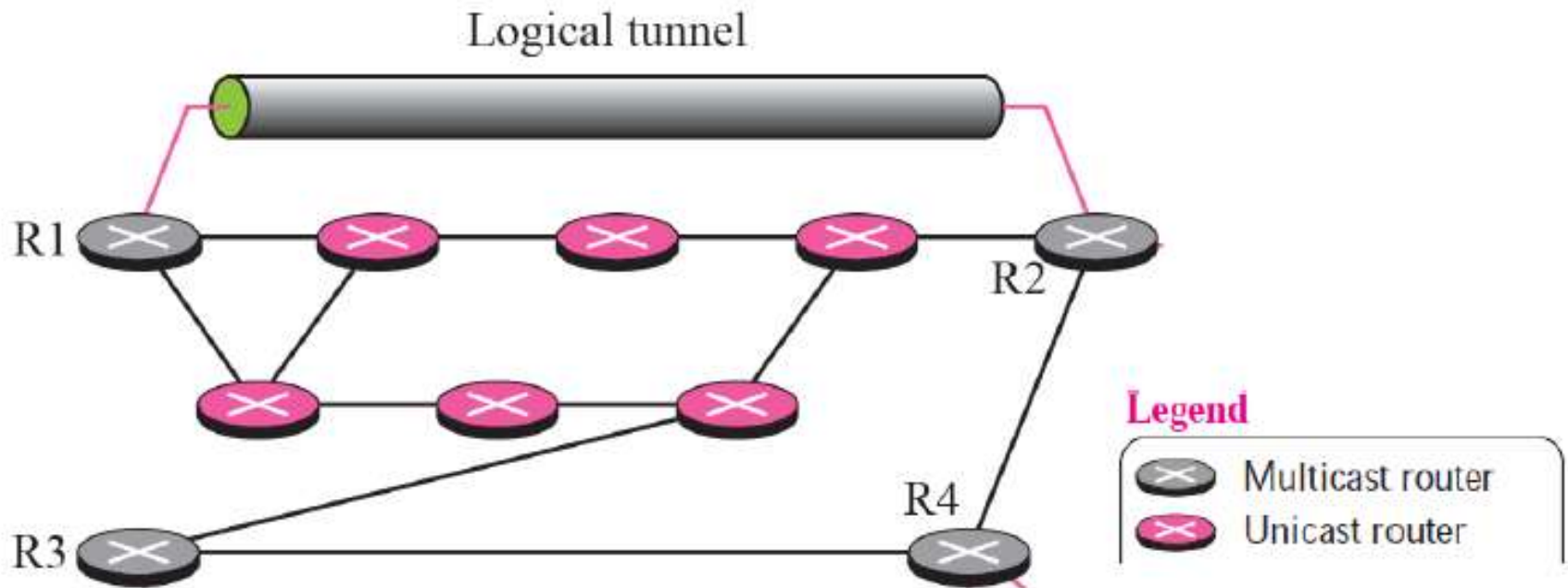


Figure 12.5 *Tunneling*

- Uses logical tunneling for multicasting between noncontiguous multicast routers



Internet Group Management Protocol (IGMP)

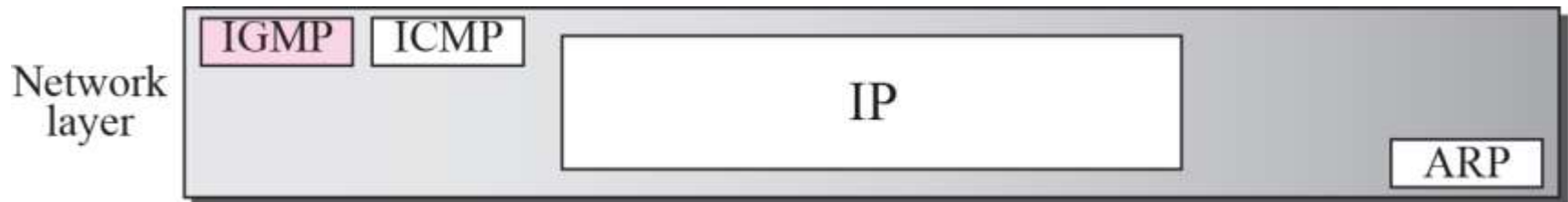
- Multicast communication means that a sender sends a message to a group of recipients that are members of the same group.
- Multicast routers need to collect information about members and share it with each other
- Information collected locally by multicast router connected to network (**IGMP protocol**)
- Collected information globally propagated to other routers (**multicast routing protocols**)

Topics Discussed in the Section

- ✓ **Group Management**
- ✓ **IGMP Messages**
- ✓ **IGMP Protocol Applied to host**
- ✓ **IGMP Protocol Applied to Router**
- ✓ **Role of IGMP in Forwarding**
- ✓ **Variables and Timers**
- ✓ **Encapsulation**
- ✓ **Compatibility with other Versions**

Figure 12.6 *Position of IGMP in the network layer*

The **Internet Group Management Protocol (IGMP)** is responsible for correcting and interpreting information about group members in a network.



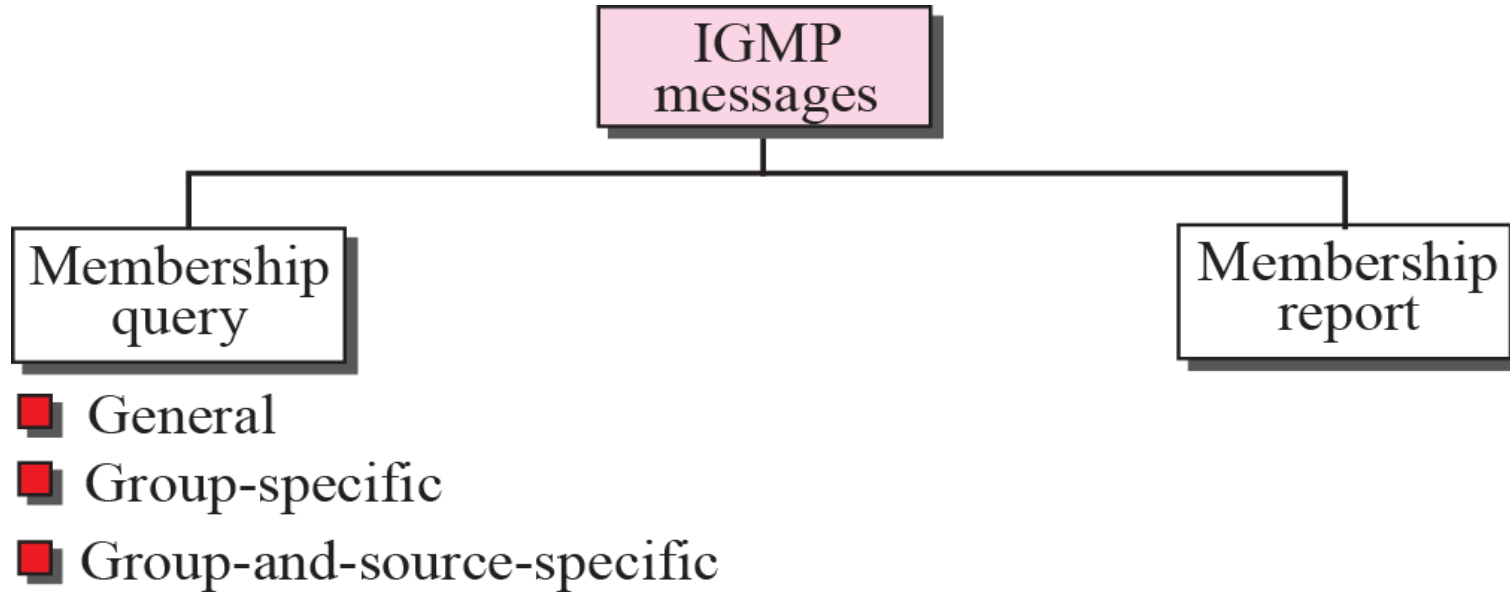
- IGMP is not a multicasting routing protocol, it is a protocol that manages group membership
- The IGMP protocol gives the multicast routers information about the membership status of hosts (routers)
- A multicast router may receive thousands of multicast packets every day for different groups. What happen If a router has no knowledge about the membership status of the hosts
- IGMP helps the multicast router create and update the list of groups



Note

IGMP is a group management protocol. It helps a multicast router create and update a list of loyal members related to each router interface.

Figure 12.7 *IGMP messages*



A membership query message is sent by a router to find active group members in the network

Figure 12.8 *Membership query message format*

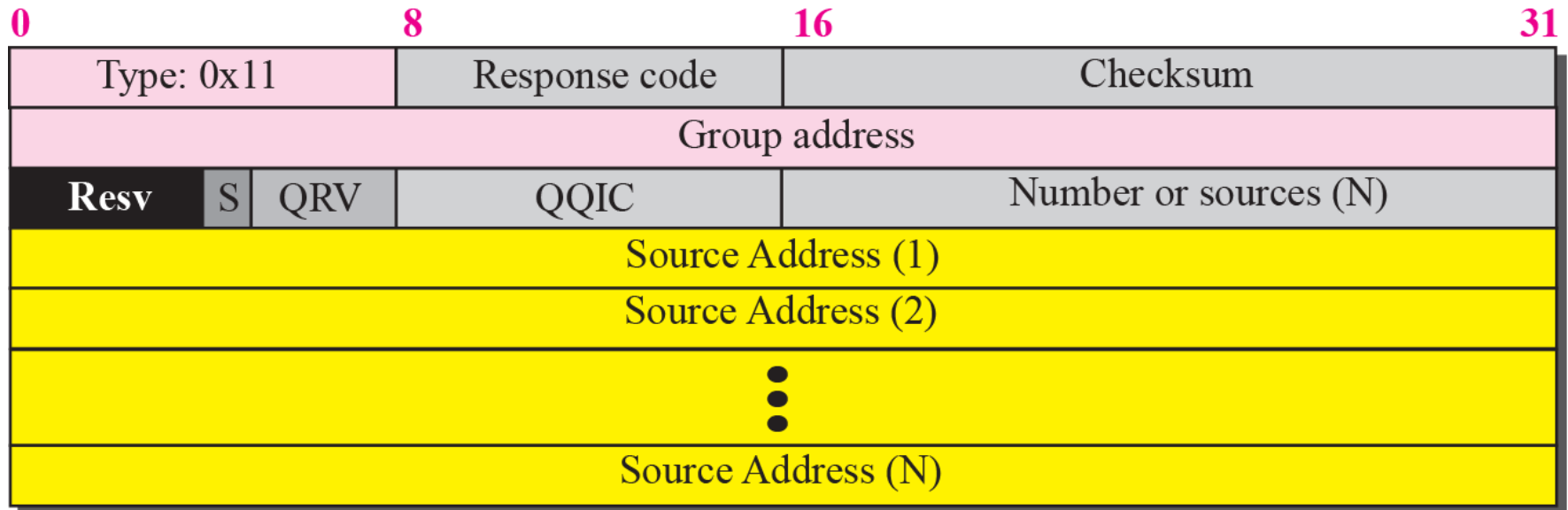
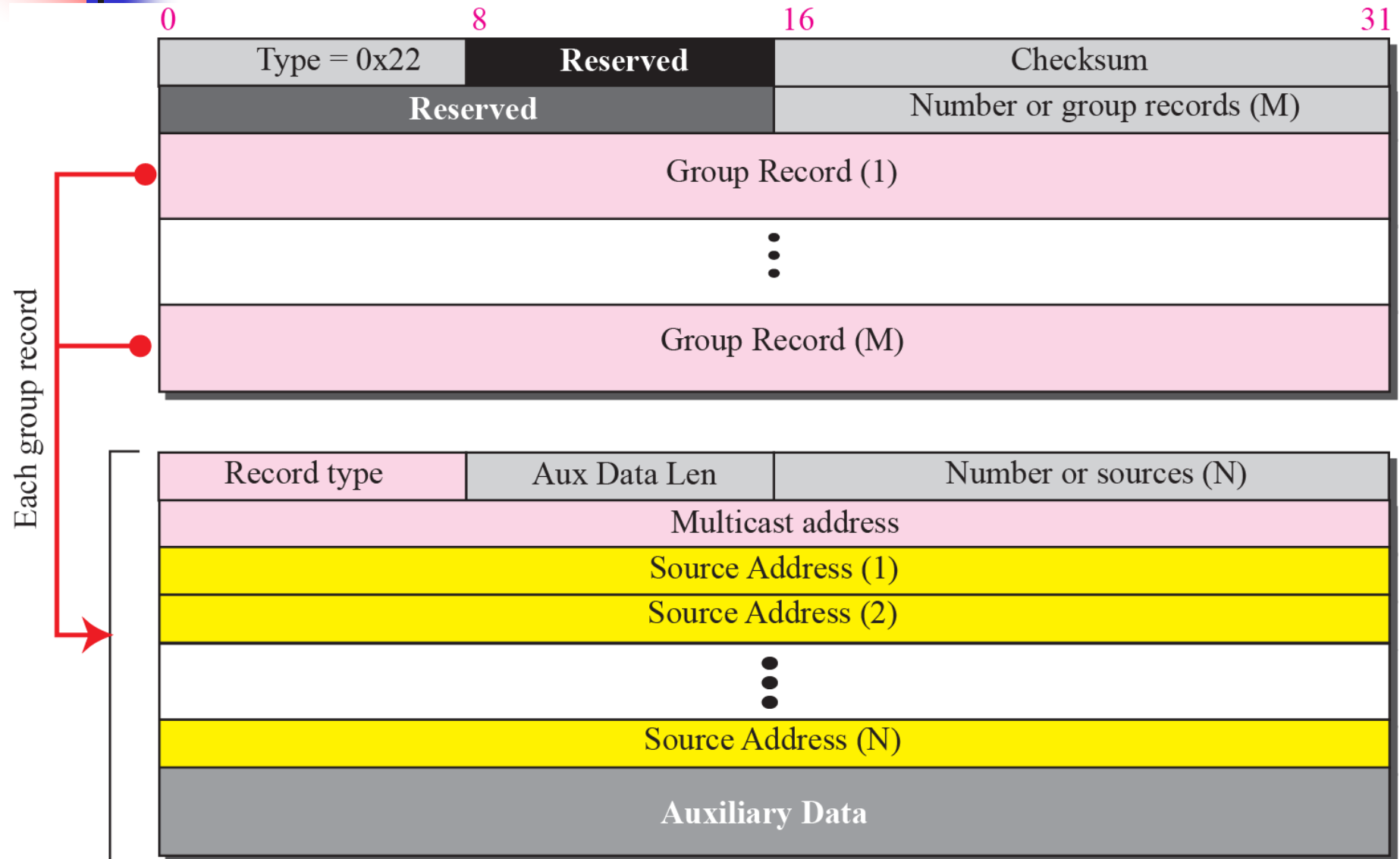


Figure 12.10 *Membership report message format*



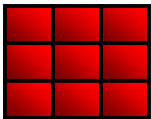


Table 12.3 *Record Type*

<i>Category</i>	<i>Type</i>	<i>Type Value</i>
Current-State-Record	Mode_Is_Include	1
	Mode_Is_Exclude	2
Filter-Mode-Change-Record	Change_To_Include_Mode	3
	Change_To_Exclude_Mode	4
Source-List-Change-Record	Allow_New_Sources	5
	Block_Old_Sources	6

Socket state

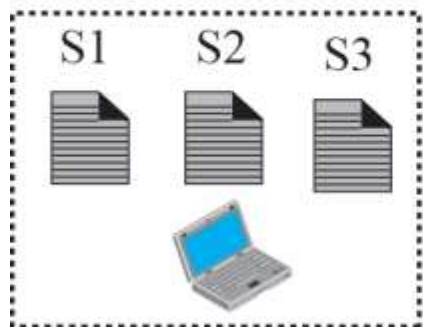
- The management of groups starts with the processes
- Each process has a record for each multicast group from which the socket wishes to receive a multicast message
- The record also shows one of the two modes: **include** mode or **exclude** mode
- **Include** mode, it lists the unicast source addresses from which the socket accepts the group messages
- **Exclude** mode, it lists the unicast source addresses that the socket will not accept the group messages

Example 12.4

Figure 12.11 shows a host with three processes: **S1**, **S2**, and **S3**. The first process has only one record; the second and the third processes each have two records. We have used lowercase alphabet to show the source address.

Figure 12.11 *Socket state*

Each **process** (associated with a **socket**) has a record for each multicast group from which the socket wishes to receive a multicast message



Legend

S: Socket
a, b, ...: Source addresses

States Table

Socket	Multicast group	Filter	Source addresses
S1	226.14.5.2	Include	a, b, d, e
S2	226.14.5.2	Exclude	a, b, c
S2	228.24.21.4	Include	b, c, f
S3	226.14.5.2	Exclude	b, c, g
S3	228.24.21.4	Include	d, e, f



Note

Each time there is a change in any socket record, the interface state will change using the above-mentioned rules.

Combine the list of resources.

1. If any of the records to be combined has the *exclusive* filter mode, then the resulting interface record will have the *exclusive* filter mode and the list of the source addresses is made as shown below:
 - a. Apply the set intersection operation on all the address lists with *exclusive* filters.
 - b. Apply the set difference operation on the result of part *a* and all the address lists with *inclusive* filters.

2. If all the records to be combined have the *inclusive* filter mode, then the resulting interface record will have the *inclusive* filter mode and the list of the source addresses is found by applying the set union operations on all the address lists.

. Or \cap : Intersection, \cup :
Union

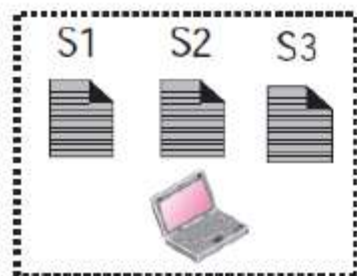
Example 12.5: Interface State

We use the two rules described above to create the interface state for the host in Example 12.4. First we found the list of source address for each multicast group.

- a.** Multicast group 226.14.5.2 has two exclude lists and one include list.

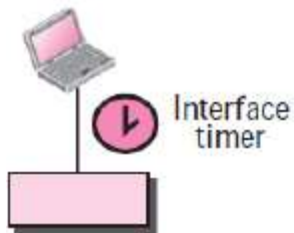
- b.** Multicast group: 228.24.21.4 has two include lists.

Figure 12.12 *Interface state*



Socket state

Socket	Multicast group	Filter	Source addresses
S1	226.14.5.2	Include	a, b, d, e
S2	226.14.5.2	Exclude	a, b, c
S2	228.24.21.4	Include	b, c, f
S3	226.14.5.2	Exclude	b, c, g
S3	228.24.21.4	Include	d, e, f



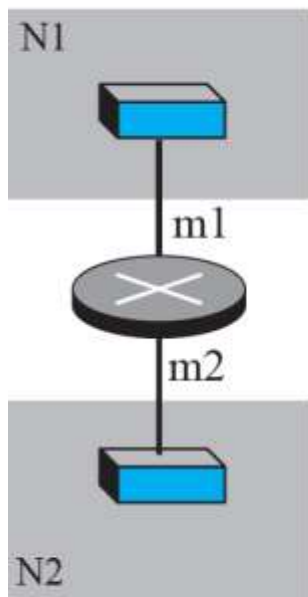
Interface state

Multicast group	Group timer	Filter	Source addresses
226.14.5.2		Exclude	c
228.24.21.4		Include	b, c, d, e, f

226.14.5.2 : **exclude source list = {a, b, c} . {b, c, g} – {a, b, d, e} = {c}**

228.24.21.4 : **include source list = {b, c, f} + {d, e, f} = {b, c, d, e, f}**

Figure 12.14 *Router States*



State for interface m1

Multicast group	Timer	Filter	Source addresses
227.12.15.21	⬇	Exclude	(a , ⬇) (c , ⬇)
228.21.25.41	⬇	Include	(b , ⬇) (d , ⬇) (e , ⬇)

State for interface m2

Multicast group	Timer	Filter	Source addresses
226.10.11.8	⬇	Exclude	(b , ⬇)
227.21.25.41	⬇	Include	(a , ⬇) (b , ⬇) (c , ⬇)
228.32.12.40	⬇	Include	(d , ⬇) (e , ⬇) (f , ⬇)

Router maintains state information for each multicast group associated with each network interface

12-4 MULTICAST ROUTING

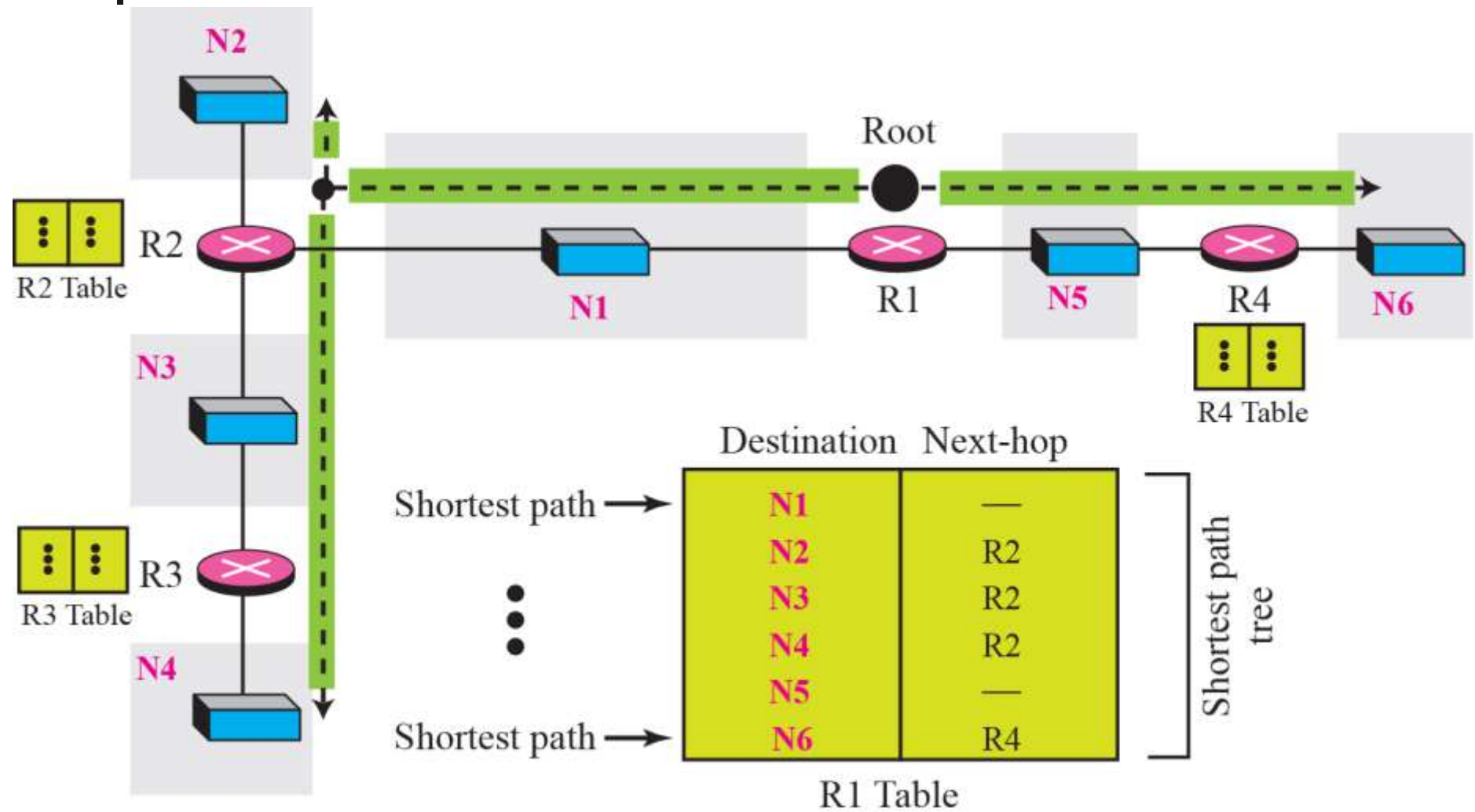
Now we show how information collected by IGMP is disseminated to other routers using multicast routing protocols. However, we first discuss the idea of optimal routing, common in all multicast protocols. We then give an overview of multicast routing protocols.



Note

In unicast routing, each router in the domain has a table that defines a shortest path tree to possible destinations.

Figure 12.18 *Unicast routing*



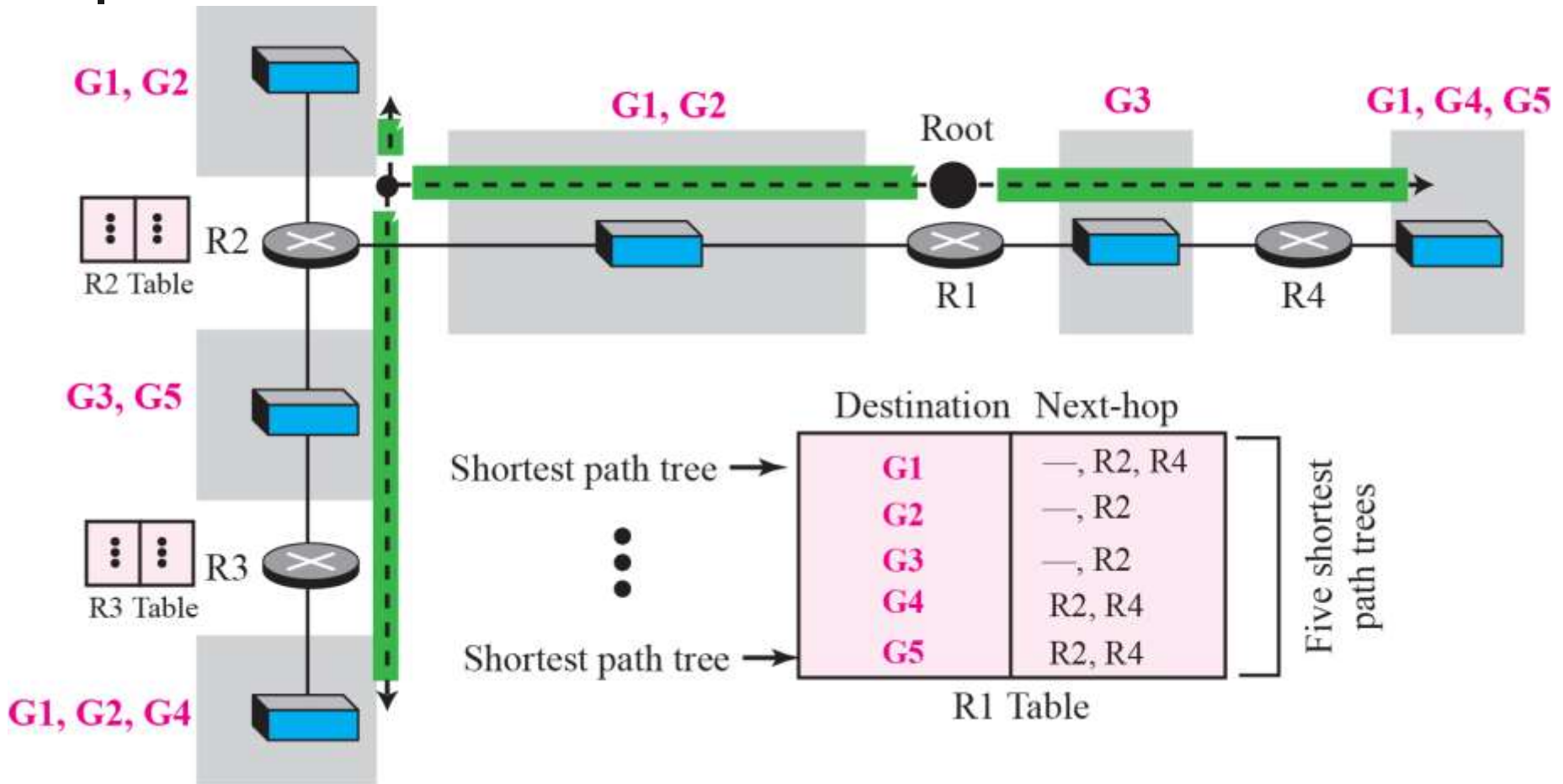
Note

In multicast routing, each involved router needs to construct a shortest path tree for each group.

In the source-based tree approach, each router needs to have one shortest path tree for each group and source.

The shortest path tree for a group defines the next hop for each network that has loyal member(s) for that group

Figure 12.19 *Source-based tree approach*

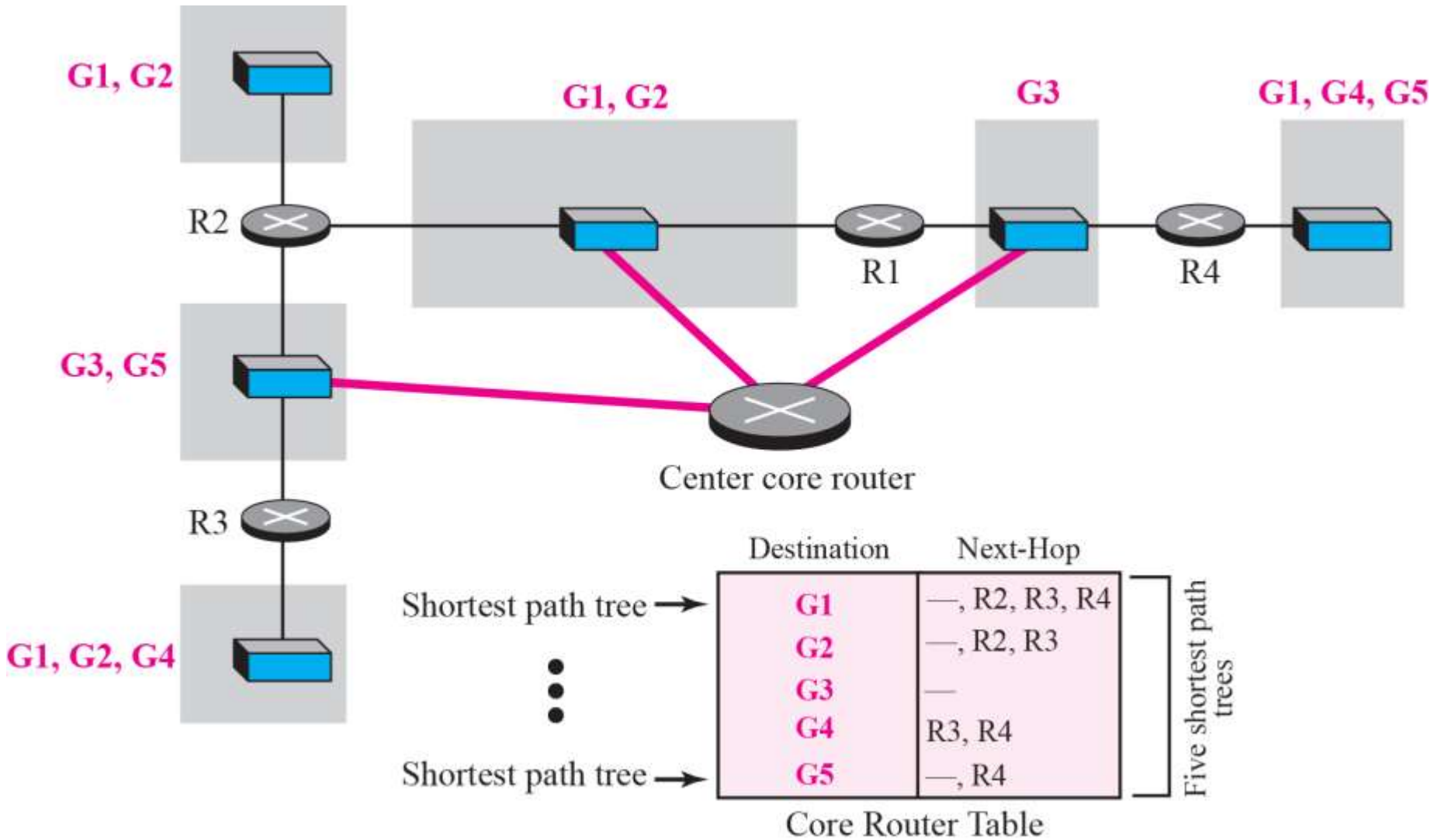


For m groups, each **router** needs to have m shortest path trees, one for each group

In the group-shared tree approach, only the core router (also called rendezvous), which has a shortest path tree for each group, is involved in multicasting.

If a router receives a multicast packet, it encapsulates the packet in a unicast packet and sends it to the core router

Figure 12.20 *Group-shared tree approach*



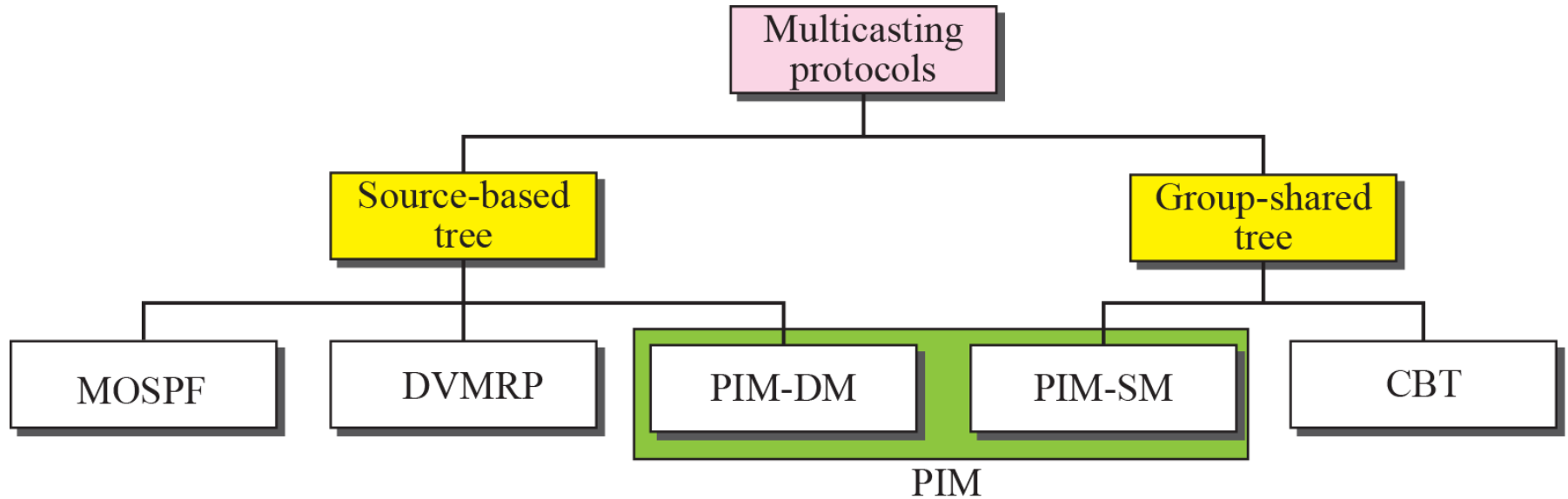
12-4 ROUTING PROTOCOLS

During the last few decades, several multicast routing protocols have emerged. Some of these protocols are extensions of unicast routing protocols; some are totally new. We discuss these protocols in the remainder of this chapter. Figure 12.21 shows the taxonomy of these protocols.

Topics Discussed in the Section

- ✓ **Multicast Link State Routing: MOSPF**
- ✓ **Multicast Distance Vector: DVMRP**
- ✓ **Core-Based Tree: CBT**
- ✓ **Protocol Independent Multicast: PIM**

Figure 12.21 *Taxonomy of common multicast protocols*



Multicast Link State Routing: **MOSPF**

Multicast Distance Vector Routing Protocol: **DVMRP**

Core-Based Tree: **CBT**

Protocol Independent Multicast: **PIM**

Multicast Link State Routing

- Uses source-based tree approach
- Extension of unicast link state routing
- Node advertises group with members on the link
- The information about the group comes from **IGMP**
- Router creates n shortest path trees (for n groups) using Dijkstra's algorithm
- **Problem:** time and space needed to create and save the many shortest path trees.
- **Solution:** Router calculates shortest path trees on demand

MOSPF Protocol

- Extension of OSPF Protocol
- Uses multicast link state routing to create source-based trees
- Uses new link state update packet to associate source with group of addresses (group-membership LSA)
- This way: we can include in the tree only the hosts (using their unicast addresses) that belong to a particular group
- The router calculates the shortest path trees on demand (when it receives the first multicast packet)

Multicast Distance Vector Routing

- Uses source-based tree approach
- Uses four strategies, each built on its predecessor
 1. Flooding
 2. Reverse Path Forwarding (RPF)
 3. Reverse Path Broadcasting (RPB)
 4. Reverse Path Multicasting (RPM)

Flooding

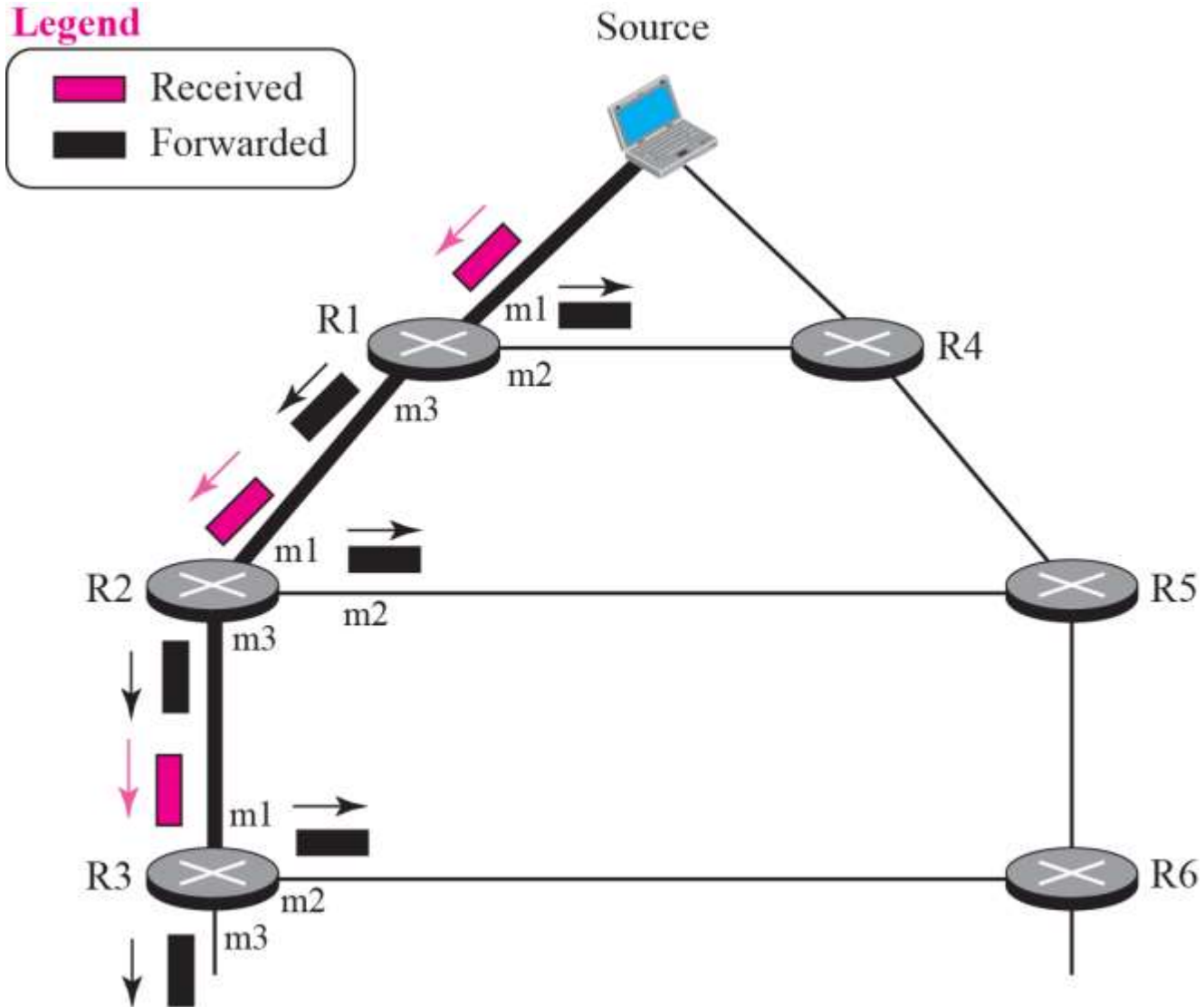
Flooding is the first strategy that comes to mind. A router receives a packet and without even looking at the destination group address

Flooding broadcasts packets but creates loops in the systems.

Reverse Path Forwarding (RPF)

- To prevent loops, only one copy is forwarded; the other copies are dropped.
- In RPF, a router forwards only the copy that has traveled the shortest path from the source to the router.
- The router extracts the source address of the multicast packet and consults its unicast routing table.
- If the packet has just come from the hop defined in the table, the packet has traveled the shortest path from the source to the router because the shortest path is reciprocal in unicast distance vector routing protocols.
- If a packet leaves the router and comes back again, it has not traveled the shortest path.

Figure 12.22 *Reverse Path Forwarding (RPF)*



RPF eliminates the loop in the flooding process.

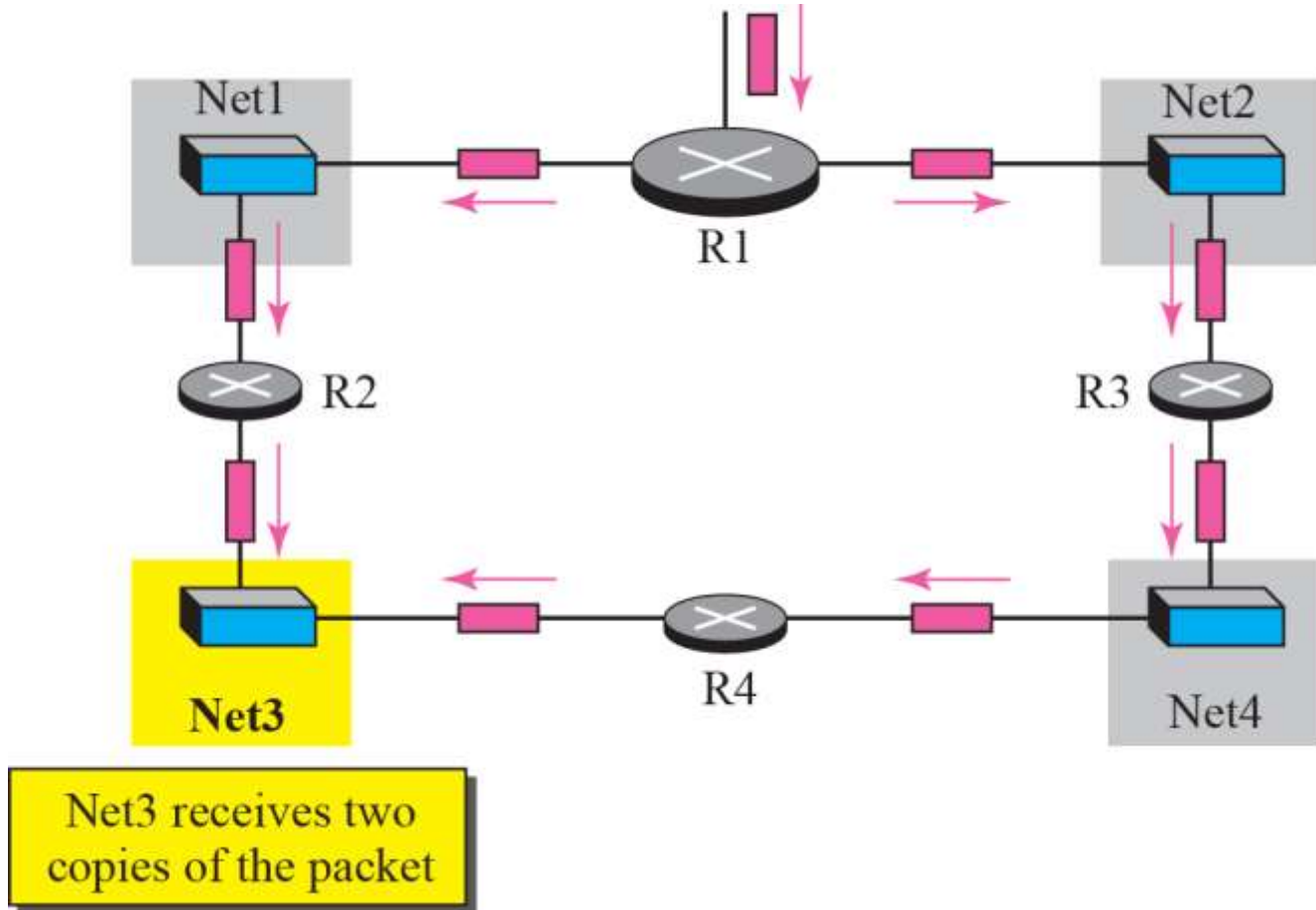


Note

RPF eliminates the loop in the flooding process.

- **RPF** guarantees that each network receives a copy of the multicast packet without formation of loops.
- However, **RPF** does not guarantee that each network receives only one copy; a network may receive two or more copies
- **RPF** is not based on the destination address (a group address); forwarding is based on the source address

Figure 12.23 *Problem with RPF*

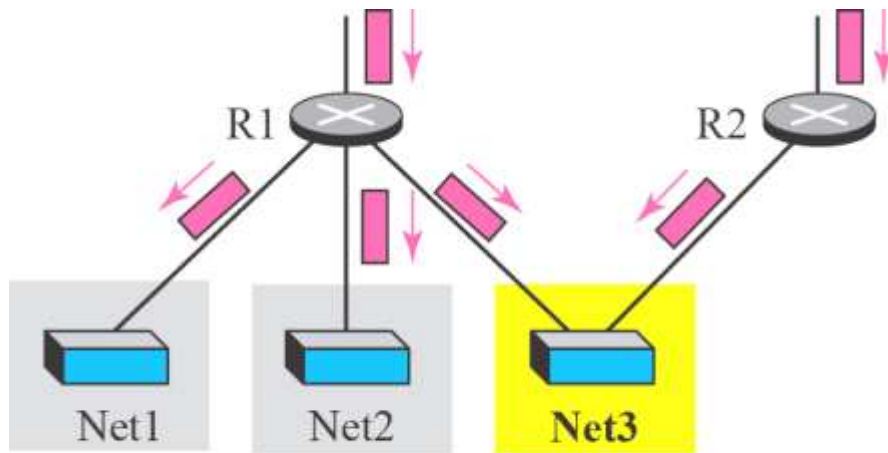




Reverse Path Broadcasting (RPB)

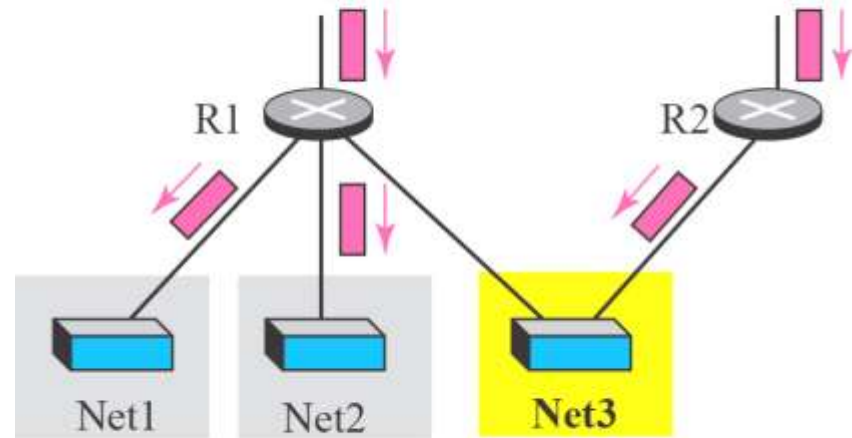
- To eliminate duplication, we must define only one parent router for each network. We must have this restriction
- A network can receive a multicast packet from a particular source only through a designated parent router.
- For each source, the router sends the packet only out of those interfaces for which it is the designated parent. This policy is called **reverse path broadcasting (RPB)**
- **RPB** guarantees that the packet reaches every network and that every network receives only one copy

Figure 12.24 *RPF versus RPB*



a. RPF

R1 is the parent of Net1 and Net2.
R2 is the parent of Net3



b. RPB

- **RPB** does not multicast the packet, it broadcasts it.
- Multicast packet must reach only those networks that have active members for that particular group.
- This is called reverse path multicasting (**RPM**).
- **RPM** uses two procedures, pruning and grafting

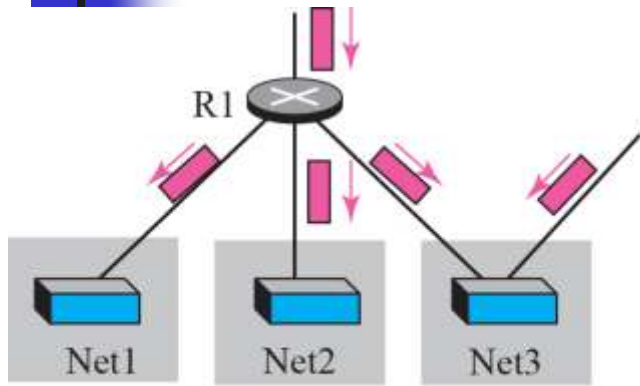


Note

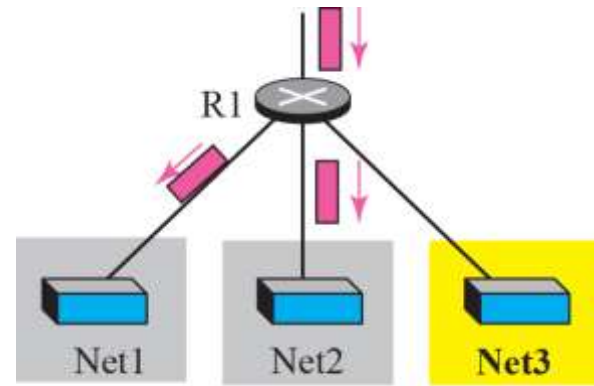
RPB creates a shortest path broadcast tree from the source to each destination.

It guarantees that each destination receives one and only one copy of the packet.

Figure 12.25 Reverse Path Multicasting (RPM)



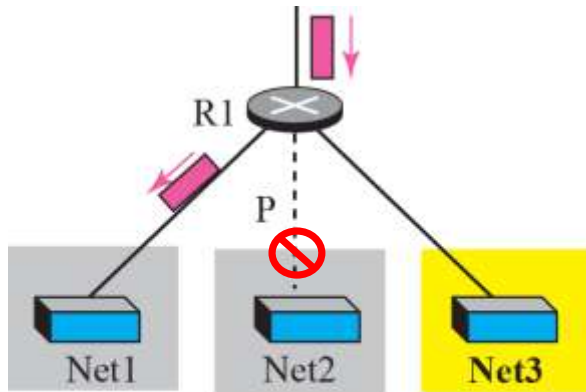
a. RPF



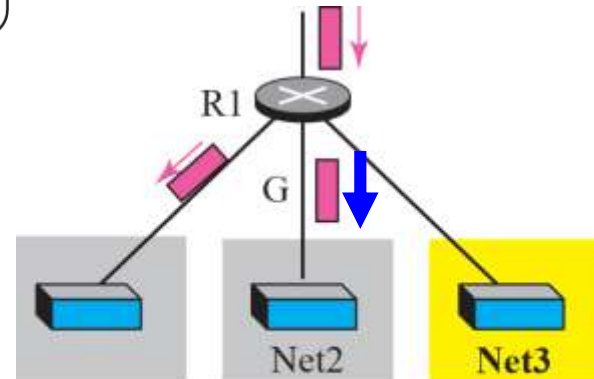
b. RPB

Legend

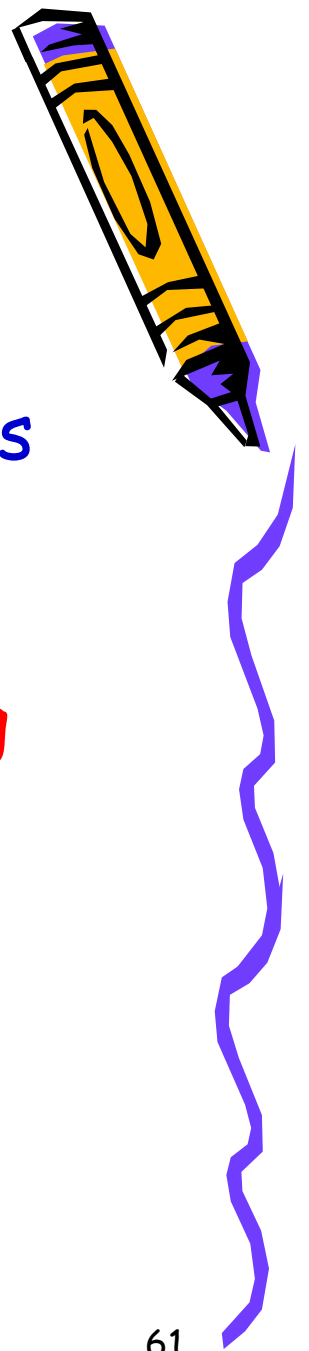
P: Pruned route
G: Grafted route



c. RPM (after pruning)



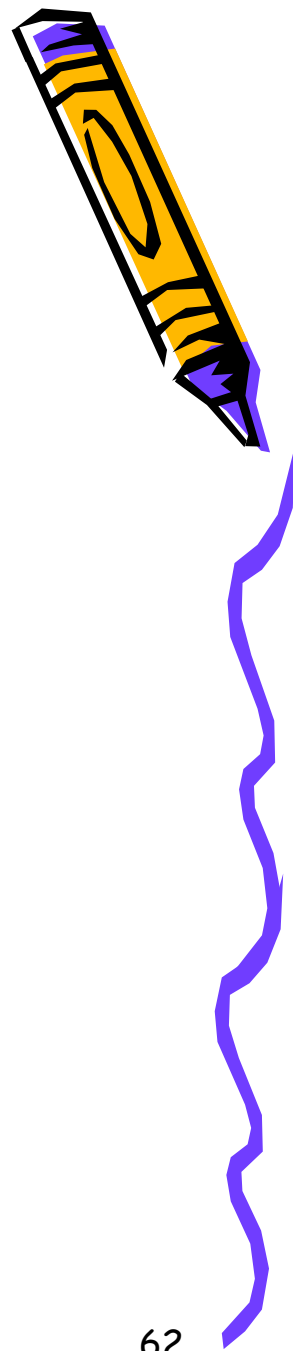
d. RPM (after grafting)



Reverse Path Multicasting (1)

- To increase efficiency, the multicast packet must reach only those networks that have active members for that particular group
- RPM adopts the procedures of **Pruning** and **Grafting**
- **Pruning**
 - The designated parent router of each network is responsible for holding the membership information (through IGMP)





Reverse Path Multicasting (2)

- The router sends a prune message to the upstream router so that it can prune the corresponding interface
- That is, the upstream router can stop sending multicast message for this group through that interface
- **Grafting**
 - The graft message forces the upstream router to resume sending the multicast messages





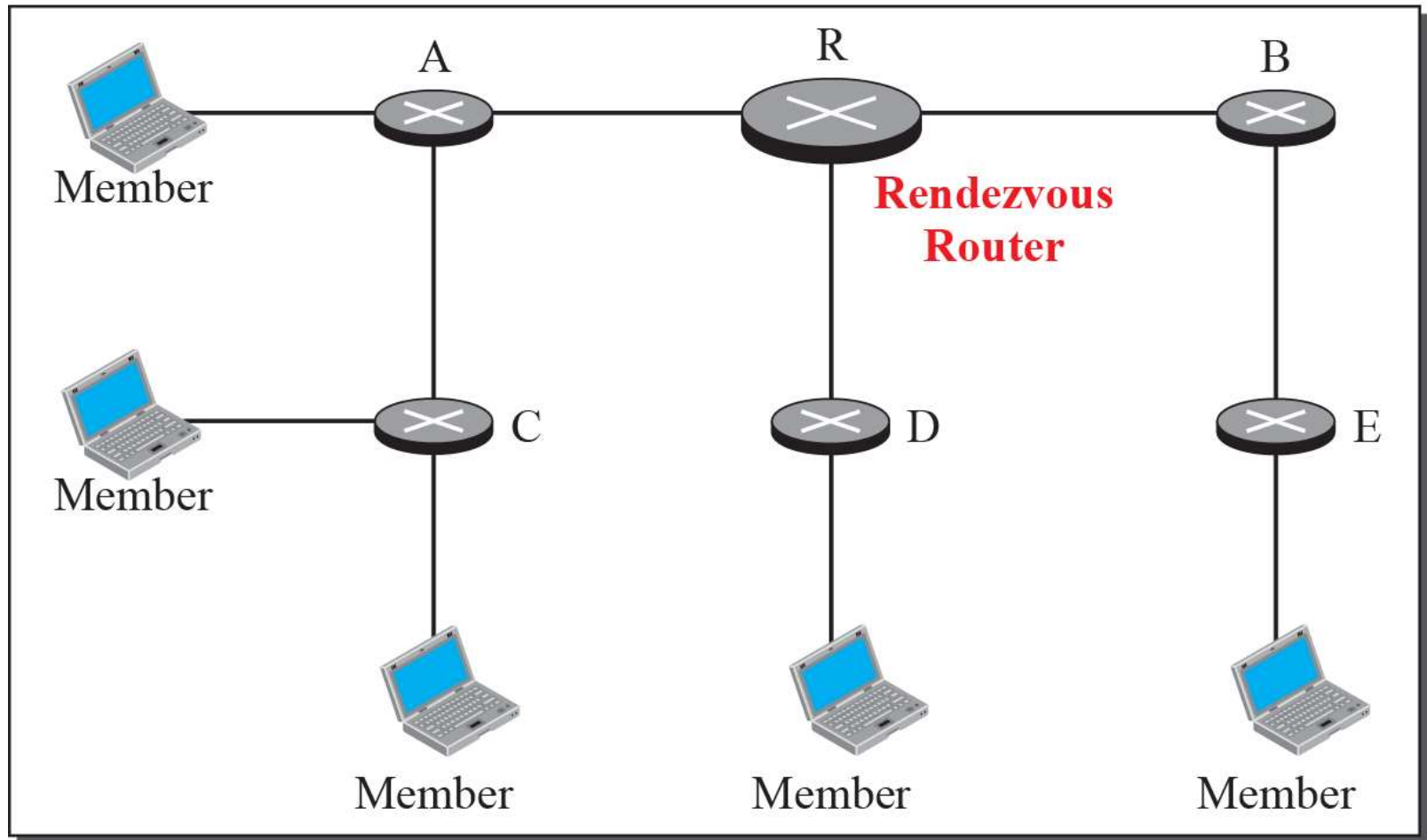
Note

RPM adds pruning and grafting to RPB to create a multicast shortest path tree that supports dynamic membership changes.

Figure 12.26 *Group-shared tree with rendezvous router*

Core-Based Tree (CBT) Protocol

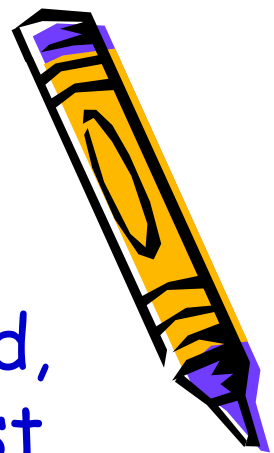
Shared Tree

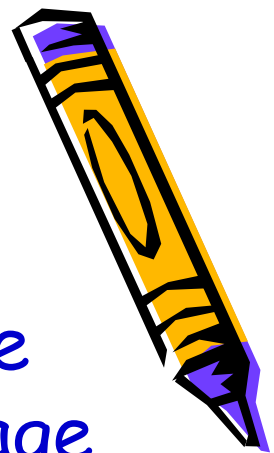


Formation of CBT tree (1)

Core-Based Tree (CBT) Protocol

- After the **rendezvous point** is selected, every router is informed of the unicast address of the selected router
- Each router sends a unicast join message to show that it wants to join the group
- This message passes through all routers that are located between the sender and the rendezvous router



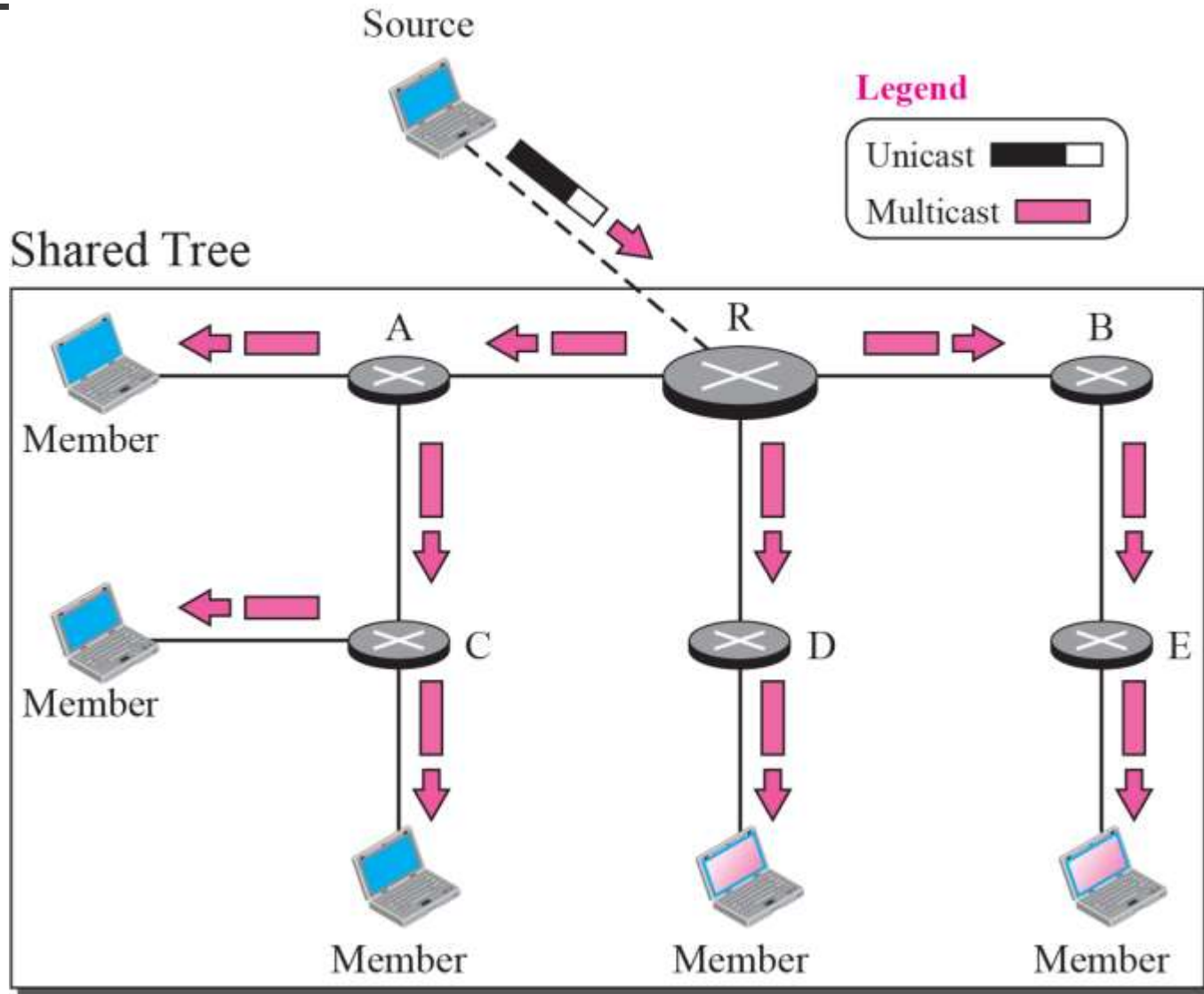


Formation of CBT tree (2)

- Each intermediate router extracts the necessary information from the message
 - Unicast address of the sender
 - Interface through which the packet has arrived
- Every router knows its upstream router and the downstream router
- If a router wants to leave the group, it sends a leave message to its upstream router, ...



Figure 12.27 *Sending a multicast packet to the rendezvous router*



Source Hosts host can be inside the shared tree or any host outside the shared tree



Note

In CBT, the source sends the multicast packet (encapsulated in a unicast packet) to the core router. The core router decapsulates the packet and forwards it to all interested interfaces.

Comparisons

- The tree for **DVMRP** and **MOSPf** is made from the root up (source-based)
- The tree for **CBT (Core-based tree)** is formed from the leaves down (Group-based)
- In **DVMRP**, the tree is first made (broadcasting) and then pruned
- In **CBT**, the joining gradually makes the tree, and the source in **CBT** may or may not be part of the tree





Protocol Independent Multicast (PIM)

- Protocol Independent Multicast, Dense Mode (**PIM-DM**)
- Protocol Independent Multicast, Sparse Mode (**PIM-SM**).
- Both protocols are unicast-protocol dependent

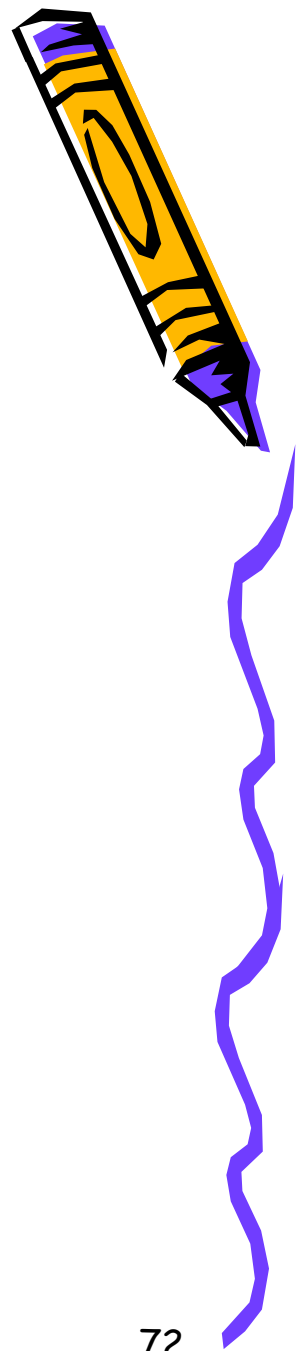


Note

PIM-DM is used in a dense multicast environment, such as a LAN.

PIM-DM (Dense Mode)

- It is used when there is a possibility that each router is involved in multicasting (dense mode)
- In this environment, the use of a protocol that broadcasts the packet is justified because almost all routers are involved in the process





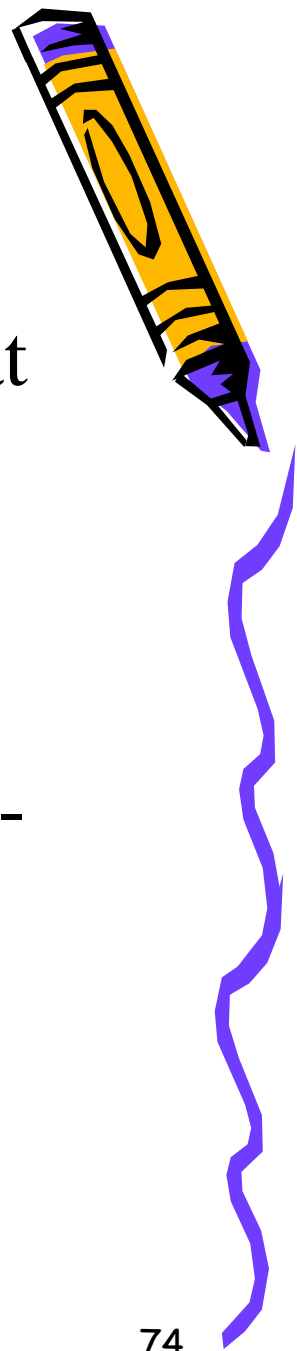
PIM-DM (Dense Mode)

PIM-DM uses RPF and pruning/grafting strategies to handle multicasting.

However, it is independent from the underlying unicast protocol.

PIM-SM (Sparse Mode)

- Used when there is a slight possibility that each router is involved in multicasting
- In this environment, the use of a protocol that broadcasts the packet is not justified
- A protocol such as CBT that uses a group-shared tree is more appropriate.





PIM-SM (Sparse Mode)

PIM-SM is used in a sparse multicast environment such as a WAN.

PIM-SM is similar to CBT but uses a simpler procedure.

12-6 MBONE

- Multimedia and real-time communication have increased the need for multicasting in the Internet
- However, only a small fraction of Internet routers are multicast routers
- The solution is tunneling. The multicast routers are seen as a group of routers on top of unicast routers
- The multicast routers may not be connected directly, but they are connected logically
- To enable multicasting, we make a multicast backbone (**MBONE**) out of these isolated routers using the concept of tunneling.

Figure 12.28 *Logical tunneling*

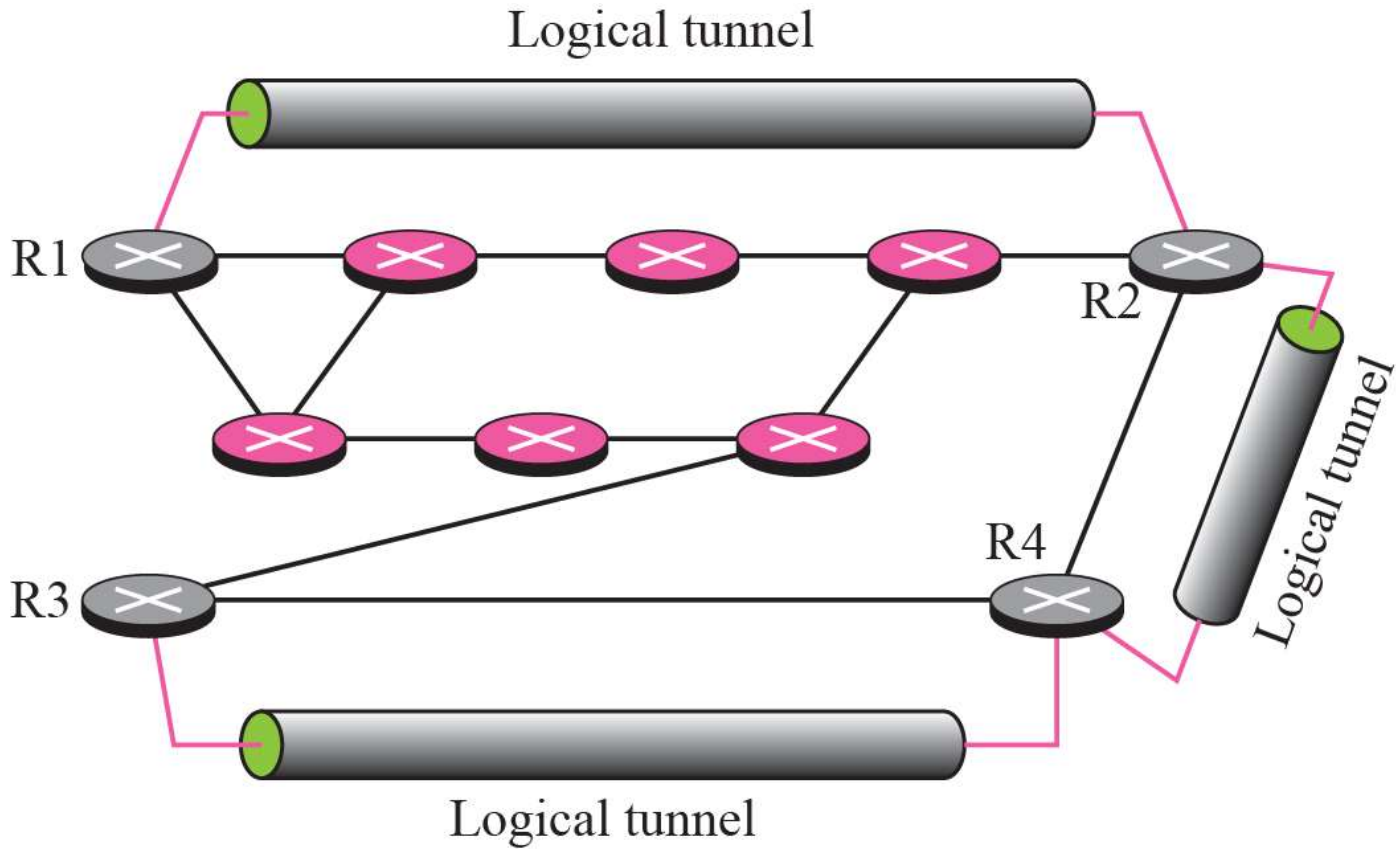
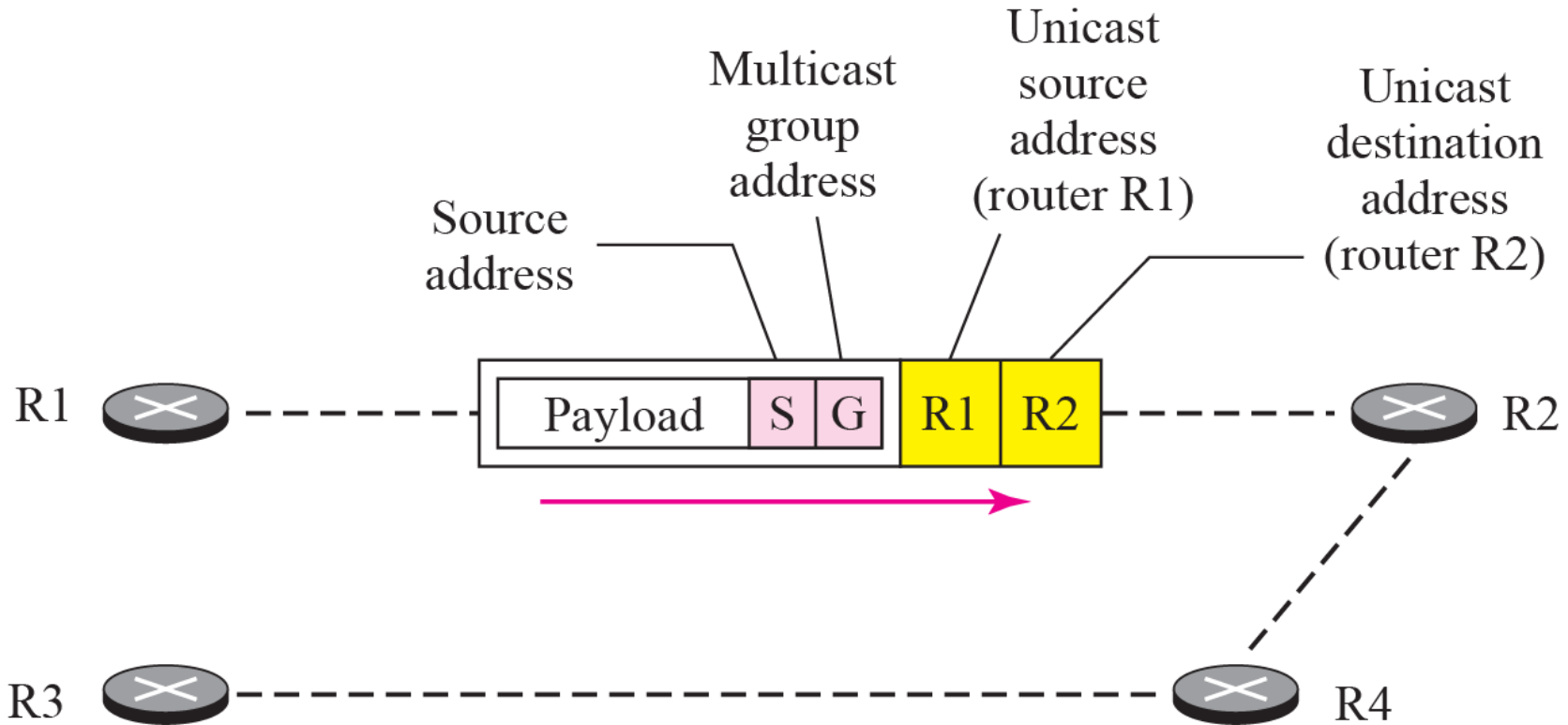


Figure 12.29 MBONE



The multicast packet becomes the payload (data) of the unicast packet

The only protocol that supports **MBONE** and tunneling is **DVMRP**

Chapter 13

Introduction to the Transport Layer

Edited & Presented by:

Dr. Mohammad Alhammouri

TCP/IP Protocol Suite (B A. Forouzan)

OBJECTIVES:

- ❑ To define process-to-process communication at the transport layer and compare it with host-to-host communication at the network layer.**
- ❑ To discuss the addressing mechanism at the transport layer, to discuss port numbers, and to define the range of port numbers used for different purposes.**
- ❑ To explain the packetizing issue at the transport layer: encapsulation and decapsulation of messages.**
- ❑ To discuss multiplexing (many-to-one) and demultiplexing (one-to-many) services provided by the transport layer.**
- ❑ To discuss flow control and how it can be achieved at the transport layer.**

OBJECTIVES (*continued*):

- ❑ **To discuss error control and how it can be achieved at the transport layer.**
- ❑ **To discuss congestion control and how it can be achieved at the transport layer.**
- ❑ **To discuss the connectionless and connection-oriented services at the transport layer and show their implementation using an FSM.**
- ❑ **To discuss the behavior of four generic transport-layer protocols and their applications: simple protocol, Stop-and-Wait protocol, Go-Back-N protocol, and Selective-Repeat protocol.**
- ❑ **To describe the idea of bidirectional communication at the transport layer using the piggybacking method.**

Chapter Outline

13.1 Transport-Layer Services

13.2 Transport-Layer Protocols

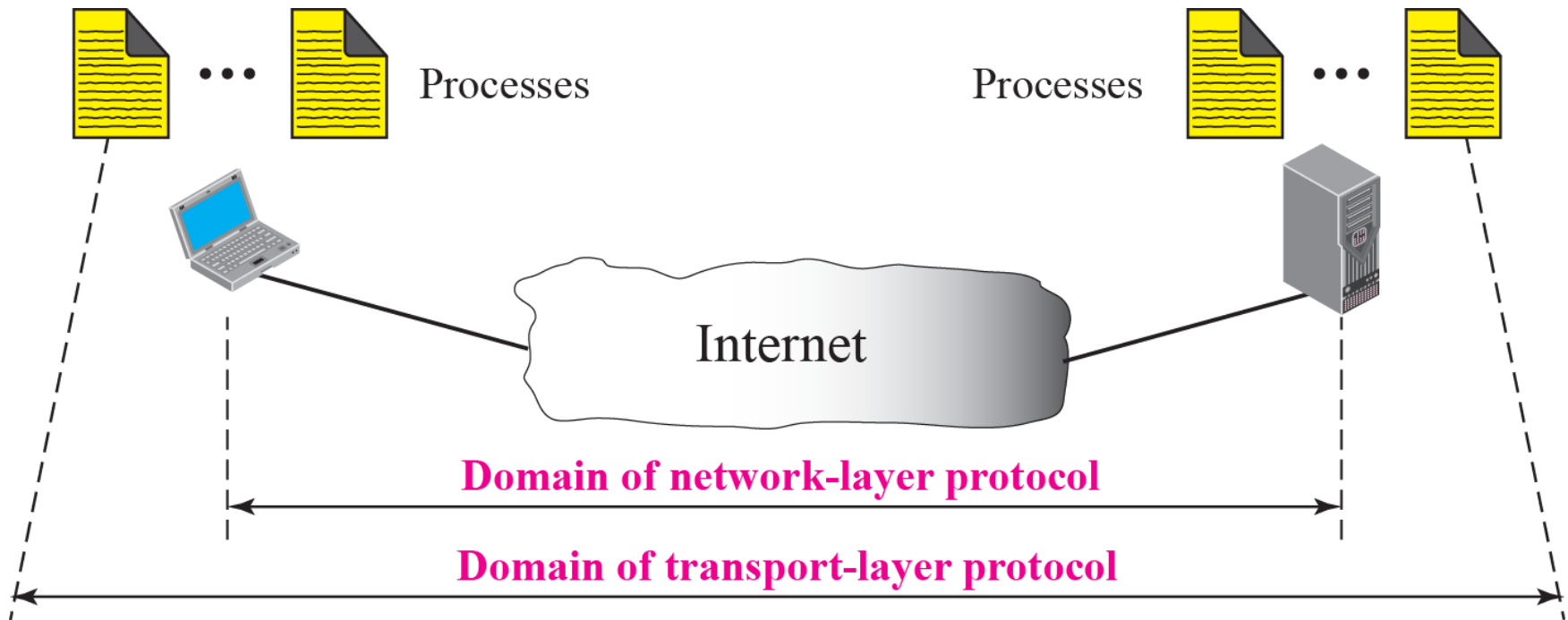
13-1 TRANSPORT-LAYER SERVICES

As we discussed in Chapter 2, the transport layer is located between the network layer and the application layer. The transport layer is responsible for providing services to the application layer; it receives services from the network layer. In this section, we discuss the services that can be provided by a transport layer; in the next section, we discuss the principle beyond several transport layer protocols.

Topics Discussed in the Section

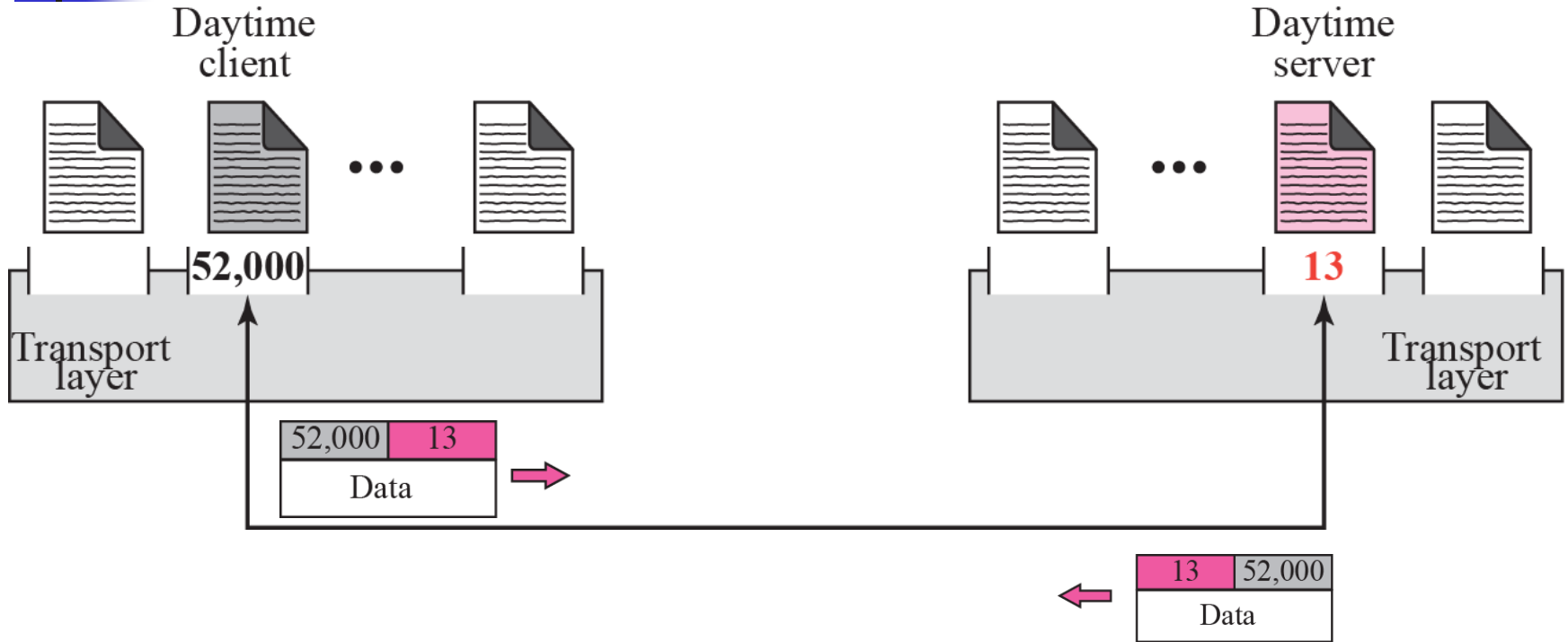
- ✓ **Process-to-Process Communication**
- ✓ **Addressing: Port Numbers**
- ✓ **Encapsulation and Decapsulation**
- ✓ **Multiplexing and Demultiplexing**
- ✓ **Flow Control**
- ✓ **Error Control**
- ✓ **Congestion Control**
- ✓ **Connectionless and Connection-Oriented Services**

Figure 13.1 *Network layer versus transport layer*



Host-to-Host vs Process-to-Process communications

Figure 13.2 *Port numbers*



TCP/IP has decided to use universal port numbers for servers; these are called **well-known port numbers**, 13 is an example. The server port number cannot be chosen randomly

The daytime client process uses an **ephemeral** (temporary) port number 52,000 to identify itself

Figure 13.3 *IP addresses versus port numbers*

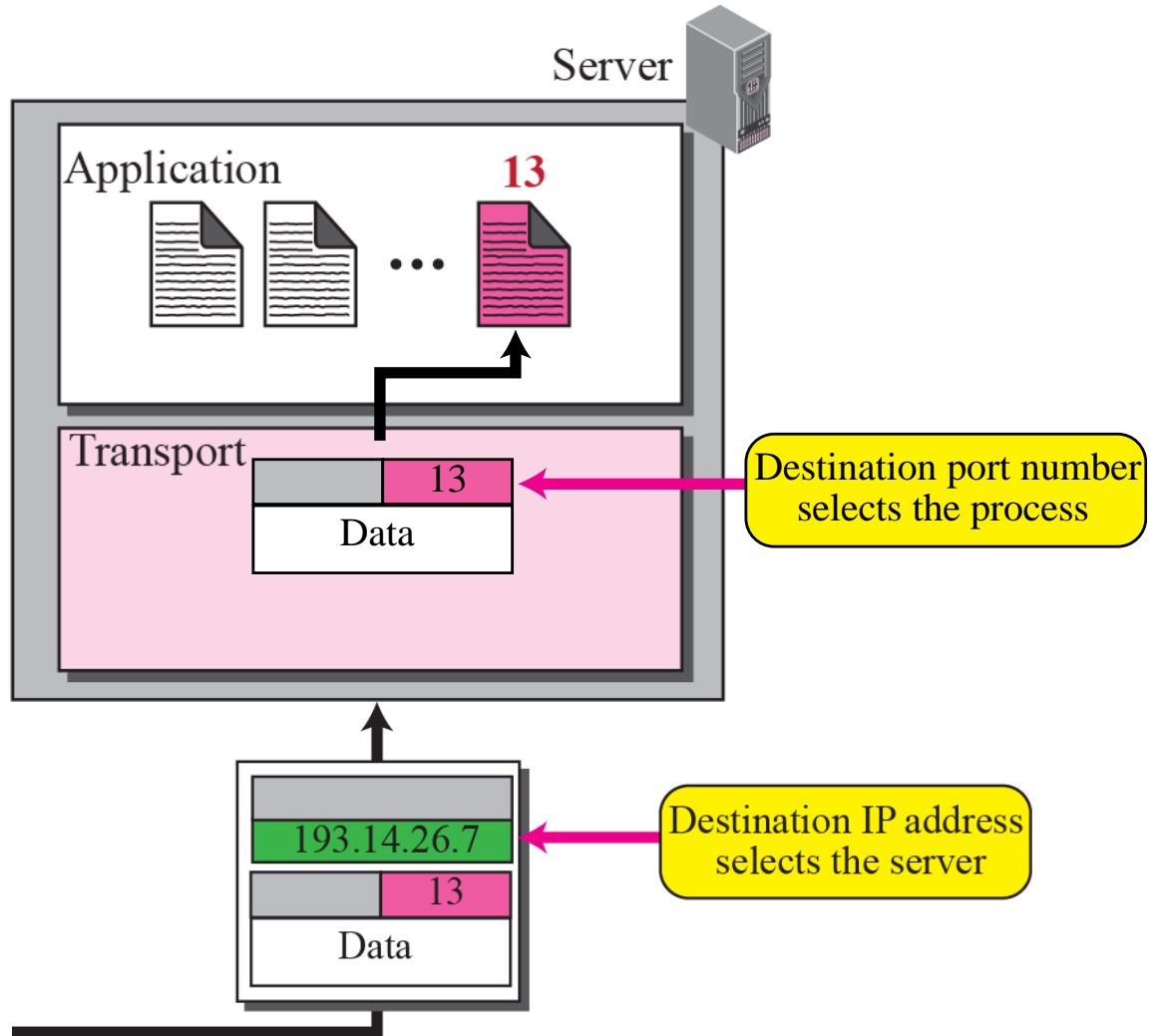
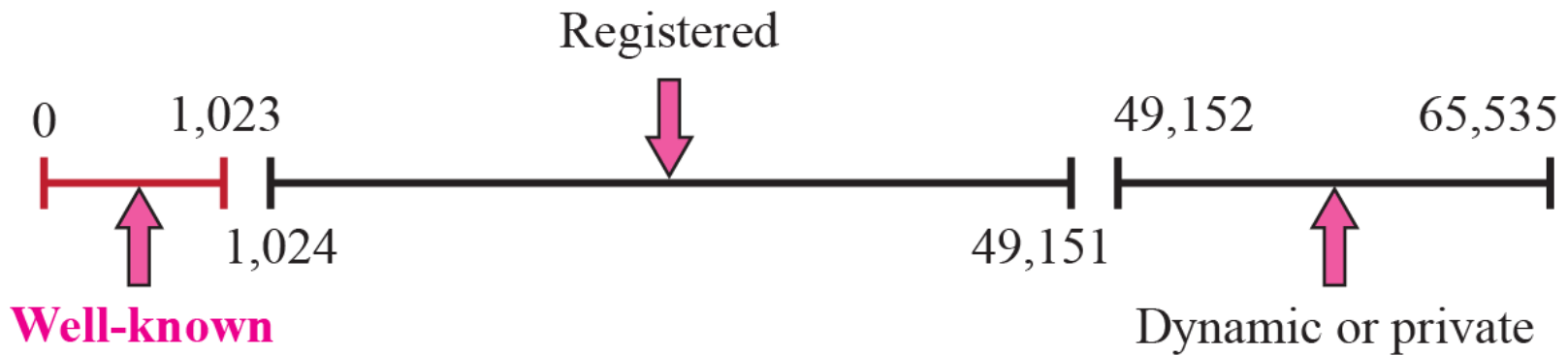


Figure 13.4 *ICANN ranges*

ICANN has divided the **port numbers** into three ranges: well-known, registered, and dynamic (or private)





Note

The well-known port numbers are less than 1,024.

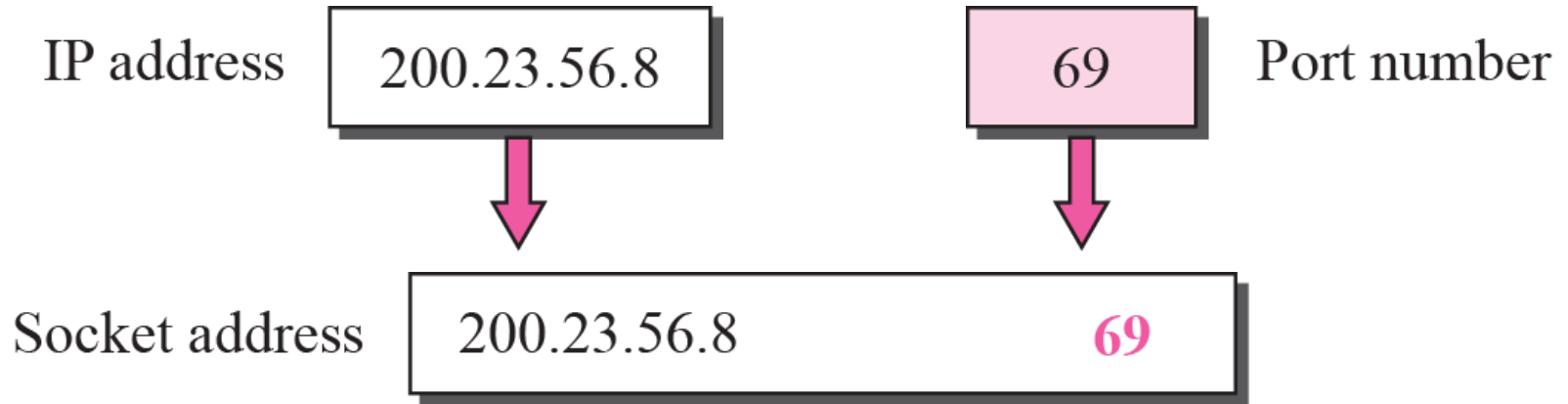
Example 13.1

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 that TFTP can use port 69 on either UDP or TCP. SNMP (see Chapter 24) uses two port numbers (161 and 162), each for a different purpose.

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

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

Figure 13.5 *Socket address*



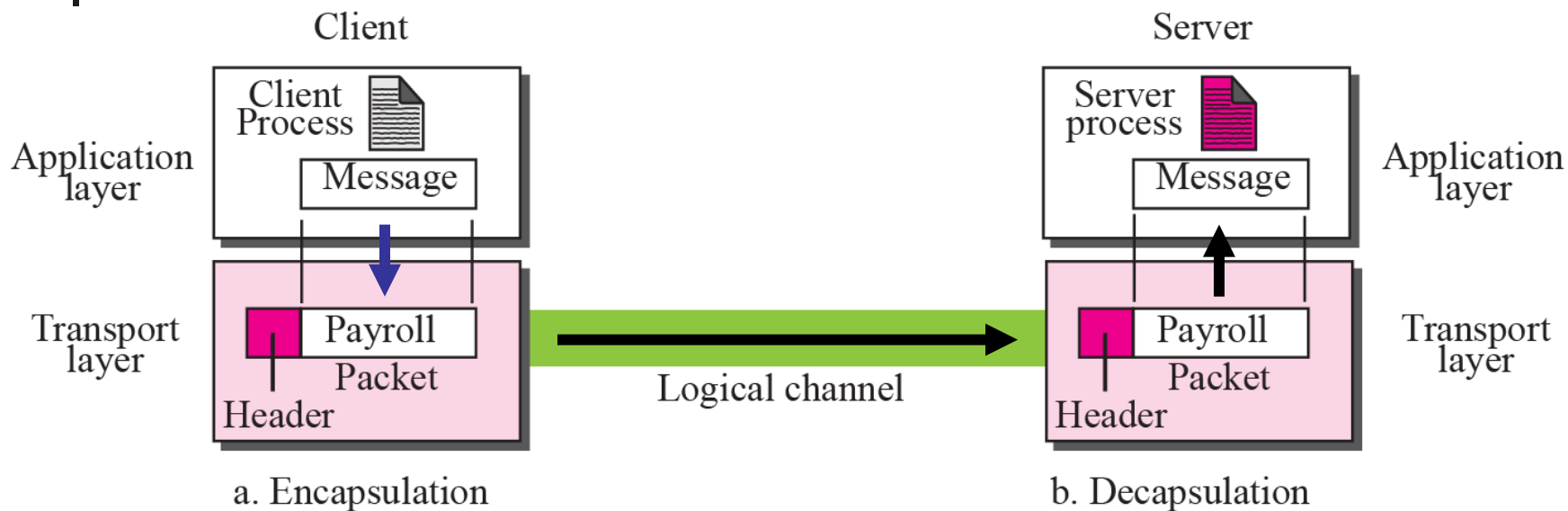
To use the services of transport layer in the Internet, we need a pair of socket addresses: **the client socket address** and the **server socket address**.

These four pieces of information are:

part of the network-layer packet header and the transport-layer packet header.

The first header contains the IP addresses; the second header contains the port numbers.

Figure 13.6 *Encapsulation and decapsulation*



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

Figure 13.7 *Multiplexing and demultiplexing*

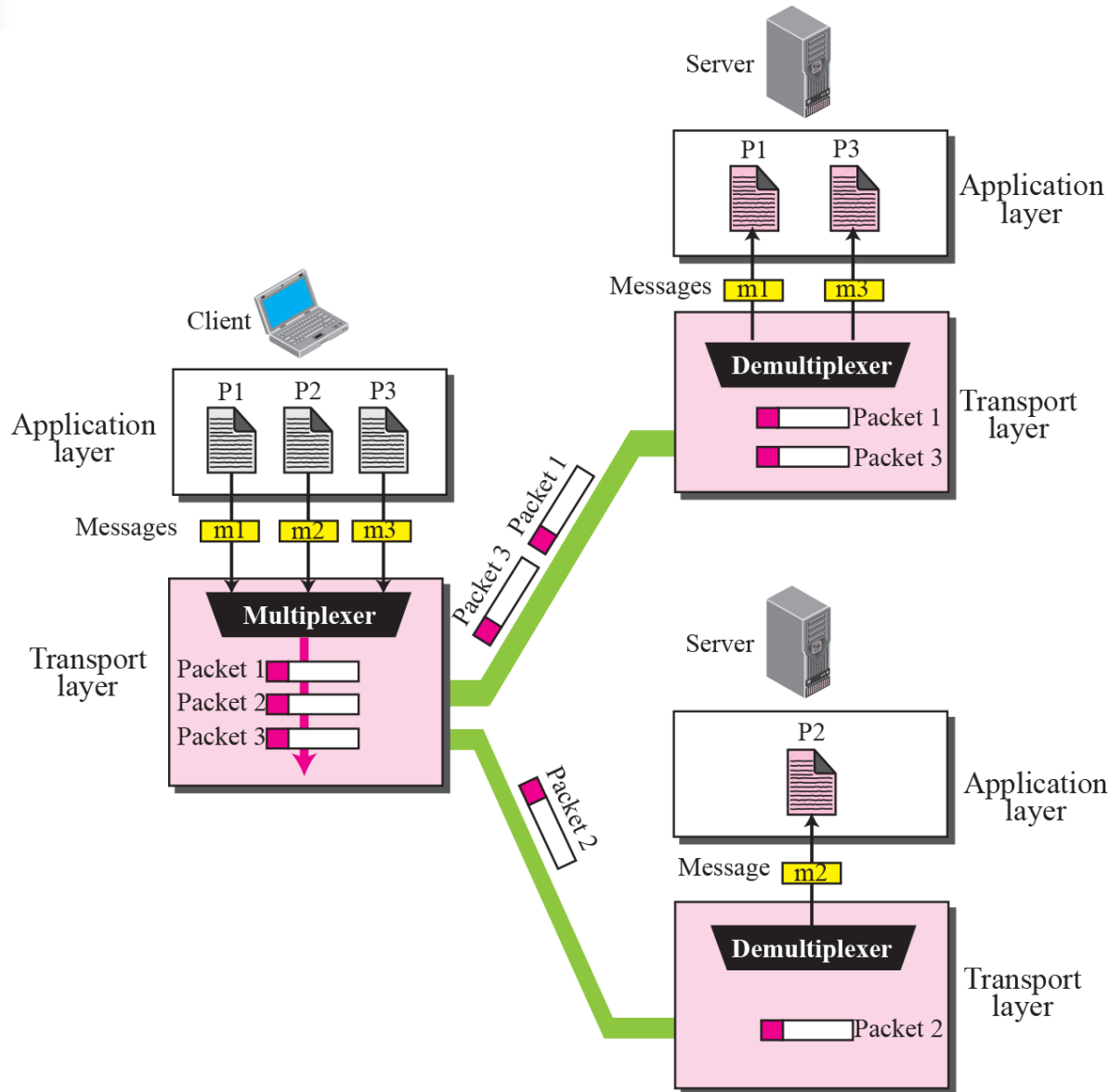
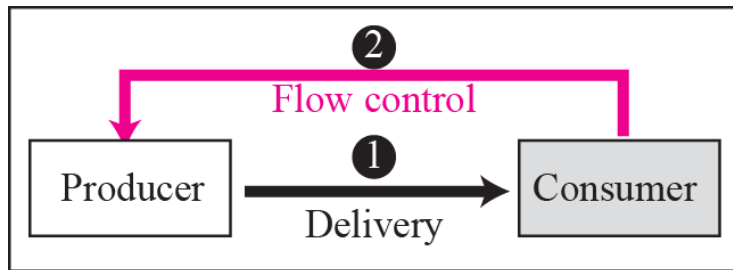


Figure 13.8 *Pushing or pulling*

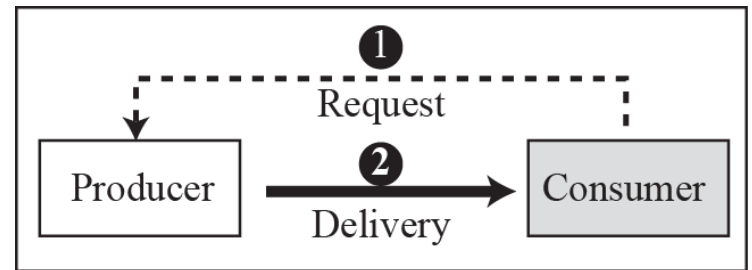
Delivery of items from a producer to a consumer can occur in one of the two ways:

Pushing: If the sender delivers items whenever they are produced

Pulling: If the producer delivers the items after the consumer has requested them,

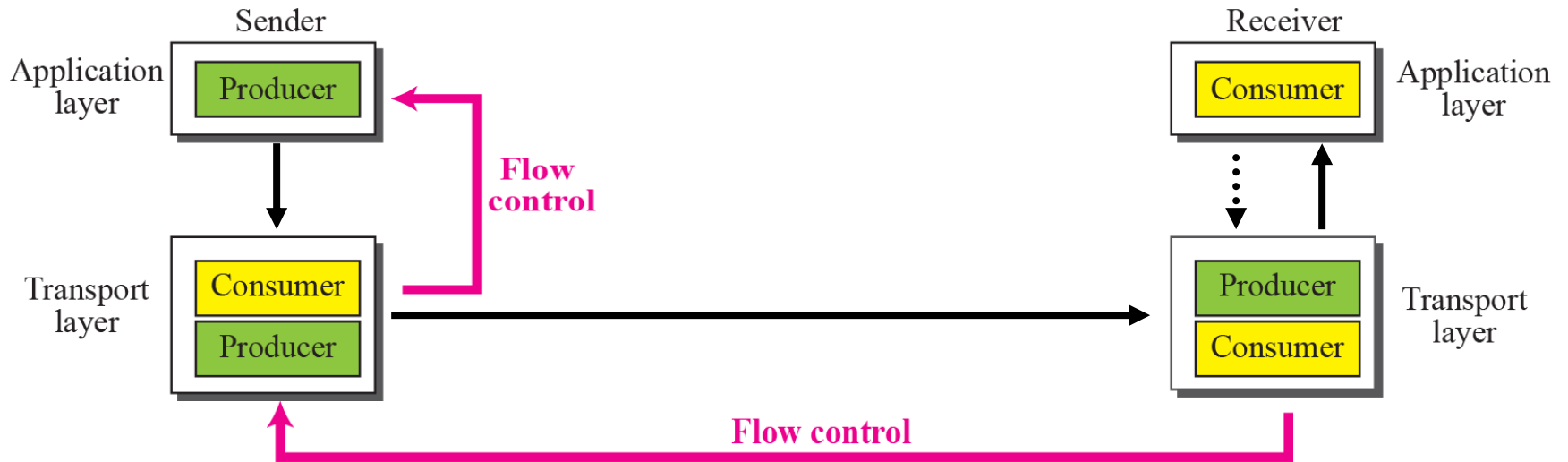


a. Pushing



b. Pulling

Figure 13.9 *Flow control at the transport layer*





Buffers

- Flow control can be implemented in several ways, one of the solutions is normally to use two buffers
- One buffer at the sending transport layer and the other at the receiving transport layer
- buffer is a set of memory locations that can hold packets at the sender and receiver
- The flow control communication can occur by sending signals from the consumer to producer
- When the buffer of the **sending transport layer** is full, it informs the application layer to stop passing chunks of messages. When there are some vacancies, it informs the sending transport layer that it can send message again.
- When the buffer of the **receiving transport layer** is full, it informs the sending transport layer to stop sending packets

Example 13.2

The above discussion requires that the consumers communicate with the producers in two occasions: when the buffer is full and when there are vacancies. If the two parties use a buffer of only one slot, the communication can be easier. Assume that each transport layer uses one single memory location to hold a packet. When this single slot in the sending transport layer is empty, the sending transport layer sends a note to the application layer to send its next chunk; when this single slot in the receiving transport layer is empty, it sends an acknowledgment to the sending transport layer to send its next packet. As we will see later, this type of flow control, using a single-slot buffer at the sender and the receiver, is inefficient.

Figure 13.10 *Error control at the transport layer*

- **Error control**, unlike the **flow control**, involves only the sending and receiving transport layers

Error control at the transport layer is responsible to:

- 1- Detect and discard corrupted packets.
2. Keep track of lost and discarded packets and resend them.
3. Recognize duplicate packets and discard them.
4. Buffer out-of-order packets until the missing packets arrive.





Error control: Sequence Numbers

- Error control requires that the sending transport layer knows which packet is to be resent and packet is duplicate or out order.
- This can be done if the packets are numbered.
- We can add a field to the transport layer packet to hold the sequence number of the packets
- When a packet is corrupted or lost, the receiving transport layer can somehow inform the sending transport layer to resend that packet
- The out-of-order packets can be recognized by observing gaps in the sequence numbers.
- Packets are numbered sequentially

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

For example, if m is 4, the only sequence numbers are 0 through 15, inclusive.

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ...



Error control: Acknowledgment

- We can use both positive and negative signals as error control
- The receiver side can send an acknowledgement (ACK) for each or a collection of packets that have arrived correctly.
- The sender can detect lost packets if it uses a timer
- If an ACK does not arrive before the timer expires, the sender resends the packet
- Duplicate packets can be silently discarded by the receiver
- Out-of-order packets can be either discarded (to be treated as lost packets by the sender), or stored until the missing ones arrives.

- **Flow control** requires the use of two buffers, one at the sender site and the other at the receiver site
- **Error control** requires the use of sequence and acknowledgment numbers by both sides
- These two requirements can be combined if we use two numbered buffers at both sides

Figure 13.11 *Sliding window in circular format*

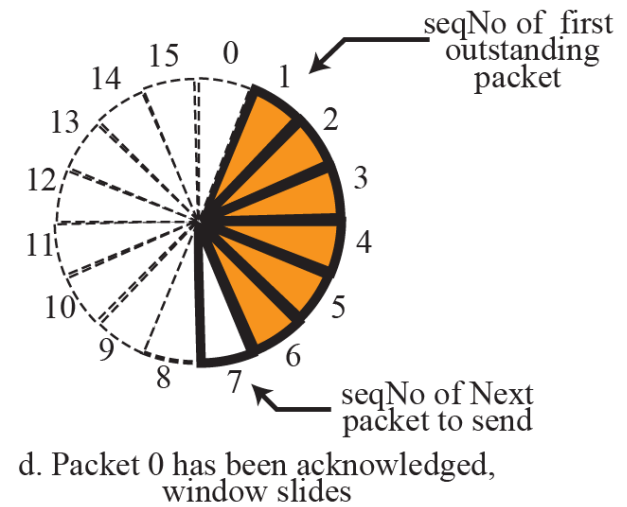
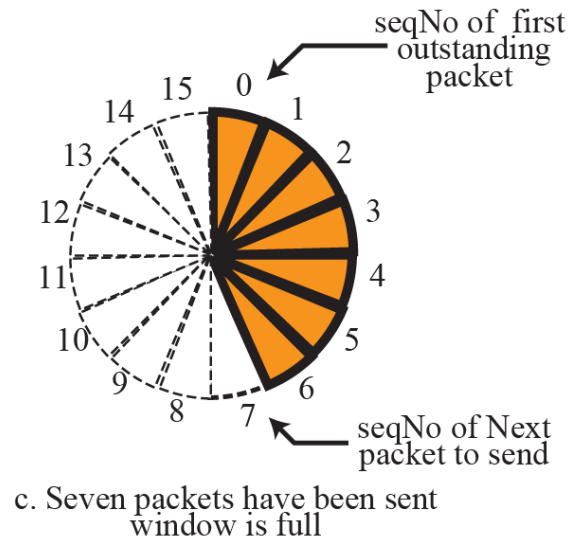
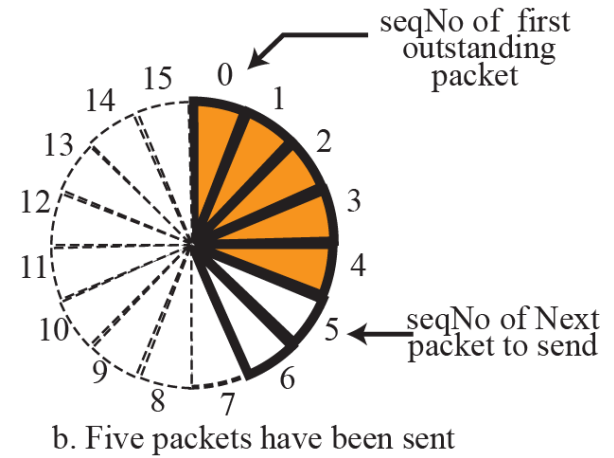
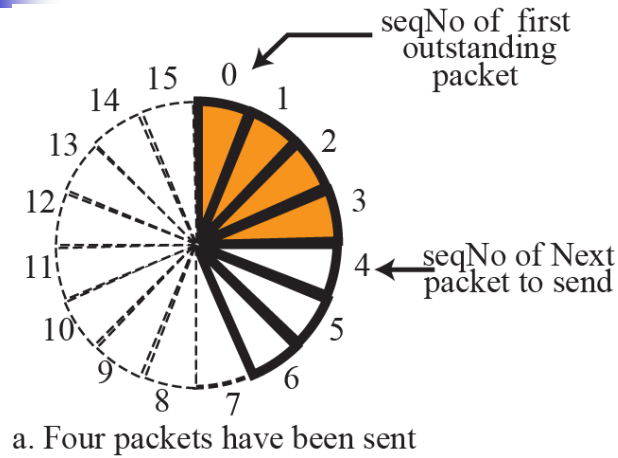


Figure 13.12 *Sliding window in linear format*



a. Four packets have been sent



b. Five packets have been sent



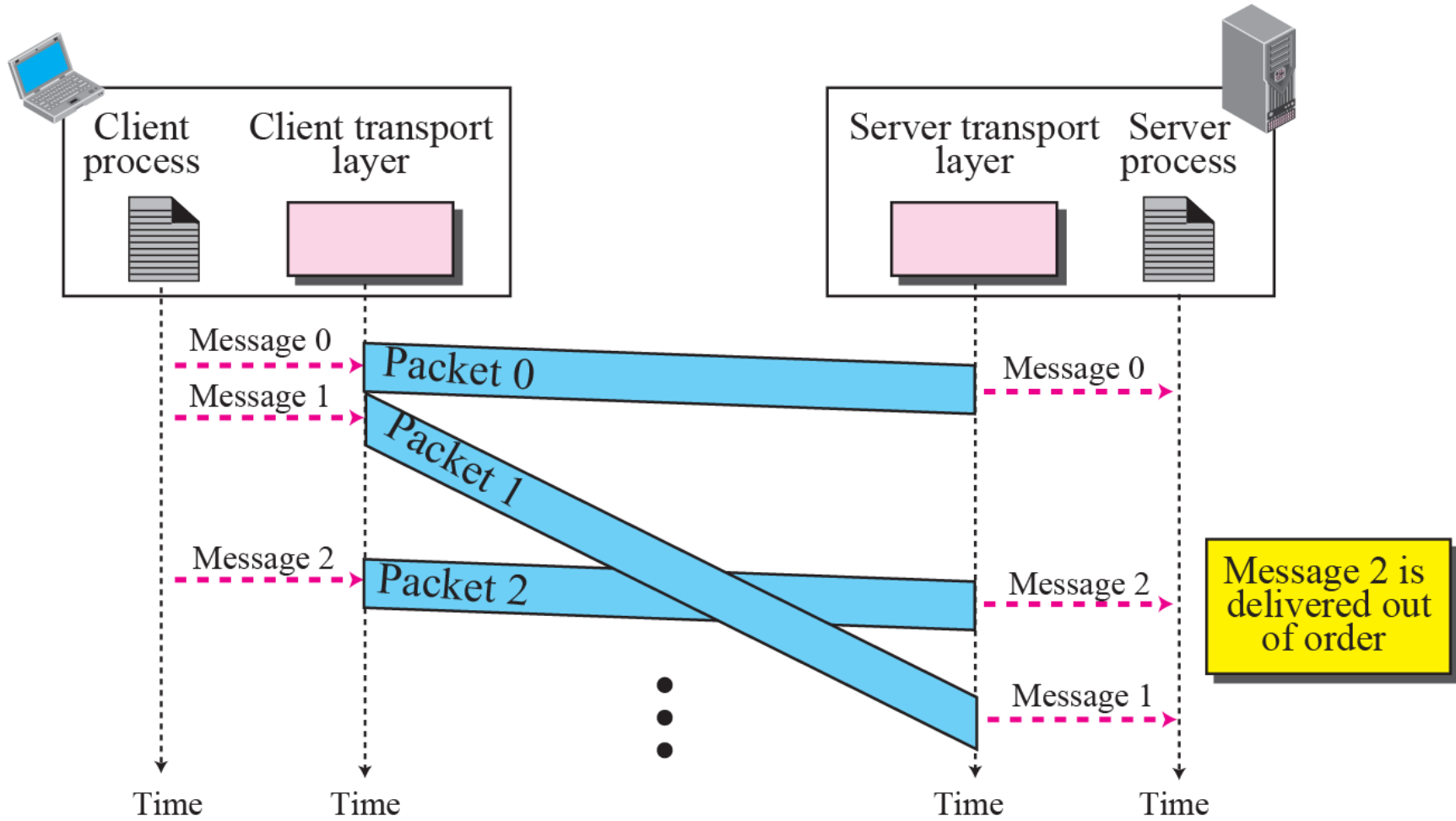
c. Seven packets have been sent
window is full



d. Packet 0 have been acknowledged
and window slid

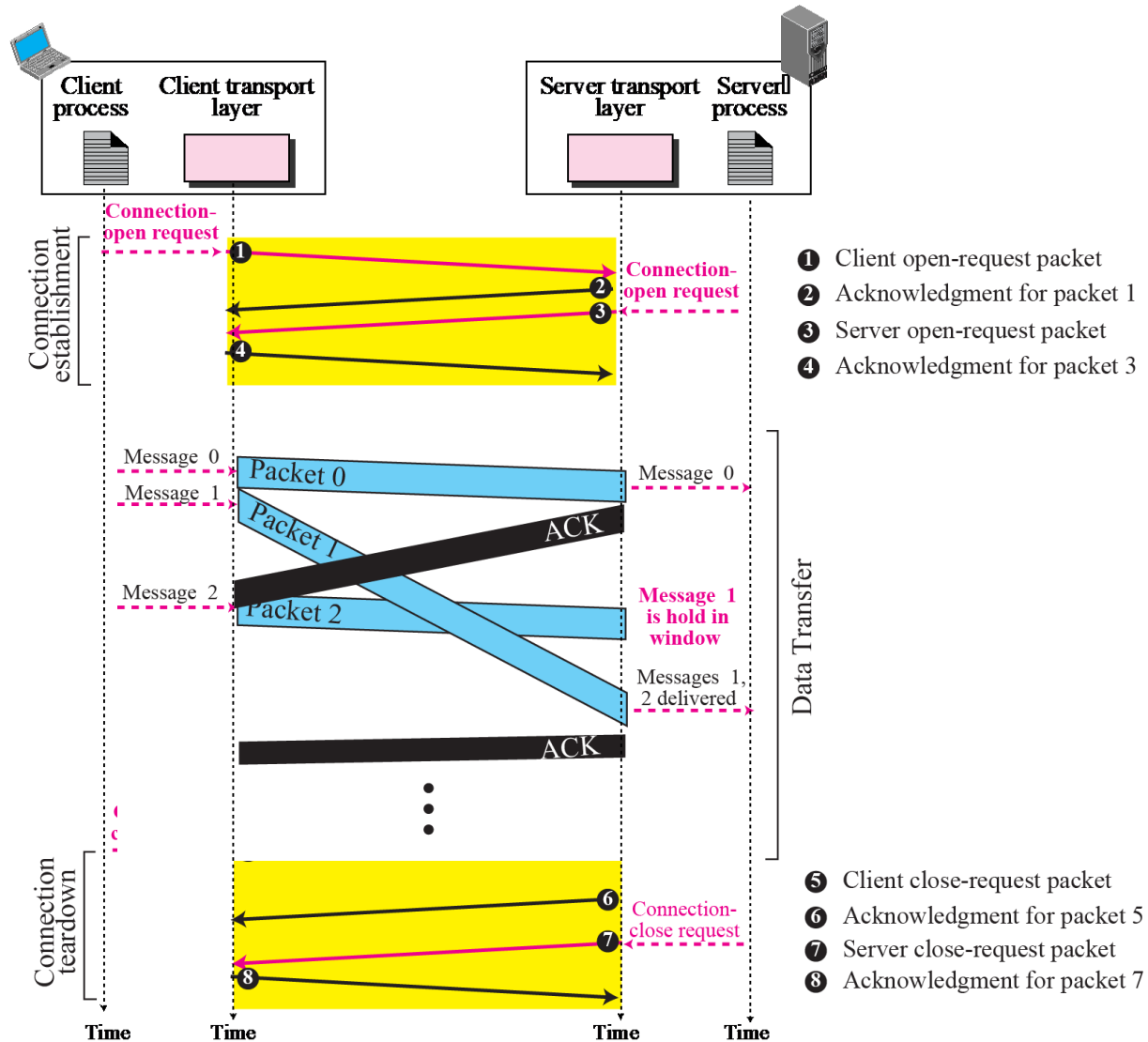
- The source process (application program) divide its message into chunks of data
- The transport layer treats each chunk as a single unit without any relation between the chunks
- The packets may arrive out of order at the destination and will be delivered out of order to the server process.
- The situation would be worse if one of the packets were lost
- The receiving transport layer has no idea that one of the messages has been lost (no numbering)
- No flow control, error control, or congestion control can be effectively implemented in a connectionless service

Figure 13.13 *Connectionless service*



- The client and the server first need to establish a connection between themselves
- Data exchange can only happen after the connection establishment
- We can implement flow control, error control, and congestion control in a connection-oriented protocol.

Figure 13.14 *Connection-oriented service*



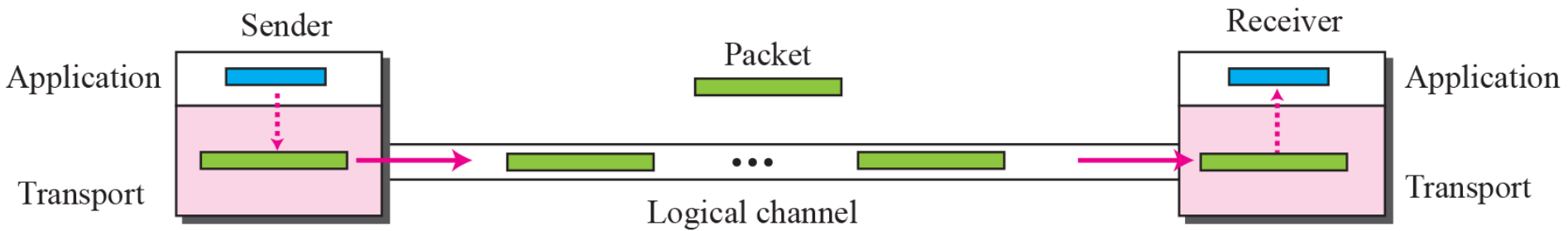
13-2 TRANSPORT-LAYER PROTOCOLS

We can create a transport-layer protocol by combining a set of services described in the previous sections. To better understand the behavior of these protocols, we start with the simplest one and gradually add more complexity. The TCP/IP protocol uses a transport layer protocol that is either a modification or a combination of some of these protocols.

Topics Discussed in the Section

- ✓ **Simple Protocol**
- ✓ **Stop-and-Wait Protocol**
- ✓ **Go-Back- N Protocol**
- ✓ **Selective-Repeat Protocol**

Figure 13.16 *Simple protocol*



- A connectionless protocol with neither flow nor error control
- We assume that the receiver can immediately handle any packet it receives
- The receiver can never be overwhelmed with incoming packets.



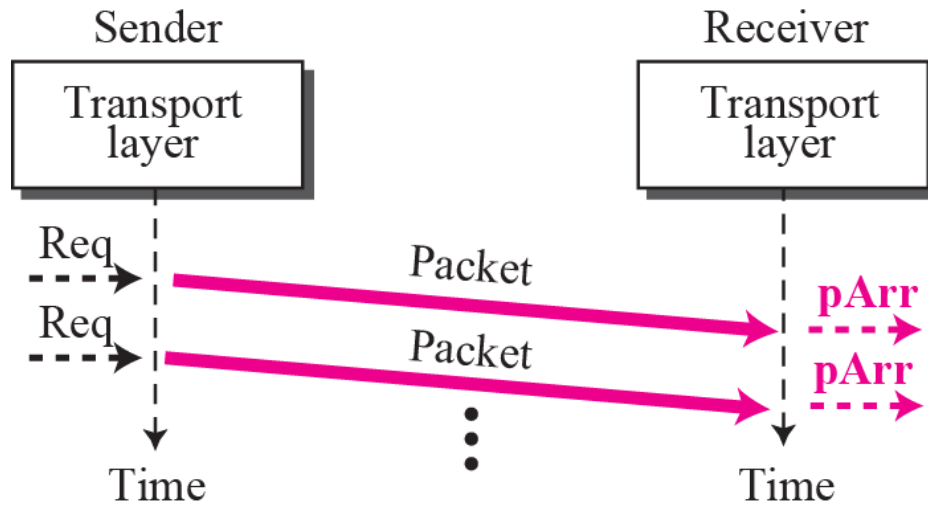
Note

The simple protocol is a connectionless protocol that provides neither flow nor error control.

Example 13.3

Figure 13.18 shows an example of communication using this protocol. It is very simple. The sender sends packets one after another without even thinking about the receiver.

Figure 13.18 *Example 13.3*

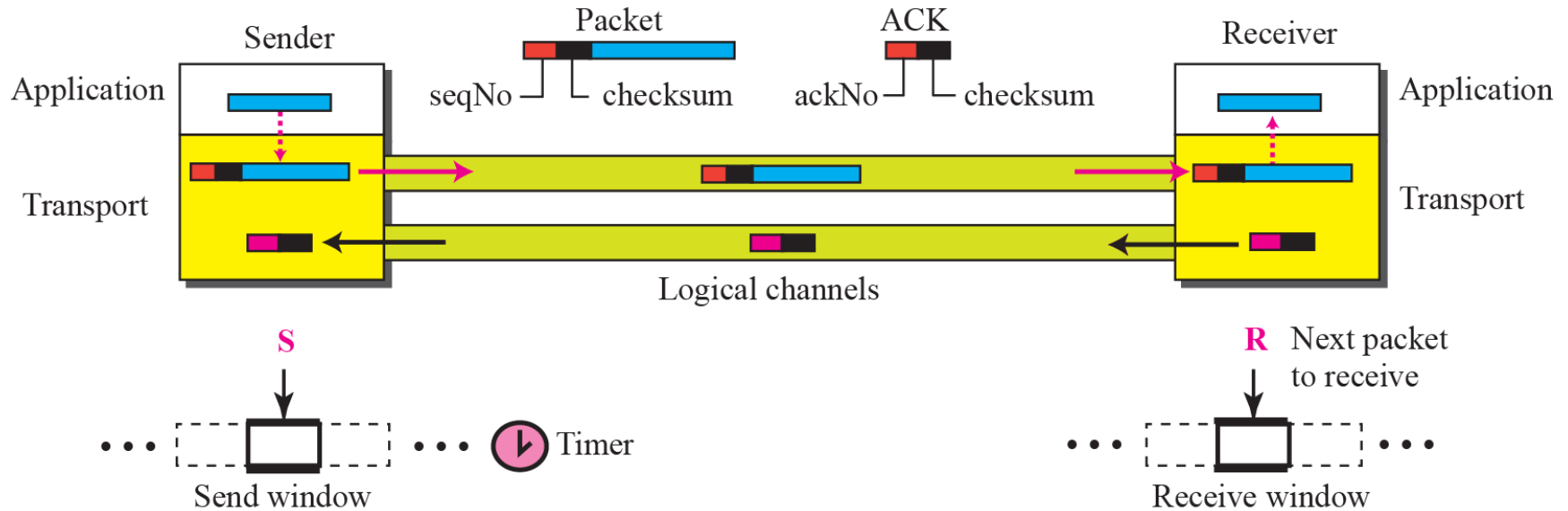


Events:

Req: Request came from process

pArr: Packet arrived

Figure 13.19 *Stop-and-wait protocol*




- Connection-oriented protocol, which uses both flow and error control
- Both the sender and the receiver use a sliding window of size 1
- The sender sends one packet at a time and waits for an acknowledgment before sending the next one.



Note

In Stop-and-Wait protocol, flow control is achieved by forcing the sender to wait for an acknowledgment, and error control is achieved by discarding corrupted packets and letting the sender resend previous packet when the timer expires.



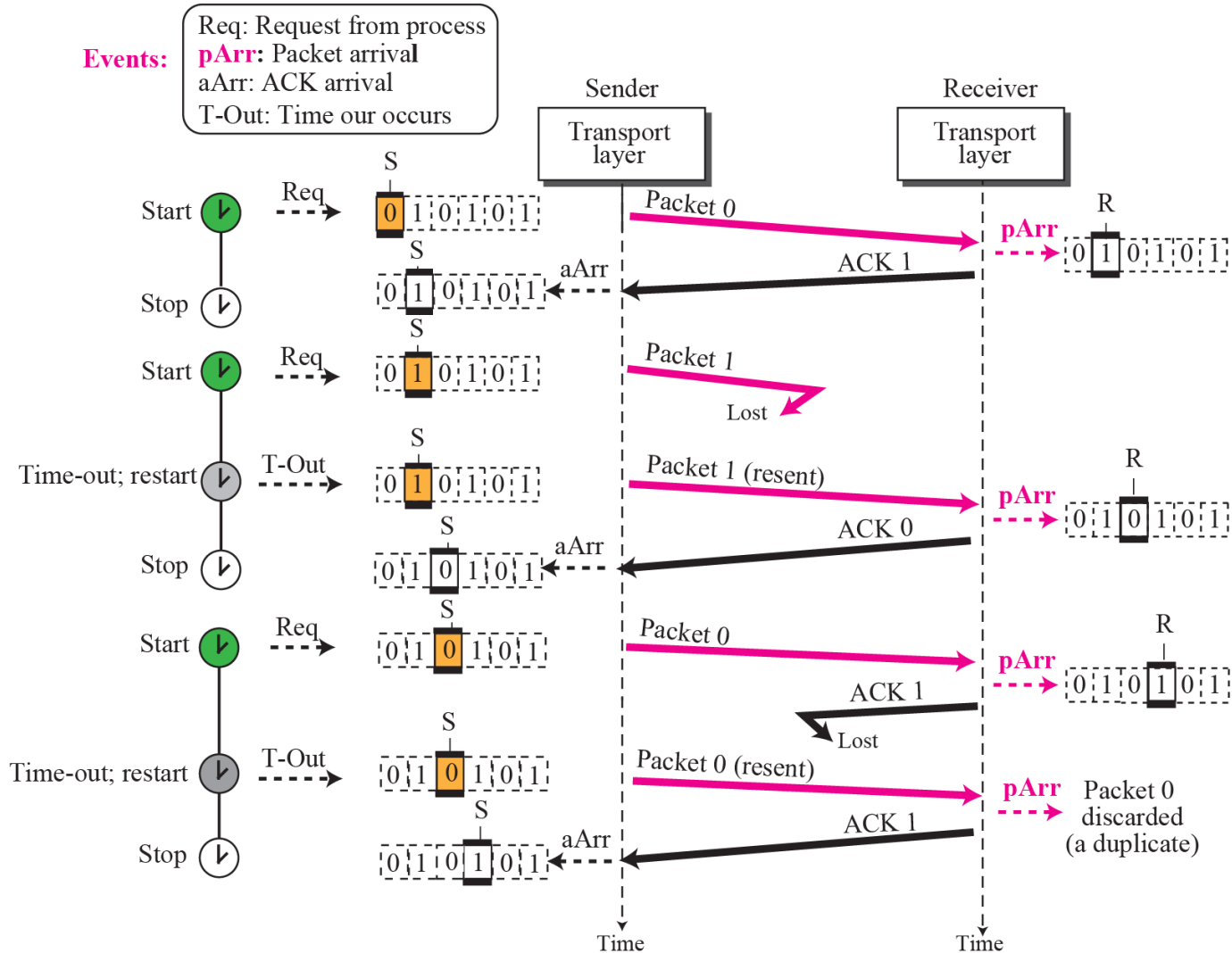
In the Stop-and-Wait protocol, we can use a 1-bit field to number the packets. The sequence numbers are based on modulo-2 arithmetic.

In the Stop-and-Wait protocol, the acknowledgment number is in modulo-2 arithmetic

Example 13.4

Figure 13.21 shows an example of Stop-and-Wait protocol. Packet 0 is sent and acknowledged. Packet 1 is lost and resent after the time-out. The resent packet 1 is acknowledged and the timer stops. Packet 0 is sent and acknowledged, but the acknowledgment is lost. The sender has no idea if the packet or the acknowledgment is lost, so after the time-out, it resends packet 0, which is acknowledged.

Figure 13.21 Example 13.4



Example 13.5

In a **Stop-and-Wait system**, the bandwidth of the line is **1 Mbps**, and 1 bit takes 20 milliseconds to make a round trip. What is the bandwidth-delay product? If the system data packets are 1,000 bits in length, what is the utilization percentage of the link?


Solution

The bandwidth-delay product is:

$$(1 \times 10^6) \times (20 \times 10^{-3}) = 20,000 \text{ bits.}$$

The system can send 20,000 bits during the time it takes for the data to go from the sender to the receiver and the acknowledgment to come back.

However, the system sends only 1,000 bits. We can say that the link utilization is only $1,000/20,000$, or 5 percent. **For this reason, for a link with a high bandwidth or long delay, the use of Stop-and-Wait wastes the capacity of the link.**



The Stop-and-Wait protocol is very inefficient if our channel is thick and long. By thick, we mean that our channel has a large bandwidth (high data rate); by long, we mean the round-trip delay is long.

Example 13.6

What is the utilization percentage of the link in Example 13.5 if we have a protocol that can send up to 15 packets before stopping and worrying about the acknowledgments?

Solution

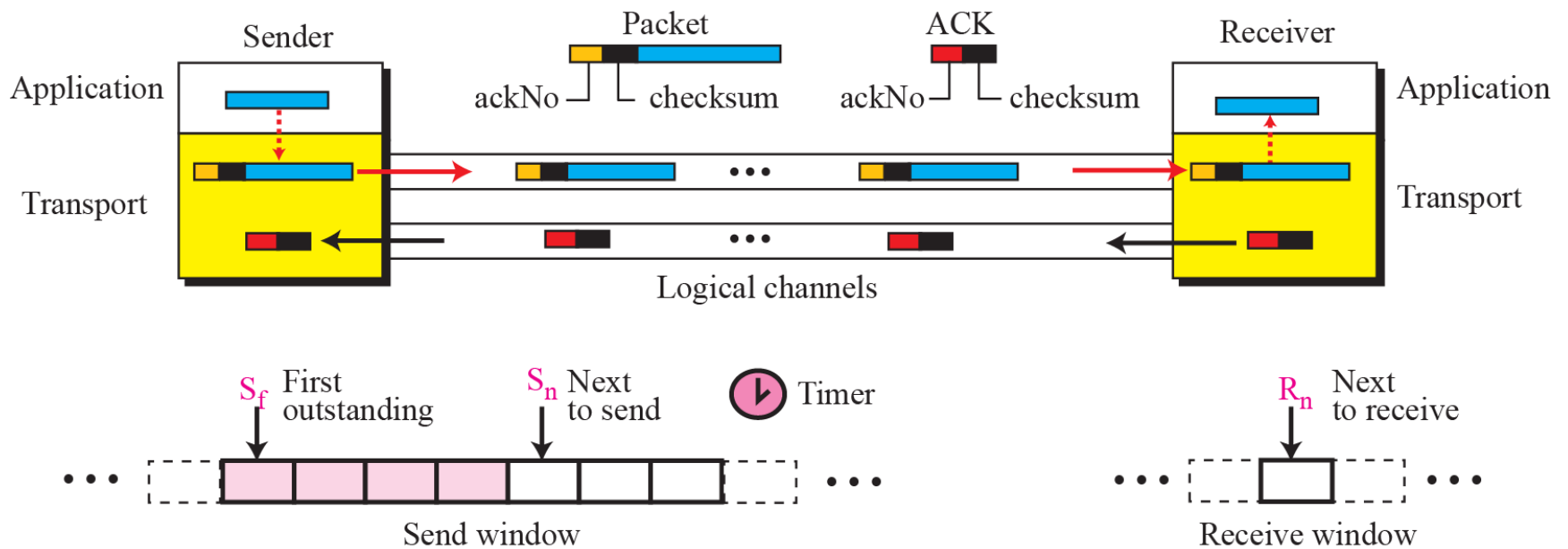
The bandwidth-delay product is still 20,000 bits. The system can send up to 15 packets or 15,000 (1 packet is 1000 bits) bits during a round trip. This means the utilization is $15,000/20,000$, or 75 percent. Of course, if there are damaged packets, the utilization percentage is much less because packets have to be resent.


To improve efficiency:

- **First:** multiple packets must be in transition while the sender is waiting for acknowledgment (Go-back-N protocol)
- **Second:** more than one packet be outstanding to keep the channel busy while the sender is waiting for acknowledgment (Selective-Repeat (SR) protocol)

- The key to **Go-back-N** is that we can send several packets before receiving acknowledgments
- The receiver can only buffer one packet
- keep a copy of the sent packets until the acknowledgments arrive
- Several data packets and acknowledgments can be in the channel at the same time.


Figure 13.22 *Go-Back-N protocol*





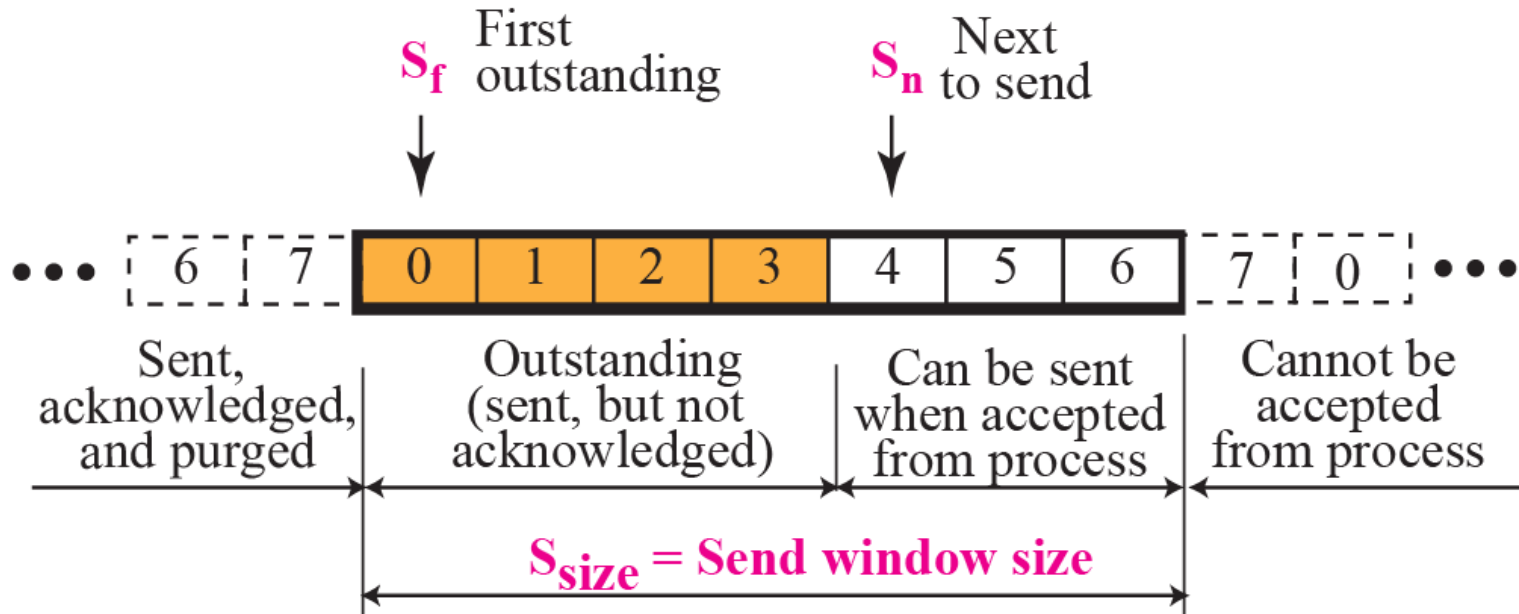
In the Go-Back-N Protocol, the sequence numbers are modulo 2^m , where m is the size of the sequence number field in bits.

In the Go-Back-N protocol, the acknowledgment number is cumulative and defines the sequence number of the next packet expected to arrive.




For example, if the acknowledgment number (ackNo) is 7, it means all packets with sequence number up to 6 have arrived, safe and sound, and the receiver is expecting the packet with sequence number 7

Figure 13.23 *Send window for Go-Back-N*



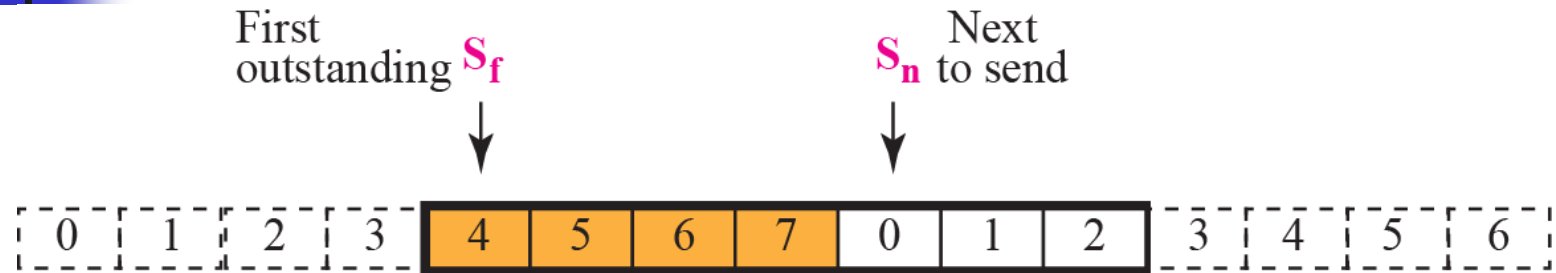
The **outstanding region** is a range of sequence numbers belonging to the packets that are sent, but have an unknown status. The sender needs to wait to find out if these packets have been received or were lost



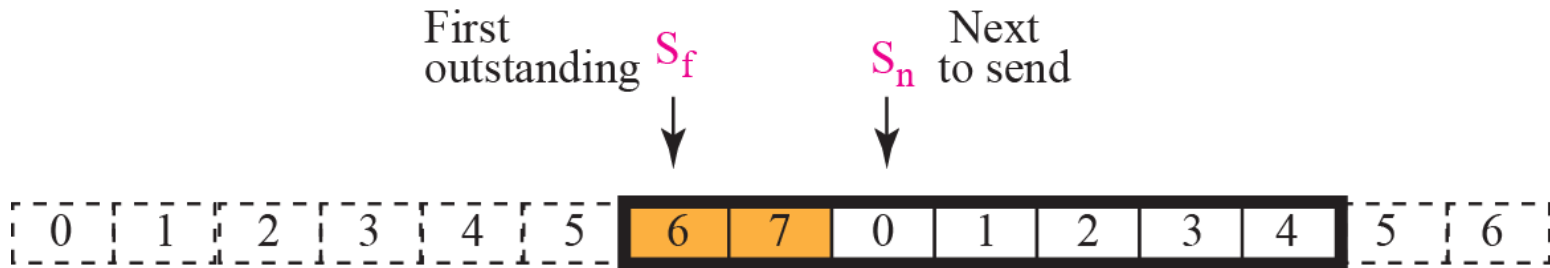
The send window is an abstract concept defining an imaginary box of maximum size = $2^m - 1$ with three variables: S_f , S_n , and S_{size} .

The send window can slide one or more slots when an error-free ACK with ackNo between S_f and S_n arrives.

Figure 13.24 *Sliding the send window*



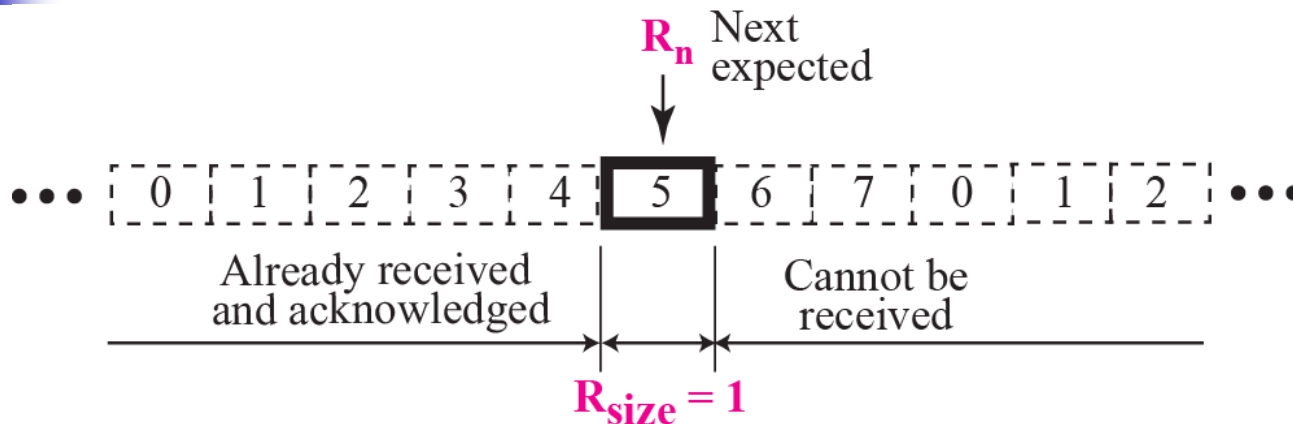
a. Window before sliding



b. Window after sliding (an ACK with ackNo = 6 has arrived)

ackNo = 6 has arrived. This means that the receiver is waiting for packets with sequence number 6.

Figure 13.25 *Receive window for Go-Back-N*



- The size of the receive window is always 1.
- The receiver is always looking for the arrival of a specific packet (R_n). Any packet arriving out of order is discarded and needs to be resent.
- The sequence numbers to the left of the window belong to the packets already received and acknowledged; the sequence numbers to the right of this window define the packets that cannot be received

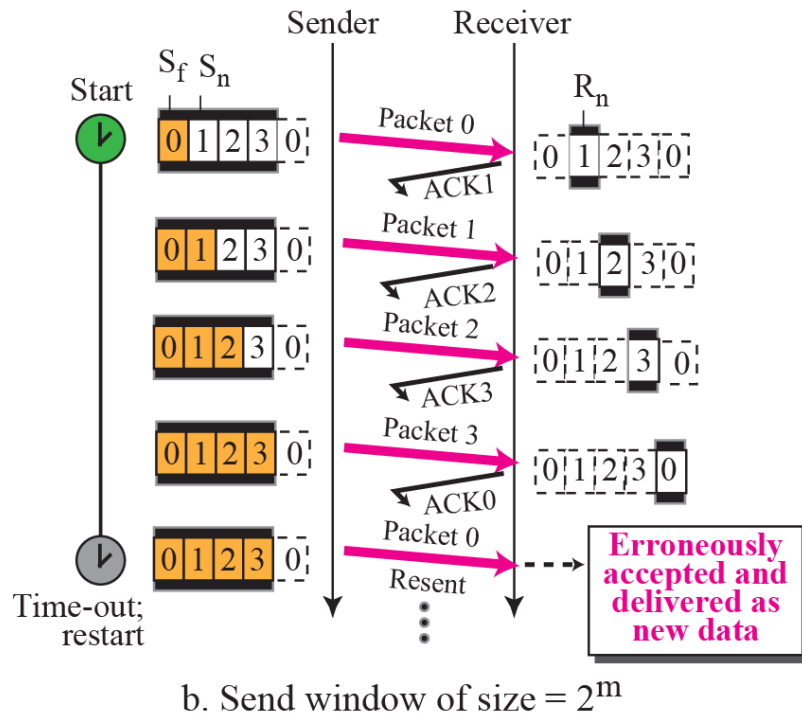
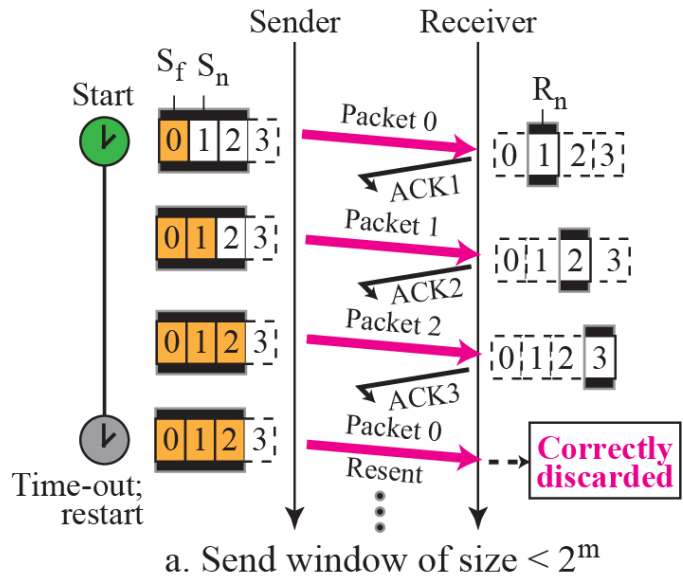


Note

The receive window is an abstract concept defining an imaginary box of size 1 with one single variable R_n .

The window slides when a correct packet has arrived; sliding occurs one slot at a time.

Figure 13.27 *Send window size for Go-Back-N*

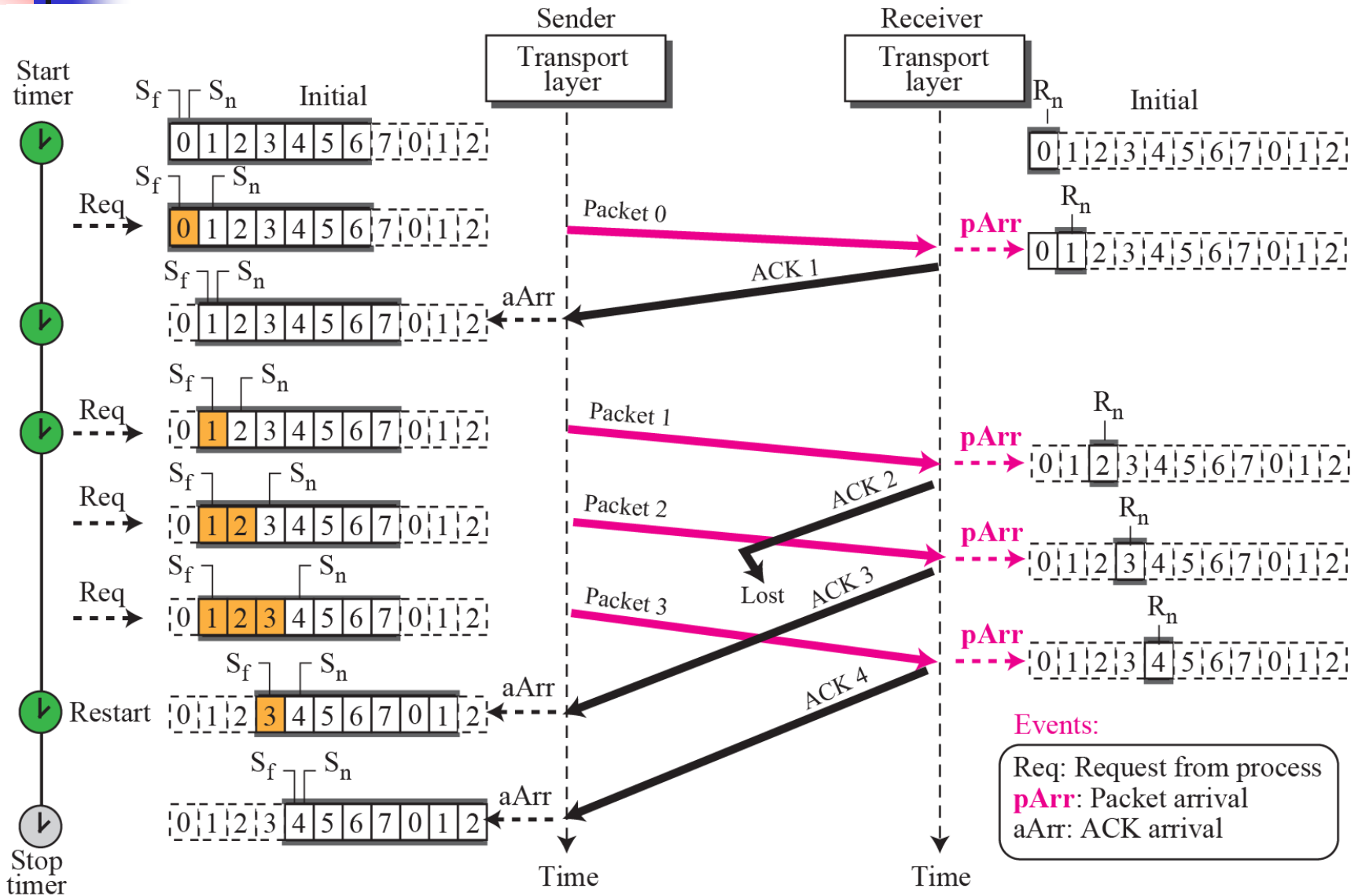




Note

In the Go-Back-N protocol, the size of the send window must be less than 2^m ; the size of the receive window is always 1.

Figure 13.28 Example 13.7



Example 13.7 explanation

No data packets are lost, but some ACKs are delayed and one is lost. The example also shows how **cumulative acknowledgments** can help if acknowledgments are delayed or lost.

There is **no time-out event** here because all outstanding packets are acknowledged before the timer expires. Note that although ACK 2 is lost, ACK 3 is cumulative and serves as both ACK 2 and ACK 3. There are four events at the receiver site.

Example 13.8

Figure 13.29 shows what happens when a packet is lost. Packets 0, 1, 2, and 3 are sent. **However, packet 1 is lost.** The receiver receives packets 2 and 3, but they are discarded because they are received out of order (packet 1 is expected). When the receiver receives packets 2 and 3, it sends ACK1 to show that it expects to receive packet 1.

However, these ACKs are not useful for the sender because the `ackNo` is equal S_f , not greater than S_f . So the sender discards them. When the time-out occurs, the sender resends packets 1, 2, and 3, which are acknowledged..

Figure 13.29 *Example 13.8*

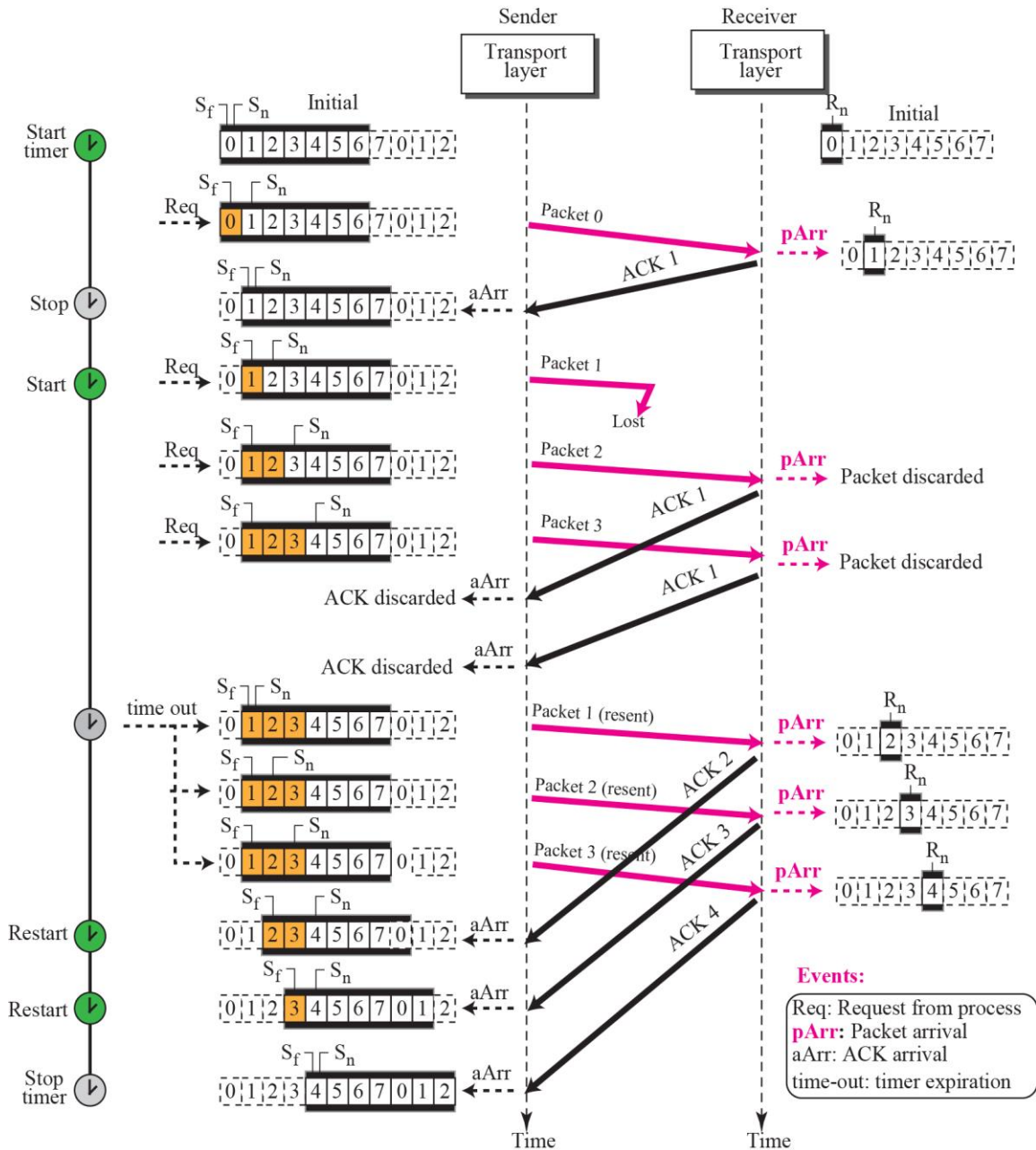
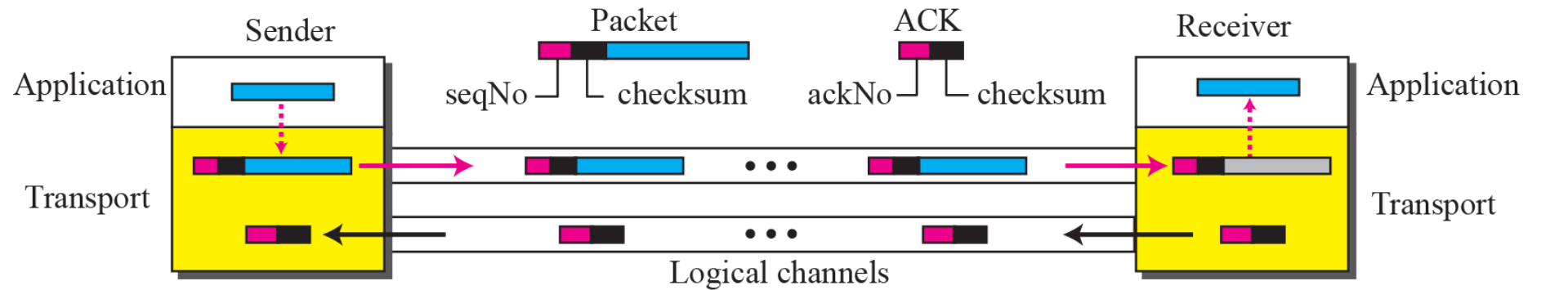


Figure 13.30 *Outline of Selective-Repeat*



- Sent, but not acknowledged
- Acknowledged out of order



- Packet received out of order

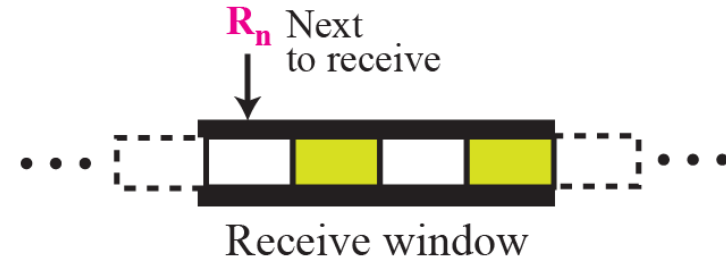
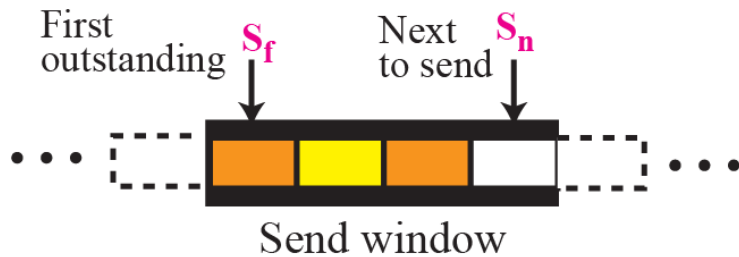


Figure 13.31 *Send window for Selective-Repeat protocol*

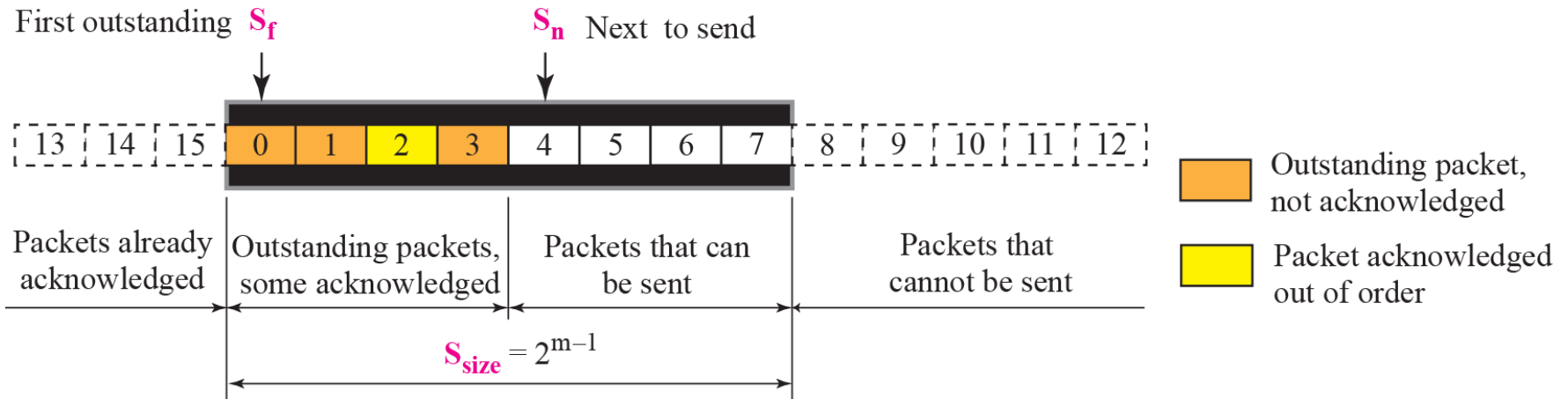
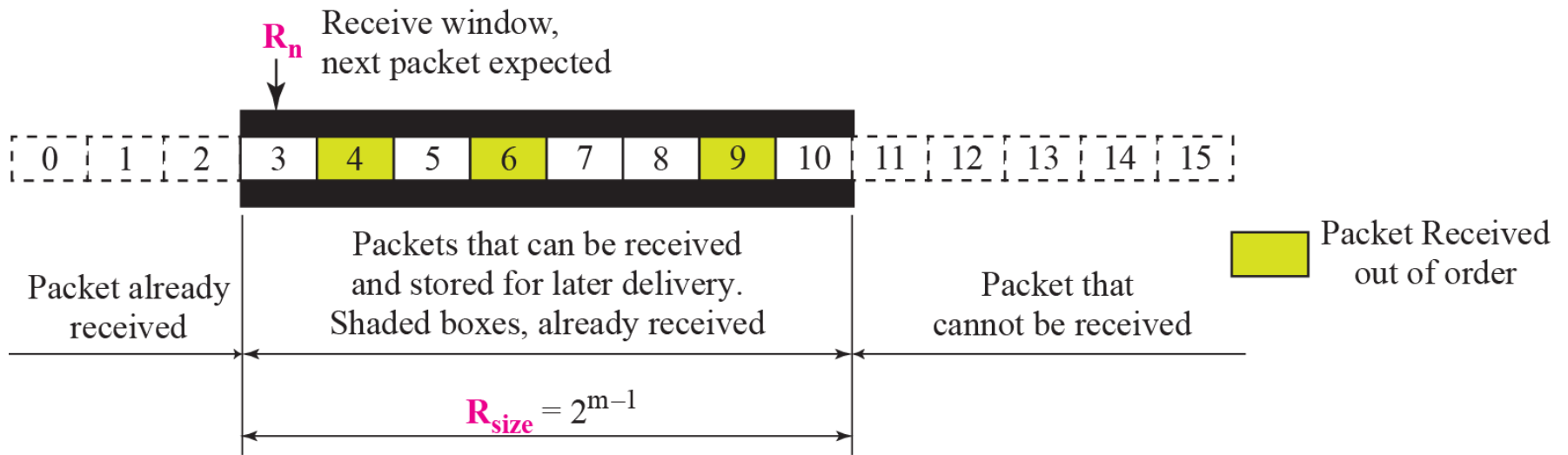


Figure 13.32 *Receive window for Selective-Repeat protocol*





Note

In the Selective-Repeat protocol, an acknowledgment number defines the sequence number of the error-free packet received.

Example 13.9

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

Solution

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

Example 13.10

This example is similar to Example 3.8 (Figure 13.29) in which packet 1 is lost. We show how Selective-Repeat behaves in this case. Figure 13.34 shows the situation. At the sender, packet 0 is transmitted and acknowledged. Packet 1 is lost. Packets 2 and 3 arrive out of order and are acknowledged. When the timer times out, packet 1 (the only unacknowledged packet) is resent and is acknowledged. The send window then slides.

Figure 13.34 Example 13.10

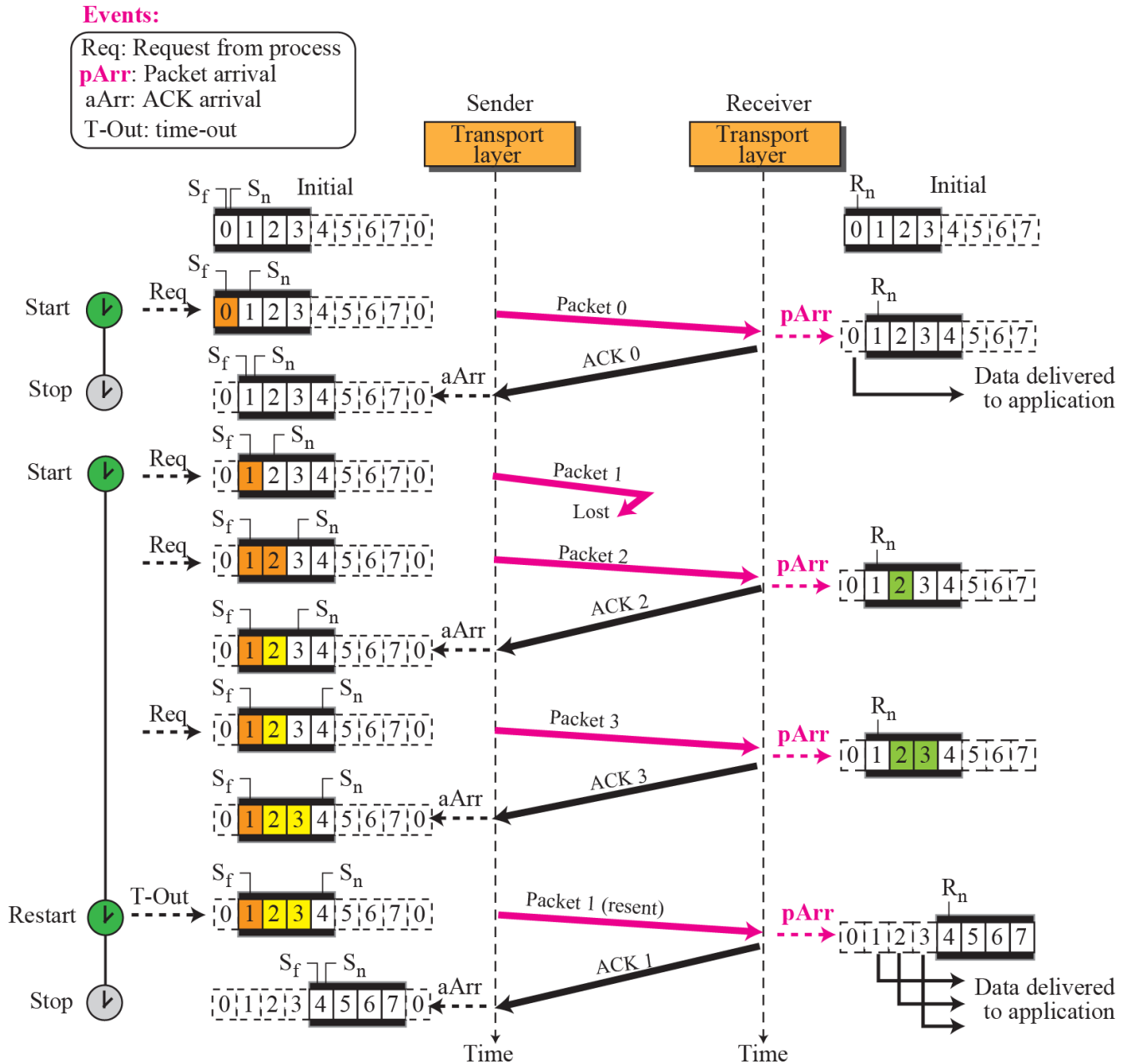
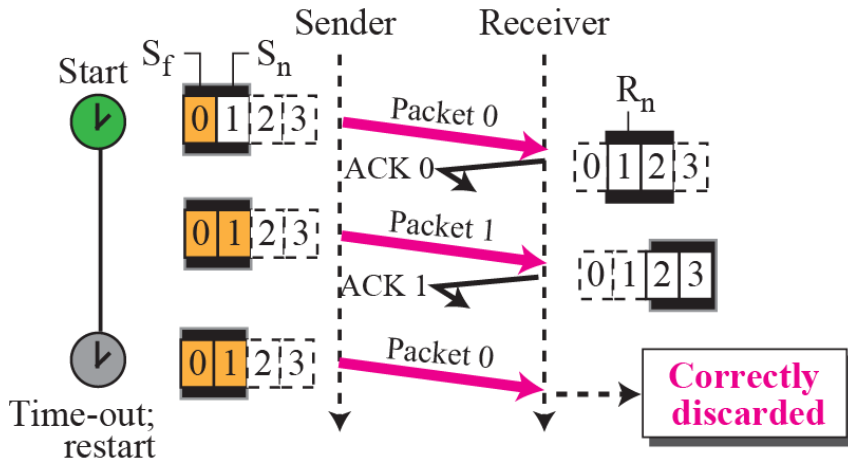
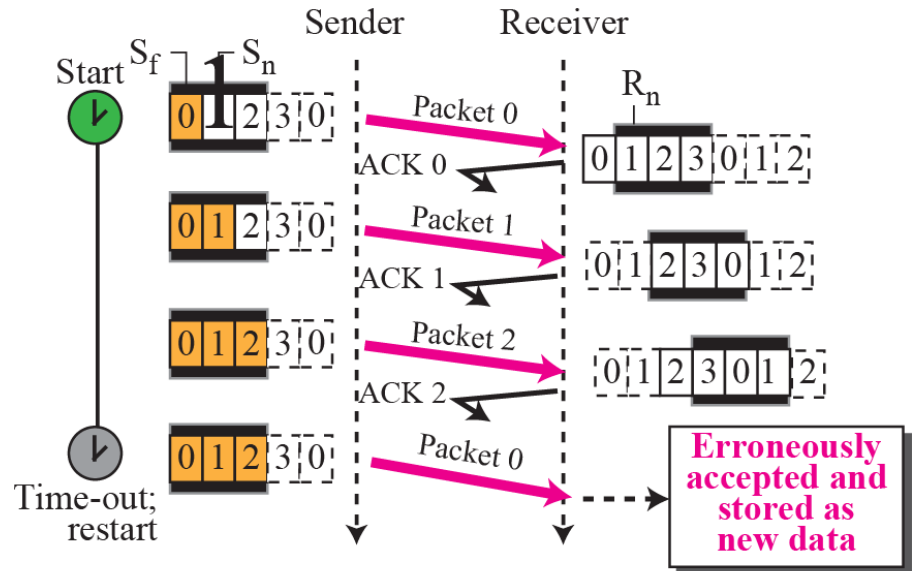


Figure 13.35 *Selective-Repeat window size*



a. Send and receive windows of size = 2^{m-1}



b. Send and receive windows of size $> 2^{m-1}$



Note

In Selective-Repeat, the size of the sender and receiver window can be at most one-half of $2m$.

Chapter 14

User Datagram Protocol (UDP)

Edited & Presented by:

Dr. Mohammad Alhammouri

TCP/IP Protocol Suite (B A. Forouzan)

OBJECTIVES:

- ❑ To introduce UDP and show its relationship to other protocols in the TCP/IP protocol suite.**
- ❑ To explain the format of a UDP packet and discuss the use of each field in the header.**
- ❑ To discuss the services provided by the UDP such as process-to-process delivery, multiplexing/demultiplexing, and queuing.**
- ❑ To show how to calculate the optional checksum and the sender the needs to add a pseudoheader to the packet when calculating the checksum.**
- ❑ To discuss how some application programs can benefit from the simplicity of UDP.**
- ❑ To briefly discuss the structure of the UDP package.**

Chapter Outline

14.1 Introduction

14.2 User Datagram

14.3 UDP Services

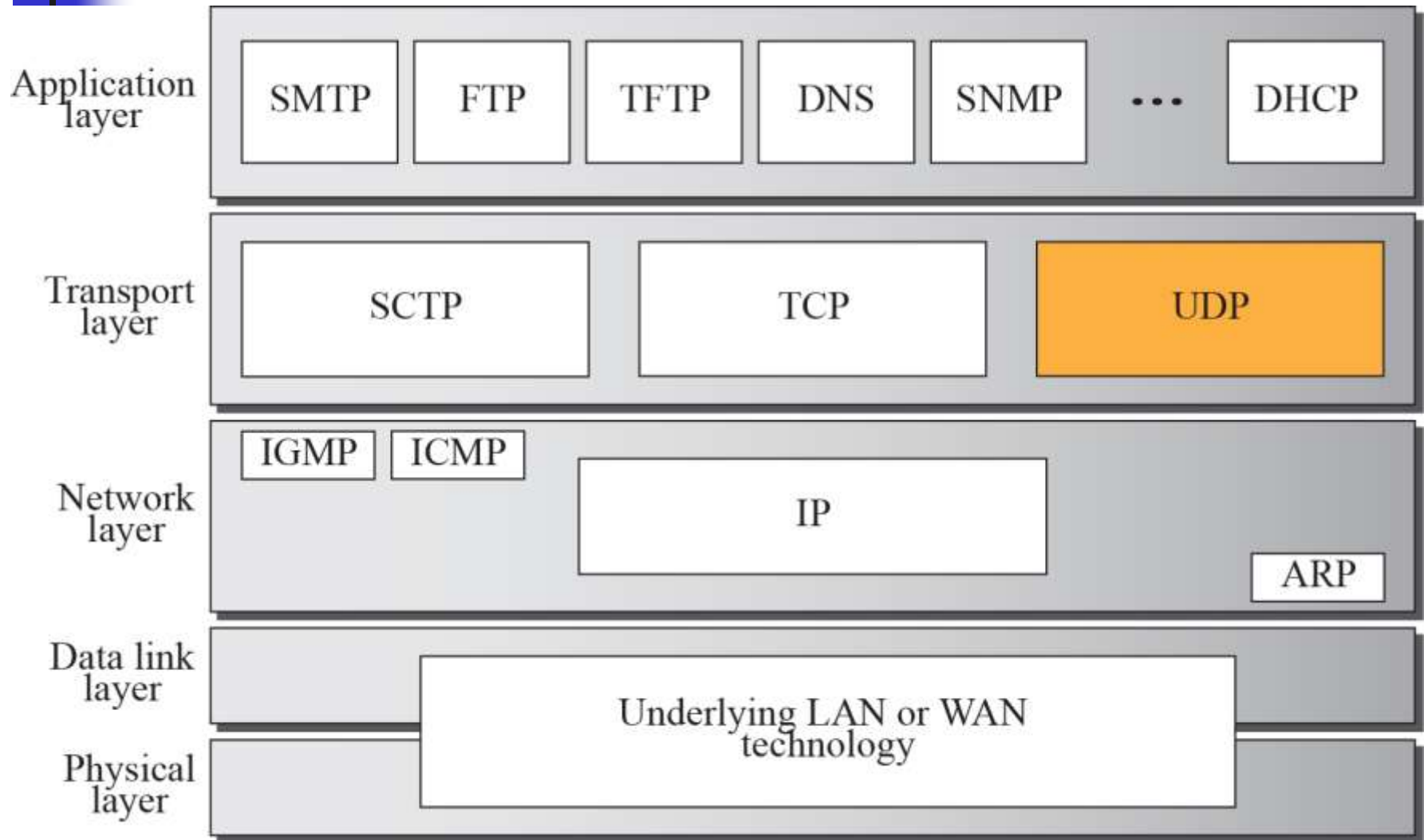
14.4 UDP Application

14.5 UDP Package

14-1 INTRODUCTION

Figure 14.1 shows the relationship of the User Datagram Protocol (UDP) to the other protocols and layers of the TCP/IP protocol suite: UDP is located between the application layer and the IP layer, and serves as the intermediary between the application programs and the network operations.

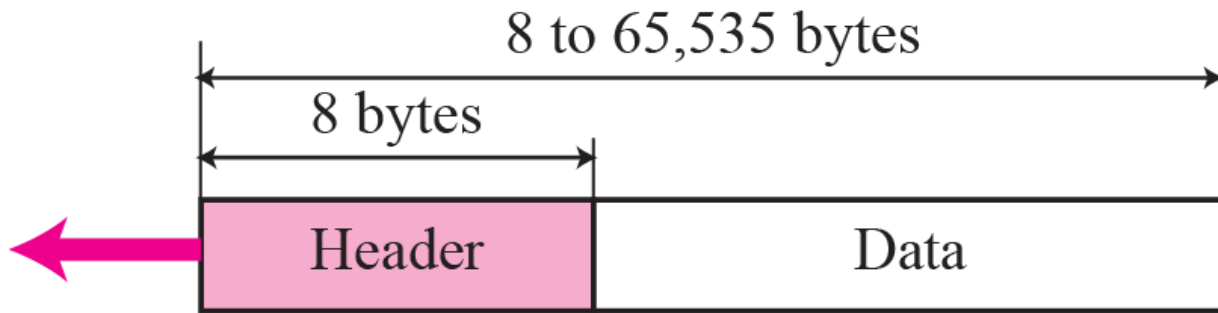
Figure 14.1 *Position of UDP in the TCP/IP protocol suite*



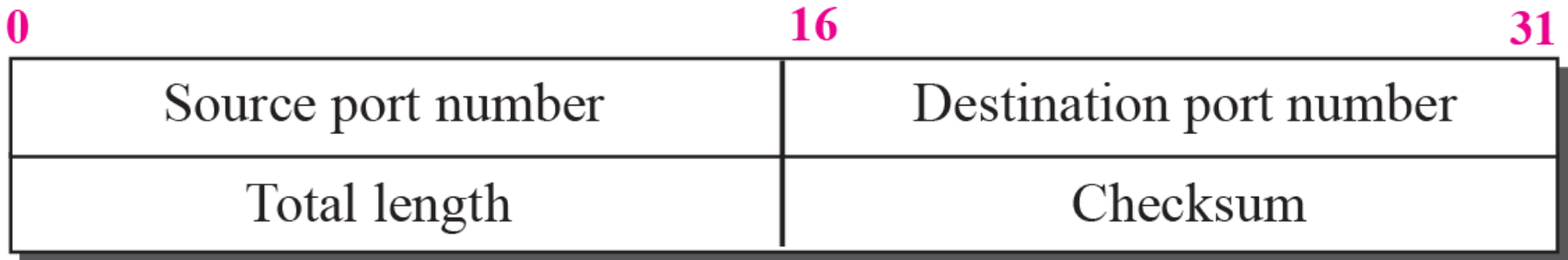
14-2 USER DATAGRAM

UDP packets, called user datagrams, have a fixed-size header of 8 bytes. Figure 14.2 shows the format of a user datagram.

Figure 14.2 *User datagram 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?

Example 14.1 *Continued*

Solution

- a. The source port number is the first four hexadecimal digits $(CB84)_{16}$ or 52100.
- b. The destination port number is the second four hexadecimal digits $(000D)_{16}$ or 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).

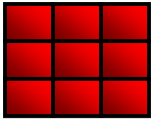


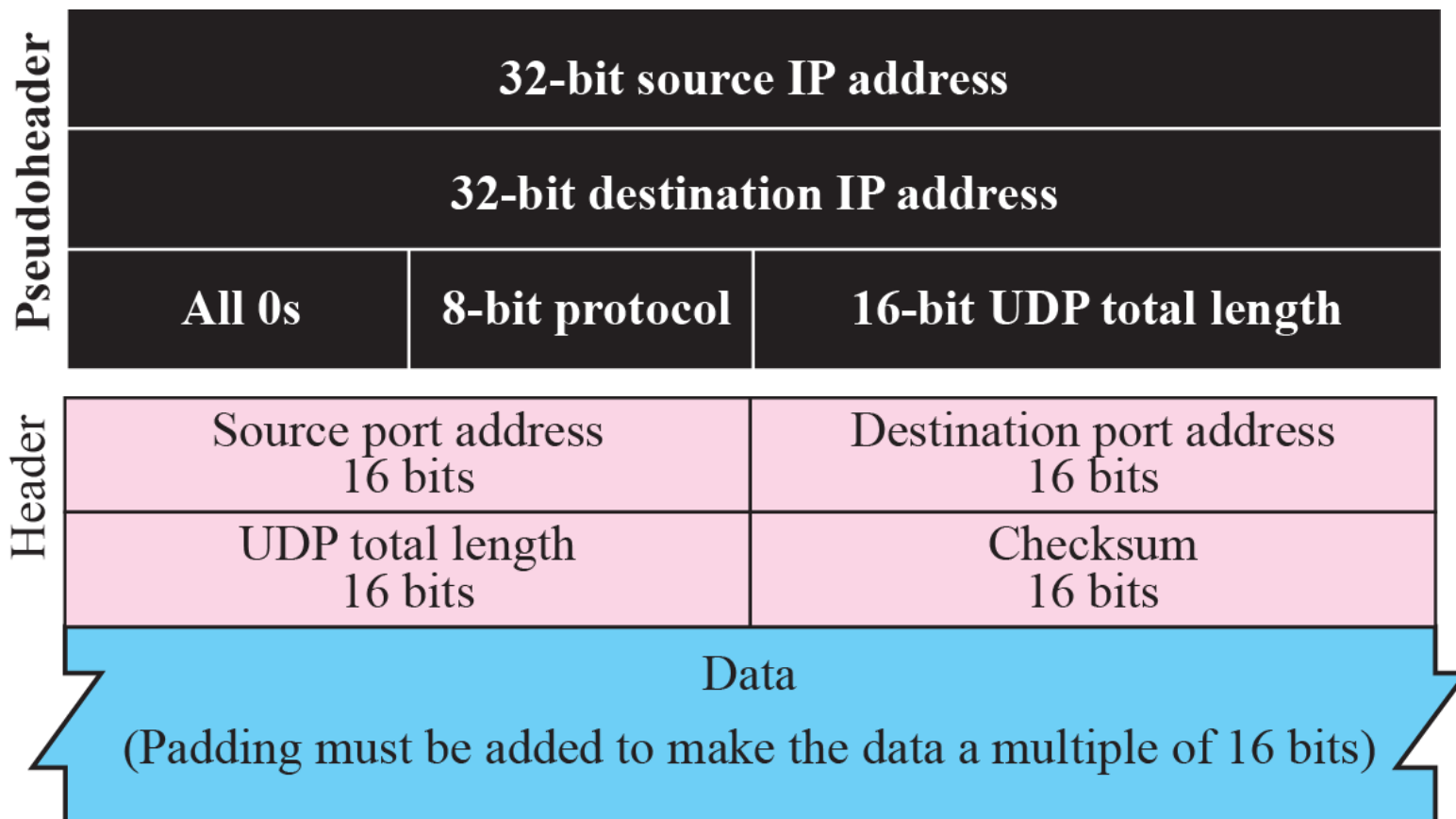
Table 14.1 *Well-known Ports used with UDP*

<i>Port</i>	<i>Protocol</i>	<i>Description</i>
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)

14-3 UDP Services

We discussed the general services provided by a transport layer protocol in Chapter 13. In this section, we discuss what portions of those general services are provided by UDP.

Figure 14.3 *Pseudoheader for checksum calculation*



Example 14.2

Figure 14.4 shows the checksum calculation for a very small user datagram with only 7 bytes of data. Because the number of bytes of data is odd, padding is added for checksum calculation. The pseudoheader as well as the padding will be dropped when the user datagram is delivered to IP (see Appendix F).

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

153.18.8.105			
171.2.14.10			
All 0s	17	15	
1087		13	
15		All 0s	
T	E	S	T
I	N	G	Pad

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)
<hr/>			
10010110	11101011	→	Sum
01101001	00010100	→	Checksum

Example 14.3

What value is sent for the checksum in one of the following hypothetical situations?

- a.** The sender decides not to include the checksum.
- b.** The sender decides to include the checksum, but the value of the sum is all 1s.
- c.** The sender decides to include the checksum, but the value of the sum is all 0s.

Example 14.3 *Continued*

Solution

- a.** The value sent for the checksum field is all 0s to show that the checksum is not calculated.

- b.** When the sender complements the sum, the result is all 0s; the sender complements the result again before sending. The value sent for the checksum is all 1s. The second complement operation is needed to avoid confusion with the case in part a.

- c.** This situation never happens because it implies that the value of every term included in the calculation of the sum is all 0s, which is impossible; some fields in the pseudoheader have nonzero values (see Appendix D).

Figure 14.5 *Encapsulation and decapsulation*

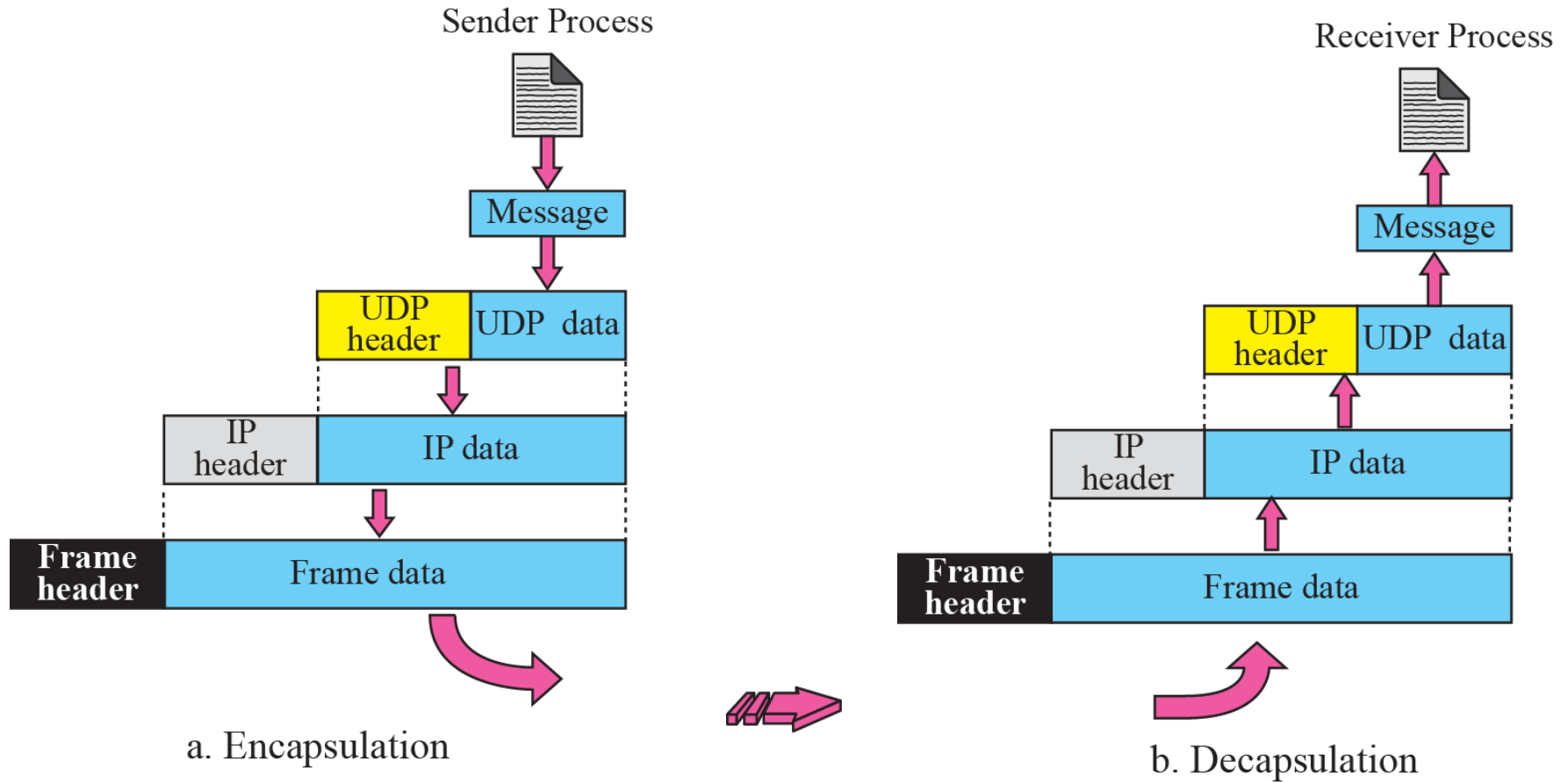


Figure 14.6 *Queues in UDP*

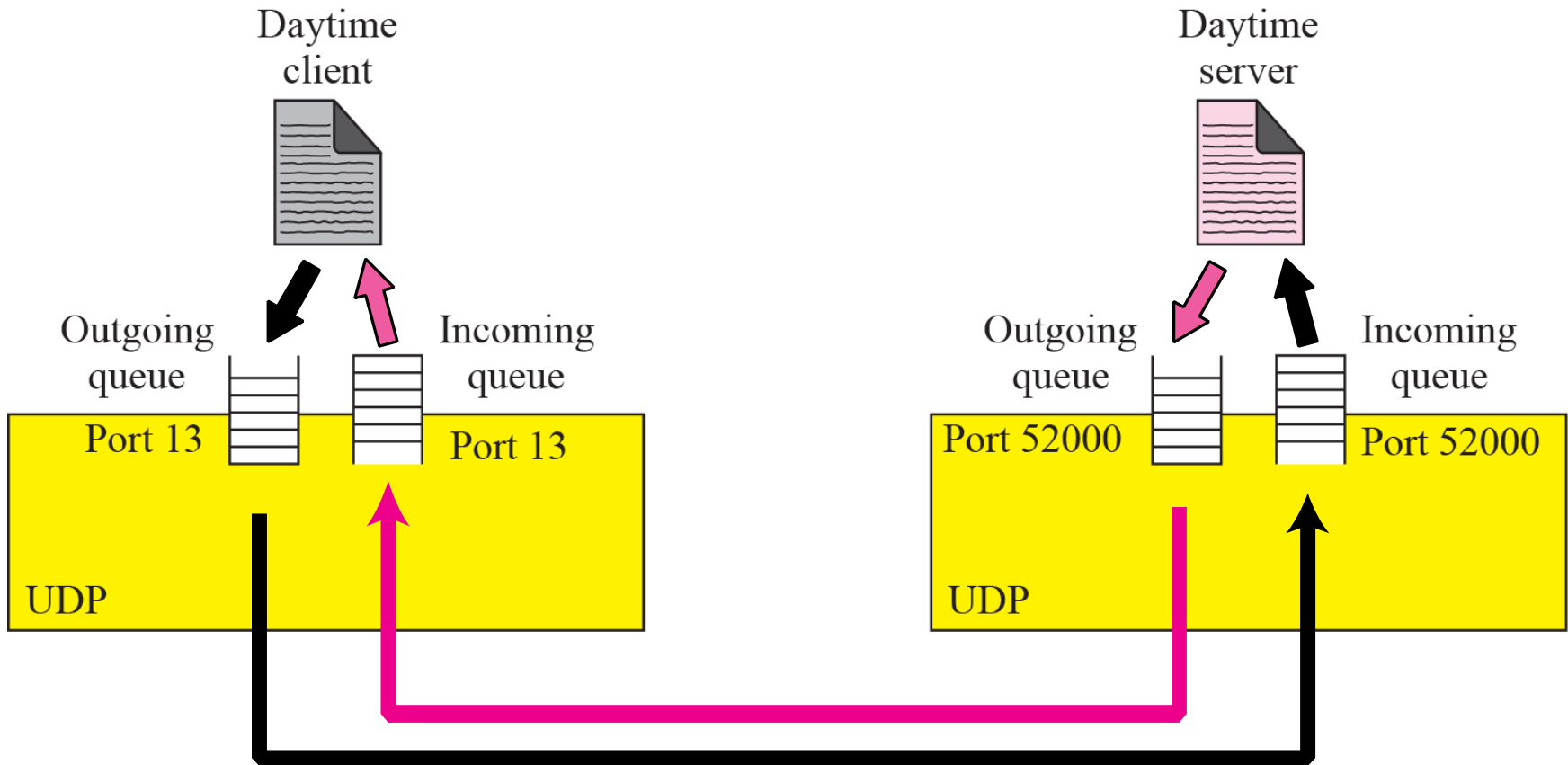
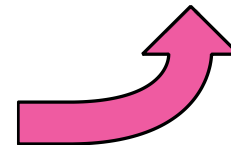
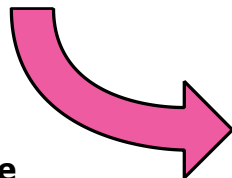
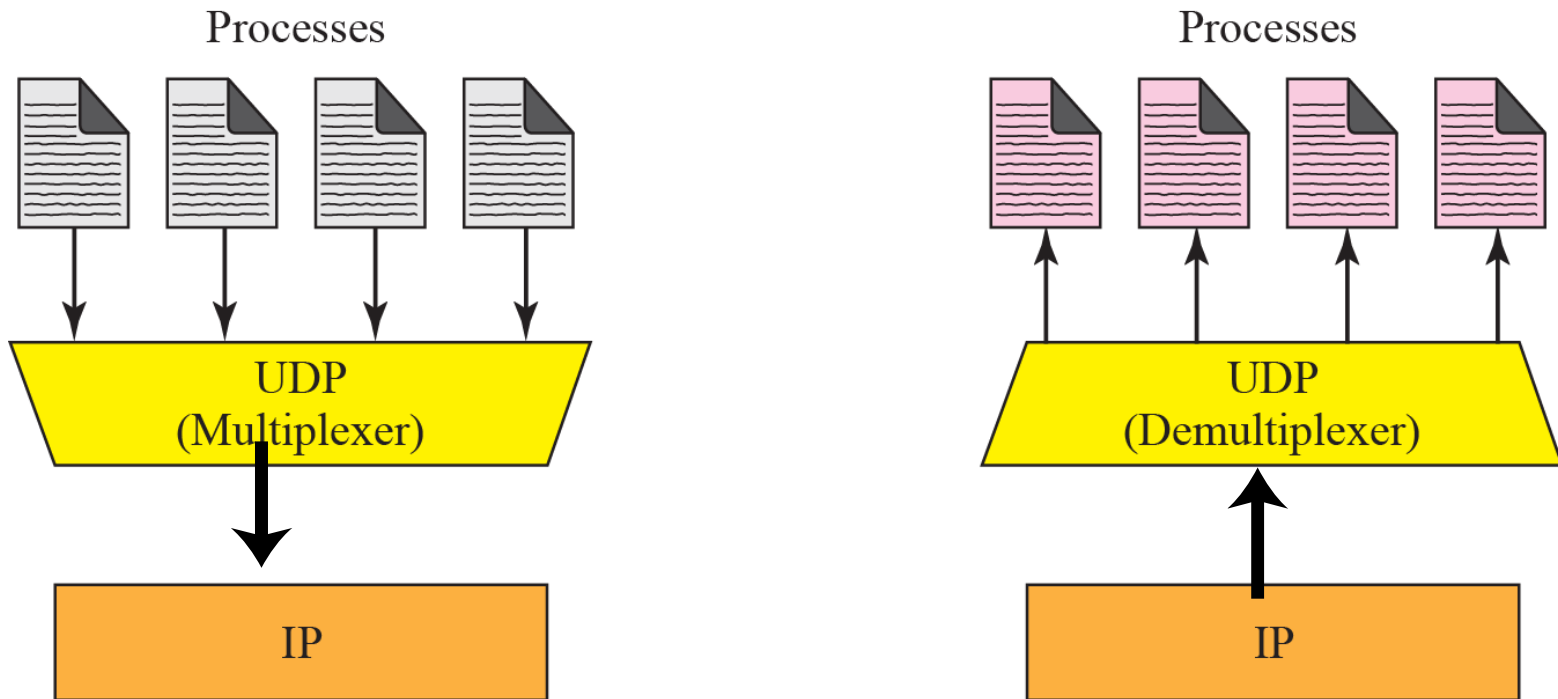


Figure 14.7 *Multiplexing and demultiplexing*

**Several processes that need to send user datagrams.
However, there is only one UDP**





Note

UDP is an example of the connectionless simple protocol we discussed in Chapter 13 with the exception of an optional checksum added to packets for error detection.

14-4 UDP APPLICATION

Although UDP meets almost none of the criteria we mentioned in Chapter 13 for a reliable transport-layer protocol, UDP is preferable for some applications. An application designer needs sometimes to compromise to get the optimum.

For example: **The connectionless service** provides less delay; the connection-oriented service creates more delay. If delay is an important issue for the application, the connectionless service is preferred.

Example 14.4

A client–server application such as DNS (see Chapter 19) 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.

Example 14.5

A client-server application such as SMTP (see Chapter 23), 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.

Example 14.6

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 delivery of the parts are not an overriding 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.

Example 14.7

Assume we are watching a real-time stream video on our computer. Such a program is considered a long file; it is divided into many small parts and broadcast in real time. The parts of the message are 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 the time, which most viewers do not even notice. However, video cannot be viewed out of order, so streaming audio, video, and voice applications that run over UDP must reorder or drop frames that are out of sequence.

14-5 UDP PACKAGE

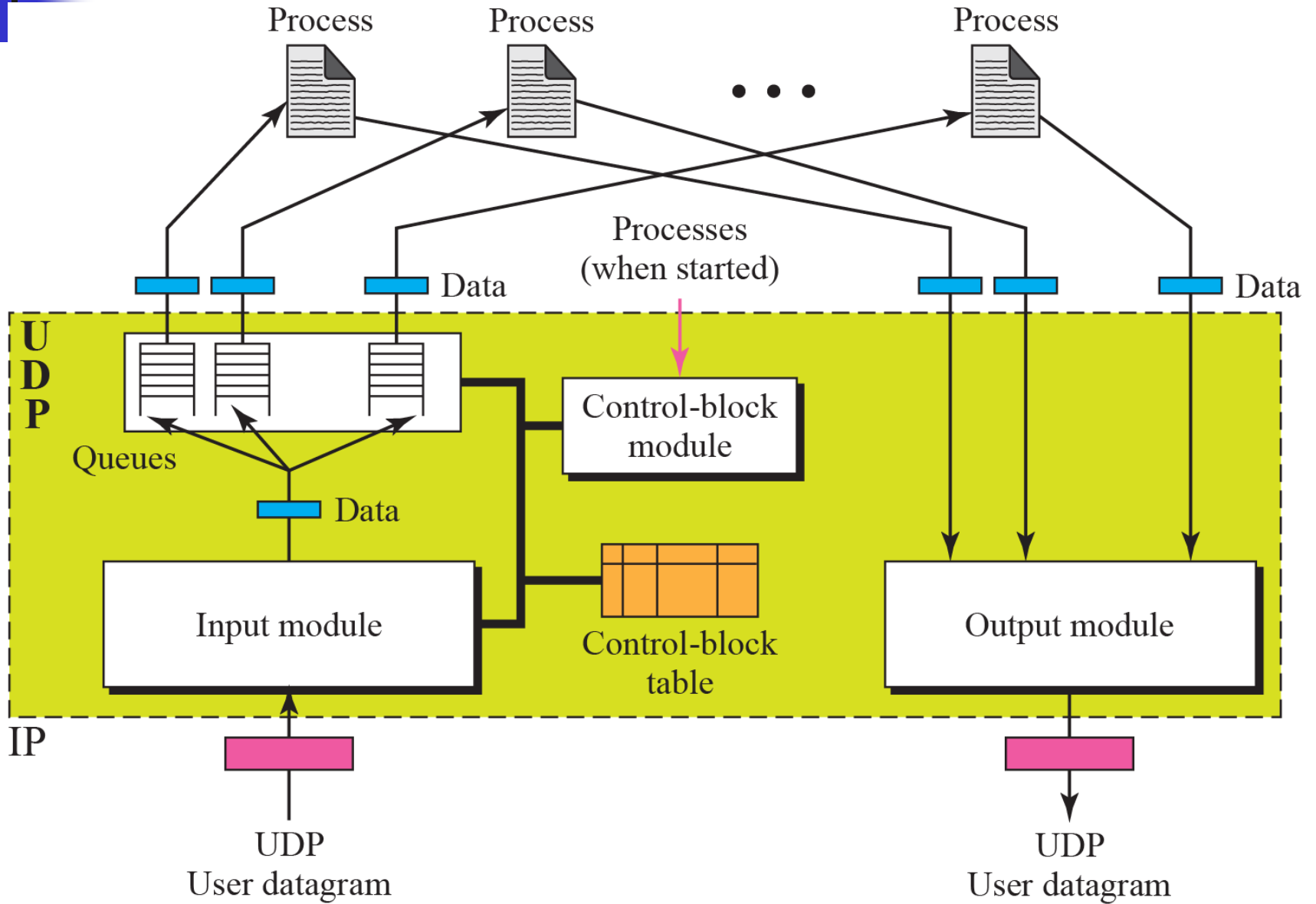
To show how UDP handles the sending and receiving of UDP packets, we present a simple version of the UDP package.

We can say that the UDP package involves five components: a control-block table, input queues, a control-block module, an input module, and an output module.

Topics Discussed in the Section

- ✓ **Control-Block Table**
- ✓ **Input Queues**
- ✓ **Control-Block Module**
- ✓ **Input Module**
- ✓ **Output Module**

Figure 14.8 *UDP design*



In our package, UDP has a **control-block table** to keep track of the open ports. Each entry in this table has a minimum of four fields:

- the state, which can be: FREE or IN-USE,
- the process ID,
- the port number,
- and the corresponding queue number.

- The **control-block module** is responsible for the management of the control-block table.
- When a process starts, it asks for a port number from the operating system.
- The operating system assigns well-known port numbers to servers and ephemeral port numbers to clients.
- The process passes the process ID and the port number to the control-block module to create an entry in the table for the process
- The Module does not create the queues

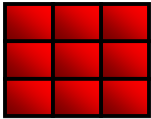


Table 14.2 *Control Block Module*

```
1  UDP_Control_Block_Module (process ID, port number)
2  {
3      Search the table for a FREE entry.
4      if (not found)
5          Delete one entry using a predefined strategy.
6      Create a new entry with the state IN-USE
7      Enter the process ID and the port number.
8      Return.
9  } // End module
```

- The **input module** receives a user datagram from the IP.
- It searches the control-block table to find an entry having the same port number as this user datagram.
- If the entry is found, the module uses the information in the entry to enqueue the data.
- If the entry is not found, it generates an ICMP message.
- **Output Module** is responsible for creating and sending user datagrams

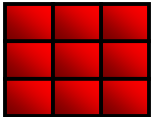


Table 14.3 *Input Module*

```
1  UDP_INPUT_Module (user_datagram)
2  {
3      Look for the entry in the control_block table
4      if (found)
5          {
6              Check to see if a queue is allocated
7              If (queue is not allocated)
8                  allocate a queue
9              else
10                 enqueue the data
11          } //end if
12      else
13          {
14              Ask ICMP to send an "unreachable port" message
15              Discard the user datagram
16          } //end else
17
18      Return.
19  } // end module
```

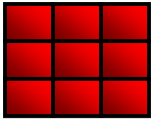



Table 14.4 *Output Module*

```
1  UDP_OUTPUT_MODULE (Data)
2  {
3      Create a user datagram
4      Send the user datagram
5      Return.
6  }
```

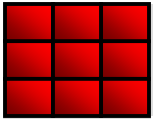


Table 14.5 *The Control-Block Table at the Beginning of Examples*

<i>State</i>	<i>Process ID</i>	<i>Port Number</i>	<i>Queue Number</i>
IN-USE	2,345	52,010	34
IN-USE	3,422	52,011	
FREE			
IN-USE	4,652	52,012	38
FREE			

Example 14.8

The first activity is the arrival of a user datagram with destination port number 52,012. The input module searches for this port number and finds it. Queue number 38 has been assigned to this port, which means that the port has been previously used. The input module sends the data to queue 38. The control-block table does not change.

Example 14.9

After a few seconds, a process starts. It asks the operating system for a port number and is granted port number 52,014. Now the process sends its ID (4,978) and the port number to the control-block module to create an entry in the table. The module takes the first FREE entry and inserts the information received. The module does not allocate a queue at this moment because no user datagrams have arrived for this destination (see Table 14.6).

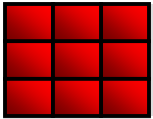


Table 14.6 *Control-Block Table after Example 14.9*

<i>State</i>	<i>Process ID</i>	<i>Port Number</i>	<i>Queue Number</i>
IN-USE	2,345	52,010	34
IN-USE	3,422	52,011	
IN-USE	4,978	52,014	
IN-USE	4,652	52,012	38
FREE			

Example 14.10

A user datagram now arrives for port 52,011. The input module checks the table and finds that no queue has been allocated for this destination since this is the first time a user datagram has arrived for this destination. The module creates a queue and gives it a number (43). See Table 14.7.

Table 14.7 *Control-Block Table after Example 14.10*

<i>State</i>	<i>Process ID</i>	<i>Port Number</i>	<i>Queue Number</i>
IN-USE	2,345	52,010	34
IN-USE	3,422	52,011	43
IN-USE	4,978	52,014	
IN-USE	4,652	52,012	38
FREE			

Example 14.11

After a few seconds, a user datagram arrives for port 52,222. The input module checks the table and cannot find an entry for this destination. The user datagram is dropped and a request is made to ICMP to send an unreachable port message to the source.

Chapter 15

Transmission Control Protocol (TCP)

Edited & Presented by:

Dr. Mohammad Alhammouri

TCP/IP Protocol Suite (B A. Forouzan)

OBJECTIVES:

- ❑ To introduce TCP as a protocol that provides reliable stream delivery service.**
- ❑ To define TCP features and compare them with UDP features.**
- ❑ To define the format of a TCP segment and its fields.**
- ❑ To show how TCP provides a connection-oriented service, and show the segments exchanged during connection establishment and connection termination phases.**
- ❑ To discuss the state transition diagram for TCP and discuss some scenarios.**
- ❑ To introduce windows in TCP that are used for flow and error control.**

OBJECTIVES (*continued*):

- ❑ **To discuss how TCP implements flow control in which the receive window controls the size of the send window.**
- ❑ **To discuss error control and FSMs used by TCP during the data transmission phase.**
- ❑ **To discuss how TCP controls the congestion in the network using different strategies.**
- ❑ **To list and explain the purpose of each timer in TCP.**
- ❑ **To discuss options in TCP and show how TCP can provide selective acknowledgment using the SACK option.**
- ❑ **To give a layout and a simplified pseudocode for the TCP package.**

Chapter Outline

- 15.1 TCP Services***
- 15.2 TCP Features***
- 15.3 Segment***
- 15.4 A TCP Connection***
- 15.5 State Transition Diagram***
- 15.6 Windows in TCP***
- 15.7 Flow Control***
- 15.8 Error Control***
- 15.9 Congestion Control***
- 15.10 TCP Timers***
- 15.11 Options***
- 15.12 TCP Package***

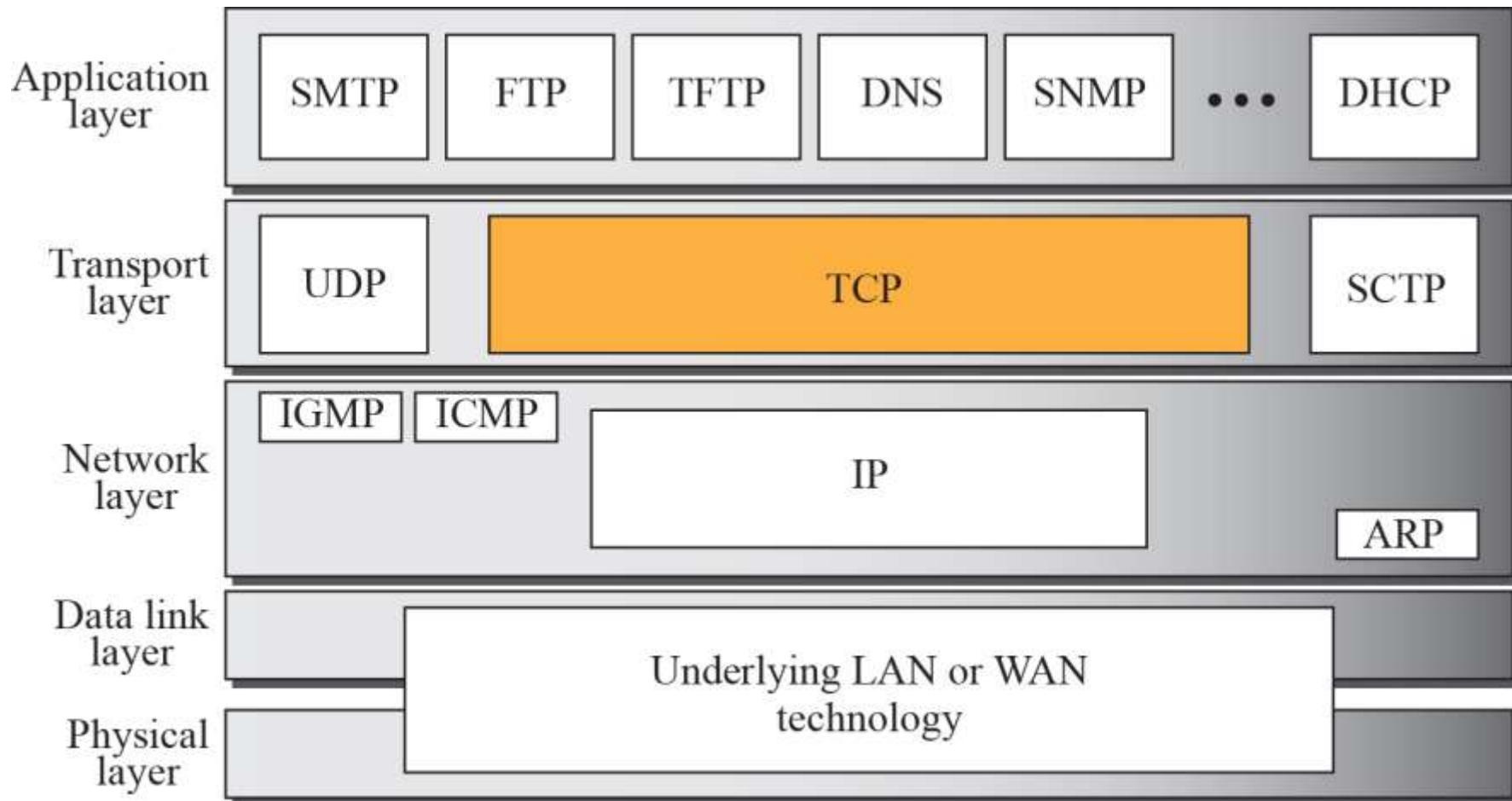
15-1 TCP SERVICES

Figure 15.1 shows the relationship of TCP to the other protocols in the TCP/IP protocol suite. TCP lies between the application layer and the network layer, and serves as the intermediary between the application programs and the network operations.

Topics Discussed in the Section

- ✓ **Process-to-Process Communication**
- ✓ **Stream Delivery Service**
- ✓ **Full-Duplex Communication**
- ✓ **Multiplexing and Demultiplexing**
- ✓ **Connection-Oriented Service**
- ✓ **Reliable Service**

Figure 15.1 *TCP/IP protocol suite*



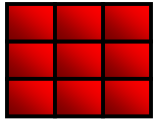


Table 15.1 *Well-known Ports used by TCP*

<i>Port</i>	<i>Protocol</i>	<i>Description</i>
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 and 21	FTP	File Transfer Protocol (Data and Control)
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

Figure 15.2 *Stream delivery*

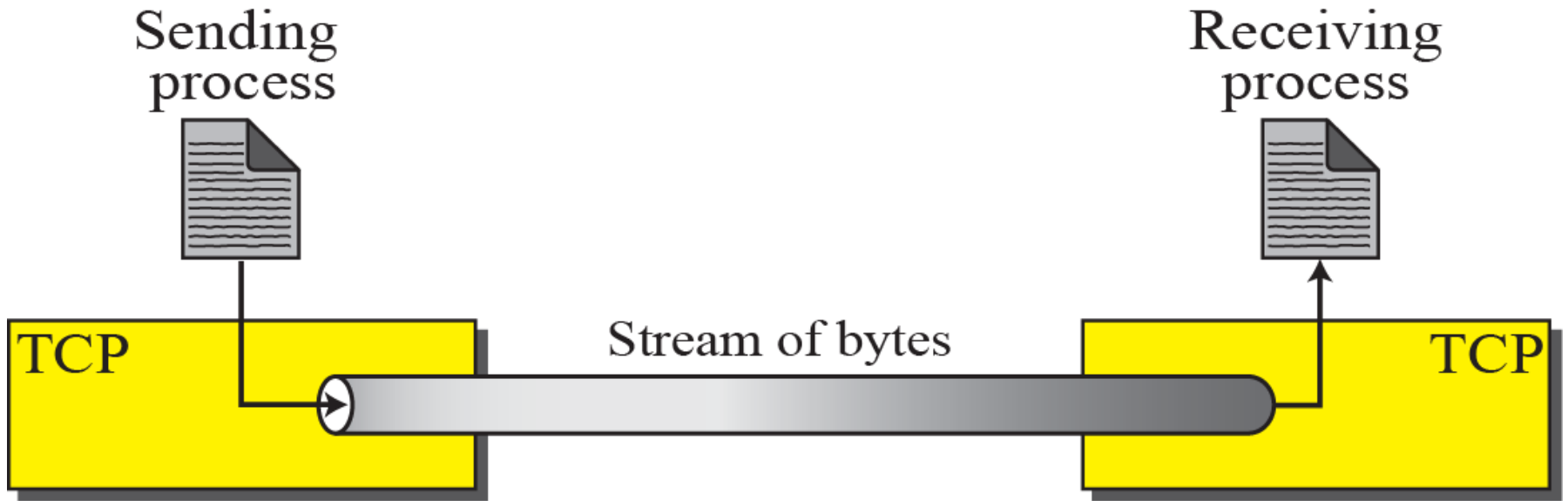


Figure 15.3 *Sending and receiving buffers*

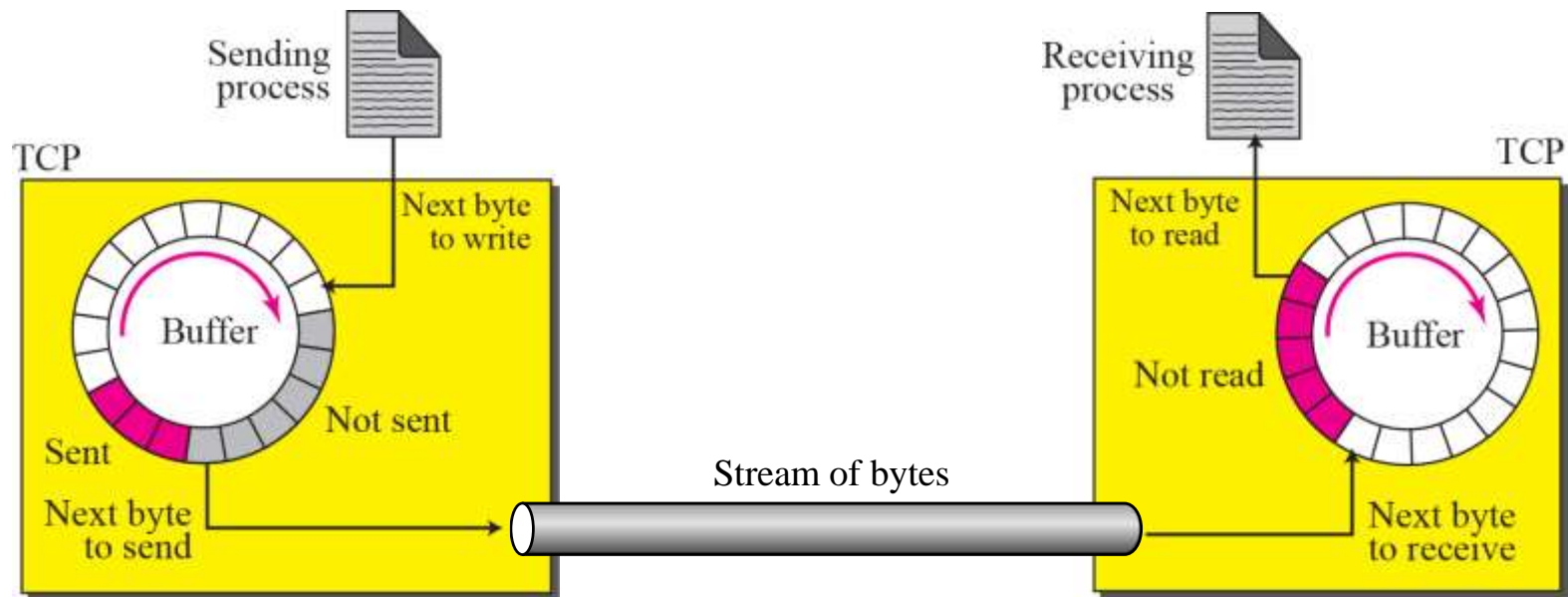
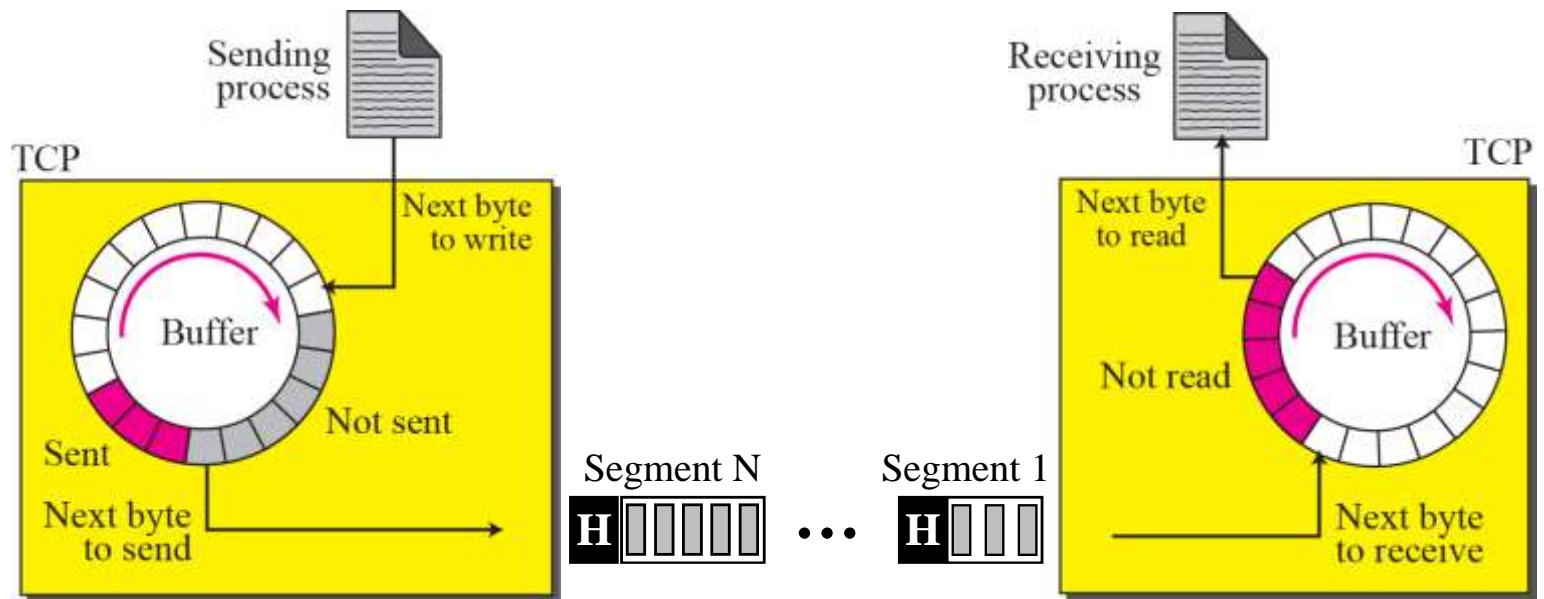


Figure 15.4 *TCP segments*



15-2 TCP FEATURES

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

Topics Discussed in the Section

- ✓ **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 an arbitrarily generated number.

Example 15.1

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



Note

The value in the sequence number field of a segment defines the number assigned to 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.

15-3 SEGMENT

Before discussing TCP in more detail, let us discuss the TCP packets themselves. A packet in TCP is called a segment.

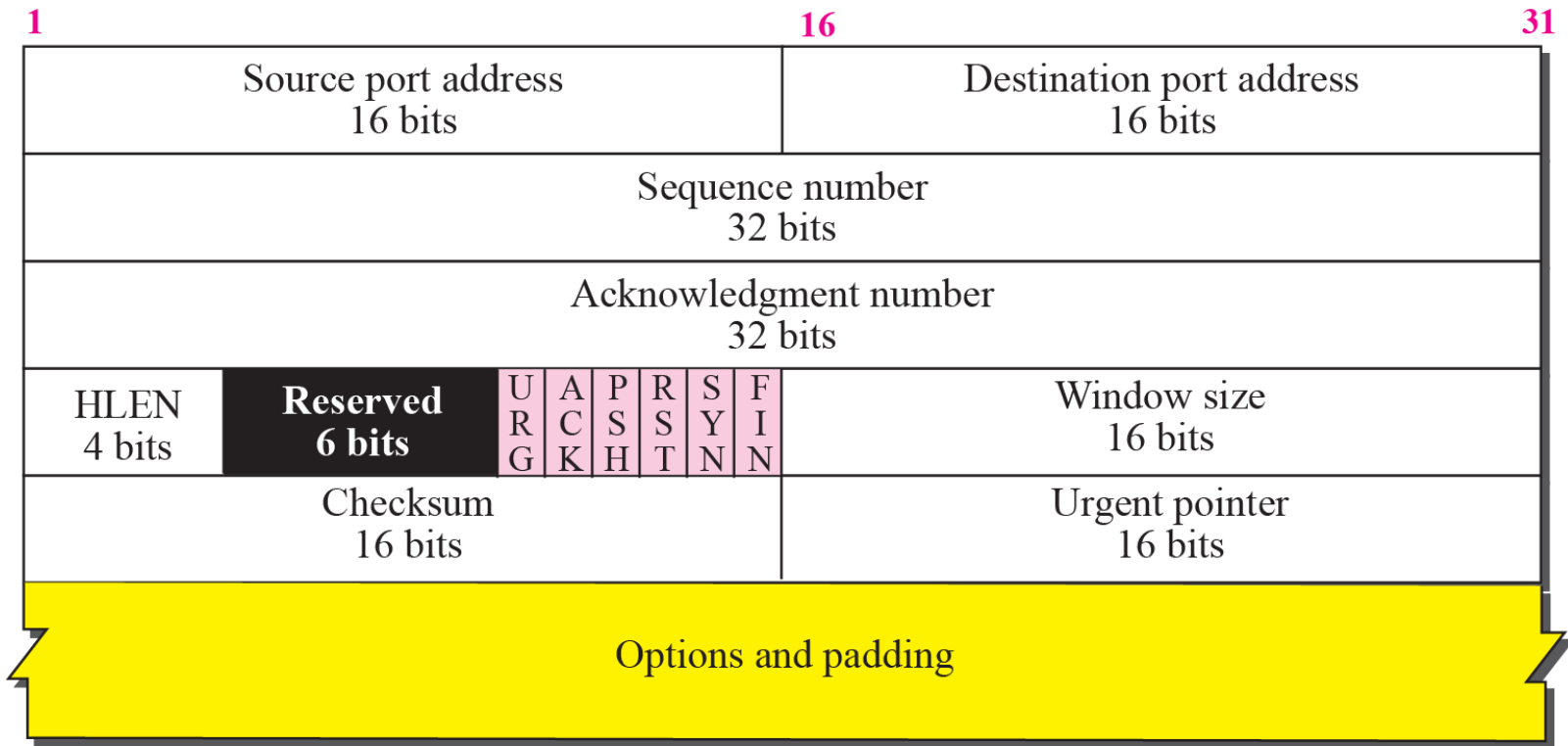
Topics Discussed in the Section

- ✓ **Format**
- ✓ **Encapsulation**

Figure 15.5 *TCP segment format*



a. Segment



b. Header

Figure 15.6 *Control field*

URG: Urgent pointer is valid

ACK: Acknowledgment is valid

PSH: Request for push

RST: Reset the connection

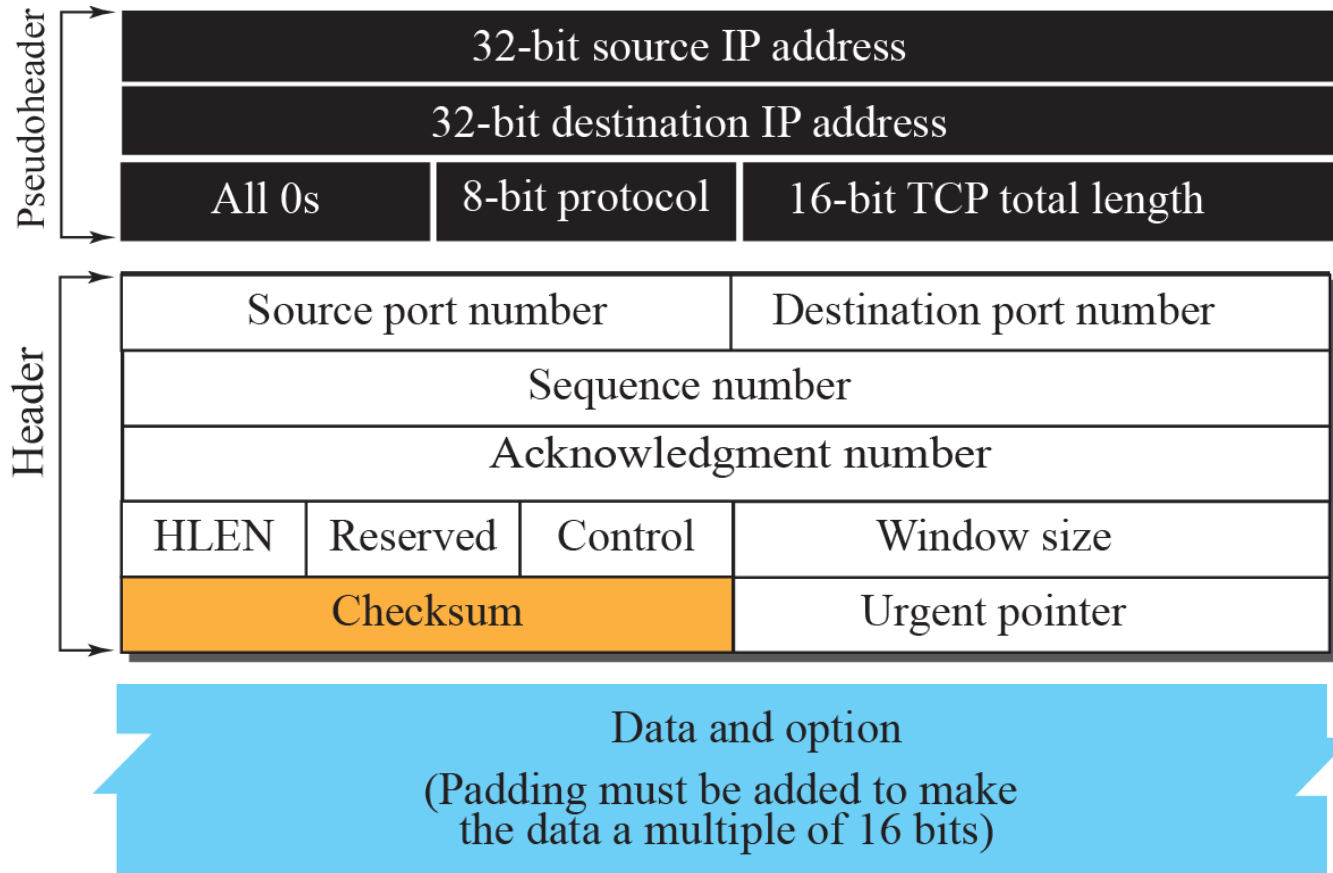
SYN: Synchronize sequence numbers

FIN: Terminate the connection



6 bits

Figure 15.7 *Pseudoheader added to the TCP segment*

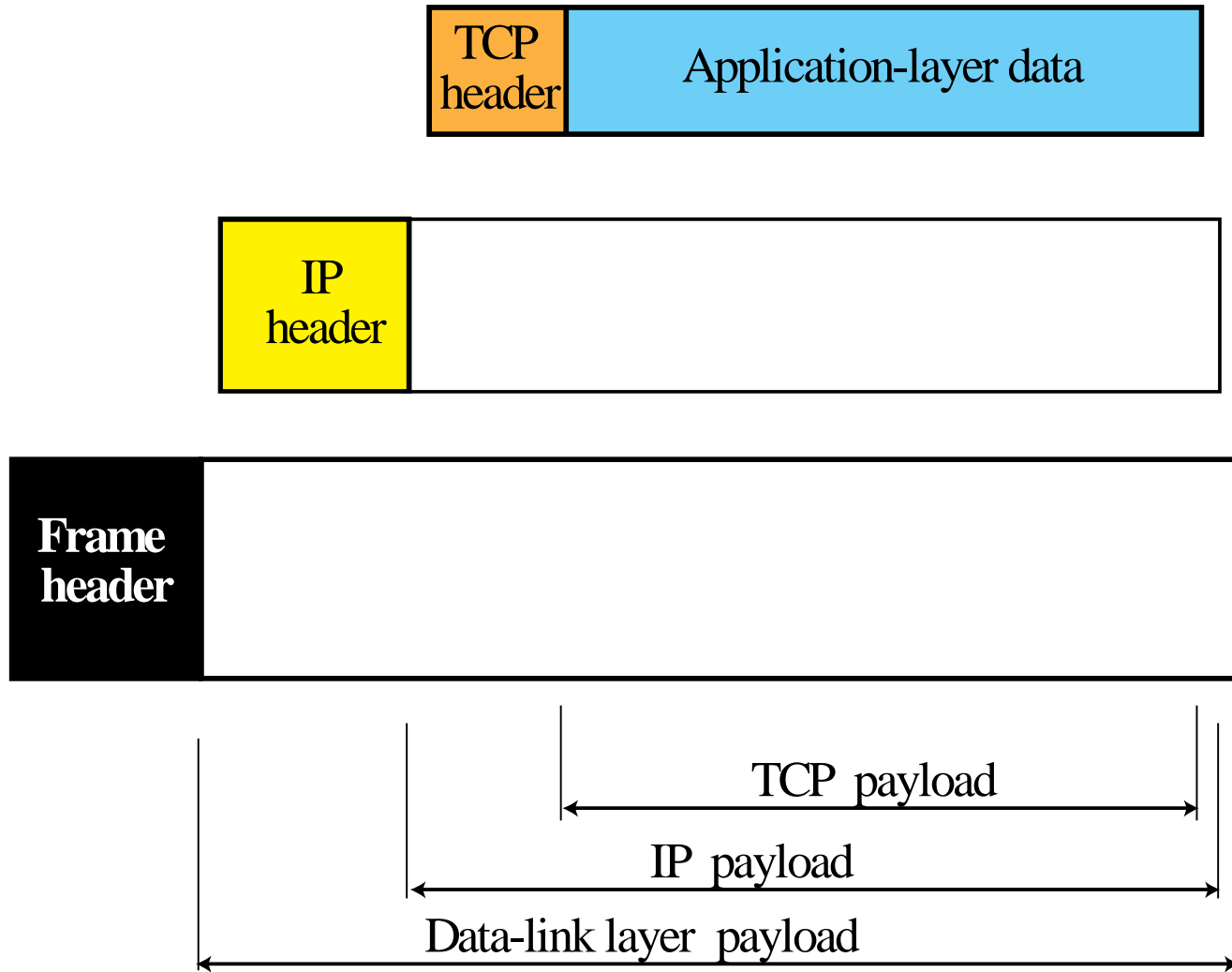




Note

The use of the checksum in TCP is mandatory.

Figure 15.8 *Encapsulation*



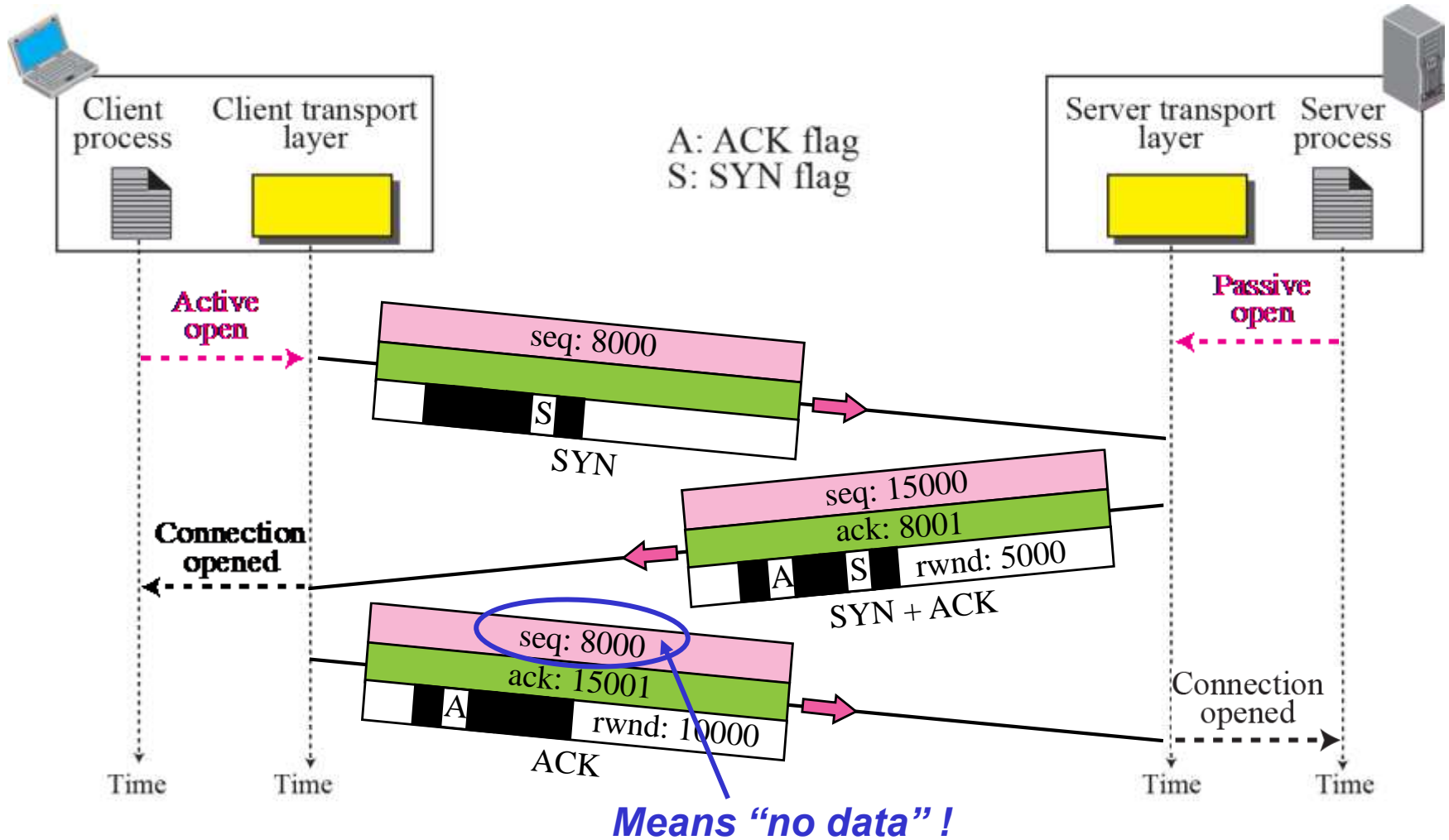
15-4 A TCP CONNECTION

TCP is connection-oriented. It establishes a virtual path between the source and destination. All of the segments belonging to a message are then sent over this virtual path. You may wonder how TCP, which uses the services of IP, a connectionless protocol, can be connection-oriented. The point is that a TCP connection is virtual, not physical. TCP operates at a higher level. TCP uses the services of IP to deliver individual segments to the receiver, but it controls the connection itself. If a segment is lost or corrupted, it is retransmitted.

Topics Discussed in the Section

- ✓ **Connection Establishment**
- ✓ **Data Transfer**
- ✓ **Connection Termination**
- ✓ **Connection Reset**

Figure 15.9 Connection establishment using three-way handshake





Note

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



Note

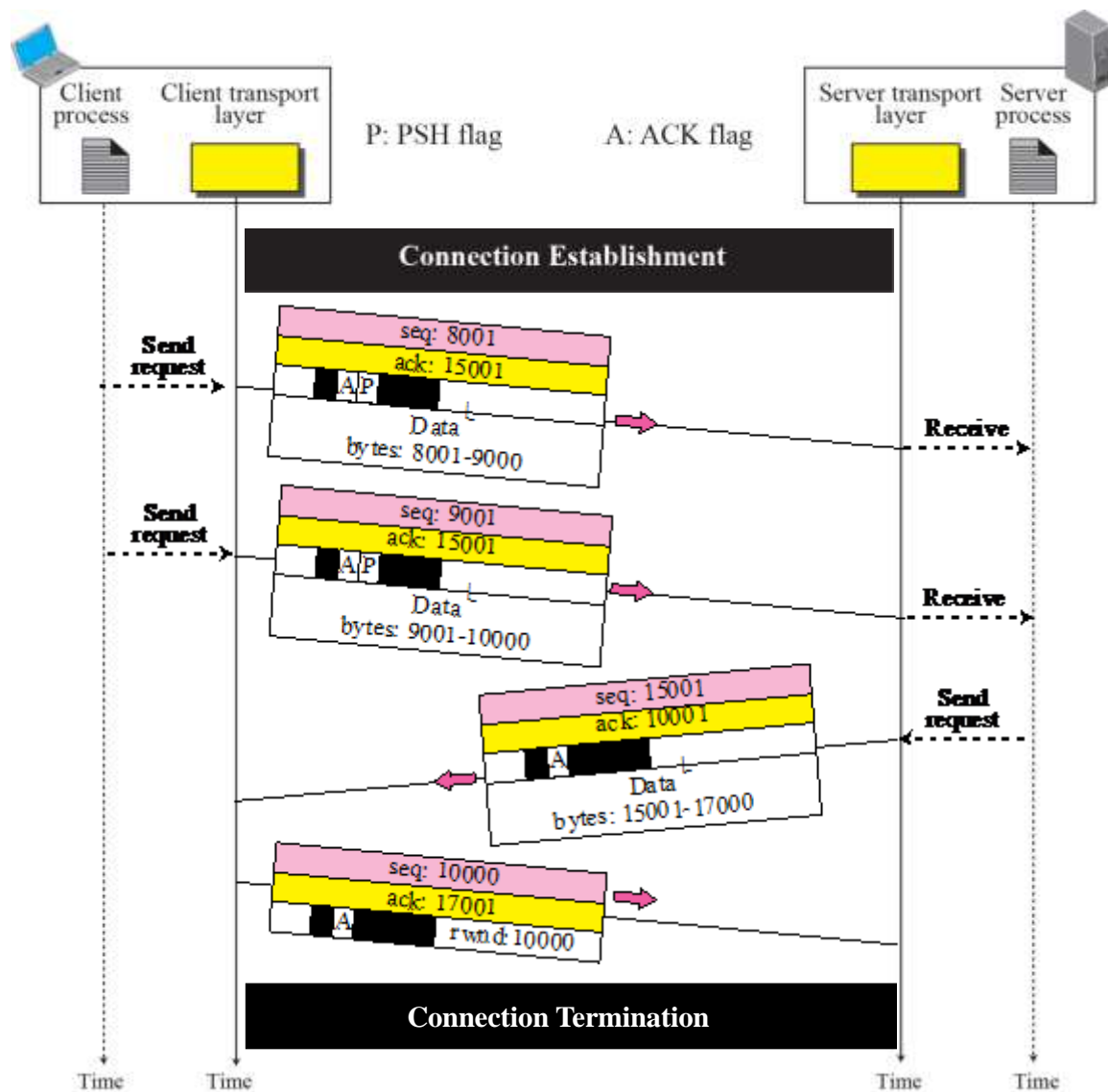
A SYN + ACK segment cannot carry data, but does consume one sequence number.



Note

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

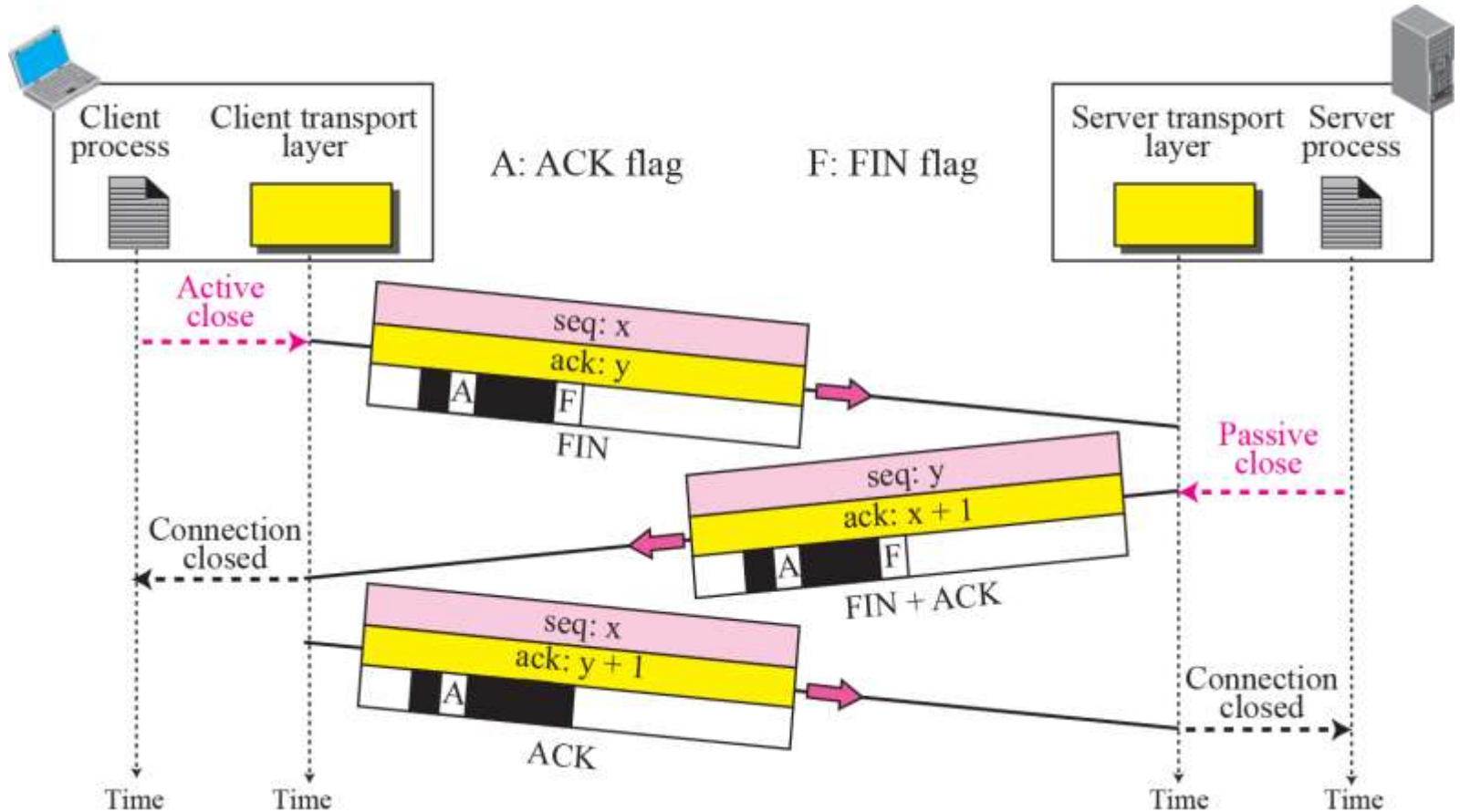
Figure 15.10 Data Transfer



P: PSH (Push Flag)

- The application program at the sender can request a *push* operation.
- This means that the sending TCP must not wait for the window to be filled.
- After the segment is created, it will be sent immediately
- Segment includes data that must be delivered to the receiving application program as soon as possible and not to wait for more data to come.
- Although the push operation can be requested by the application program, most current TCP implementations ignore such requests. TCP can choose whether or not to use this feature.

Figure 15.11 *Connection termination using three-way handshake*

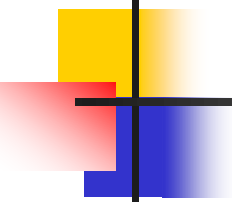




The FIN segment consumes one sequence number if it does not carry data.

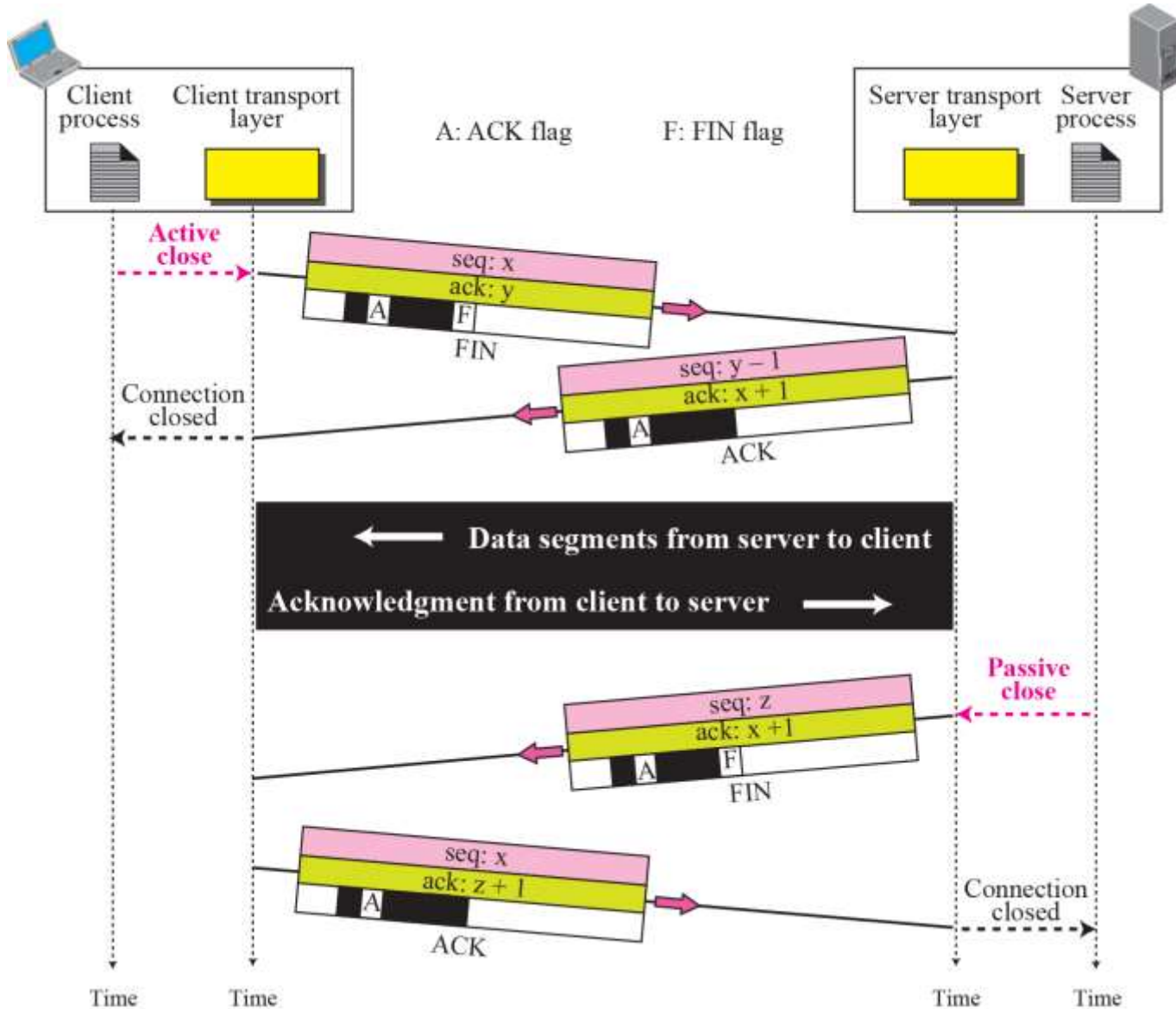


The FIN + ACK segment (from server) consumes one sequence number if it does not carry data.



The ACK (from client): This segment cannot carry data and consumes no sequence numbers

Figure 15.12 *Half-Close*



15-6 WINDOWS IN TCP

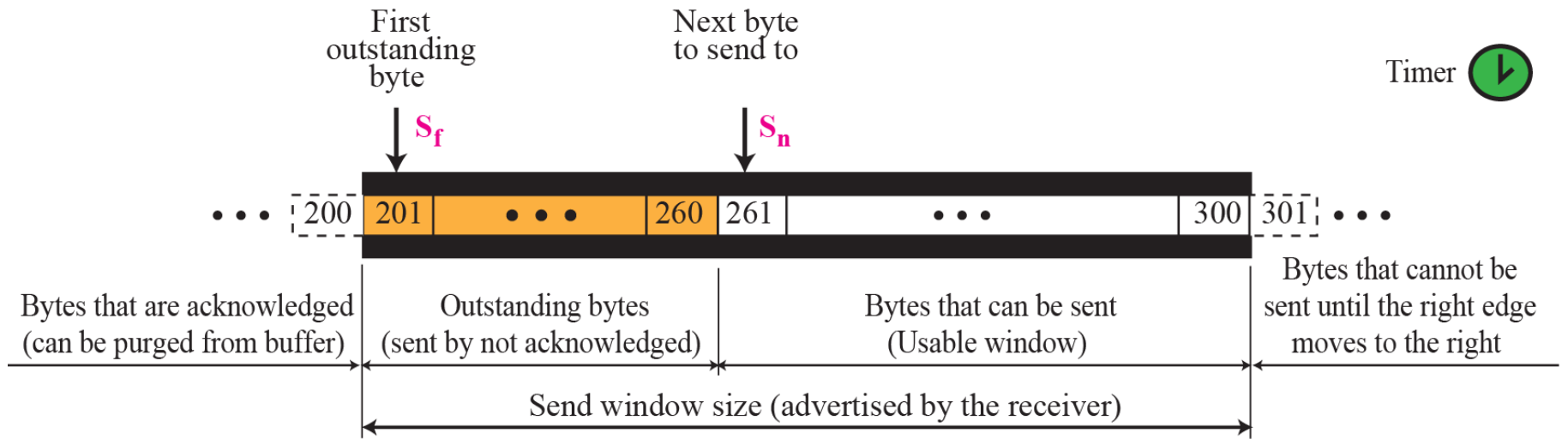
Before discussing data transfer in TCP and the issues such as flow, error, and congestion control, we describe the windows used in TCP. TCP uses two windows (send window and receive window) for each direction of data transfer, which means four windows for a bidirectional communication.

To make the discussion simple, we make an assumption that communication is only unidirectional; the bidirectional communication can be inferred using two unidirectional communications with **piggybacking** (Data and Ack can travel in both direction).

Topics Discussed in the Section

- ✓ **Send Window**
- ✓ **Receive Window**

Figure 15.22 *Send window in TCP*

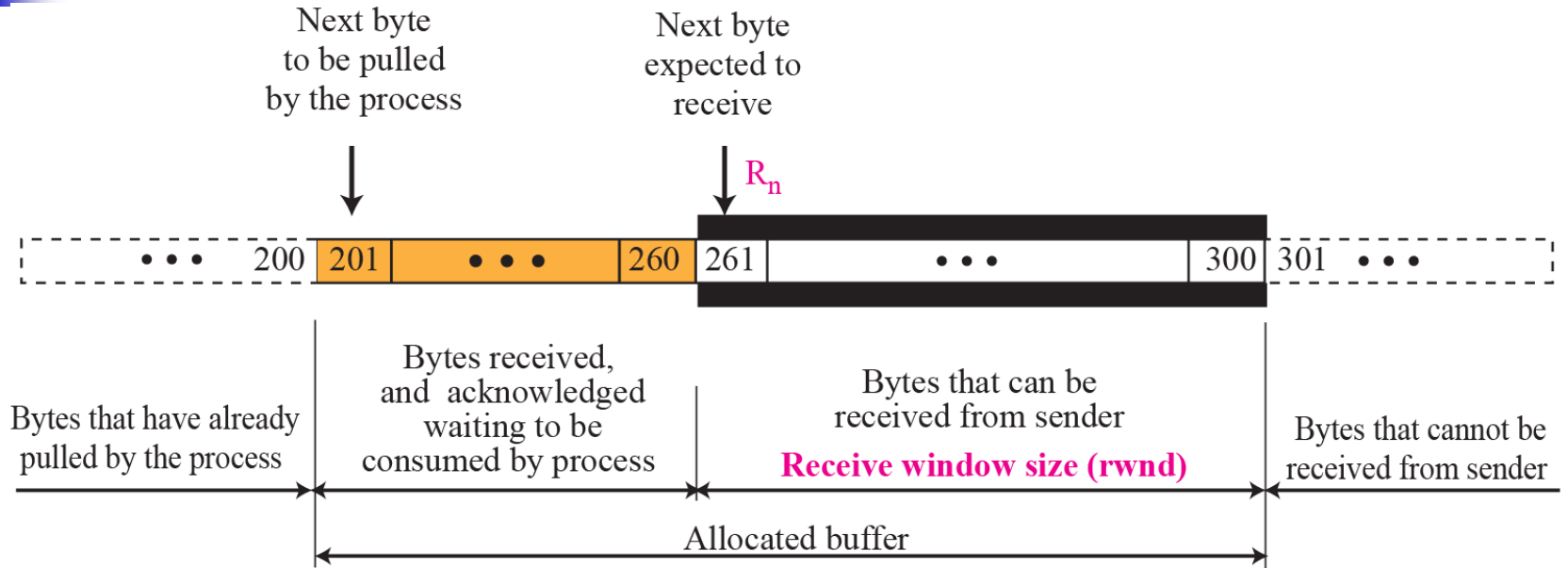


a. Send window

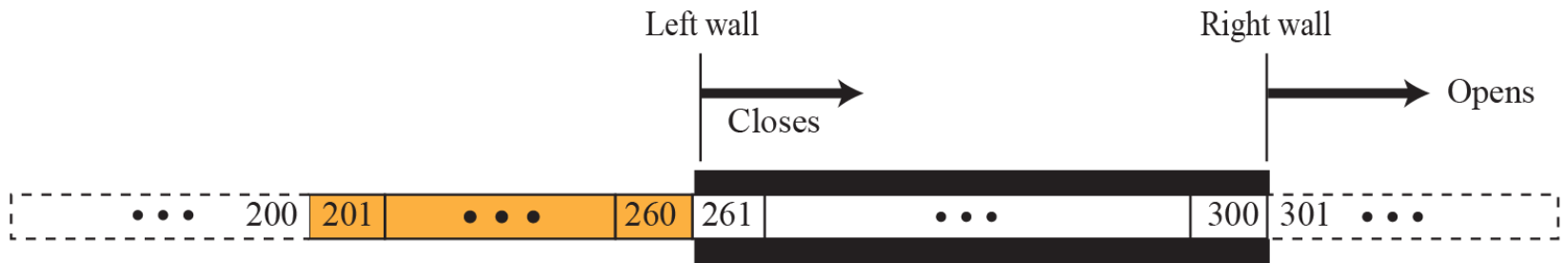


b. Opening, closing, and shrinking send window

Figure 15.23 *Receive window in TCP*



a. Receive window and allocated buffer



b. Opening and closing of receive window

Receive Window

- There are two differences between the receive window in TCP and the one we used for SR in Chapter 13:
- (1) TCP allows the receiving process to pull data at its own NEED
- Part of the allocated buffer at the receiver may be occupied by bytes that have been received and acknowledged, but are waiting to be pulled by the receiving process
- The receive window size is then always smaller or equal to the buffer size

$Rwnd(\text{window size}) = \text{buffer size} - \text{number of waiting bytes to be pulled}$

Receive Window

- (2) The second difference is the way acknowledgments are used in the TCP protocol:
- Remember that an **acknowledgement in SR is selective**, defining the uncorrupted Packets that have been received.
-
- The major acknowledgment mechanism in TCP is a **cumulative acknowledgment** announcing the next expected byte to receive
-
- The new versions of TCP, however, uses both cumulative and selective acknowledgements as we will discuss later in the option section.

15-7 FLOW CONTROL

As discussed in Chapter 13, flow control balances the rate a producer creates data with the rate a consumer can use the data. TCP separates flow control from error control. In this section we discuss flow control, ignoring error control. We temporarily assume that the logical channel between the sending and receiving TCP is error-free. Figure 15.24 shows unidirectional data transfer between a sender and a receiver; bidirectional data transfer can be obtained from unidirectional one as discussed in Chapter 13.

Figure 15.24 *TCP/IP protocol suite*

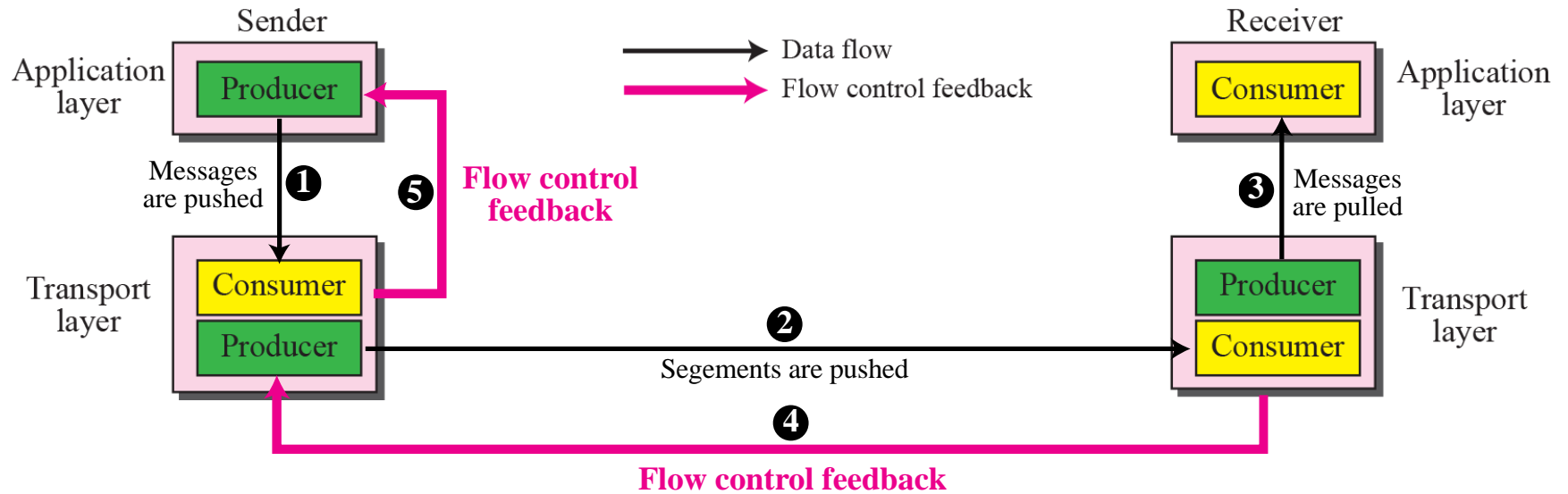


Figure 15.25 *An example of flow control*

Note: We assume only unidirectional communication from client to server. Therefore, only one window at each side is shown.

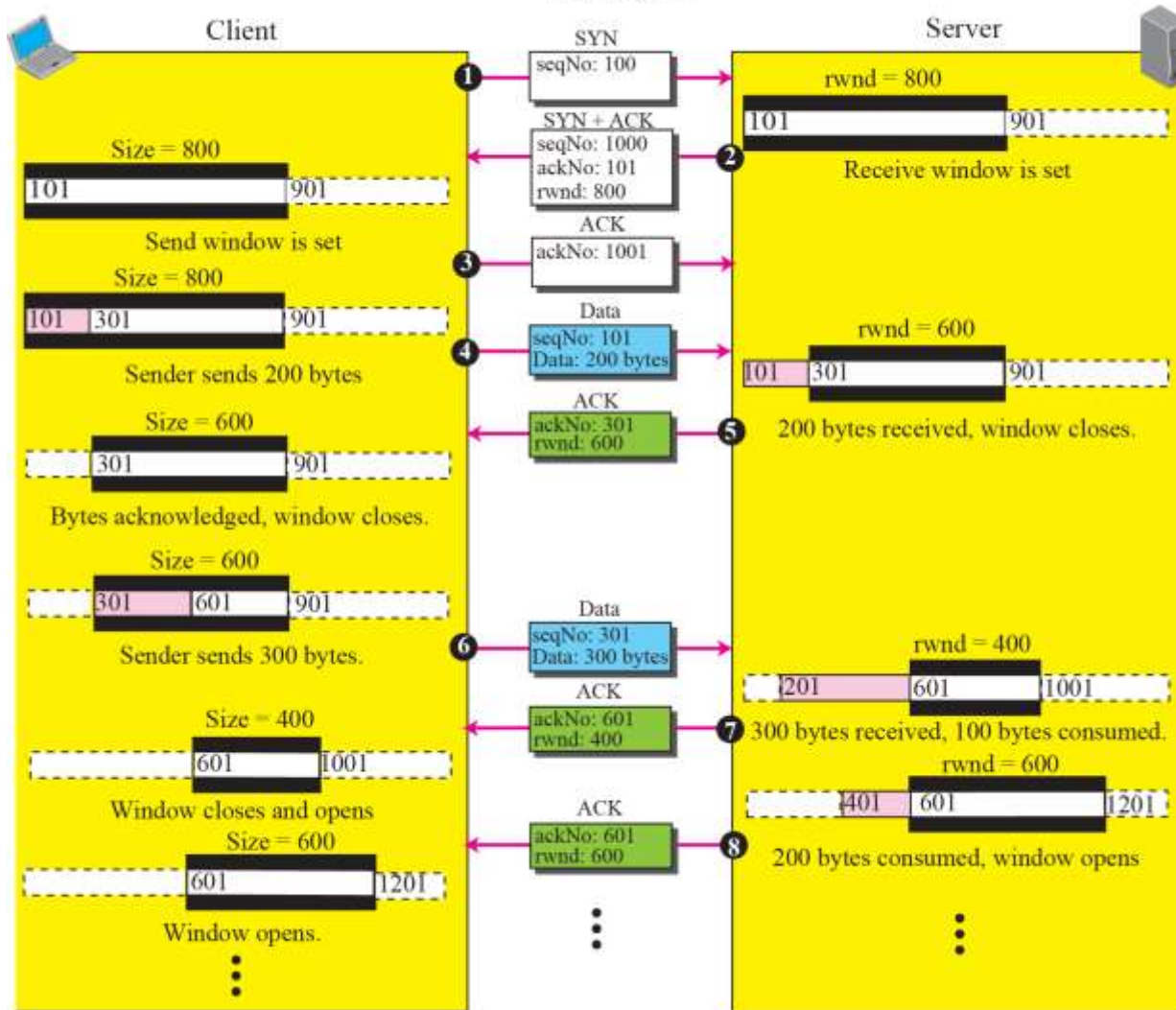
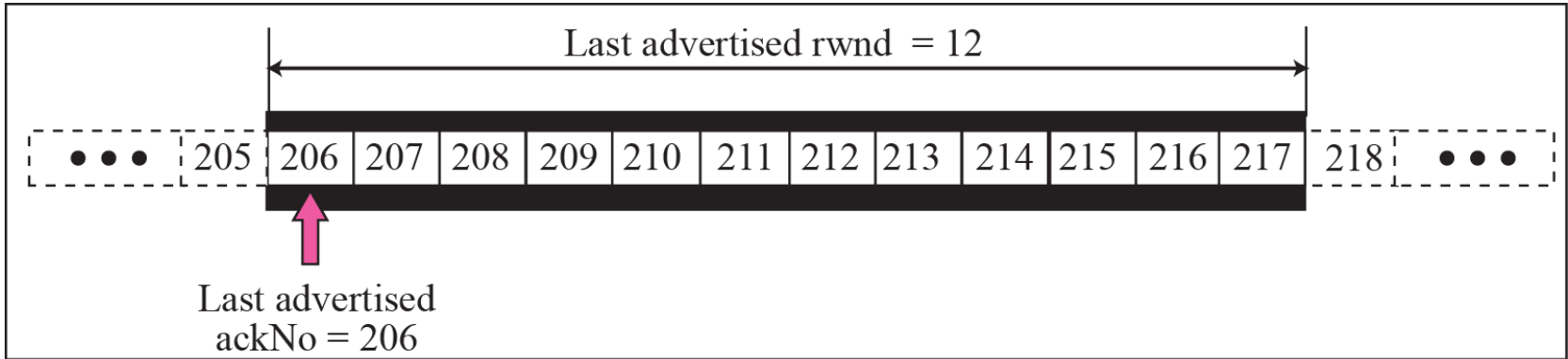
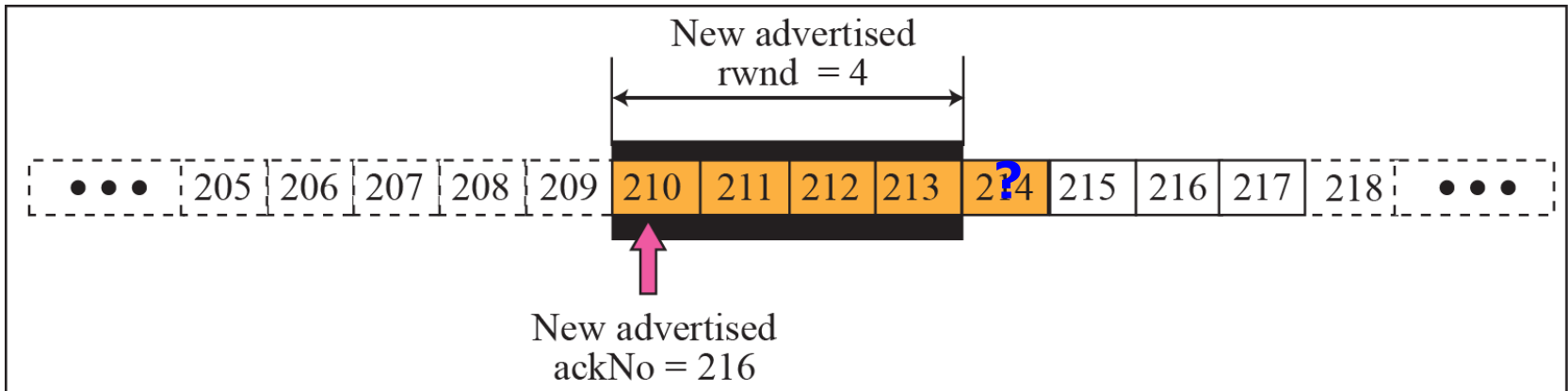


Figure 15.26 Example 15.2

Prevent the shrinking of the send window:
 $\text{new ackNo} + \text{new rwnd} \geq \text{last ackNo} + \text{last rwnd}$



a. The window after the last advertisement



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

****Bytes 206 to 209 are acknowledged and consumed**

Example 15.2

Figure 15.26 shows the reason for the mandate in window shrinking. **Part a** of the figure shows values of last acknowledgment and *rwnd*. **Part b** shows the situation in which the sender has sent bytes **206 to 214**. Bytes **206 to 209** are acknowledged and purged.

Example 15.2 cont.

The new advertisement, however, defines the new value of *rwnd* as 4, in which $210 + 4 < 206 + 12$.

When the send window shrinks it creates a problem:

byte 214 which has been already sent is outside the window. The relation discussed before forces the receiver to maintain the right-hand wall of the window to be as shown in part a because the receiver does not know which of the bytes 210 to 217 has already been sent.

One way to prevent this situation: is to let the receiver postpone its feedback until enough buffer locations are available in its window. In other words, the receiver should wait until more bytes are consumed by its process.

15-8 ERROR CONTROL

TCP is a reliable transport layer protocol. This means that an application program that delivers a stream of data to TCP relies on TCP to deliver the entire stream to the application program on the other end in the correct order, without error, and without any part lost or duplicated.

Error control in TCP is achieved through the use of three tools: *checksum*, *acknowledgment*, and *time-out*.

Topics Discussed in the Section

- ✓ **Checksum**
- ✓ **Acknowledgment**
- ✓ **Retransmission**
- ✓ **Out-of-Order Segments**
- ✓ **FSMs for Data Transfer in TCP**
- ✓ **Some Scenarios**



Note

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

Acknowledgement Type

- In the past, TCP used only one type of acknowledgement: Accumulative Acknowledgement (**ACK**), also namely **accumulative positive acknowledgement**
- More and more implementations are adding another type of acknowledgement: **Selective Acknowledgement (SACK)**, SACK is implemented as an option at the end of the TCP header.





Note

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



Note

***TCP can be best modeled as a
Selective Repeat protocol.***

Rules for Generating ACK (1)

- 1. When one end sends a data segment to the other end, it must include an ACK. That gives the next sequence number it expects to receive. (Piggyback)
- 2. The receiver needs to **delay sending** (until **another segment arrives or 500ms**) an ACK segment if there is only one outstanding in-order segment. It prevents ACK segments from creating extra traffic.
- 3. There **should not be more than 2 in-order unacknowledged segments** at any time. It prevent the unnecessary retransmission



Rules for Generating ACK (2)

- 4. When a **segment arrives with an out-of-order sequence number** that is higher than expected, the receiver immediately sends an ACK segment announcing the sequence number of the next expected segment. (for **fast retransmission**)
- 5. When a **missing segment** arrives, the receiver sends an ACK segment to announce the next sequence number expected.
- 6. If a **duplicate segment** arrives, the receiver immediately sends an ACK.

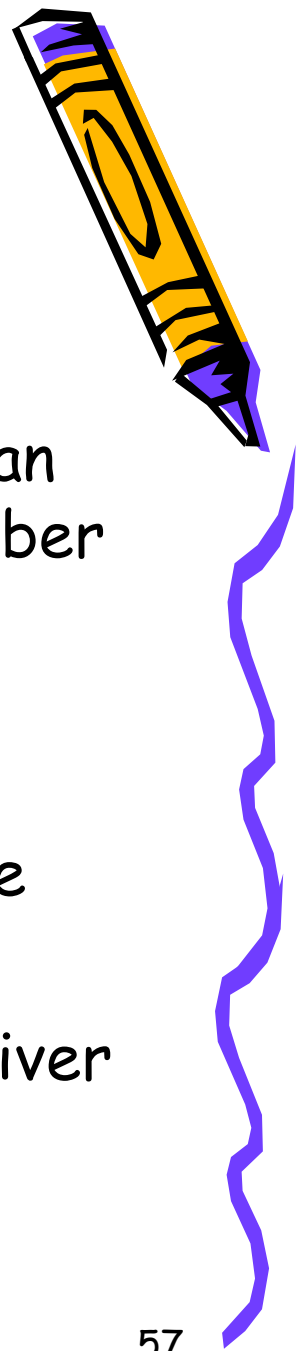


Figure 15.29 Normal operation

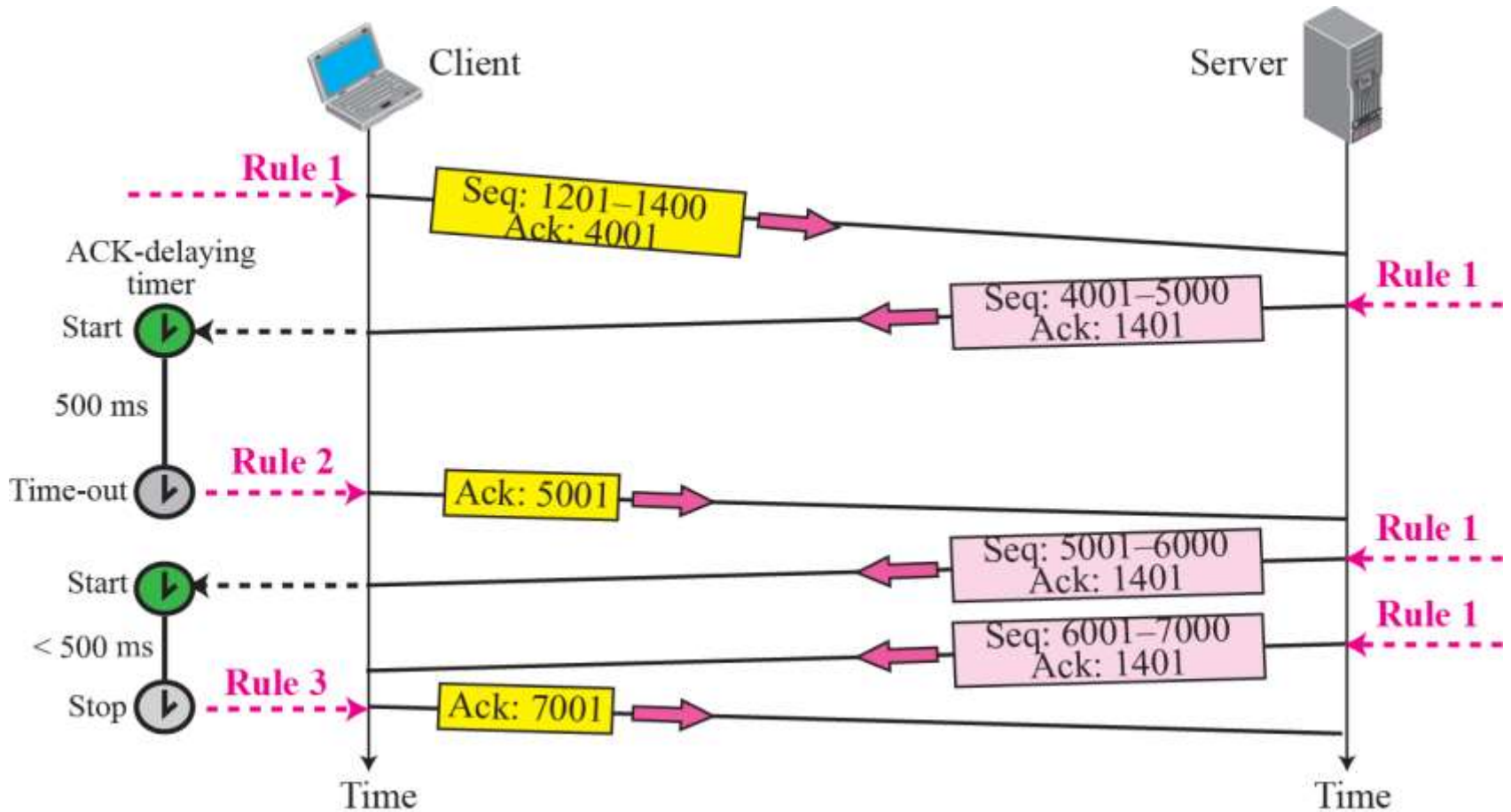
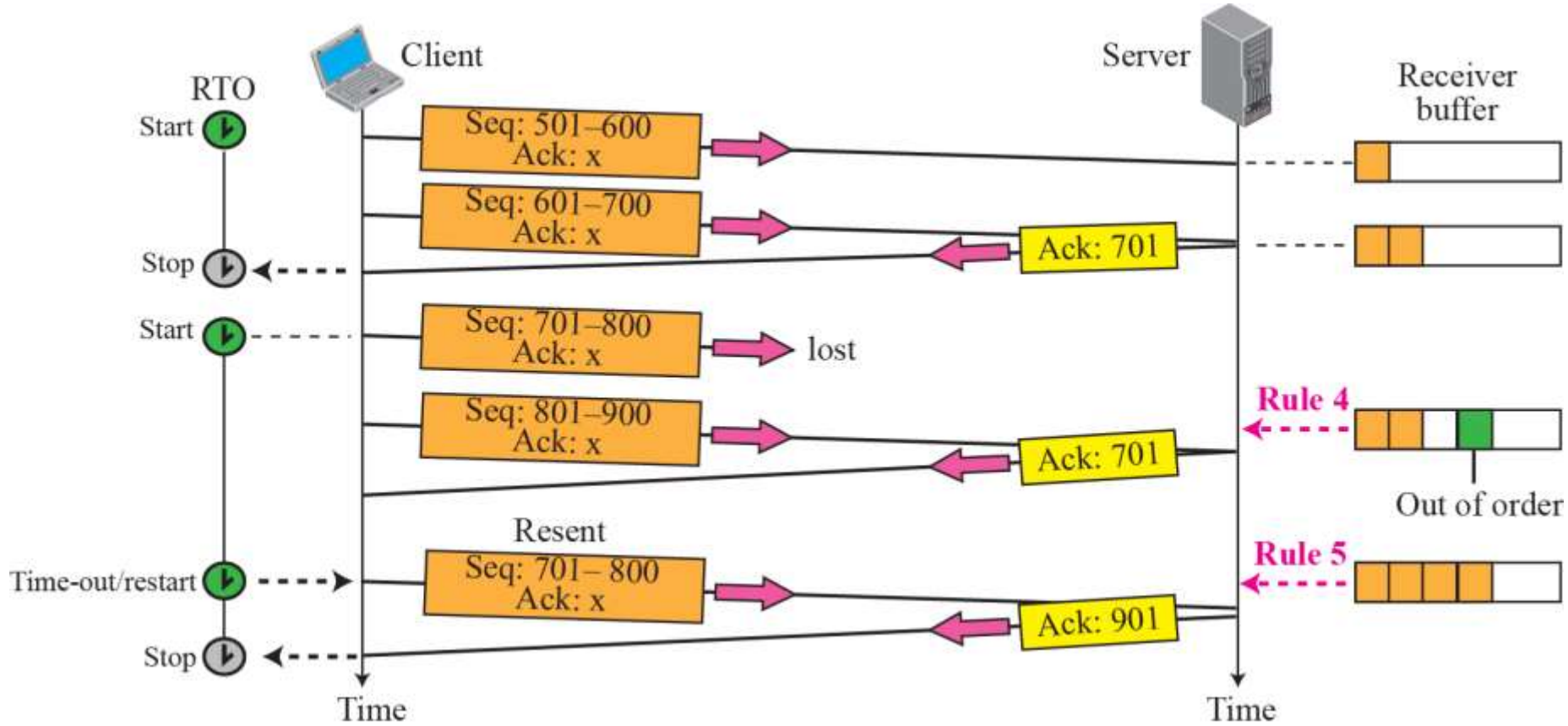


Figure 15.30 *Lost segment*

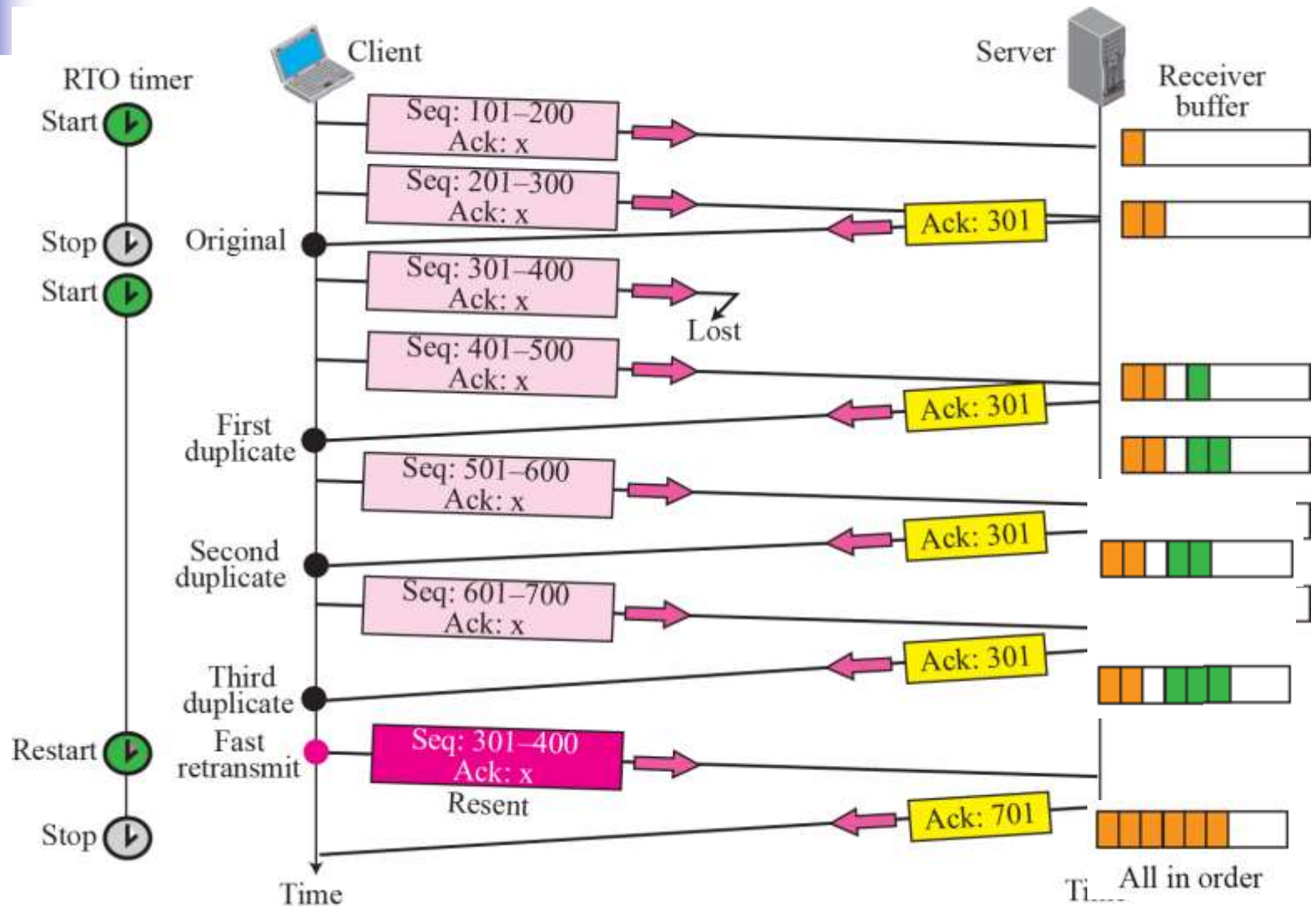


retransmission time-out (RTO)



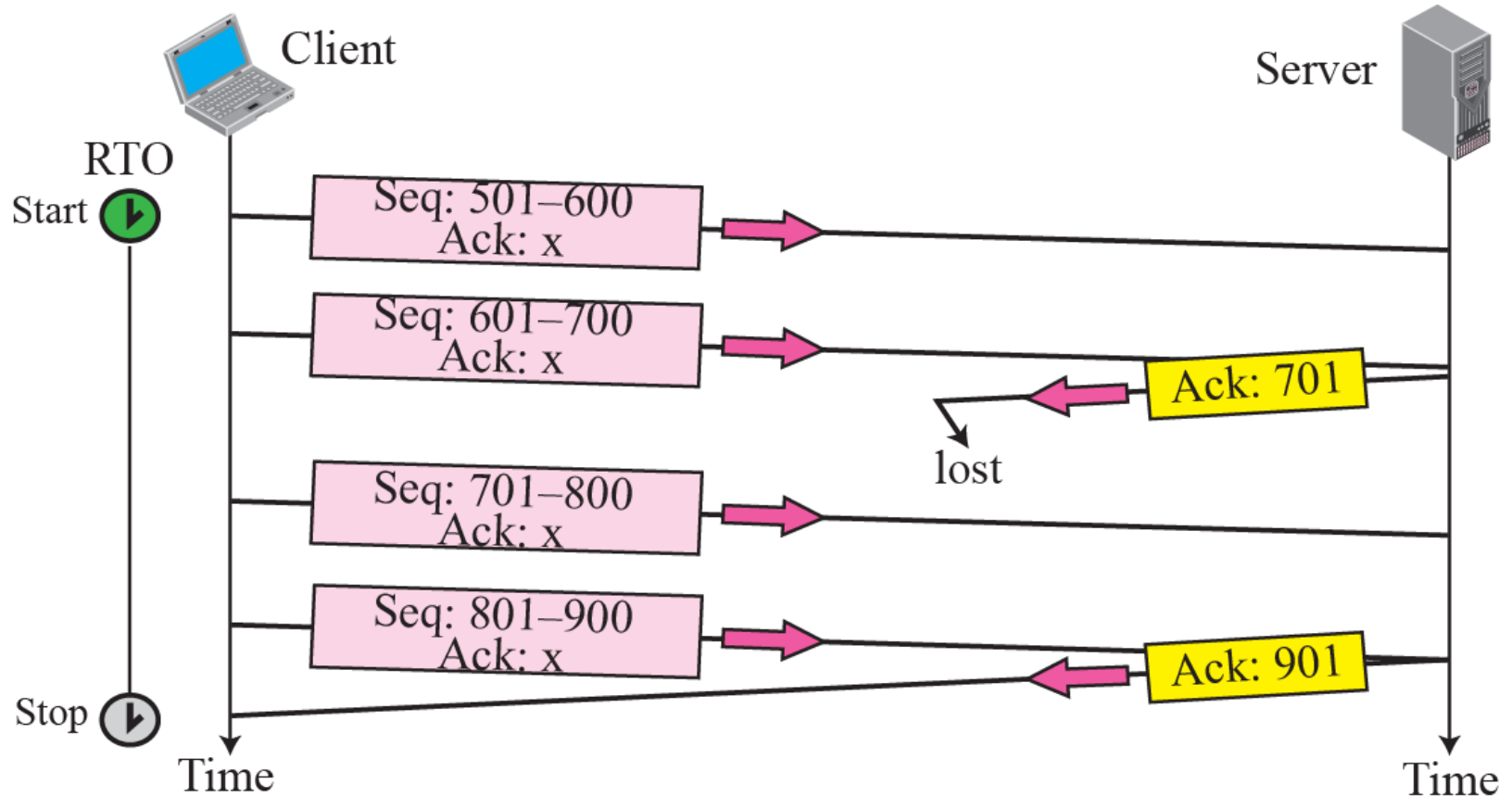
The receiver TCP delivers only ordered data to the process.

Figure 15.31 *Fast retransmission*



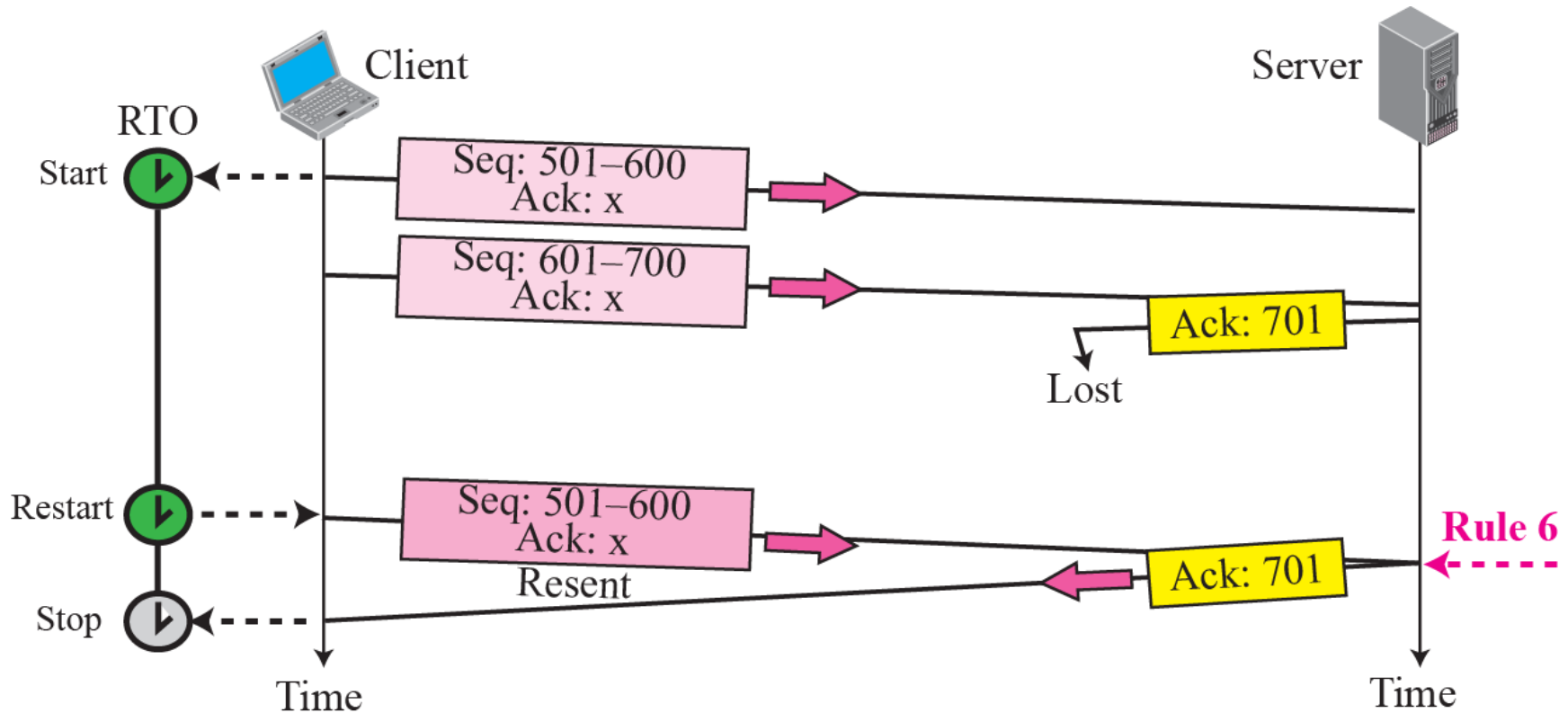
Fast retransmission :Retransmission after Three Duplicate ACK Segments

Figure 15.32 *Lost acknowledgment*



Advantage of cumulative acknowledgments

Figure 15.33 *Lost acknowledgment corrected by resending a segment*



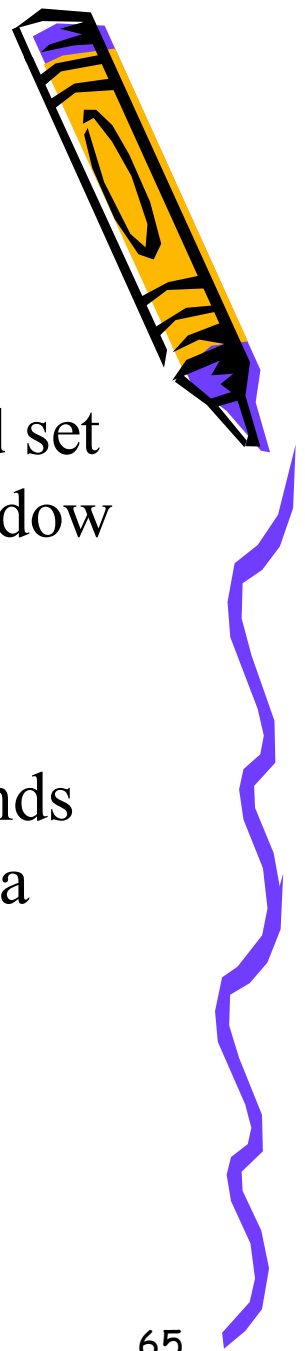
Rule 6: If a duplicate segment arrives, the receiver immediately sends an ACK



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

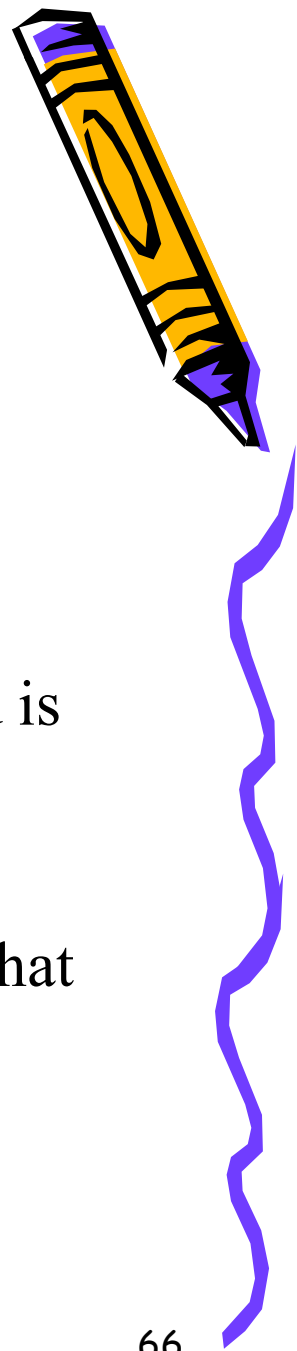
Deadlock Created by Lost Acknowledgment

- The receiver sends an acknowledgment with **rwnd** set to 0 and requests that the sender shut down its window temporarily
- After a while, the receiver wants to remove the restriction; however, if it has no data to send. It sends an ACK segment and removes the restriction with a nonzero value for **rwnd**
- A problem arises if this acknowledgment is lost



Deadlock Created by Lost Acknowledgment

- The sender is waiting for an acknowledgment that announces the nonzero **rwnd**
- The receiver thinks that the sender has received this and is waiting for data. This situation is called a **deadlock**
- To prevent deadlock, a persistence timer was designed that we will study later in the chapter



15-9 CONGESTION CONTROL

We discussed congestion control in Chapter 13. Congestion control in TCP is based on both open loop and closed-loop mechanisms. TCP uses a congestion window and a congestion policy that avoid congestion and detect and alleviate congestion after it has occurred.

Congestion Window

- **Flow control:** solution when the receiver is overwhelmed with data
- We said that the sender window size is determined by the available buffer space in the receiver (**rwnd**).
- We assumed that it is only the receiver that can dictate to the sender the size of the sender's window.
- What about the network
- If the network cannot deliver the data as fast as it is created by the sender, it must tell the sender to slow down.
- The sender has two pieces of information: the receiver-advertised window size (**rwnd**), and the congestion window size (**cwnd**)

Congestion Window

- The sender has two pieces of information: the receiver-advertised window size (**rwnd**), and the congestion window size (**cwnd**)

Actual window size = minimum (rwnd, cwnd)

Topics Discussed in the Section

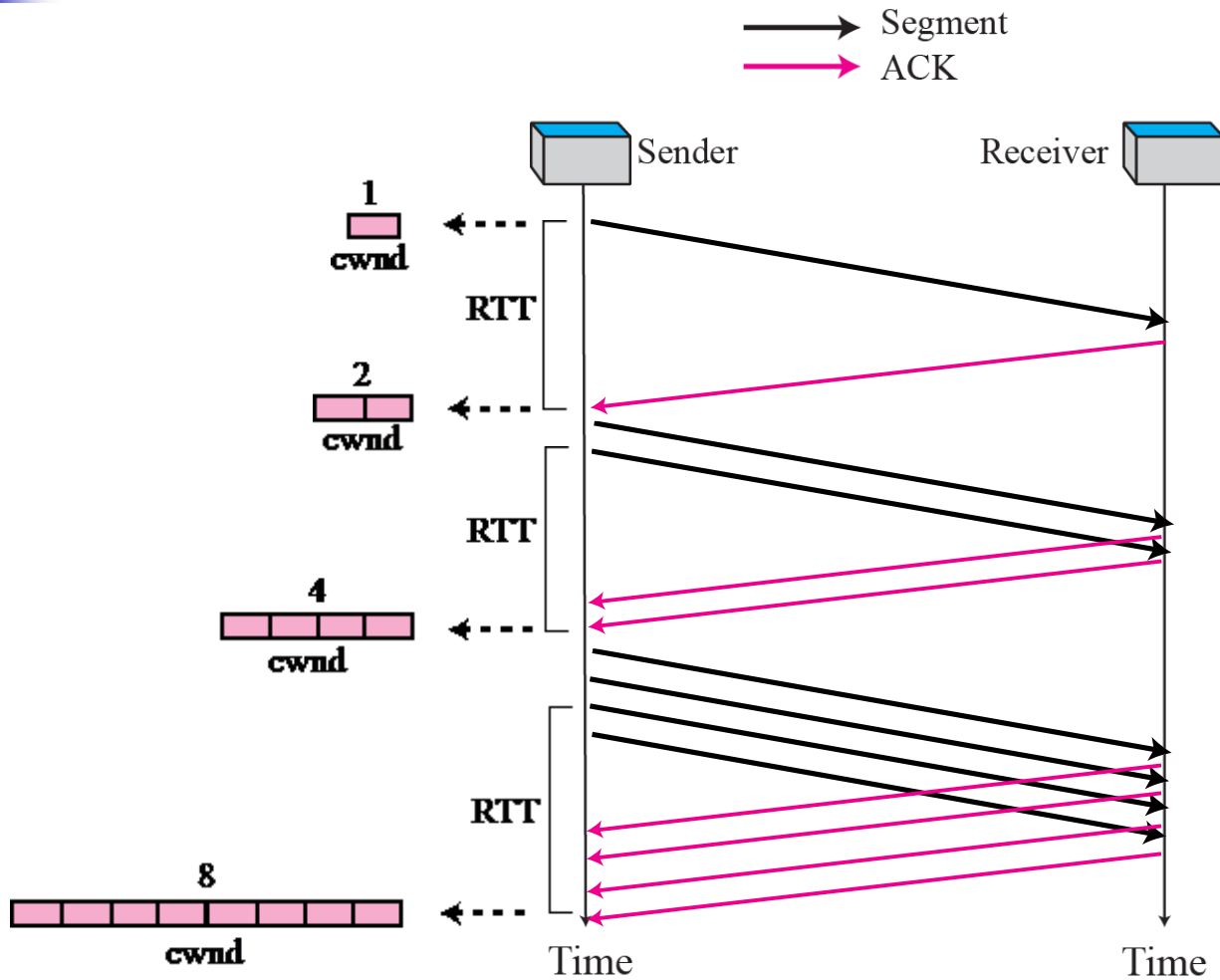
- ✓ **Congestion Window**
- ✓ **Congestion Policy**

Congestion Avoidance: Slow start Algorithm

- The sender has two pieces of information: the receiver-advertised window size (**rwnd**), and the congestion window size (**cwnd**)

Actual window size = minimum (rwnd, cwnd)

Figure 15.34 *Slow start, exponential increase*





Note

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

Figure 15.35 *Congestion avoidance, additive increase*

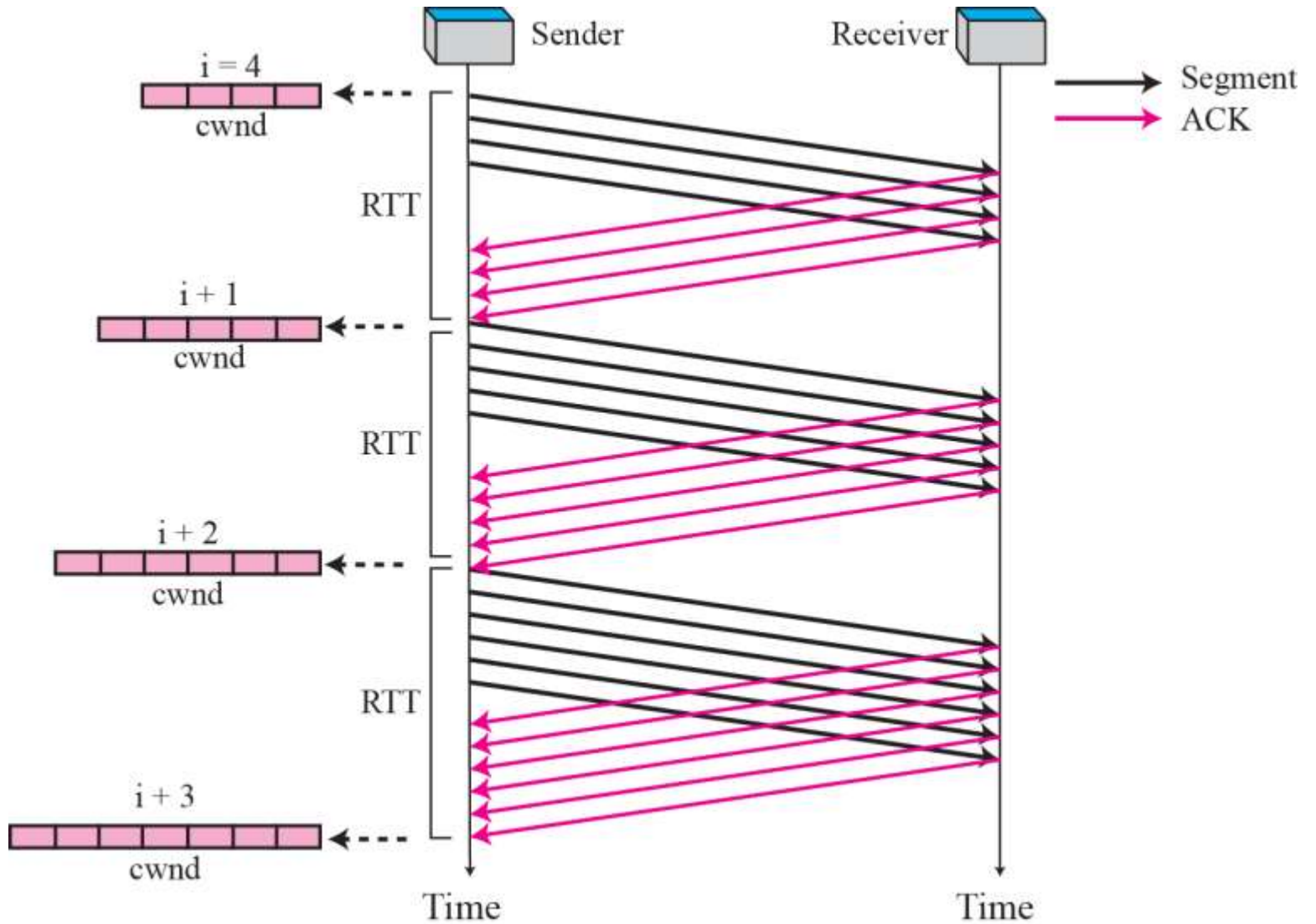


Figure 15.34 *Slow start, exponential increase*

If we look at the size of the cwnd in terms of round-trip times (RTTs), we find that the growth rate is exponential as shown below:

Start	→	cwnd = 1
After 1 RTT	→	cwnd = 1 × 2 = 2 → 2 ¹
After 2 RTT	→	cwnd = 2 × 2 = 4 → 2 ²
After 3 RTT	→	cwnd = 4 × 2 = 8 → 2 ³



Note

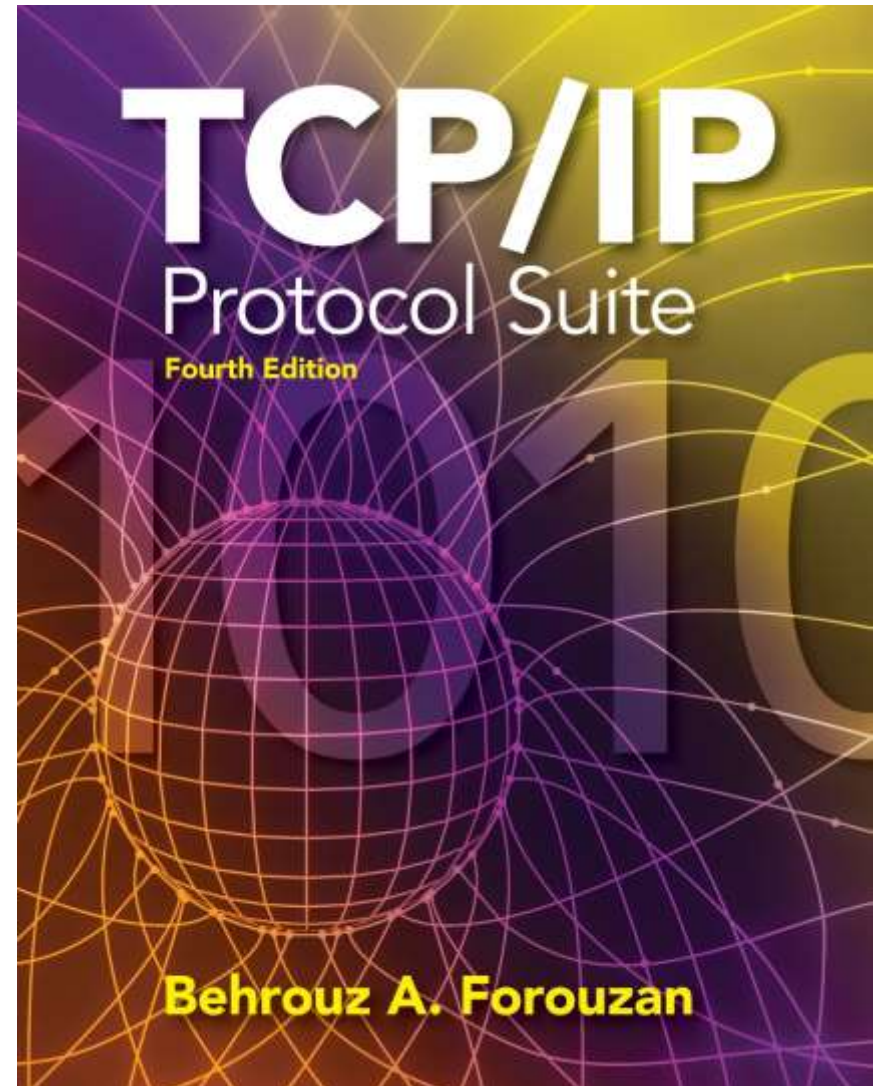
In the congestion avoidance algorithm the size of the congestion window increases additively until congestion is detected.

Congestion Avoidance

- Slow start strategy is slower in the case of delayed acknowledgments.
- For each ACK, the **cwnd** is increased by only 1 MSS (Maximum segment size).
- If three segments are acknowledged accumulatively, the size of the **cwnd** increases by only 1 MSS, not 3 MSS.
- The growth is still exponential, but it is not a power of 2.

Chapter 16

Stream Control Transmission Protocol (SCTP)



Given By: Dr. Mohammad Al-hammouri

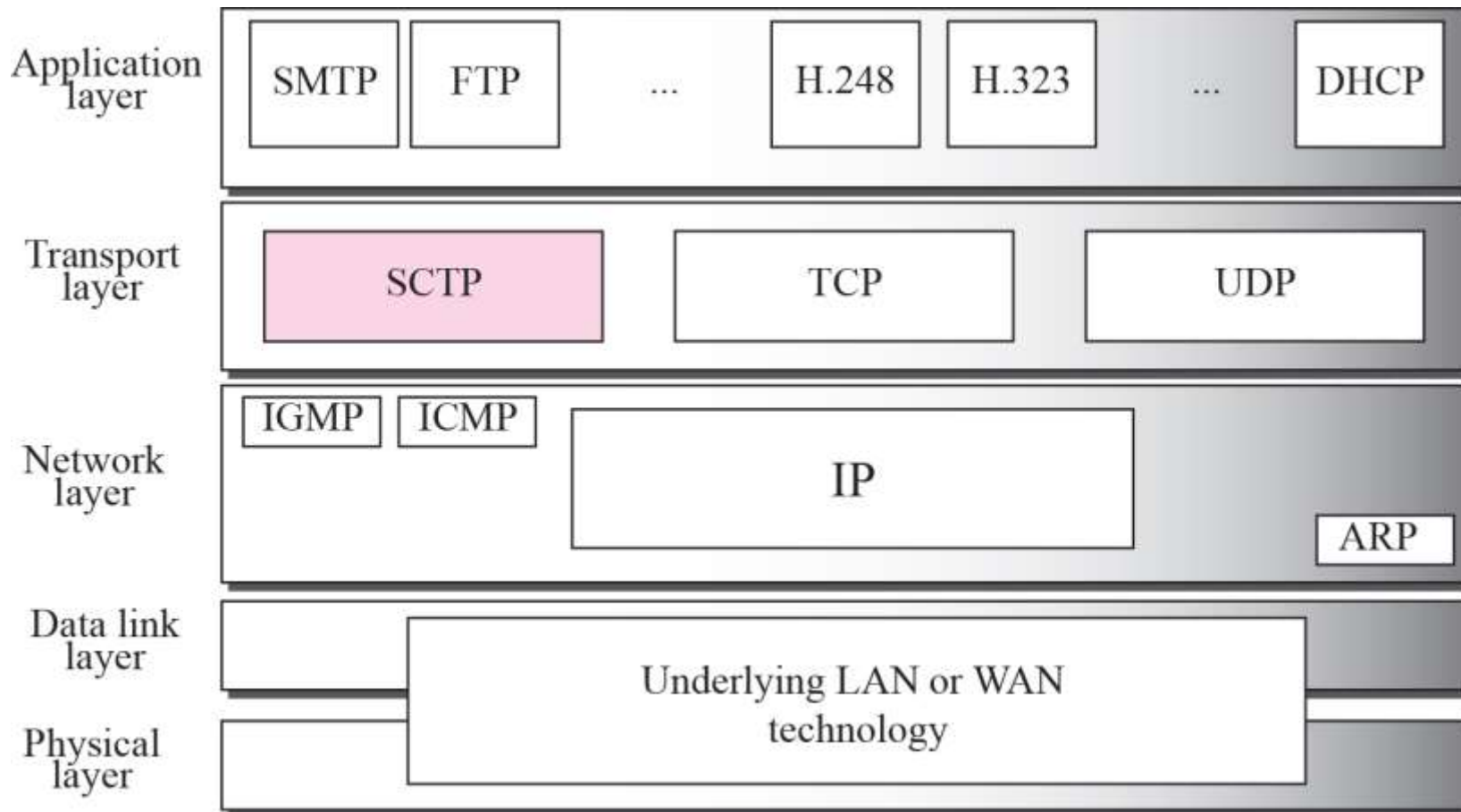
OBJECTIVES:

- ❑ To introduce SCTP as a new transport-layer protocol.**
- ❑ To discuss SCTP services and compare them with TCP.**
- ❑ To list and explain different packet types used in SCTP and discuss the purpose and of each field in each packet.**
- ❑ To discuss SCTP association and explain different scenarios such as association establishment, data transfer, association termination, and association abortion.**
- ❑ To compare and contrast the state transition diagram of SCTP with the corresponding diagram of TCP.**
- ❑ To explain flow control, error control, and congestion control mechanism in SCTP and compare them with the similar mechanisms in TCP.**

16-1 INTRODUCTION

Stream Control Transmission Protocol (SCTP) is a new **reliable, message-oriented transport-layer** protocol. Figure 16.1 shows the relationship of SCTP to the other protocols in the Internet protocol suite. SCTP lies between the application layer and the network layer and serves as the intermediary between the application programs and the network operations.

Figure 16.1 *TCP/IP Protocol suite*

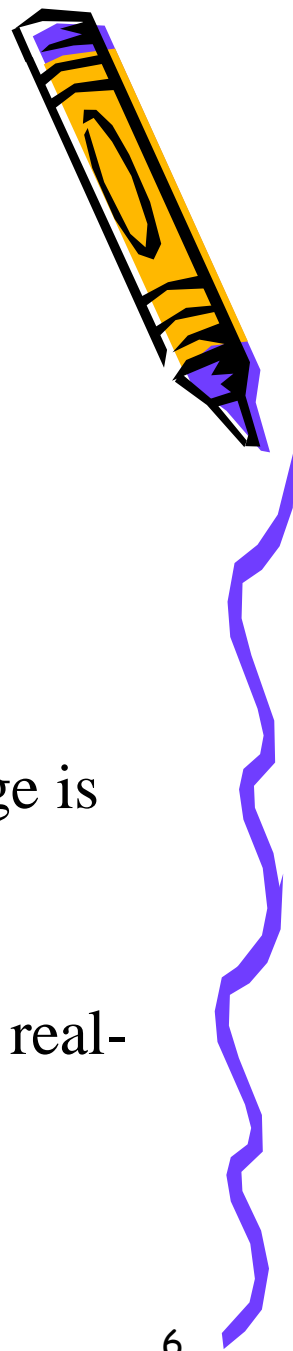




Note

SCTP is a message-oriented, reliable protocol that combines the best features of UDP and TCP.

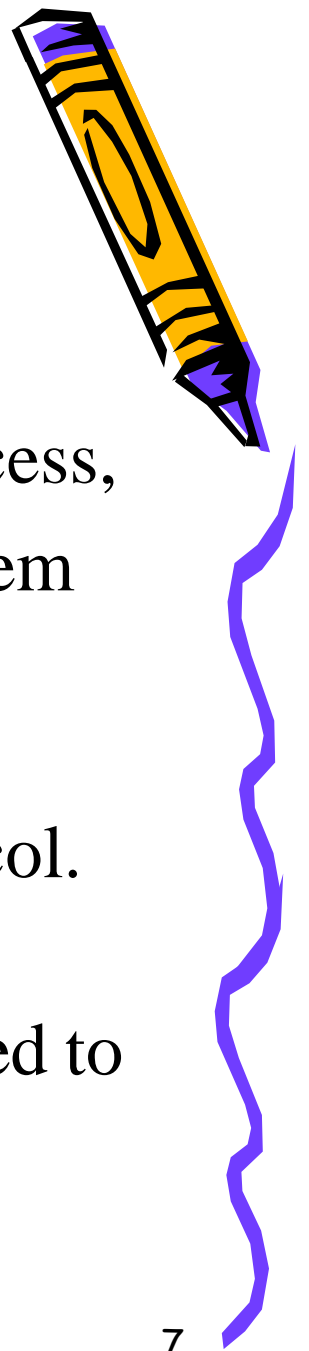
Comparison



- **UDP:** Message-oriented, Unreliable
- A process delivers a message to UDP, which is encapsulated in a user datagram and sent over the network.
- UDP *conserves the message boundaries*; each message is independent from any other message.
- This is a desirable feature when we are dealing with applications such as IP telephony and transmission of real-time data



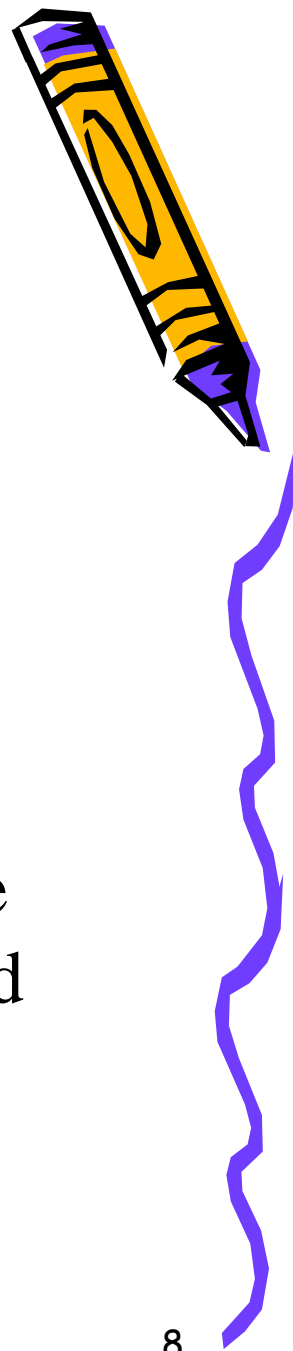
Comparison



- **TCP:** Byte-oriented, Reliable
- It receives a message or messages from a process,
- stores them as a stream of bytes, and sends them in segments.
- There is no preservation of the message boundaries. However, TCP is a reliable protocol.
- The duplicate segments are detected, the lost segments are resent, and the bytes are delivered to the end process in order



Comparison



- **SCTP: Message-oriented, Reliable**
- Combines the best features of UDP and TCP.
- SCTP is a reliable message-oriented protocol.
- It preserves the message boundaries and at the same time detects lost data, duplicate data, and out-of-order



16-2 SCTP SERVICES

Before discussing the operation of SCTP, let us explain the services offered by SCTP to the application layer processes.

Topics Discussed in the Section

- ✓ **Process-to-Process Communication**
- ✓ **Multiple Streams**
- ✓ **Multihoming**
- ✓ **Full-Duplex Communication**
- ✓ **Connection-Oriented Service**
- ✓ **Reliable Service**

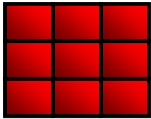


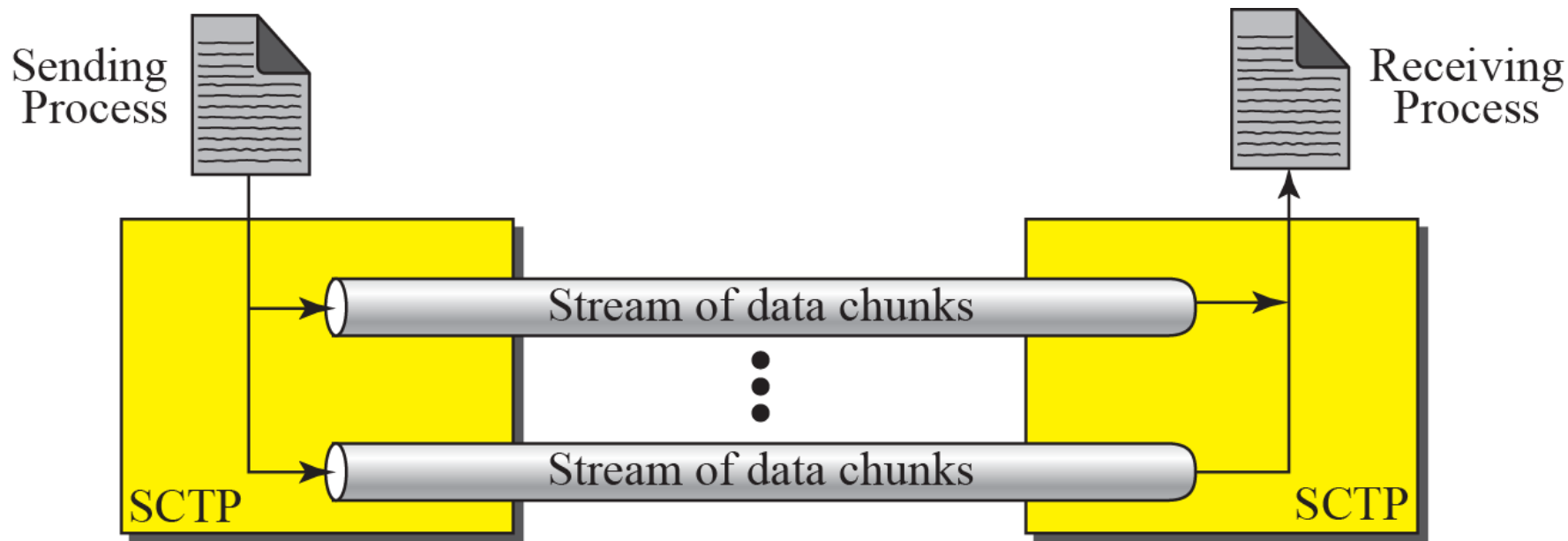
Table 16.1 *Some SCTP applications*

<i>Protocol</i>	<i>Port Number</i>	<i>Description</i>
IUA	9990	ISDN over IP
M2UA	2904	SS7 telephony signaling
M3UA	2905	SS7 telephony signaling
H.248	2945	Media gateway control
H.323	1718, 1719, 1720, 11720	IP telephony
SIP	5060	IP telephony

- Each connection between a TCP client and a TCP server involves one single stream.
- The problem with this approach is that a loss at any point in the stream blocks the delivery of the rest of the data.
- This can be acceptable when we are transferring text; it is not when we are sending real-time data such as audio or video

Figure 16.2 *Multiple-stream concept*

If one of the streams is blocked, the other streams can still deliver their data.



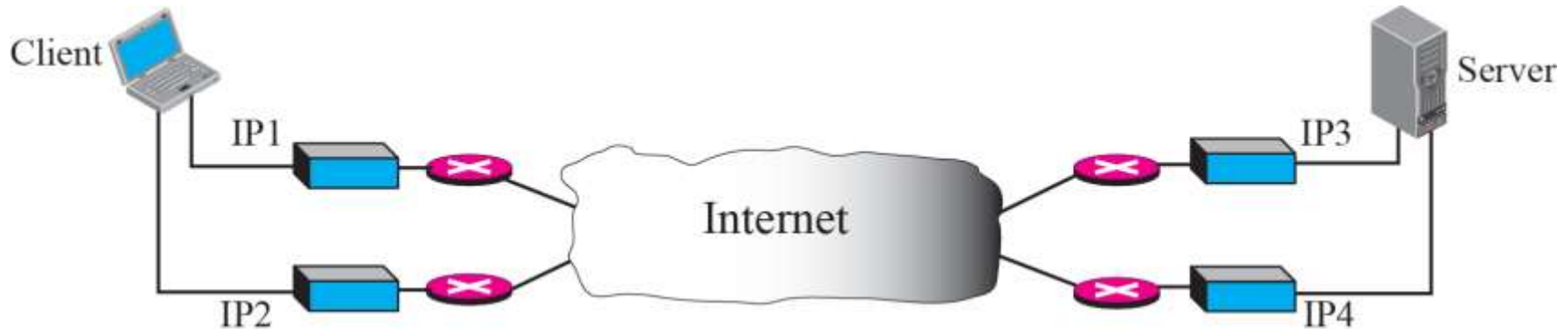
SCTP allows **multis-tream service** in each connection, which is called **association**



An association in SCTP can involve multiple streams.

Figure 16.3 *Multihoming concept*

A TCP connection involves one source and one destination IP address

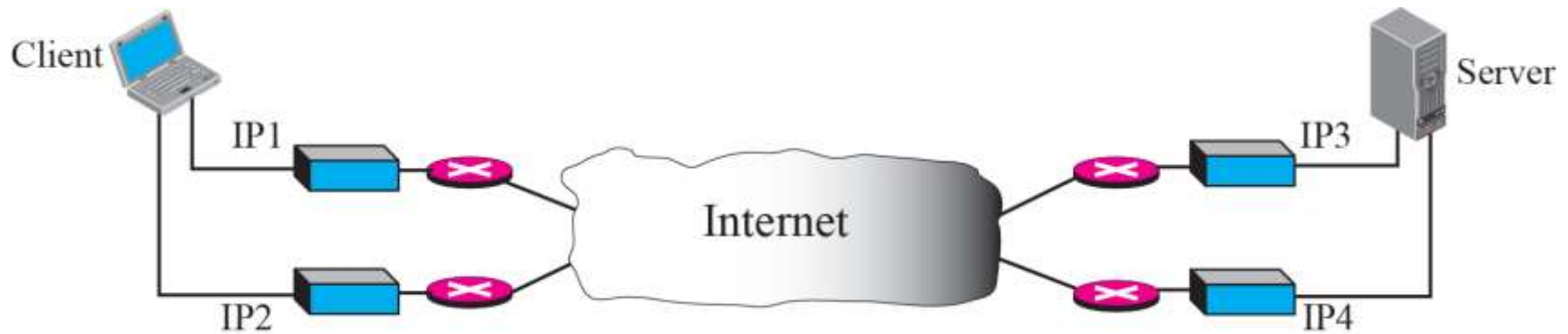


An SCTP association, on the other hand, supports **multihoming service**.

The sending and receiving host can define multiple IP addresses in each end for an association

Figure 16.3 *Multihoming concept*

In this fault-tolerant approach, when one path fails, another interface can be used for data delivery without interruption



Only one pair of IP addresses can be chosen for normal communication; the alternative is used if the main choice fails



Note

SCTP association allows multiple IP addresses for each end.



Full-Duplex Communication

Like TCP, SCTP offers **full-duplex service**, where data can flow in both directions at the same time.

Each SCTP then has a sending and receiving buffer and packets are sent in both directions.



Connection-Oriented Service

Like TCP, SCTP is a connection-oriented protocol. However, in SCTP, a connection is called an **association**.

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

1. The two SCTPs establish an association between each other.
2. Data are exchanged in both directions.
3. The association is terminated

SCTP, like TCP, is a reliable transport protocol. It uses an acknowledgment mechanism to check the safe and sound arrival of data. We will discuss this feature further in the section on error control.

16-3 SCTP FEATURES

Let us first discuss the general features of SCTP and then compare them with those of TCP.


Topics Discussed in the Section

- ✓ **Transmission Sequence Number (TSN)**
- ✓ **Stream Identifier (SI)**
- ✓ **Stream Sequence Number (SSN)**
- ✓ **Packets**
- ✓ **Acknowledgment Number**
- ✓ **Flow Control**
- ✓ **Error Control**
- ✓ **Congestion Control**



Numbering in TCP

The unit of data in TCP is a **byte**. Data transfer in TCP is controlled by numbering bytes using a sequence number.



In SCTP, a data chunk is numbered using a TSN (Transmission Sequence Number)

This 32-bit field defines the transmission sequence number. It is a sequence number that is initialized in an INIT chunk for one direction and in the INIT ACK chunk for the opposite direction.




To distinguish between different streams, SCTP uses an SI (Stream Identifier)

This 16-bit field defines each stream in an association. All chunks belonging to the same stream in one direction carry the same stream identifier.



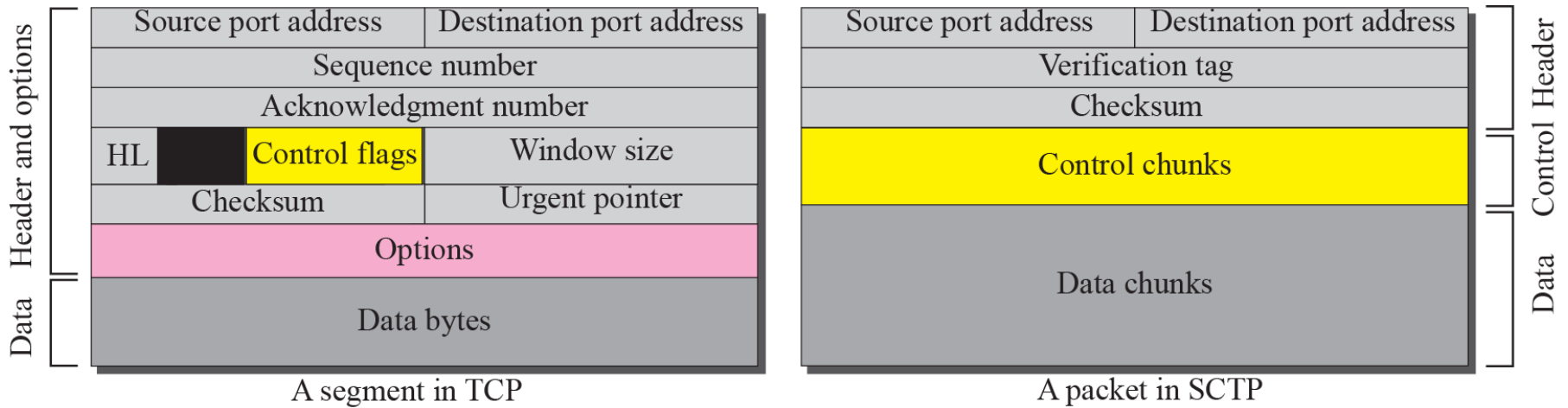
Numbering in TCP

The unit of data in TCP is a **byte**. Data transfer in TCP is controlled by numbering bytes using a sequence number.



Stream sequence number (SSN). This 16-bit field defines a chunk in a particular stream in one direction

Figure 16.4 *Comparison between a TCP segment and an SCTP packet*





Note

TCP has segments; SCTP has packets.

SCTP vs. TCP (1)

- Control information
 - TCP: part of the header
 - **SCTP: several types of control chunks**
- Data
 - TCP: one entity in a TCP segment
 - **SCTP: several data chunks in a packet**
- Option
 - TCP: part of the header
 - **SCTP: handled by defining new chunk types**



SCTP vs. TCP (2)

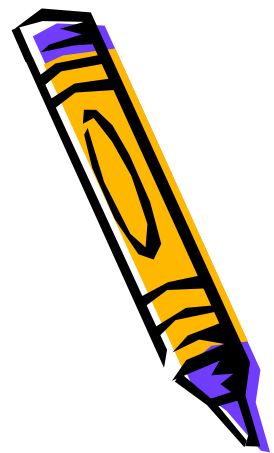
- Mandatory part of the header
 - TCP: 20 bytes, **SCTP: 12 bytes**
 - Reason:
 - **TSN in data chunk's header**
 - **Ack. # and window size are part of control chunk**
 - **No need for header length field (::no option)**
 - **No need for an urgent pointer**
- Checksum

TCP: 16 bits, **SCTP: 32 bit**



SCTP vs. TCP (3)

- Association identifier
 - TCP: none, **SCTP: verification tag**
 - **Multihoming in SCTP**
- Sequence number
 - TCP: one # in the header
 - **SCTP: TSN, SI and SSN define each data chunk**
 - SYN and FIN need to consume one seq. #
 - **Control chunks never use a TSN, SI, or SSN number**

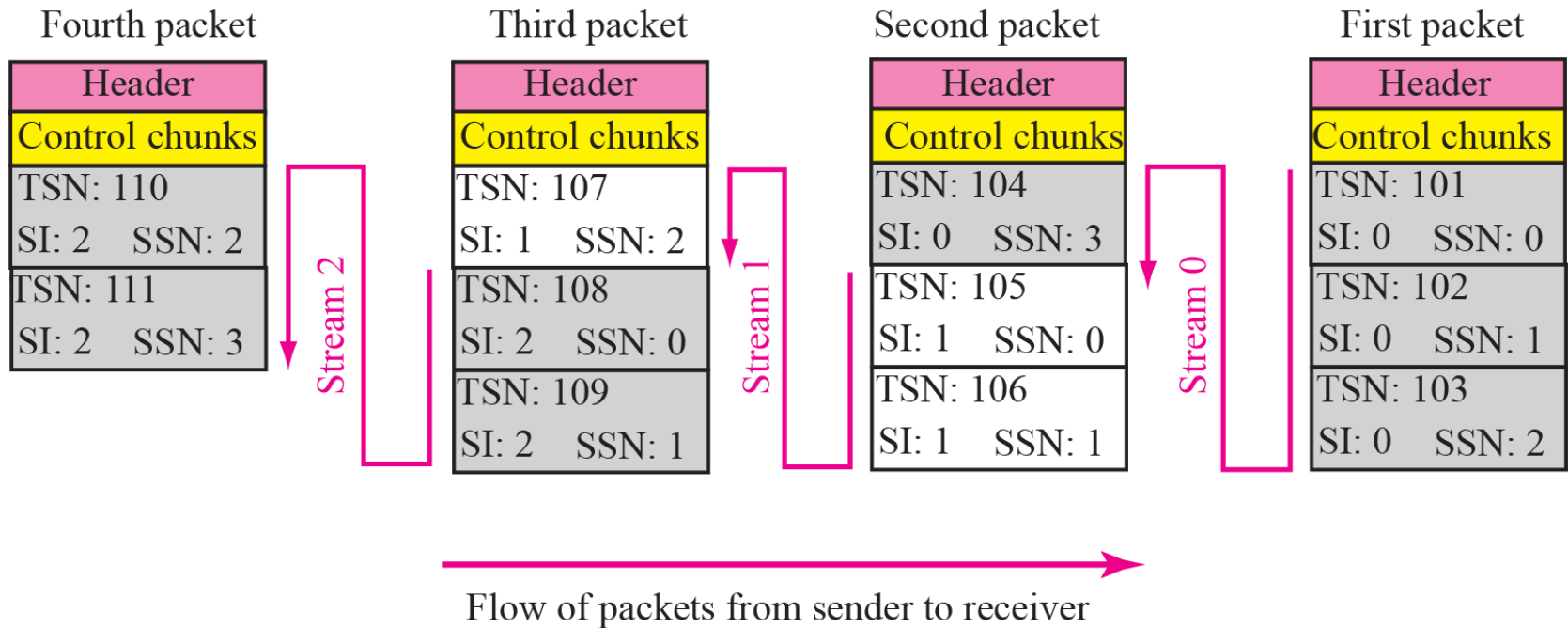




Note

In SCTP, control information and data information are carried in separate chunks.

Figure 16.5 *Packet, data chunks, and streams*





Note

Data chunks are identified by three identifiers: TSN, SI, and SSN. TSN is a cumulative number identifying the association; SI defines the stream; SSN defines the chunk in a stream.



Note

In SCTP, acknowledgment numbers are used to acknowledge only data chunks; control chunks are acknowledged by other control chunks if necessary.

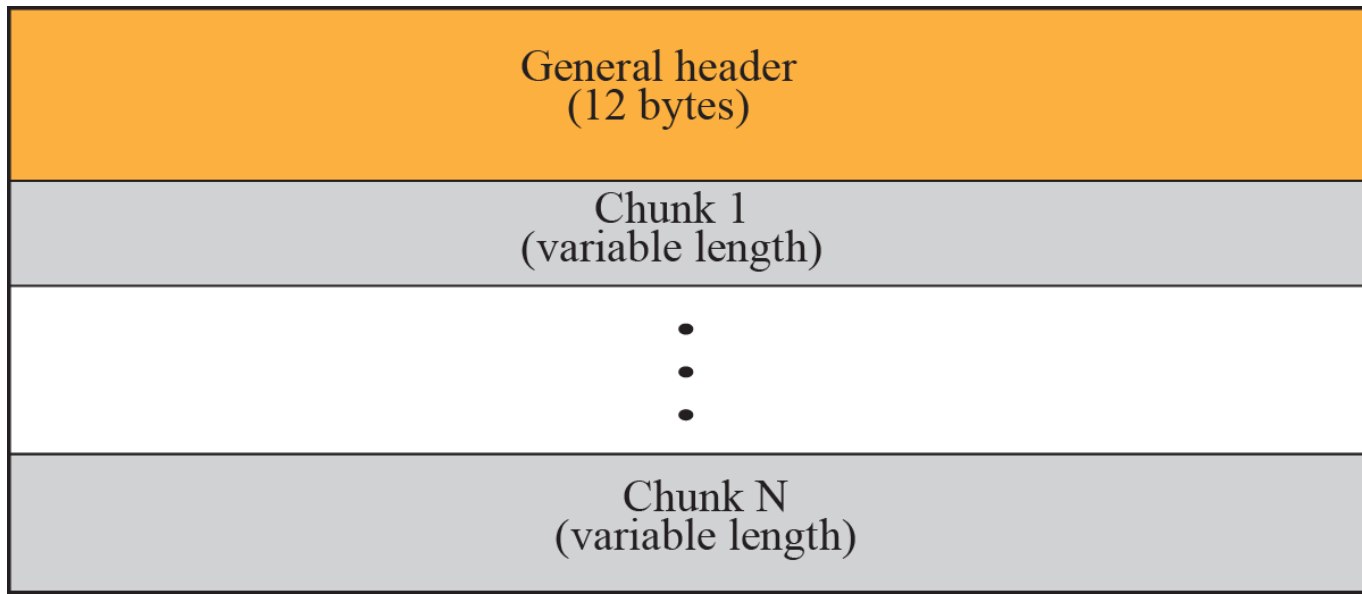
16-4 PACKET FORMAT

In this section, we show the format of a packet and different types of chunks. Most of the information presented in this section will become clear later; this section can be skipped in the first reading or used only as the reference. An SCTP packet has a mandatory general header and a set of blocks called chunks. There are two types of chunks: control chunks and data chunks.

Topics Discussed in the Section

- ✓ **General Header**
- ✓ **Chunks**

Figure 16.6 *SCTP packet format*





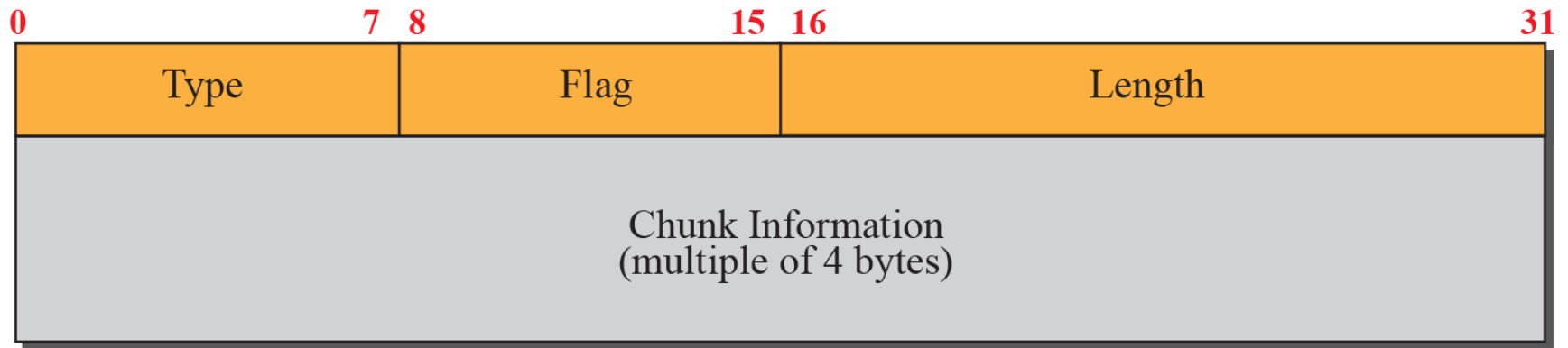
Note

In an SCTP packet, control chunks come before data chunks.

Source port address 16 bits	Destination port address 16 bits
Verification tag 32 bits	
Checksum 32 bits	


Verification tag. This is a number that matches a packet to an association. This prevents a packet from a previous association from being mistaken as a packet in this association.

Common layout of a chunk



The first three fields are common to all chunks;

The information field depends on the type of chunk (data or control)



*Chunks need to terminate on a 32-bit
(4-byte) boundary.*

SCTP requires the information section to be a multiple of 4 bytes; if not, padding bytes (eight 0s) are added at the end of the section.

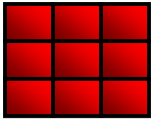


Table 16.2 *Chunks*

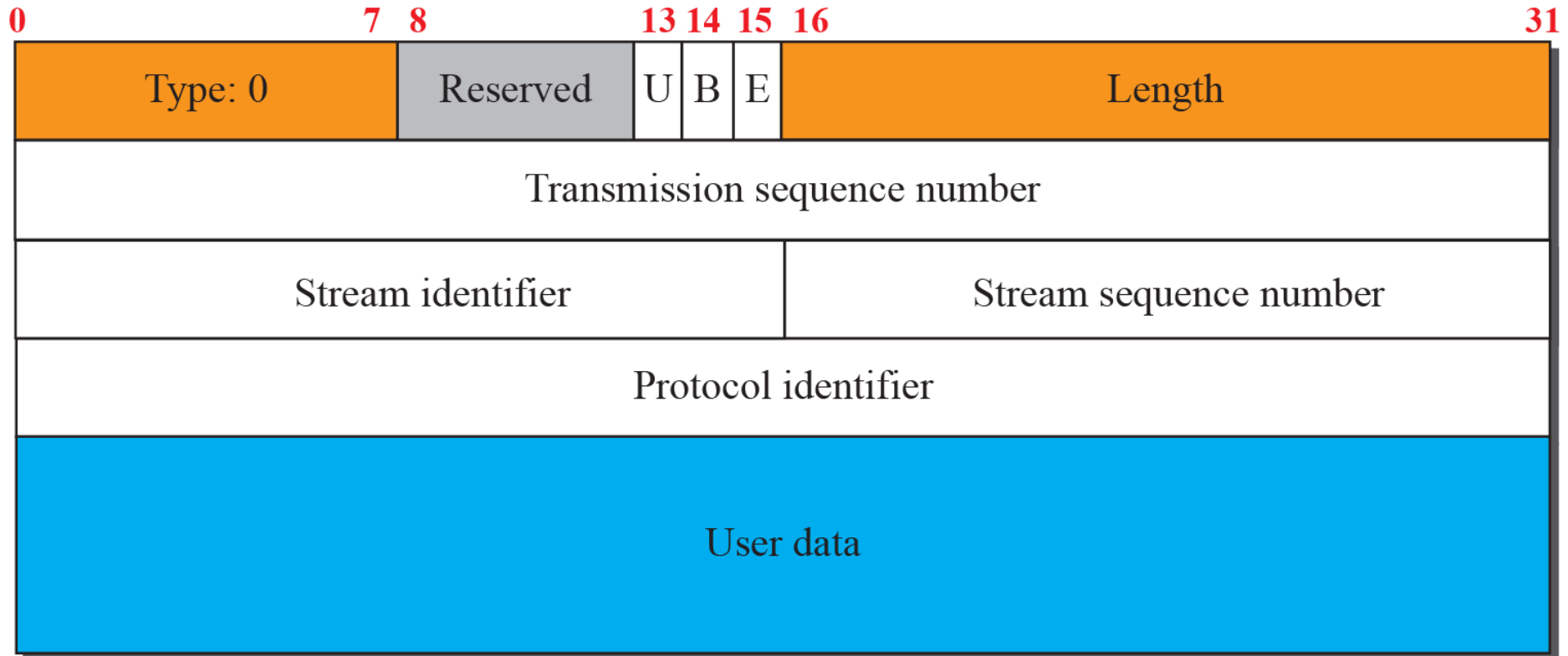
<i>Type</i>	<i>Chunk</i>	<i>Description</i>
0	DATA	User data
1	INIT	Sets up an association
2	INIT ACK	Acknowledges INIT chunk
3	SACK	Selective acknowledgment
4	HEARTBEAT	Probes the peer for liveness
5	HEARTBEAT ACK	Acknowledges HEARTBEAT chunk
6	ABORT	Abort an association
7	SHUTDOWN	Terminates an association
8	SHUTDOWN ACK	Acknowledges SHUTDOWN chunk
9	ERROR	Reports errors without shutting down
10	COOKIE ECHO	Third packet in association establishment
11	COOKIE ACK	Acknowledges COOKIE ECHO chunk
14	SHUTDOWN COMPLETE	Third packet in association termination
192	FORWARD TSN	For adjusting cumulating TSN



Note

The number of padding bytes is not included in the value of the length field.

Figure 16.9 *Data chunk*

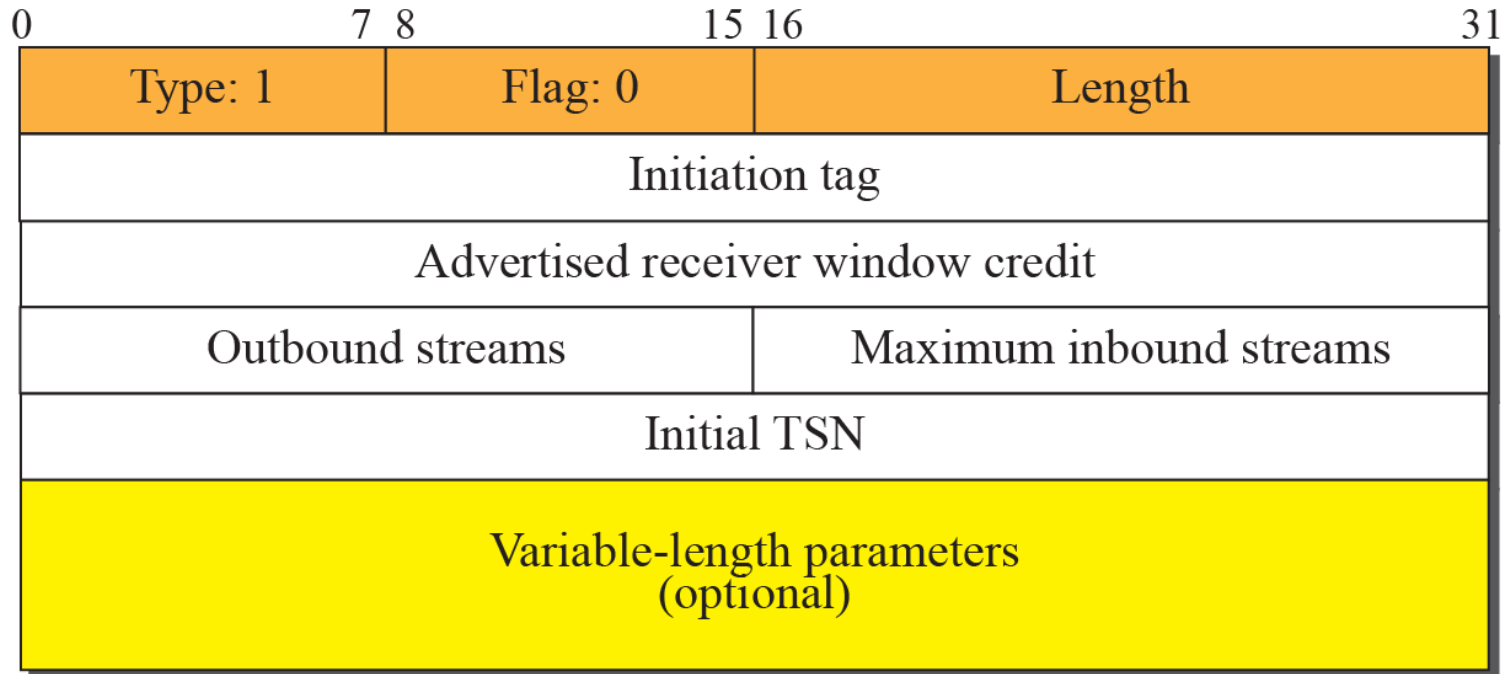




Note

A DATA chunk cannot carry data belonging to more than one message, but a message can be split into several chunks. The data field of the DATA chunk must carry at least one byte of data

Figure 16.10 *INIT chunk*



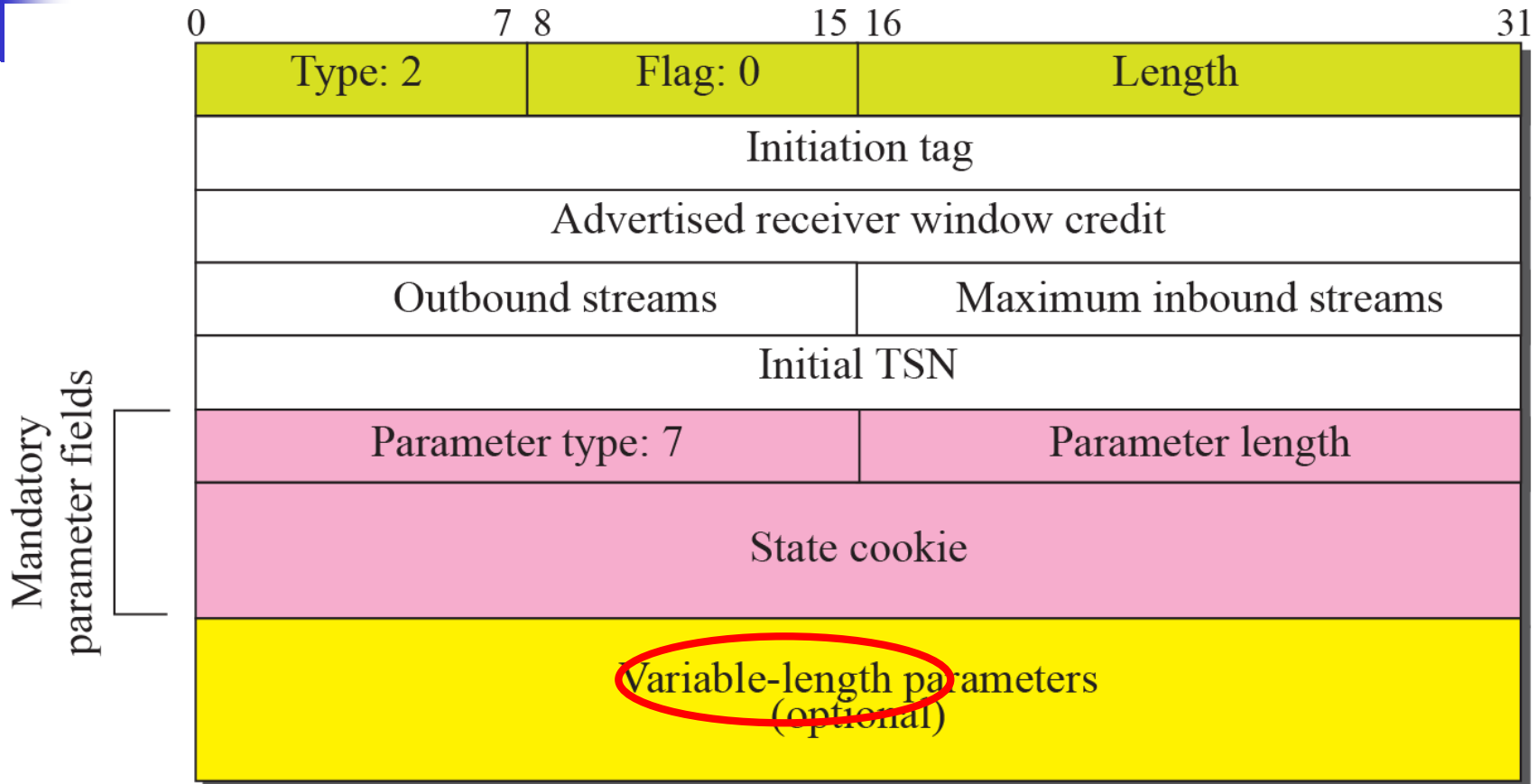


No other chunk can be carried in a packet that carries an INIT chunk.

The **INIT chunk** (initiation chunk) is the first chunk sent by an end point to establish an association.

The packet that carries this chunk cannot carry any other control or data chunks. The value of the verification tag for this packet is 0, which means no tag has yet been defined

Figure 16.11 *INIT ACK chunk*



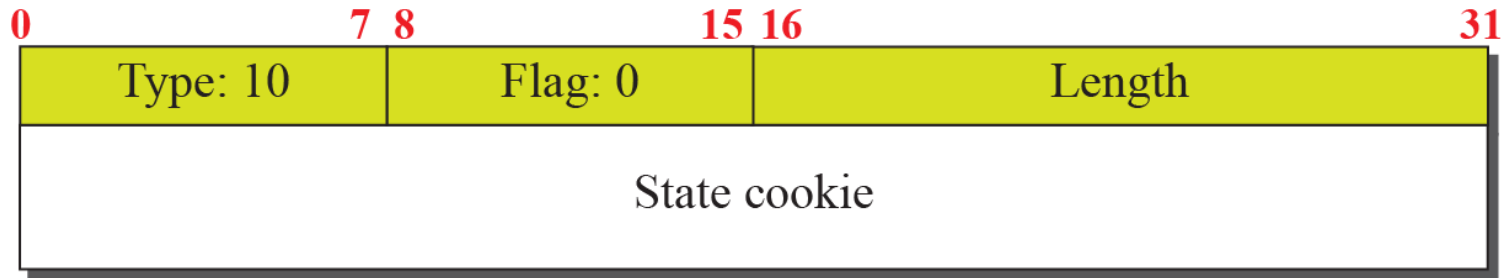
The **INIT ACK chunk** (initiation acknowledgment chunk) is the second chunk sent during association establishment. The packet that carries this chunk cannot carry any data or other control chunks



Note

No other chunk can be carried in a packet that carries an INIT ACK chunk.

Figure 16.12 *COOKIE ECHO chunk*



The **COOKIE ECHO chunk** is the third chunk sent during association establishment.

It is sent by the end point that receives an INIT ACK chunk (normally the sender of the INIT chunk).

The packet that carries this chunk can also carry user data.

Figure 16.13 *COOKIE ACK*



The **COOKIE ACK chunk** is the fourth and last chunk sent during association establishment.

The packet that carries this chunk can also carry user data.

Figure 16.14 *SACK chunk*

The **SACK chunk** (selective ACK chunk) acknowledges the receipt of data packets.

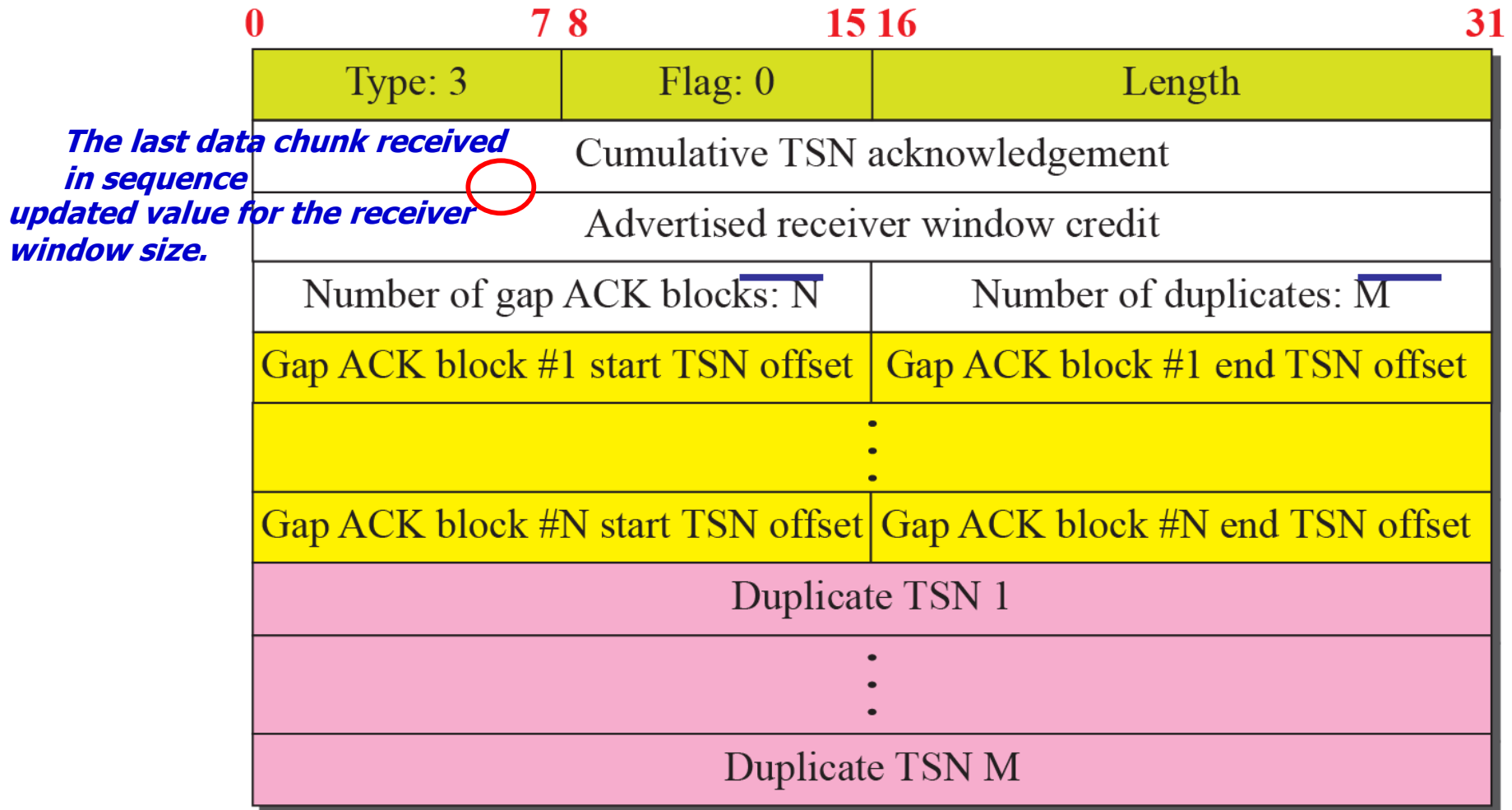
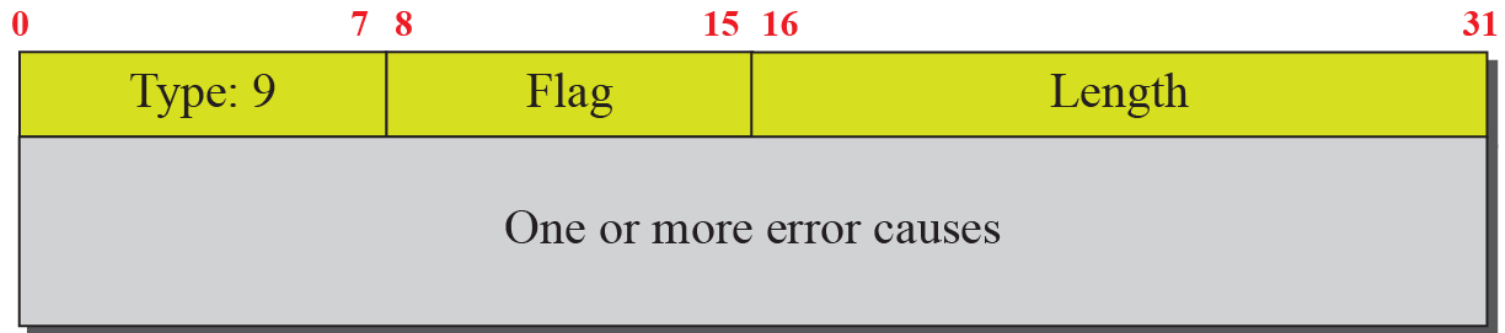


Figure 16.16 *SHUTDOWN chunks*



Figure 16.17 *ERROR chunk*

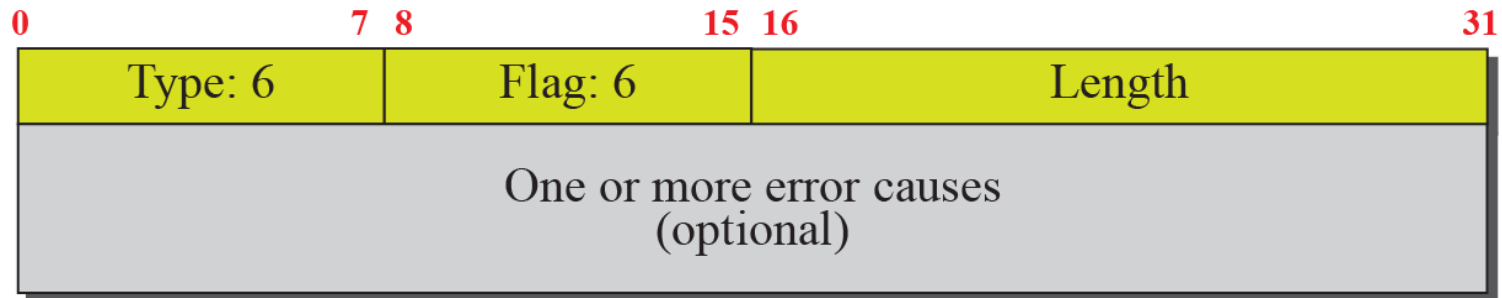


The **ERROR chunk** is sent when an end point finds some error in a received packet. Note that the sending of an ERROR chunk does not imply the aborting of the association. (This would require an ABORT chunk.)

Table 16.3 *Errors*

<i>Code</i>	<i>Description</i>
1	Invalid stream identifier
2	Missing mandatory parameter
3	State cookie error
4	Out of resource
5	Unresolvable address
6	Unrecognized chunk type
7	Invalid mandatory parameters
8	Unrecognized parameter
9	No user data
10	Cookie received while shutting down

Figure 16.18 *ABORT chunk*



The **ABORT chunk** is sent when an end point finds a fatal error and needs to abort the association. The error types are the same as those for the ERROR chunk

- Recently added to the standard (RFC 3758)
- Used to inform the receiver to adjust its cumulative TSN
- It provides partial reliable service

16-5 AN SCTP ASSOCIATION

SCTP, like TCP, is a connection-oriented protocol. However, a connection in **SCTP** is called an **association** to emphasize multihoming.

Topics Discussed in the Section

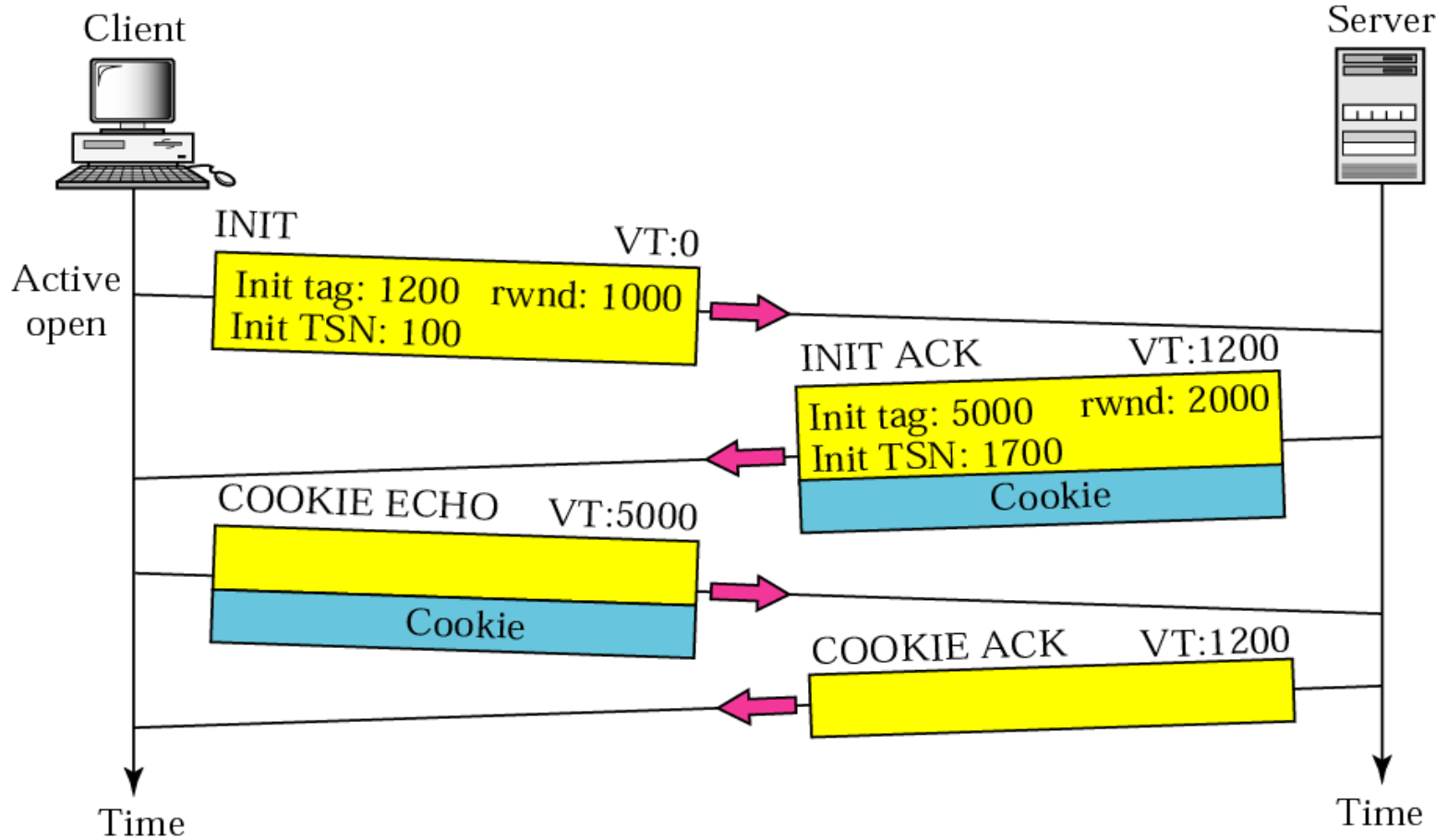
- ✓ **Association Establishment**
- ✓ **Data Transfer**
- ✓ **Association Termination**
- ✓ **Association Abortion**



Note

A connection in SCTP is called an association.

Figure 16.19 *Four-way handshaking*



Verification Tag



- In TCP, a connection is identified by a combination of IP addresses and port numbers
 - A blind attacker can send segments to a TCP server using randomly chosen source and destination port numbers
 - Delayed segment from a previous connection can show up in a new connection that uses the same source and destination port addresses (incarnation)
- Two verification tags, one for each direction, identify an association

*TIME-WAIT
timer*



Verification Tag



- In TCP, a connection is identified by a combination of IP addresses and port numbers
 - A blind attacker can send segments to a TCP server using randomly chosen source and destination port numbers
 - Delayed segment from a previous connection can show up in a new connection that uses the same source and destination port addresses . This was one of the reasons that TCP needs a TIME-WAIT timer when terminating a connection



Verification Tag



- SCTP solves these two problems by using a verification tag, a common value that is carried in all packets traveling in one direction in an association.
- A blind attacker cannot inject a random packet into an association because the packet would most likely not carry the appropriate tag (odds are 1 out of 2^{32}).
- Two verification tags, one for each direction, identify an association.



Cookie (1)



- In TCP

- Each time the server receives a SYN segment, it sets up a TCB and allocates other resources

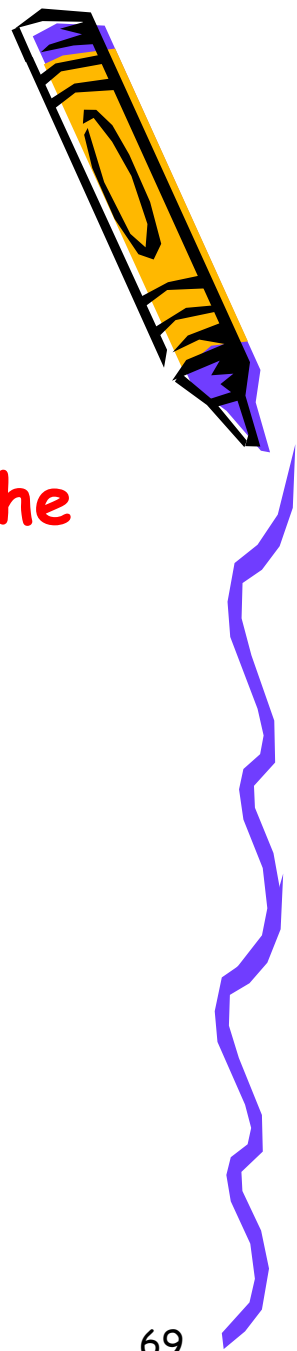
- In SCTP

- Postpone the allocation of resources until the reception of the third packet, when the IP address of the sender is verified



Cookie (2)

- In SCTP
 - The information received in the first packet must somehow be saved until the third packet arrives





Note

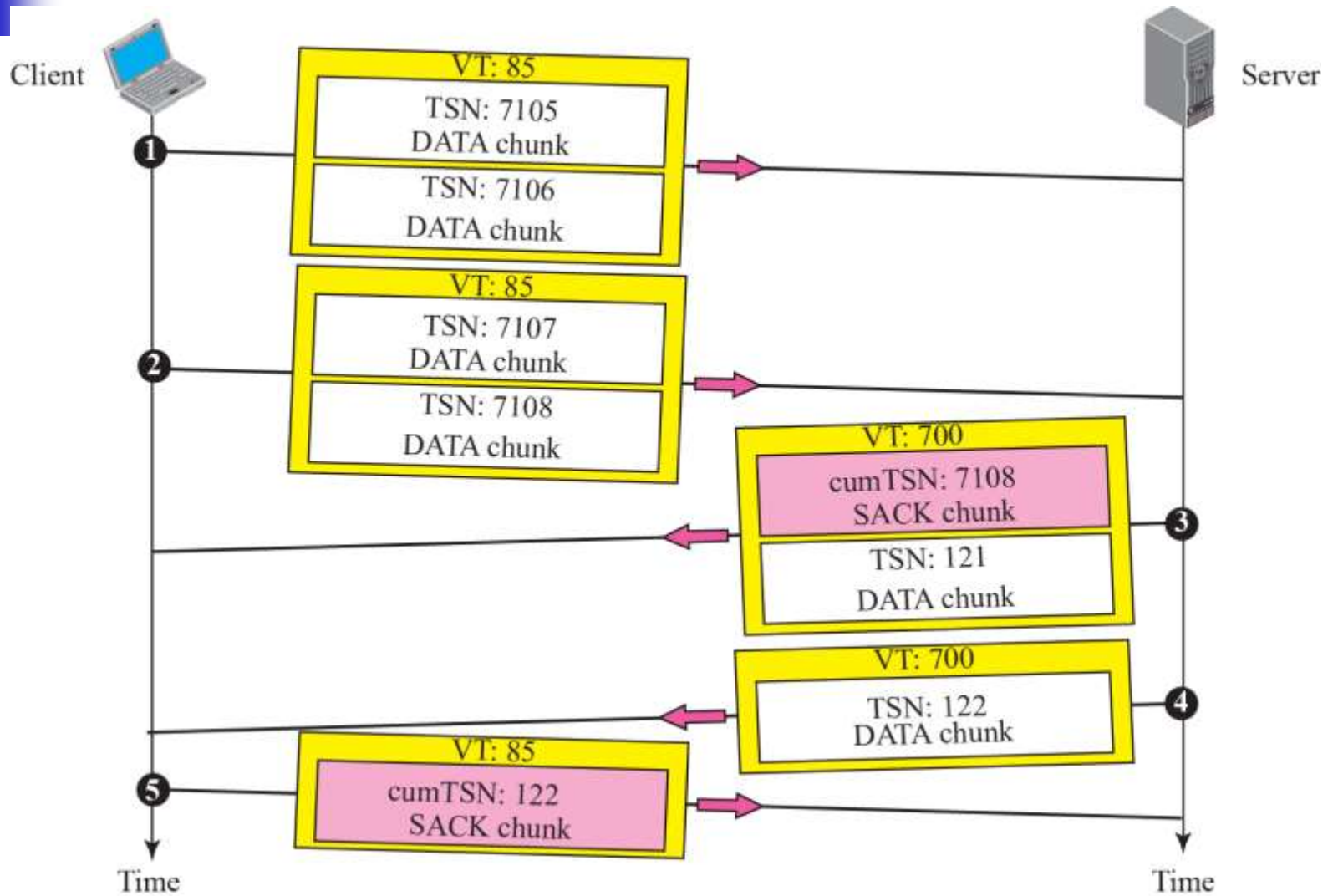
No other chunk is allowed in a packet carrying an INIT or INIT ACK chunk. A COOKIE ECHO or a COOKIE ACK chunk can carry data chunks.



Note

In SCTP, only data chunks consume TSNs; data chunks are the only chunks that are acknowledged.

Figure 16.20 Simple data transfer



The client uses the verification tag 85, the server 700



Note

The acknowledgment in SCTP defines the cumulative TSN, the TSN of the last data chunk received in order.

Multi-homing Data Transfer



- We discussed the multihoming capability of SCTP, a feature that distinguishes SCTP from UDP and TCP
- Multihoming allows both ends to define multiple IP addresses for communication
- Only one of these addresses can be defined as the primary address; the rest are alternative addresses
- The primary address is defined during association establishment



Multi-homing Data Transfer



- Primary address

- Defined during association establishment
- Determined by the other end
- The process can always override the primary address (explicitly) of the current association.
- SACK is sent to the address from which the corresponding SCTP packet originated



Multi-stream Delivery

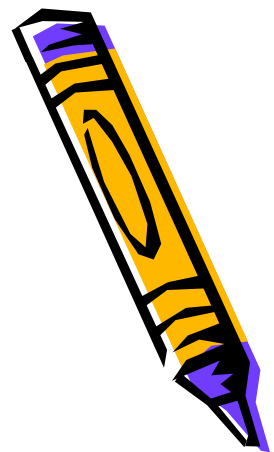


- Interesting feature in SCTP
 - Distinction between data transfer and data delivery
 - Data transfer: TSN (error/flow control)
 - Data delivery: SI, SSN
- Data delivery (in each stream)
 - Ordered (default)
 - Unordered



Multi-stream Delivery

- Data delivery (in each stream)
 - **Ordered:**
 - In ordered data delivery, data chunks in a stream use stream sequence numbers (SSNs) to define their order in the stream
 - SCTP is responsible for message delivery according to the SSN defined in the chunk
 - This may delay the delivery because some chunks may arrive out of order. In unordered data delivery



Multi-stream Delivery

- **Unordered:**

In unordered data delivery, the data chunks in a stream have the U flag set, but their SSN field value is ignored. They do not consume SSNs



Fragmentation



- IP fragmentation vs. SCTP
 - SCTP preserves the boundaries of the msg from process to process when creating a DATA chunk from a message if the size of the msg does not exceed the MTU (maximum transmission unit) of the path
- SCTP fragmentation
 - Each fragment carries a different TSN
 - All header chunks carries the same SI, SSN, payload protocol ID, and U flag



Figure 16.21 Association termination

SCTP does not allow a "half-closed" association

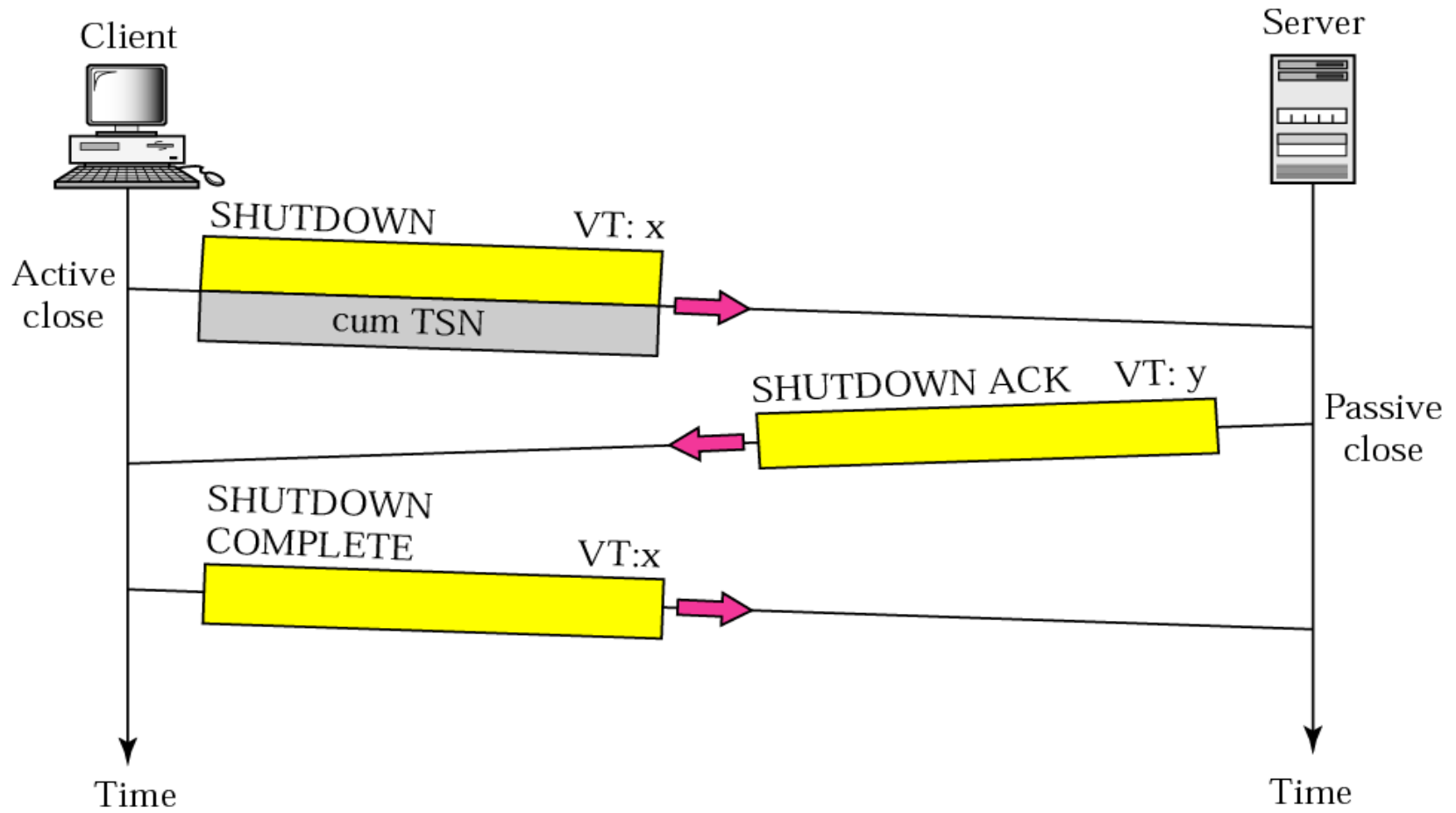


Figure 16.22 *Association abortion*

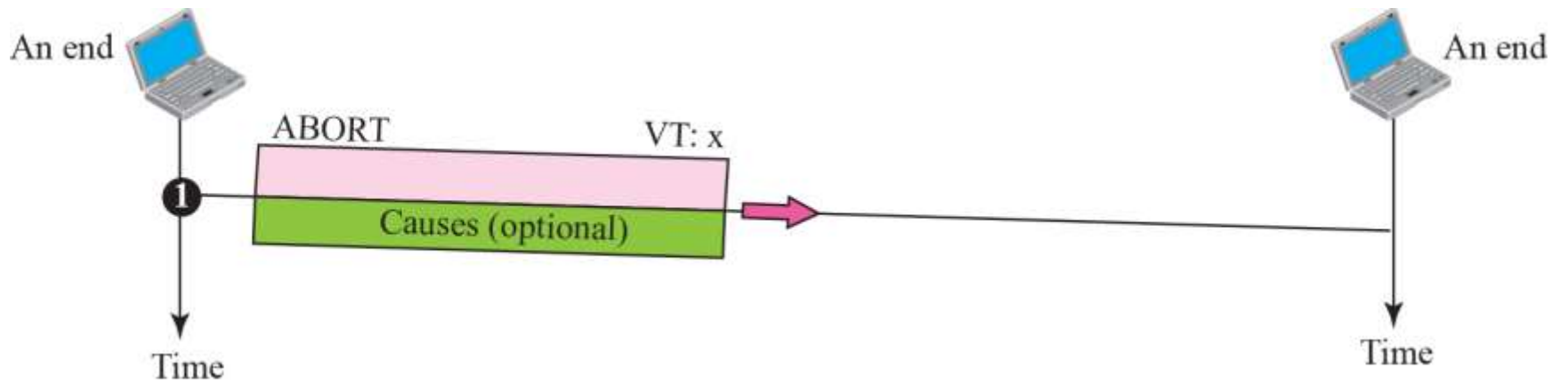


Figure 16.24 *A common scenario of state*

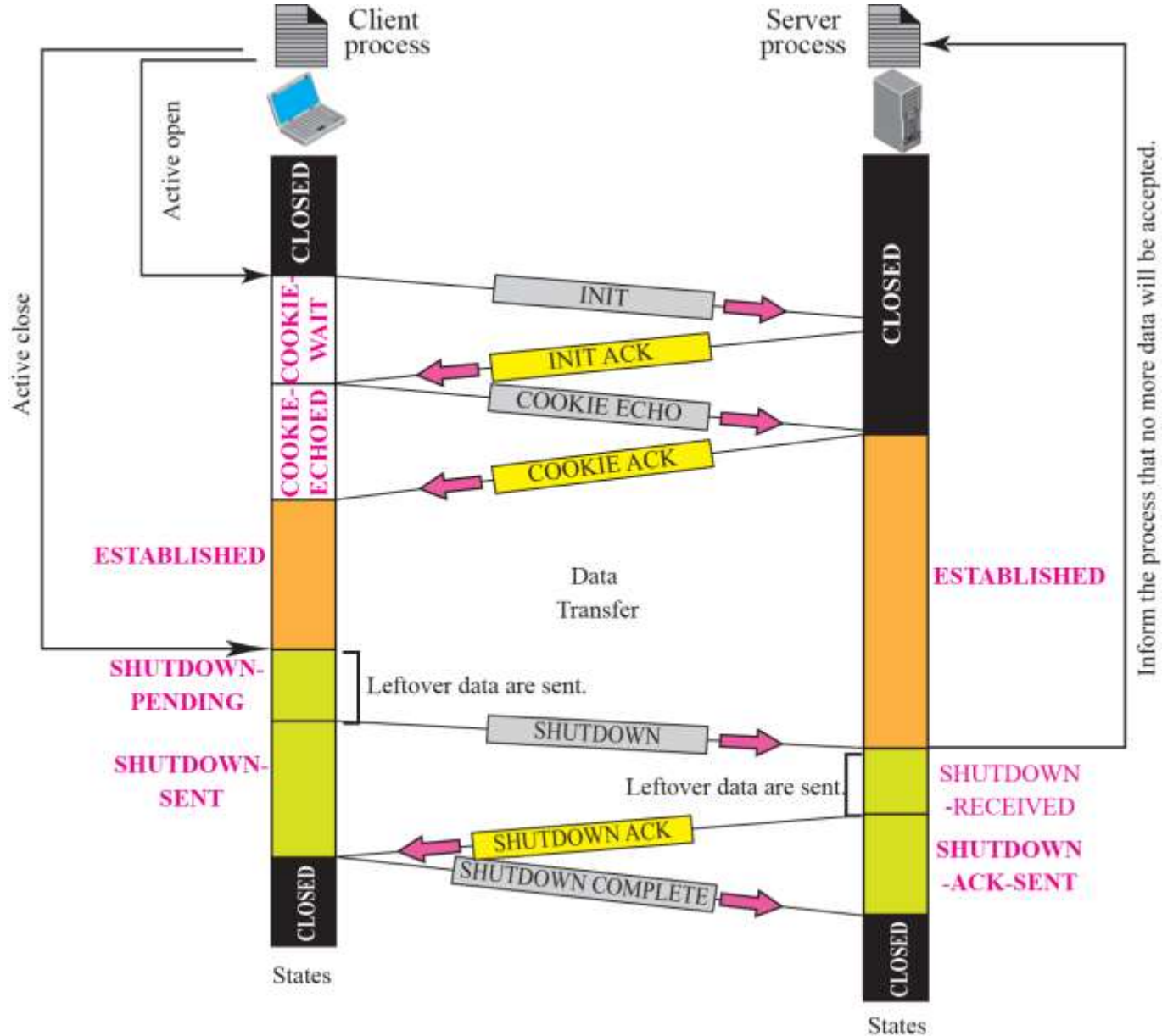
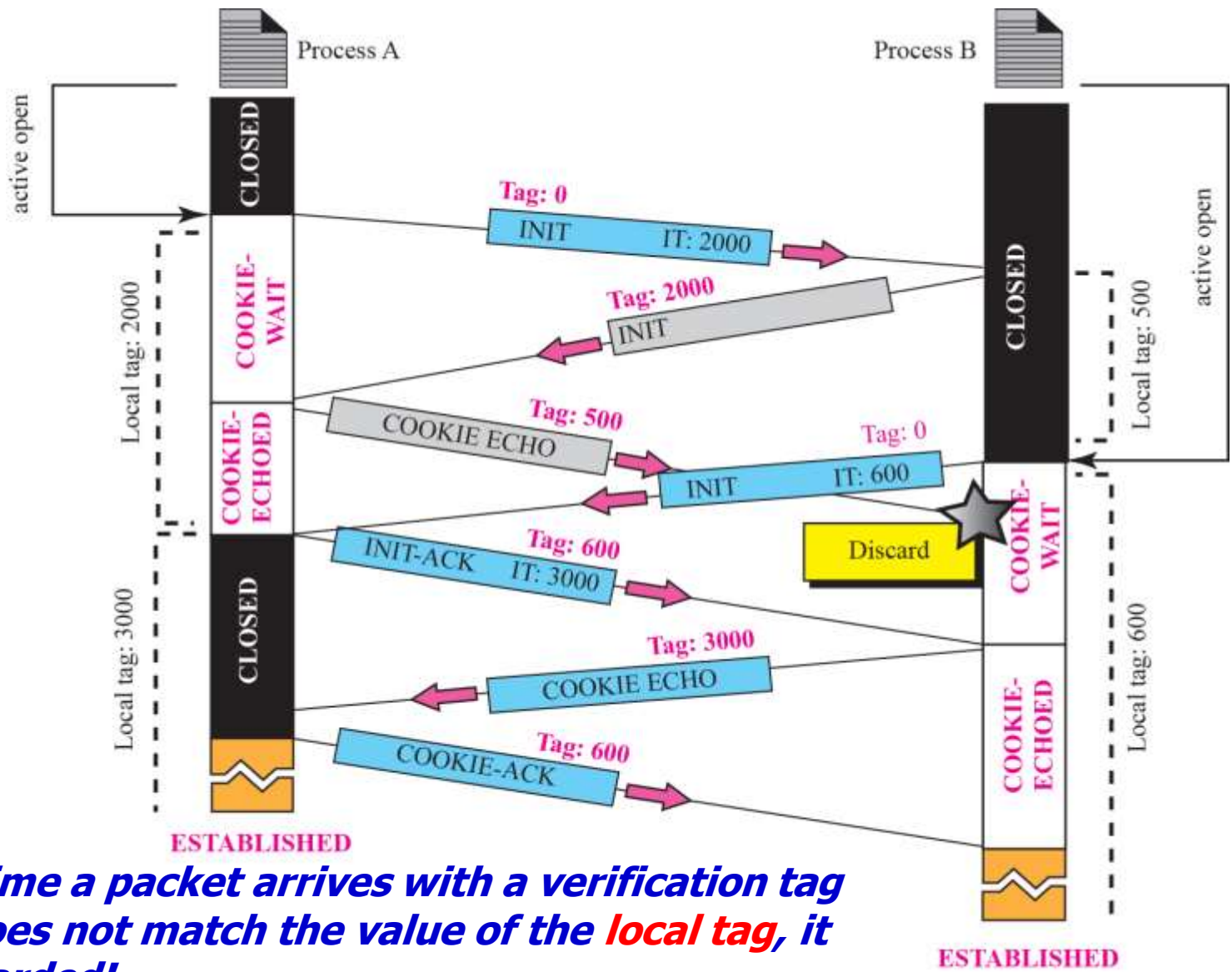
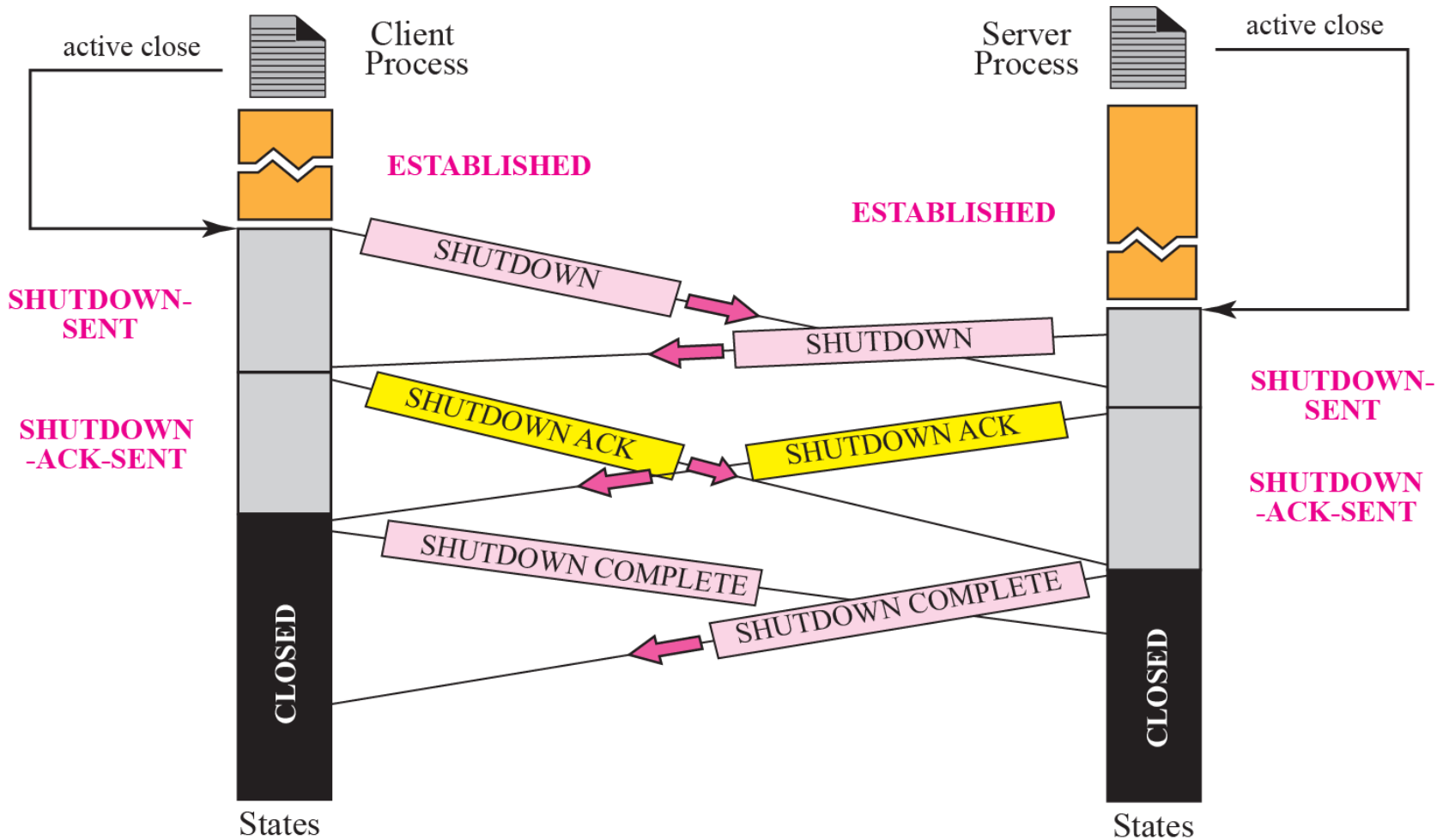


Figure 16.25 *Simultaneous open*



Each time a packet arrives with a verification tag that does not match the value of the *local tag*, it is discarded!

Figure 16.26 *Simultaneous close*



16-7 FLOW CONTROL

Flow control in SCTP is similar to that in TCP. In TCP, we need to deal with only one unit of data, the byte. In **SCTP**, we need to handle two units of data, the **byte** and the **chunk**. The values of *rwnd* and *cwnd* are expressed in bytes; the values of TSN and acknowledgments are expressed in chunks.

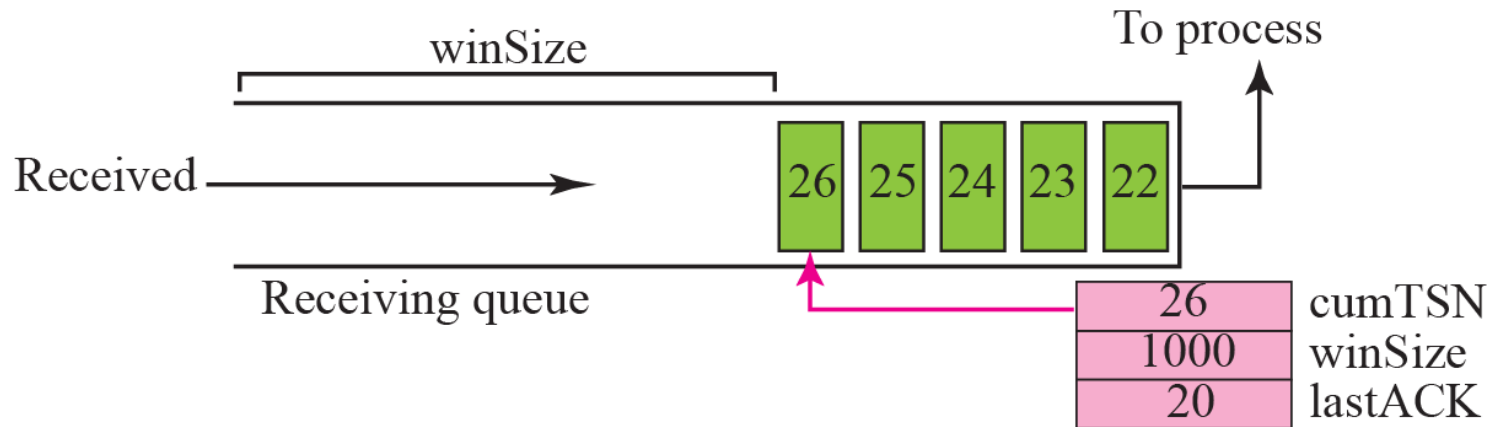
Topics Discussed in the Section

- ✓ **Receiver Site**
- ✓ **Sender Site**
- ✓ **A Scenario**

Figure 16.27 Flow control, receiver site

rwnd, cwnd: in bytes

TSN and Acknowledgement : in chunks



The first variable holds the last TSN received, *cumTSN*.

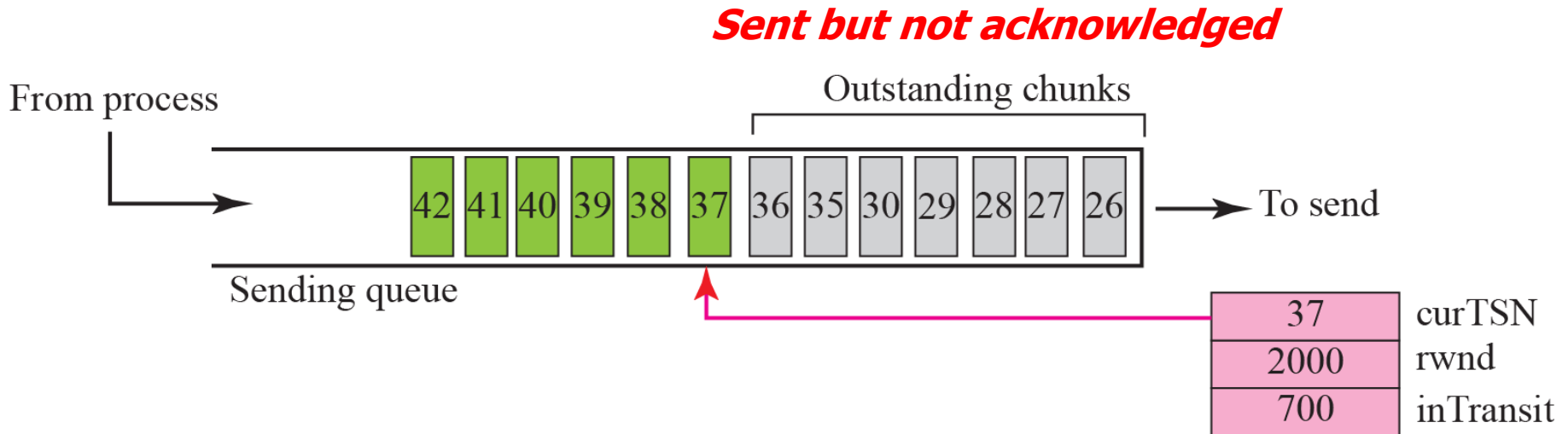
The second variable holds the available buffer size, *winsize*.

The third variable holds the last accumulative acknowledgment, *lastACK*

1. When the site receives a data chunk, it stores it at the end of the buffer (queue) and subtracts the size of the chunk from **winSize**. The TSN number of the chunk is stored in the **cumTSN** variable.
2. When the process reads a chunk, it removes it from the queue and adds the size of the removed chunk to **winSize** (recycling).
3. When the receiver decides to send a SACK, it checks the value of **lastAck**; if it is less than **cumTSN**, it sends a SACK with a cumulative TSN number equal to the **cumTSN**. It also includes the value of **winSize** as the advertised window

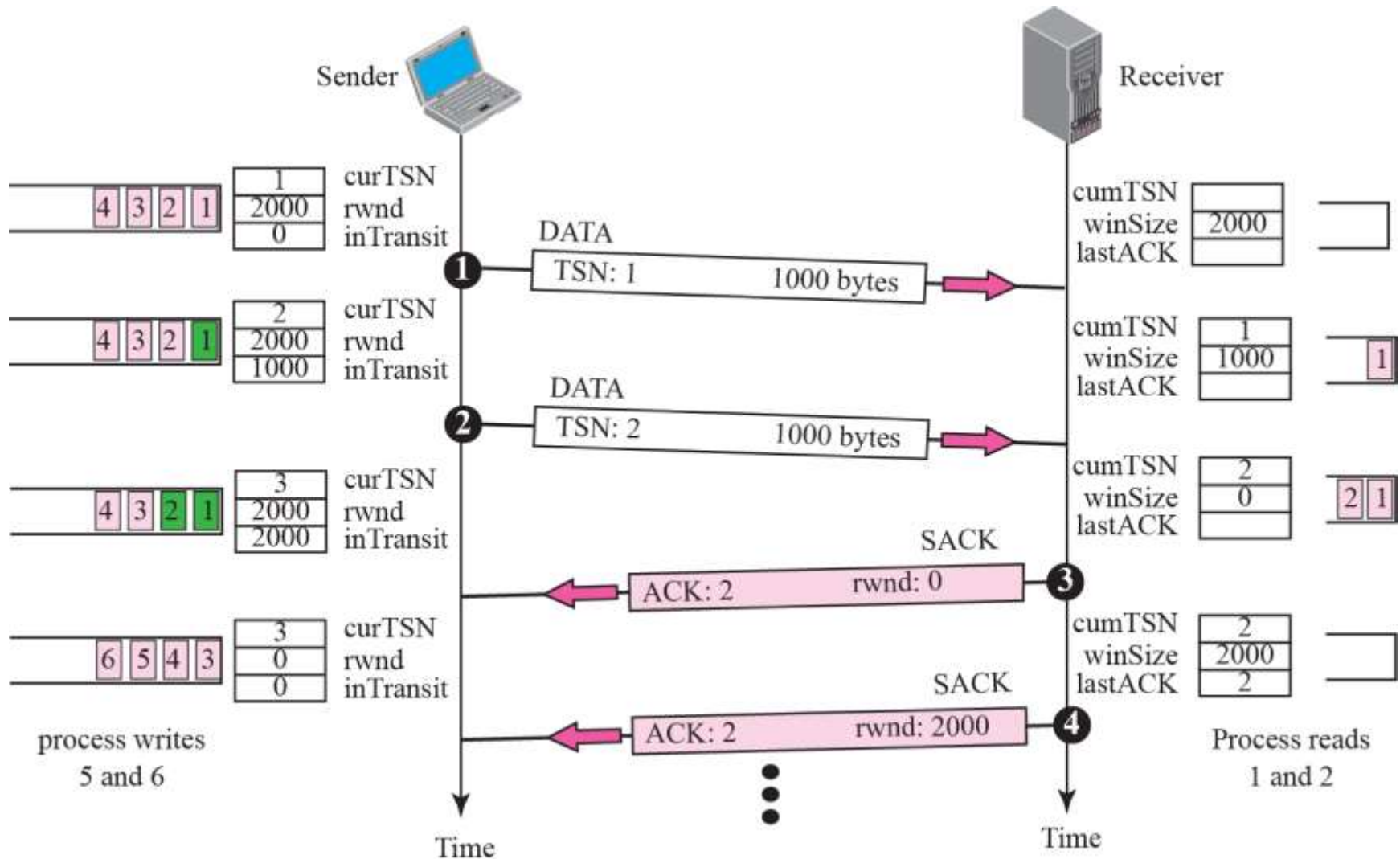
Figure 16.28 Flow control, **sender site**

1. A chunk pointed to by **curTSN** can be sent if the size of the data is less than or equal to the quantity (**rwnd-inTransit**)



2. When a SACK is received, the chunks with a TSN less than or equal to the cumulative TSN in the SACK are removed from the queue and discarded. The values of **rwnd** and **inTransit** are updated properly

Figure 16.29 *Flow control scenario*



16-8 ERROR CONTROL

SCTP, like TCP, is a reliable transport-layer protocol. It uses a SACK chunk to report the state of the receiver buffer to the sender. Each implementation uses a different set of entities and timers for the receiver and sender sites. We use a very simple design to convey the concept to the reader.

Topics Discussed in the Section

- ✓ **Receiver Site**
- ✓ **Sender Site**
- ✓ **Sending Data Chunks**
- ✓ **Generating SANK Chunks**

Figure 16.30 *Error-control receiver site*

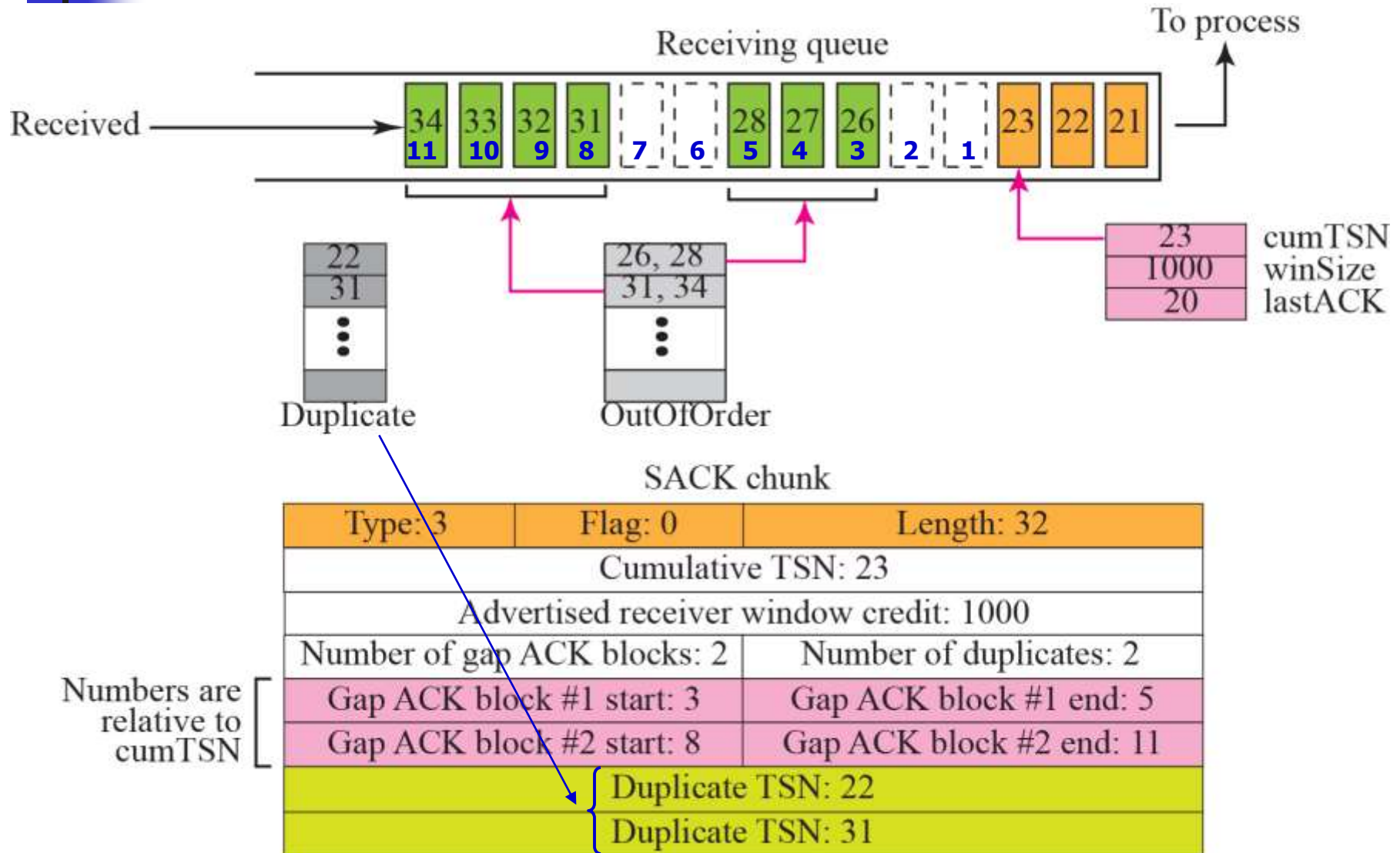
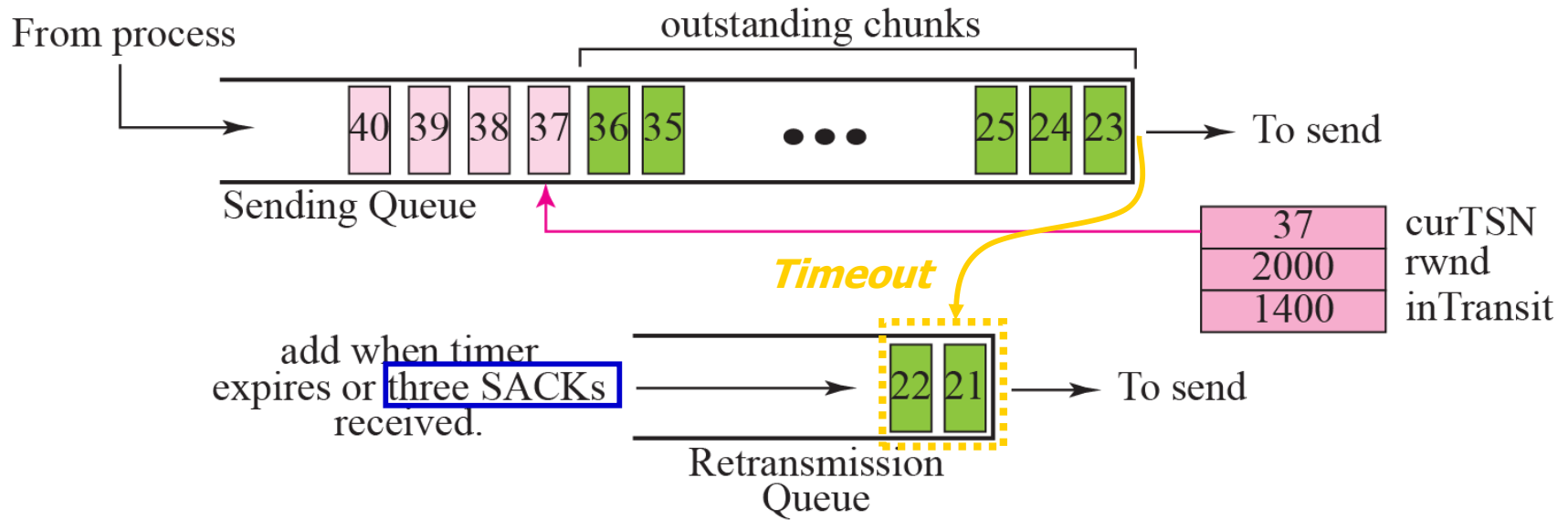


Figure 16.31 *Error control, sender site*

Assume 100 bytes per chunk

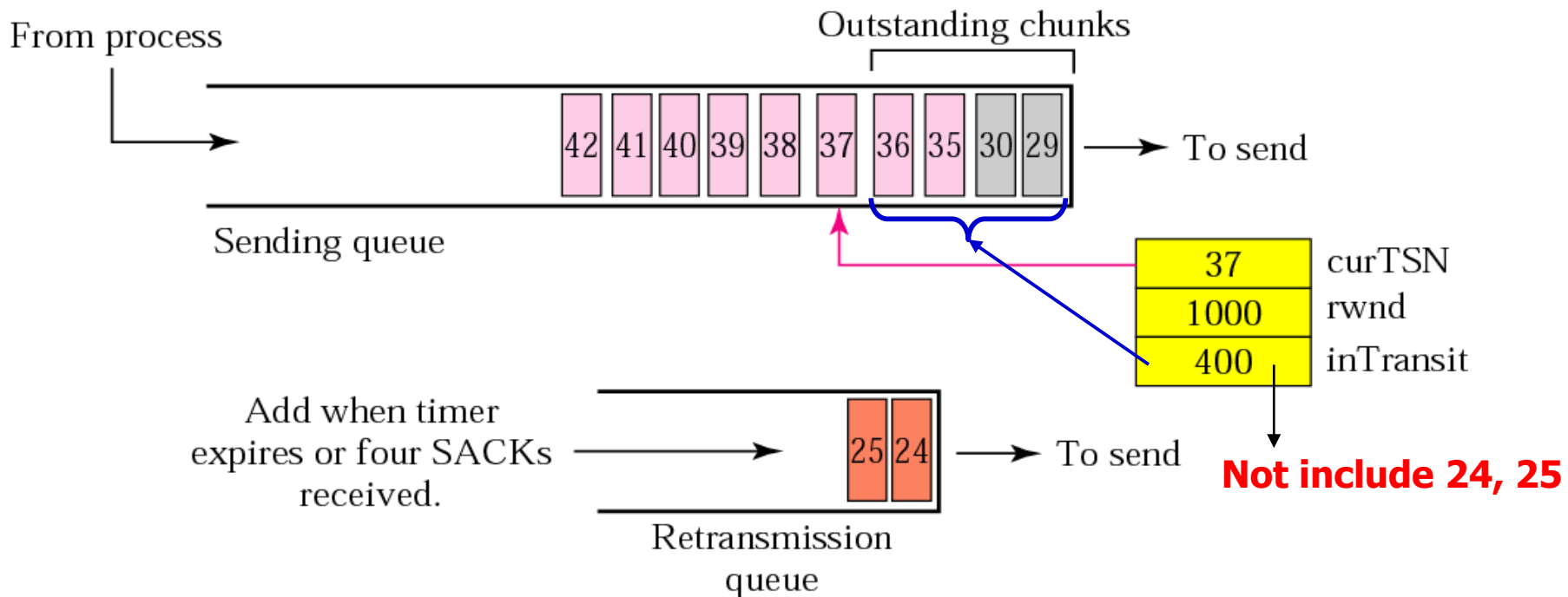


We assume that each chunk is 100 bytes, which means that 1400 bytes of data (chunks 23 to 36) are in transit

The chunks in the retransmission queue have priority

Figure 16.32 New state at the **sender site** after receiving a SACK chunk

1. **Chunks 26-28, 31-34 are removed.**
2. **The value of *rwnd* is changed to 1000 as advertised in the SACK chunk.**



3. **Also assume timer for chunks 24, 25 has expired.**
4. **4 chunks are now in transit, so *inTransit* becomes 400.**

Generating SACK Chunks



- **Piggybacking:**
 - When an end sends a DATA chunk to the other end, it must include a SACK chunk advertising the receipt of unacknowledged DATA chunks.
- **Delay sending of SACK no more than 500ms**
- **Send a SACK immediately when**
 - a packet arrives with **out-of-order data chunks**
 - a packet arrives with **duplicate data chunks** and no new data chunks



16-9 CONGESTION CONTROL

SCTP, like TCP, is a transport layer protocol with packets subject to congestion in the network. The SCTP designers have used the same strategies we described for congestion control in Chapter 15 for TCP. SCTP has slow start, congestion avoidance, and congestion detection phases. Like TCP, SCTP also uses fast retransmission and fast recovery.

- ✓ **Congestion Control and Multihoming**
 - ✓ **Explicit Congestion Notification**
- 

*Need to have different values
of cwnd for each IP address*

*It is a process that enables a receiver to explicitly inform the sender of any congestion experienced in the network.
E.g. the receiver encounters many delayed or lost packets.*

Chapter 17

Introduction to the Application Layer

Edited & Presented by:

Dr. Mohammad Alhammouri

TCP/IP Protocol Suite (B A. Forouzan)

OBJECTIVES:

- ❑ To introduce client-server paradigm.**
- ❑ To introduce socket interfaces and list some common functions in this interface.**
- ❑ To discuss client-server communication using connectionless iterative service offered by UDP.**
- ❑ To discuss client-server communication using connection-oriented concurrent service offered by TCP.**
- ❑ To give an example of a client and a server program using UDP.**
- ❑ To give an example of a client and a server program using TCP.**
- ❑ To briefly discuss the peer-to-peer paradigm and its application.**

Chapter Outline

- 17.1 Client-Server Paradigm*
- 17.2 Peer-to-Peer Paradigm*

17-1 CLIENT-SERVER PARADIGM

The purpose of a network, or an internetwork, is to provide services to users: A user at a local site wants to receive a service from a computer at a remote site. One way to achieve this purpose is to run two programs. A local computer runs a program to request a service from a remote computer; the remote computer runs a program to give service to the requesting program. This means that two computers, connected by an internet, must each run a program, one to provide a service and one to request a service.

Topics Discussed in the Section

- ✓ **Server**
- ✓ **Client**
- ✓ **Concurrency**
- ✓ **Socket Interfaces**
- ✓ **Communication Using UDP**
- ✓ **Communication Using TCP**
- ✓ **Predefined Client-Server Applications**

Concurrency in Clients

Clients can be run on a machine either iteratively or concurrently. Running clients *iteratively* means running them one by one; one client must start, run, and terminate before the machine can start another client. Most computers today, however, allow *concurrent* clients; that is, two or more clients can run at the same time.

Concurrency in Servers

An *iterative* server can process only one request at a time; it receives a request, processes it, and sends the response to the requestor before it handles another request. A concurrent server, on the other hand, can process many requests at the same time and thus can share its time between many requests.

Figure 17.1 *Server types*

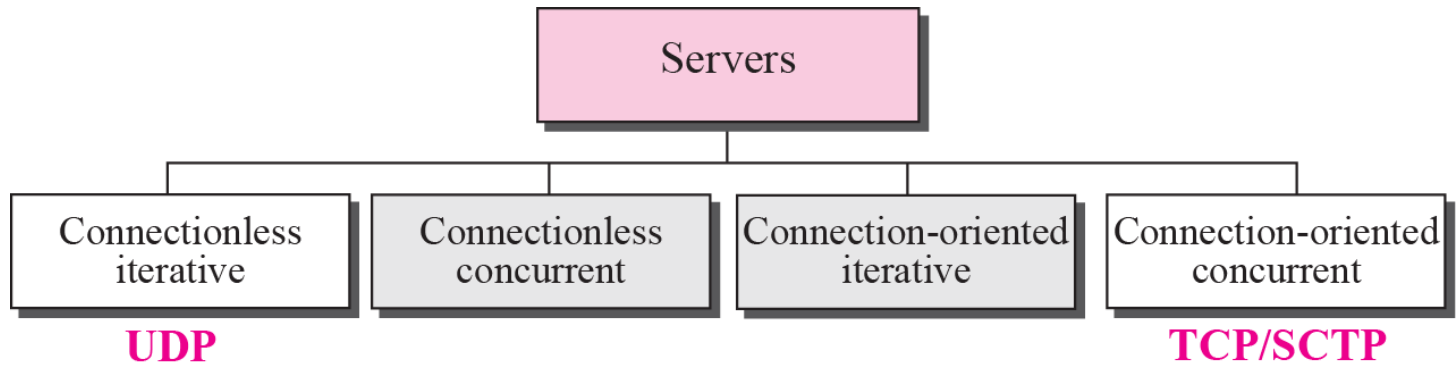
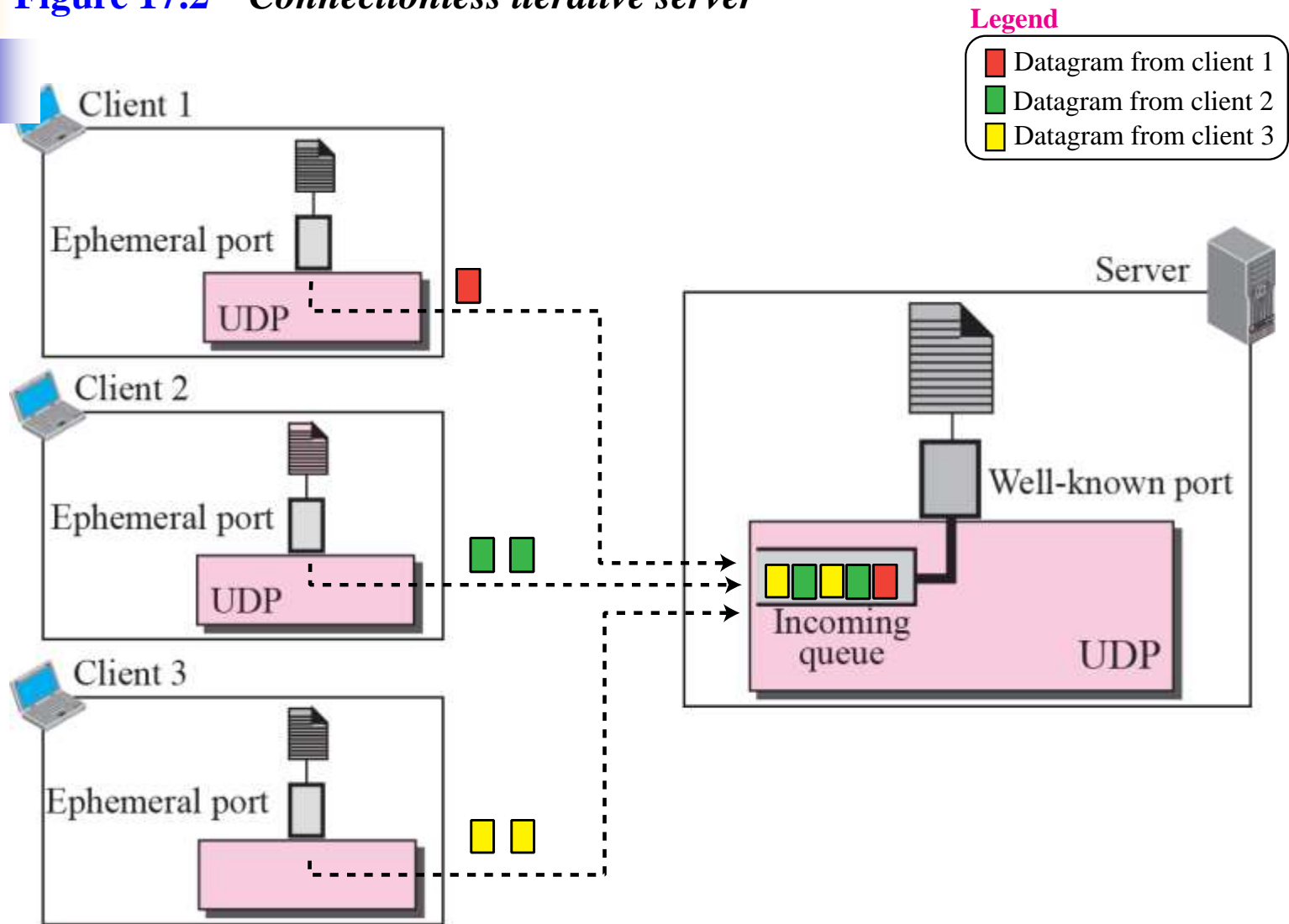


Figure 17.2 *Connectionless iterative server*

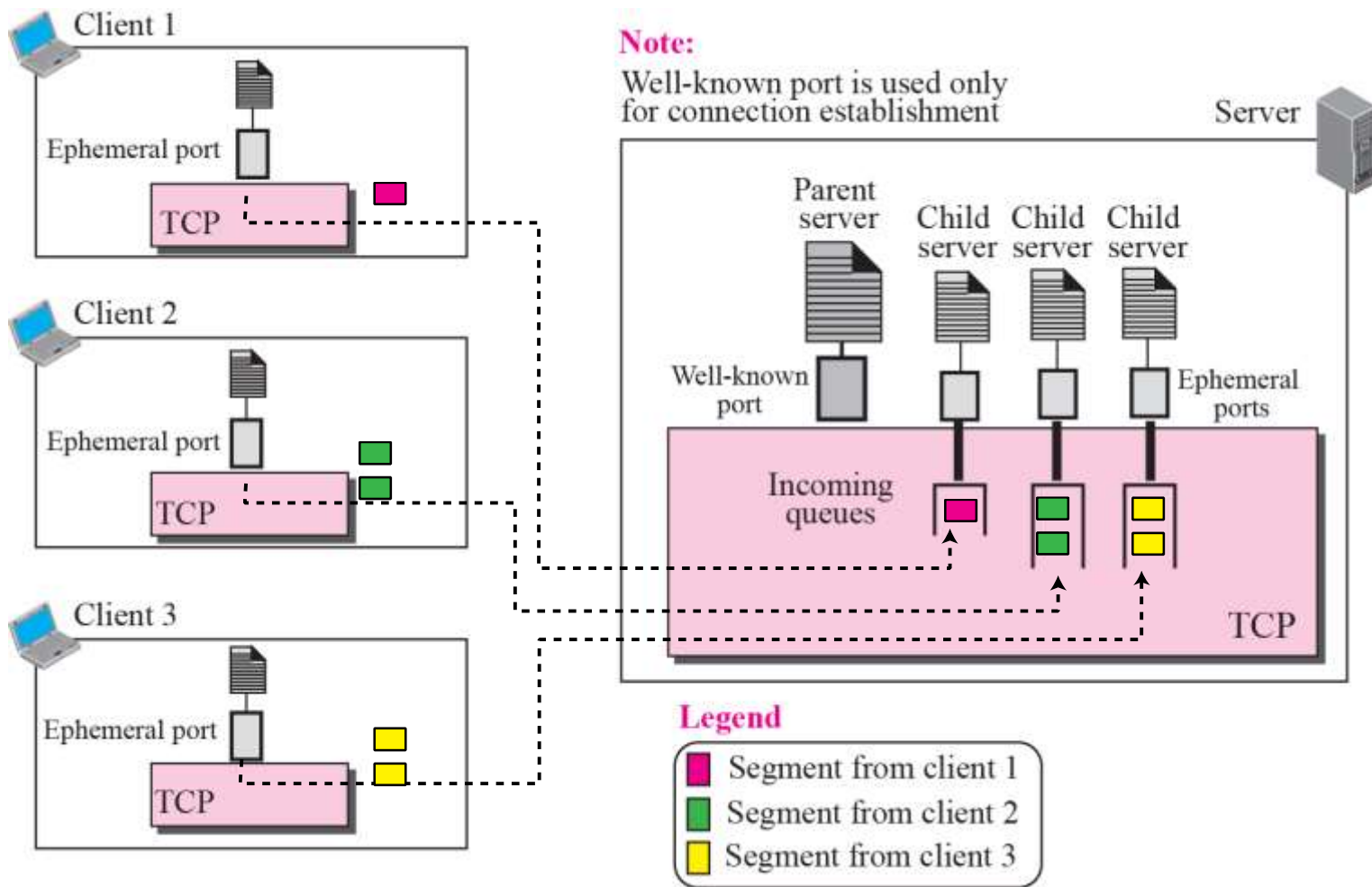


Datagrams are processed one by one in order of arrival.

The server uses one single port for this purpose

- The servers that use TCP (or SCTP) are normally concurrent
- Connection Oriented: request is a stream of bytes that can arrive in several segments and the response can occupy several segments
- connection remains open until the entire stream is processed and the connection is terminated.
- Each connection needs a port and many connections may be open at the same time
- Many ports are needed, but a server can use only one well-known port. The solution is to have one well-known port and many ephemeral ports

Figure 17.3 *Connection-oriented concurrent server*

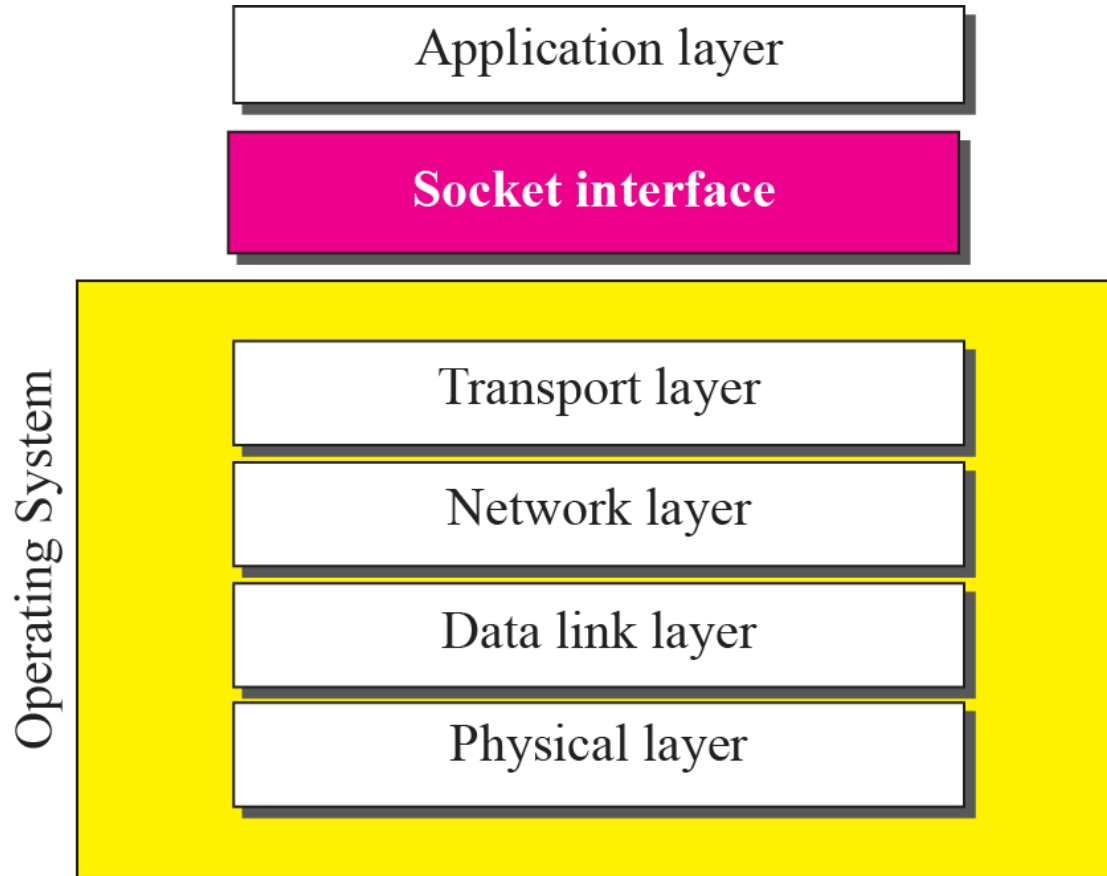


If we need a program to be able to communicate with another program running on another machine, we need a new set of instructions to tell the transport layer to open the connection, send data to and receive data from the other end, and close the connection. A set of instructions of this kind is normally referred to as an **interface**.



***An interface is a set of instructions
designed for interaction between two
entities.***

Figure 17.4 *Relation between the operating system and the TCP/IP suite*



The socket interface, as a set of instructions, located between the operating system and the application programs.

To access the services provided by the TCP/IP protocol suite, an application needs to use the instructions defined in the socket interface.

Example 17.1

Most of the programming languages have a file interface, a set of instructions that allow the programmer to open a file, read from the file, write to the file, perform other operations on the file, and finally close the file. When a program needs to open the file, it uses the name of the file as it is known to the operation system. When the file is opened, the operating system returns a reference to the file (an integer or pointer) that can be used for other instructions, such as read and write.

Figure 17.5 *Concepts of sockets*

socket is a software structure within a network node of a computer network that serves as an endpoint for sending and receiving data across the network

An application program (client or server) needs to request the operating system to create a socket

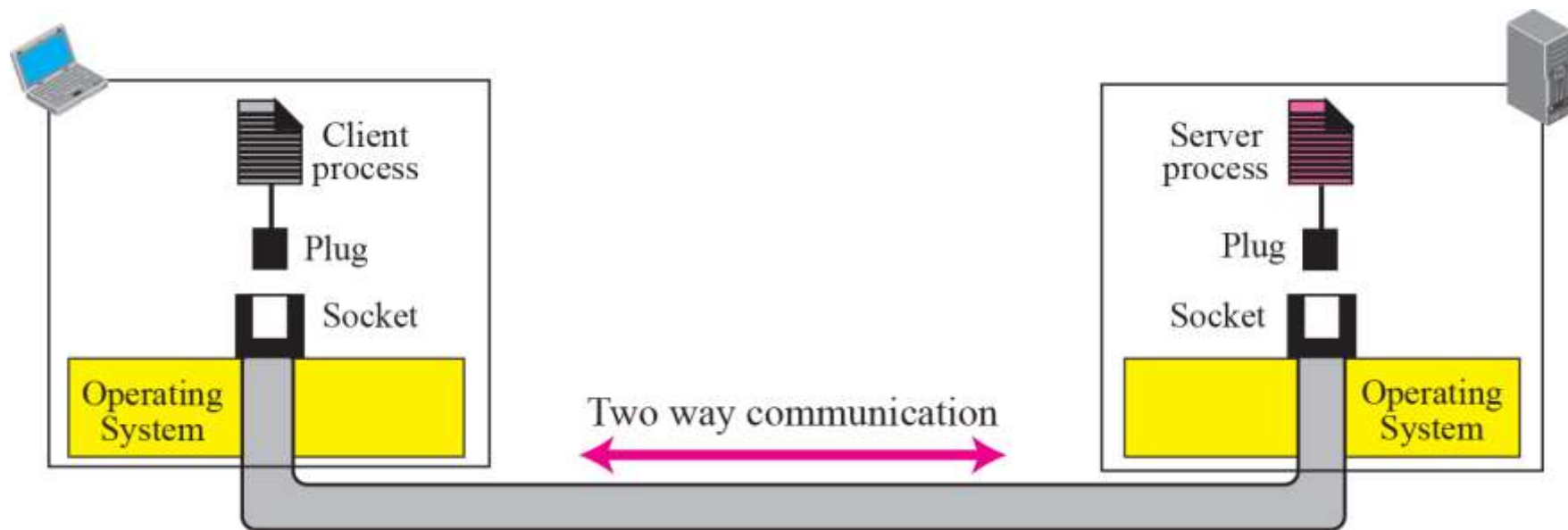
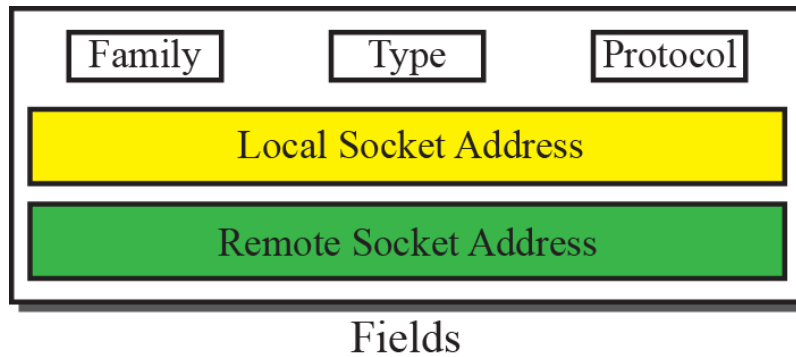


Figure 17.6 *Socket data structure*



```
struct socket
{
    int family;
    int type;
    int protocol;
    socketaddr local;
    socketaddr remote;
};
```

Generic definition

- ❑ **Family.** This field defines the protocol group: IPv4, IPv6, UNIX domain protocols, and so on. The family type we use in TCP/IP is defined by the constant `AF_INET` for IPv4 protocol and `AF_INET6` for IPv6 protocol.
- ❑ **Type.** This field defines four types of sockets: `SOCK_STREAM` (for TCP), `SOCK_DGRAM` (for UDP), `SOCK_SEQPACKET` (for SCTP), and `SOCK_RAW` (for applications that directly use the services of IP. They are shown in Figure 17.7.
- ❑ **Protocol.** This field defines the protocol that uses the interface. It is set to 0 for TCP/IP protocol suite.
- ❑ **Local socket address.** This field defines the local socket address. A socket address, as discussed in Chapter 13, is a combination of an IP address and a port number.
- ❑ **Remote socket address.** This field defines the remote socket address.

Figure 17.7 *Socket types*

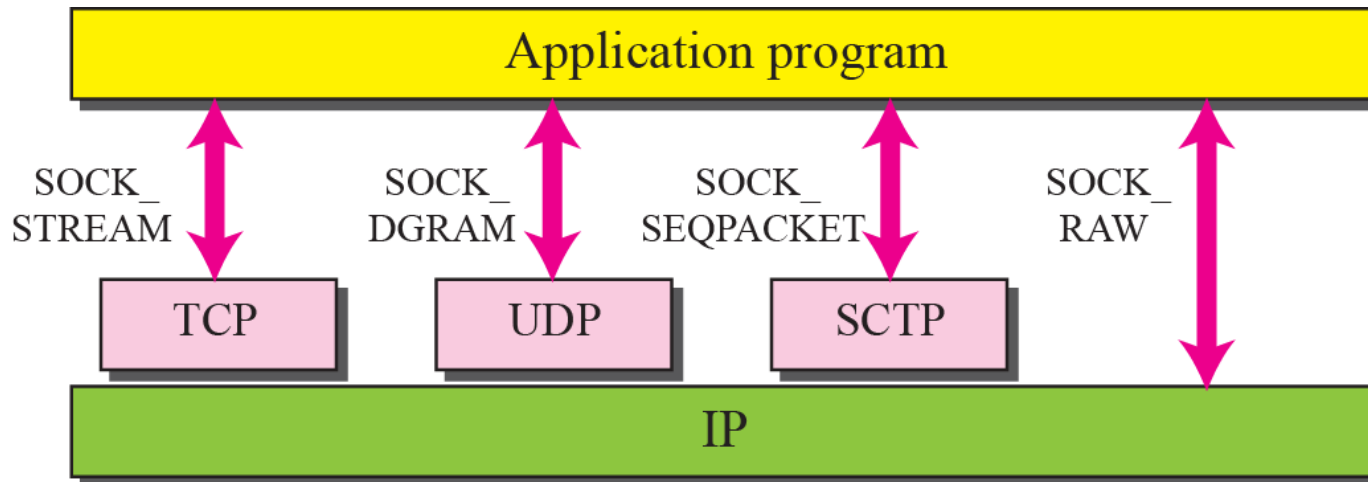
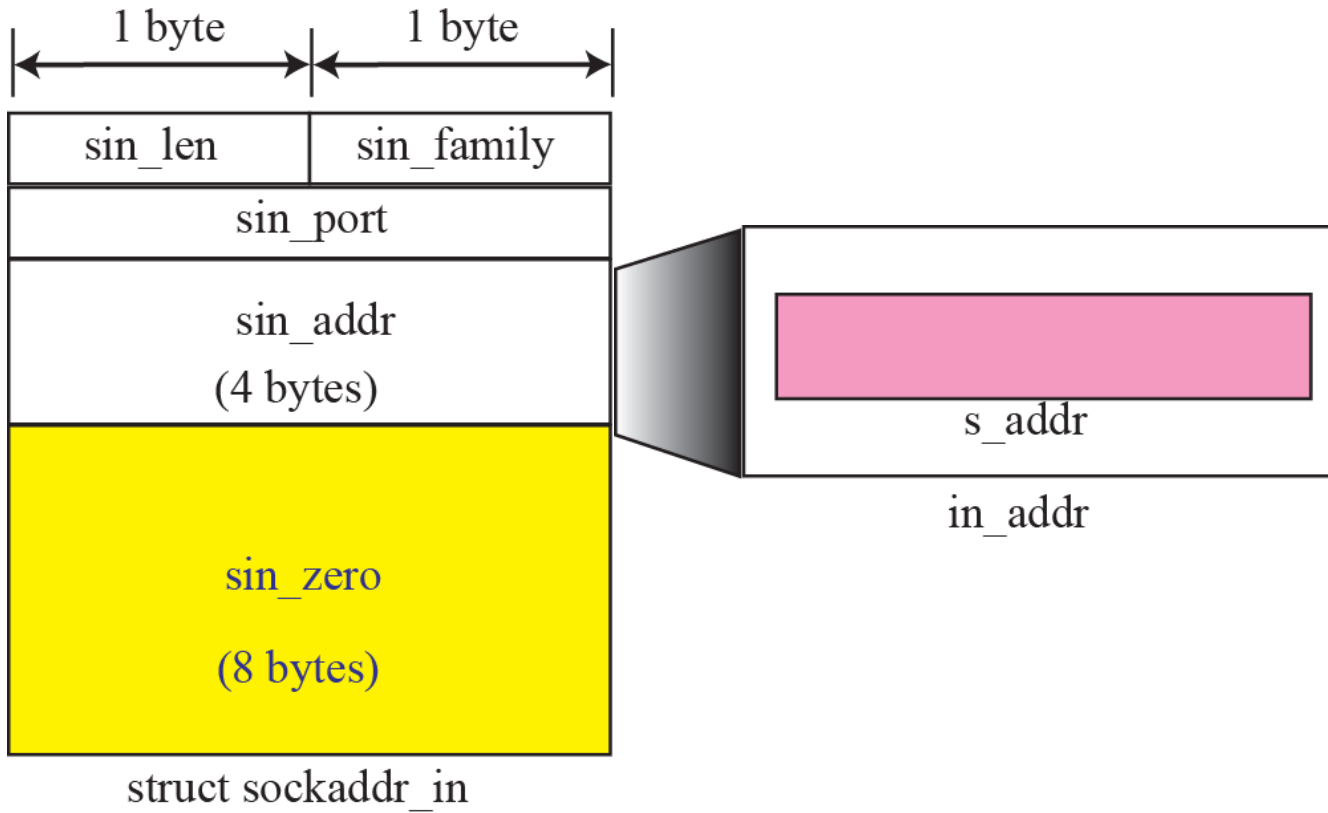


Figure 17.8 *IPv4 socket address*



structure of a socket address

Structure of a socket address, a combination of IP address and port number.

```
struct  sockaddr_in
{
    uint8_t      sin_len;           // length of structure (16 bytes)
    sa_family_t  sin_family;       // set to AF_INET
    in_port_t    sin_port;         // A 16-bit port number
    struct in_addr sin_addr;        // A 32-bit IPv4 address
    char         sin_zero[8];      // unused
}
```

□ The *socket* Function

The operating system defines the socket structure shown in Figure 17.6. The operating system, however, does not create a socket until instructed by the process. The process needs to use the *socket* function call to create a socket. The prototype for this function is shown below:

```
int socket (int family, int type, int protocol);
```

If the call is successful, the function returns a unique socket descriptor *sockfd* (a non-negative integer)

□ The *bind* Function

The socket function fills the fields in the socket partially. To bind the socket to the local computer and local port, the *bind* function needs to be called. The *bind* function, fills the value for the local socket address (local IP address and local port number). It returns -1 if the binding fails. The prototype is shown below:

```
int bind (int sockfd, const struct sockaddr* localAddress, socklen_t addrLen);
```

□ The *connect* Function

The *connect* function is used to add the remote socket address to the socket structure. It returns `-1` if the connection fails. The prototype is given below:

```
int connect (int sockfd, const struct sockaddr* remoteAddress, socklen_t addrLen);
```

The argument is the same, except that the second and third argument defines the remote address instead of the local one.

❑ The *listen* Function

The *listen* function is called only by the TCP server. After TCP has created and bound a socket, it must inform the operating system that a socket is ready for receiving client requests. This is done by calling the *listen* function. The backlog is the maximum number of connection requests. The function returns -1 if it fails. The following shows the prototype:

```
int listen (int sockfd, int backlog);
```

❑ The *accept* Function

The *accept* function is used by a server to inform TCP that it is ready to receive connections from clients. This function returns -1 if it fails. Its prototype is shown below:

```
int accept (int sockfd, const struct sockaddr* clientAddr, socklen_t* addrLen);
```

Accept function:

- a.** The call to `accept()` function makes the process check if there is any client connection request in the waiting buffer. If not, the `accept` makes the process to sleep. The process wakes up when the queue has at least one request.
- b.** After a successful call to the `accept`, a new socket is created and the communication is established between the client socket and the new socket of the server.
- c.** The address received from the *accept* function fills the remote socket address in the new socket.

.

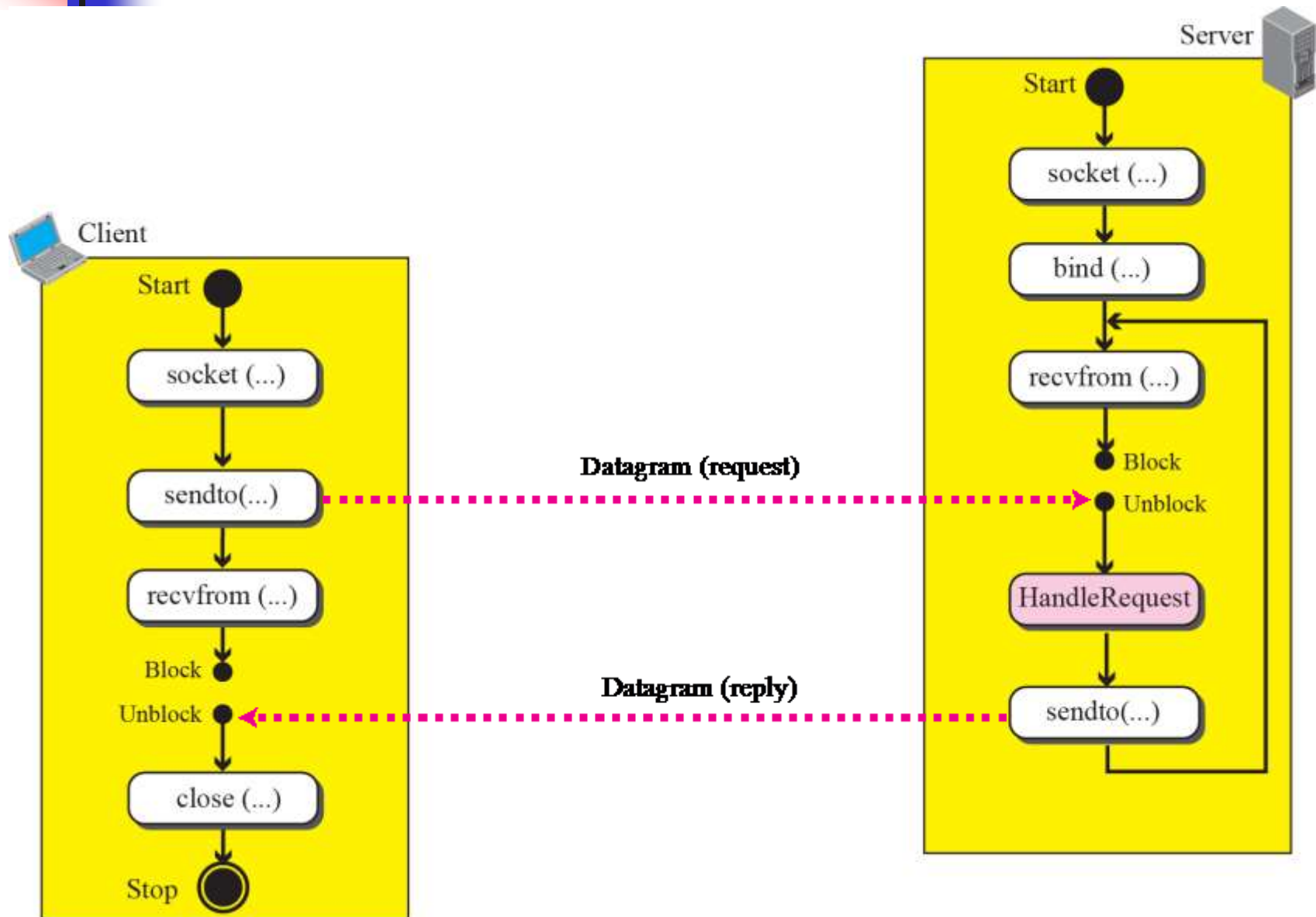
□ The *send* and *recv* Functions

The *send* function is used by a process to send data to another process running on a remote machine. The *recv* function is used by a process to receive data from another process running on a remote machine. These functions assume that there is already an open connection between two machines; therefore, it can only be used by TCP (or SCTP). These functions returns the number of bytes send or receive.

```
int send (int sockfd, const void* sendbuf, int nbytes, int flags);
```

```
int recv (int sockfd, void* recvbuf, int nbytes, int flags);
```

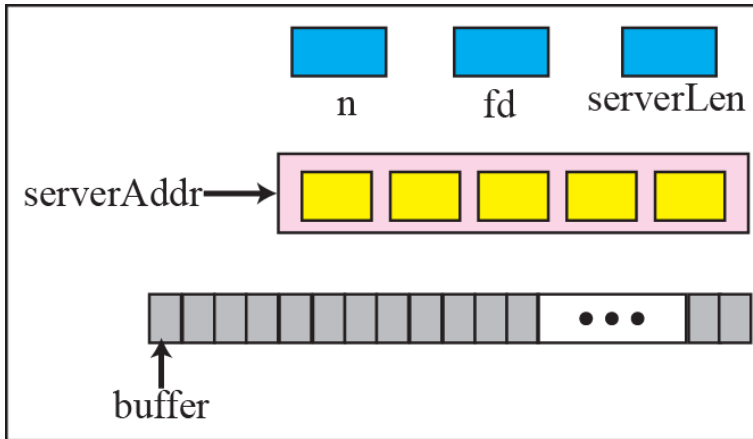

Figure 17.9 *Connectionless iterative communication using UDP*



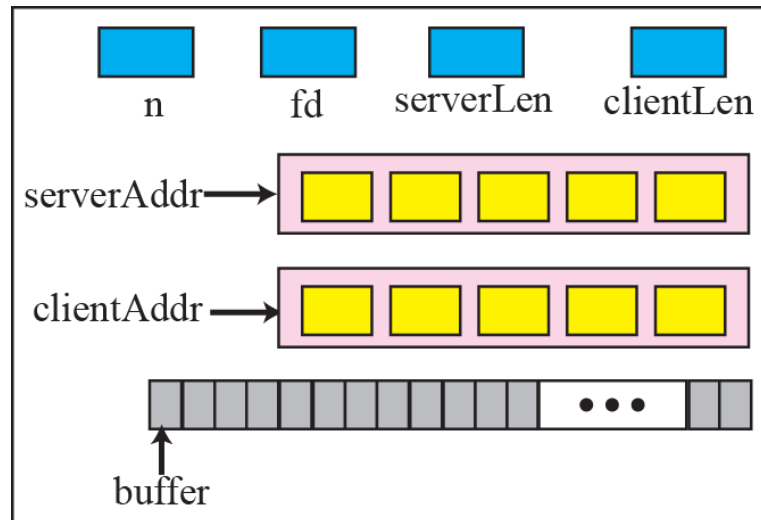
Example 17.2

As an example, let us see how we can design and write two programs: an echo server and an echo server. The client sends a line of text to the server; the server sends the same line back to the client. Although this client/server pair looks useless, it has some applications. It can be used, for example, when a computer wants to test if another computer in the network is alive. To better understand the code in a program, we first give the layout of variables used in both programs as shown in Figure 17.10.

Figure 17.10 *Variables used in echo server and echo client using UDP service*



Variables used by the client process



Variables used by the server process

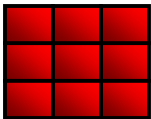


Table 17.1 *Echo Server Program using the Service of UDP*

```
01 // UDP echo server program
02 #include "headerFiles.h"
03
04 int main (void)
05 {
06     // Declaration and definition
07     int sd; // Socket descriptor
08     int nr; // Number of bytes received
09     char buffer [256]; // Data buffer
10     struct sockaddr_in serverAddr; // Server address
11     struct sockaddr_in clientAddr; // Client address
12     int clAddrLen; // Length of client Address
13     // Create socket
14     sd = socket (PF_INET, SOCK_DGRAM, 0);
15     // Bind socket to local address and port
16     memset (&serverAddr, 0, sizeof (serverAddr));
17     serverAddr.sin_family = AF_INET;
```

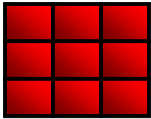


Table 17.1 *Echo Server Program using the Service of UDP (continued)*

```
18 serverAddr.sin_addr.s_addr = htonl (INADDR_ANY);      // Default address
19 serverAddr.sin_port = htons (7)    // We assume port 7
20 bind (sd, (struct sockaddr*) &serverAddr, sizeof (serverAddr));
21 // Receiving and echoing datagrams
22 for ( ; ; )    // Run forever
23 {
24     nr = recvfrom (sd, buffer, 256, 0, (struct sockaddr*)&clientAddr, &clAddrLen);
25     sendto (sd, buffer, nr, 0, (struct sockaddr*)&clientAddr, sizeof(clientAddr));
26 }
27 } // End of echo server program
```

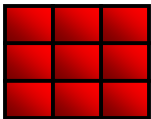


Table 17.2 *Echo Client Program using the Service of UDP*

```
01 // UDP echo client program
02 #include "headerFiles.h"
04
04 int main (void)
05 {
06     // Declaration and definition
07     int sd; // Socket descriptor
08     int ns; // Number of bytes send
09     int nr; // Number of bytes received
10     char buffer [256]; // Data buffer
11     struct sockaddr_in serverAddr; // Socket address
11     // Create socket
12     sd = socket (PF_INET, SOCK_DGRAM, 0);
13     // Create server socket address
14     memset (&servAddr, 0, sizeof(serverAddr));
15     servAddr.sin_family = AF_INET;
16     inet_pton (AF_INET, "server address", &serverAddr.sin_addr);
17     serverAddr.sin_port = htons (7);
18     // Send and receive datagrams
19     fgets (buffer, 256, stdin);
```

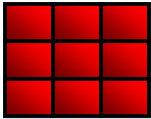


Table 17.2 *Echo Client Program using the Service of UDP (continued)*

```
20     ns = sendto (sd, buffer, strlen (buffer), 0,  
21                 (struct sockaddr)&serverAddr, sizeof(serveraddr));  
22     recvfrom (sd, buffer, strlen (buffer), 0, NULL, NULL);  
23     buffer [nr] = 0;  
24     printf ("Received from server:");  
25     fputs (buffer, stdout);  
26     // Close and exit  
27     close (sd);  
28     exit (0);  
29 } // End of echo client program
```

Listen socket: This socket is only used during connection establishment.

Bind function : bind this connection to the socket address of the server computer

The server program then calls the **accept** function. This function is a blocking function; when it is called, it is blocked until the TCP receives a connection request (SYN segment) from a client.

The *accept* function: then is unblocked and creates a new socket called the connect socket that includes the socket address of the client that sent the SYN segment

To provide **concurrency**, the server process (parent process) calls the *fork* function. This function creates a new process (child process), which is exactly the same as the parent process

Figure 17.11 *Flow diagram for connection-oriented, concurrent communication*

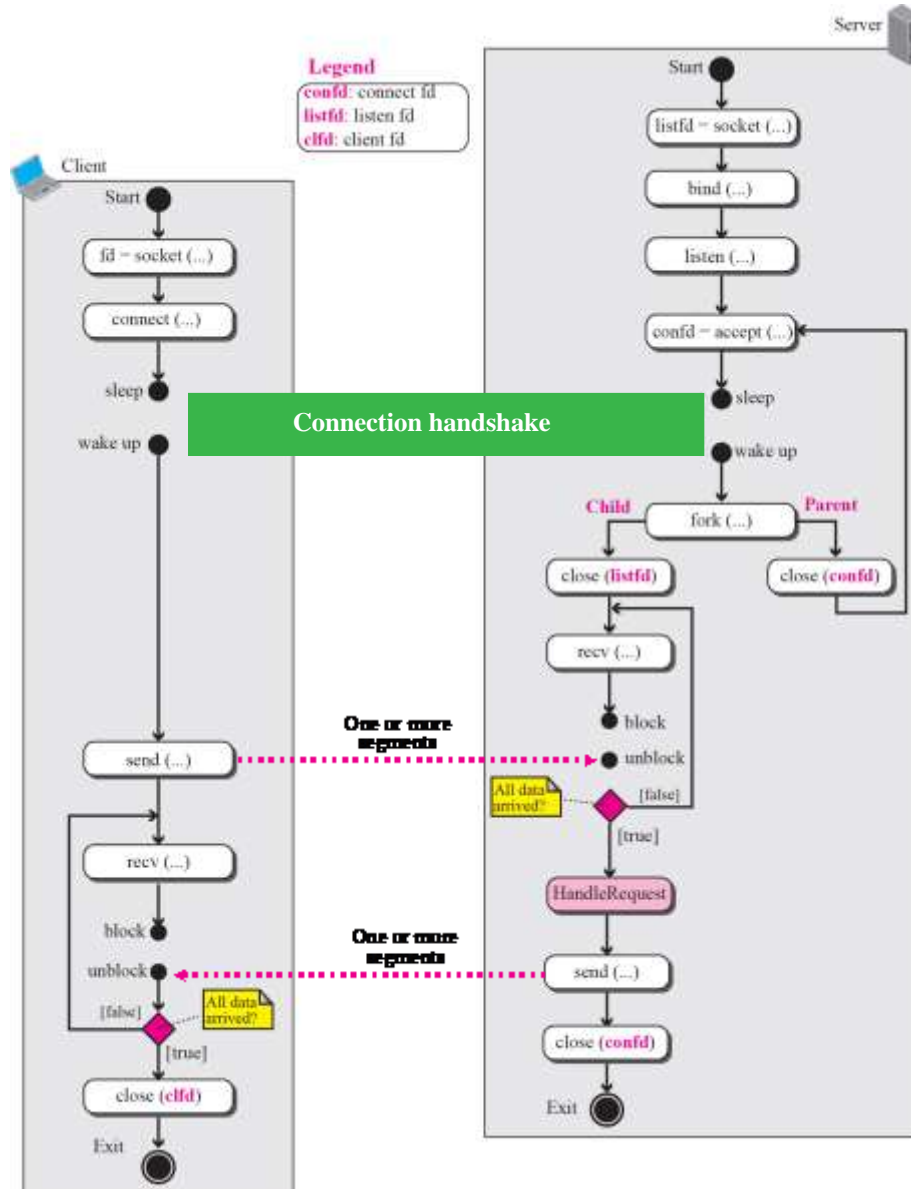
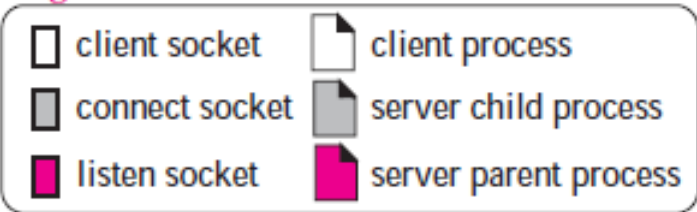


Figure 17.12 *Status of parent and child processes with respect to the socket*

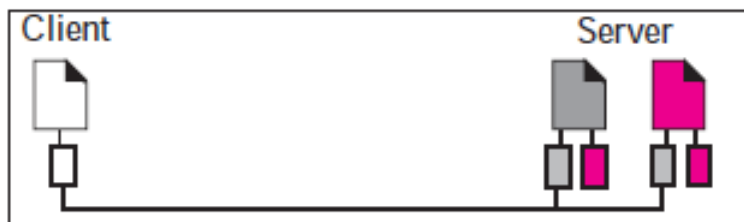
Legend



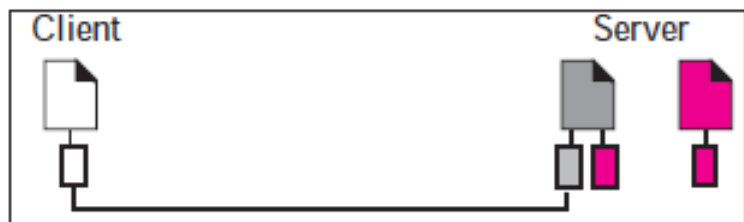
a. Before return from *accept*



b. After return from *accept*



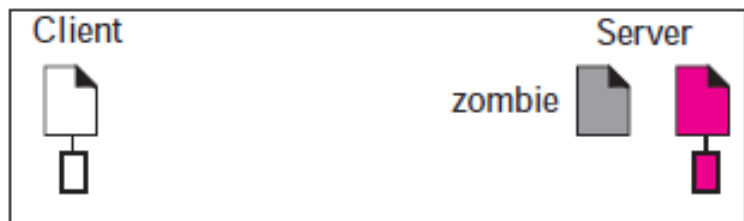
c. After *fork*



d. After parent closes connect socket



e. After child closes listen socket

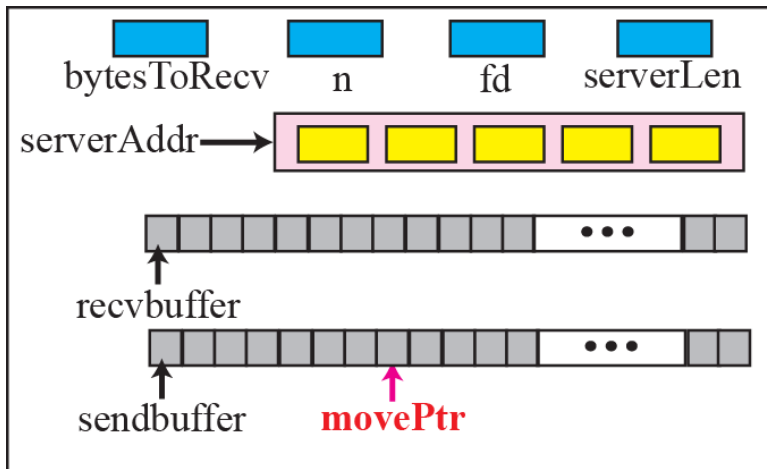


f. After child closes connect socket

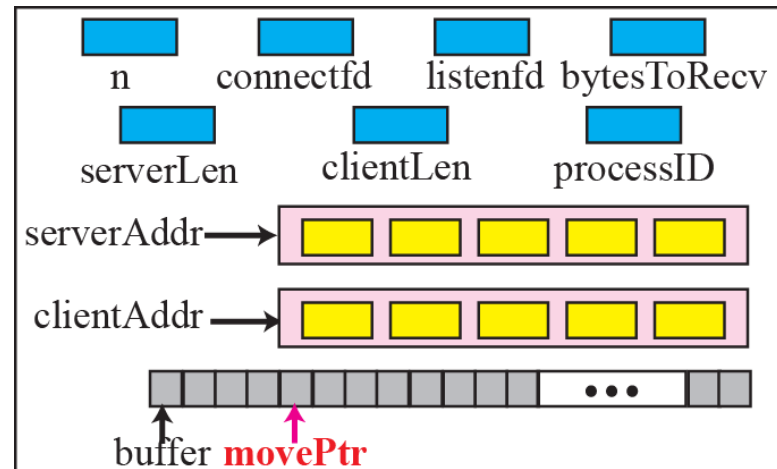
Example 17.3

We want to write two programs to show how we can have an echo client and echo server using the services of TCP. Figure 17.13 shows the variables we use in these two programs. Since data may arrive in different chunks, we need pointers to point to the buffer. The first buffer is fixed and always points to the beginning of the buffer; the second pointer is moving to let the arrived bytes be appended to the end of the previous section.

Figure 17.13 *Variable used in echo client and echo sever using TCP*



Variables used by the client process



Variables used by the server process

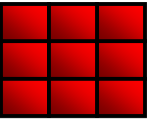


Table 17.3 *Echo Server Program using the Services of TCP*

```
01 // Echo server program
02 #include "headerFiles.h"
03
04 int main (void)
05 {
06     // Declaration and definition
07     int listensd;                // Listen socket descriptor
08     int connectsd;             // Connecting socket descriptor
09     int n;                      // Number of bytes in each reception
10     int bytesToRecv;           // Total bytes to receive
11     int processID;             // ID of the child process
12     char buffer [256];         // Data buffer
13     char* movePtr;             // Pointer to the buffer
14     struct sockaddr_in serverAddr; // Server address
15     struct sockaddr_in clientAddr; // Client address
16     int clAddrLen;             // Length of client address
17     // Create listen socket
18     listensd = socket (PF_INET, SOCK_STREAM, 0);
19     // Bind listen socket to the local address and port
20     memset (&serverAddr, 0, sizeof (serverAddr));
21     serverAddr.sin_family = AF_INET;
```

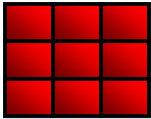


Table 17.3 *Echo Server Program using the Services of TCP (continued)*

```
22 serverAddr.sin_addr.s_addr = htonl (INADDR_ANY);
23 serverAddr.sin_port = htons (7); // We assume port 7
24 bind (listensd, &serverAddr, sizeof (serverAddr));
25 // Listen to connection requests
26 listen (listensd, 5);
27 // Handle the connection
28 for ( ; ; ) // Run forever
29 {
30     connectfd = accept (listensd, &clientAddr, &clAddrLen);
31     processID = fork ();
32     if (processID == 0) // Child process
33     {
34         close (listensd);
35         bytesToRecv = 256;
36         movePtr = buffer;
37         while ( (n = recv (connectfd, movePtr, bytesToRecv, 0)) > 0)
38         {
39             movePtr = movePtr + n;
40             bytesToRecv = movePtr - n;
41         } // End of while
```

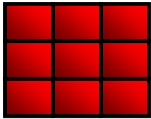


Table 17.3 *Echo Server Program using the Services of TCP (continued)*

```
42         send (connectsd, buffer, 256, 0);
43         exit (0);
44     } // End of if
45     close (connectsd);           // Back to parent process
46 } // End of for loop
47 } // End of echo server program
```

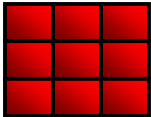



Table 17.4 *Echo Client Program using the services of TCP*

```
01 // TCP echo client program
02 #include "headerFiles.h"
03
04 int main (void)
05 {
06     // Declaration and definition
07     int sd; // Socket descriptor
08     int n; // Number of bytes received
09     int bytesToRecv; // Number of bytes to receive
10     char sendBuffer [256]; // Send buffer
11     char recvBuffer [256]; // Receive buffer
12     char* movePtr; // A pointer the received buffer
13     struct sockaddr_in serverAddr; // Server address
14
15     // Create socket
16     sd = socket (PF_INET, SOCK_STREAM, 0);
17     // Create server socket address
18     memset (&serverAddr, 0, sizeof(serverAddr));
19     serverAddr.sin_family = AF_INET;
20     inet_pton (AF_INET, "server address", &serverAddr.sin_addr);
21     serverAddr.sin_port = htons (7); // We assume port 7
22     // Connect
```

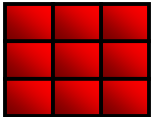


Table 17.4 *Echo Client Program using the services of TCP (continued)*

```
23 connect (sd, (struct sockaddr*)&serverAddr, sizeof(serverAddr));
24 // Send and receive data
25 fgets (sendBuffer, 256, stdin);
26 send (fd, sendBuffer, strlen (sendbuffer), 0);
27 bytesToRecv = strlen (sendbuffer);
28 movePtr = recvBuffer;
29 while ( (n = recv (sd, movePtr, bytesToRecv, 0) ) > 0)
30 {
31     movePtr = movePtr + n;
32     bytesToRecv = bytesToRecv - n;
33 } // End of while loop
34 recvBuffer[bytesToRecv] = 0;
35 printf ("Received from server:");
36 fputs (recvBuffer, stdout);
37 // Close and exit
38 close (sd);
39 exit (0);
40 } // End of echo client program
```



Note

In Appendix F we give some simple Java versions of programs in Table 17.1 to 17.4

17-2 PEER-TO-PEER PARADIGM

Although most of the applications available in the Internet today use the client-server paradigm, the idea of using peer-to-peer (P2P) paradigm recently has attracted some attention. In this paradigm, two peer computers can communicate with each other to exchange services. This paradigm is interesting in some areas such file as transfer in which the client-server paradigm may put a lot of the load on the server machine. However, we need to mention that the P2P paradigm does not ignore the client-server paradigm; it is based on this paradigm.