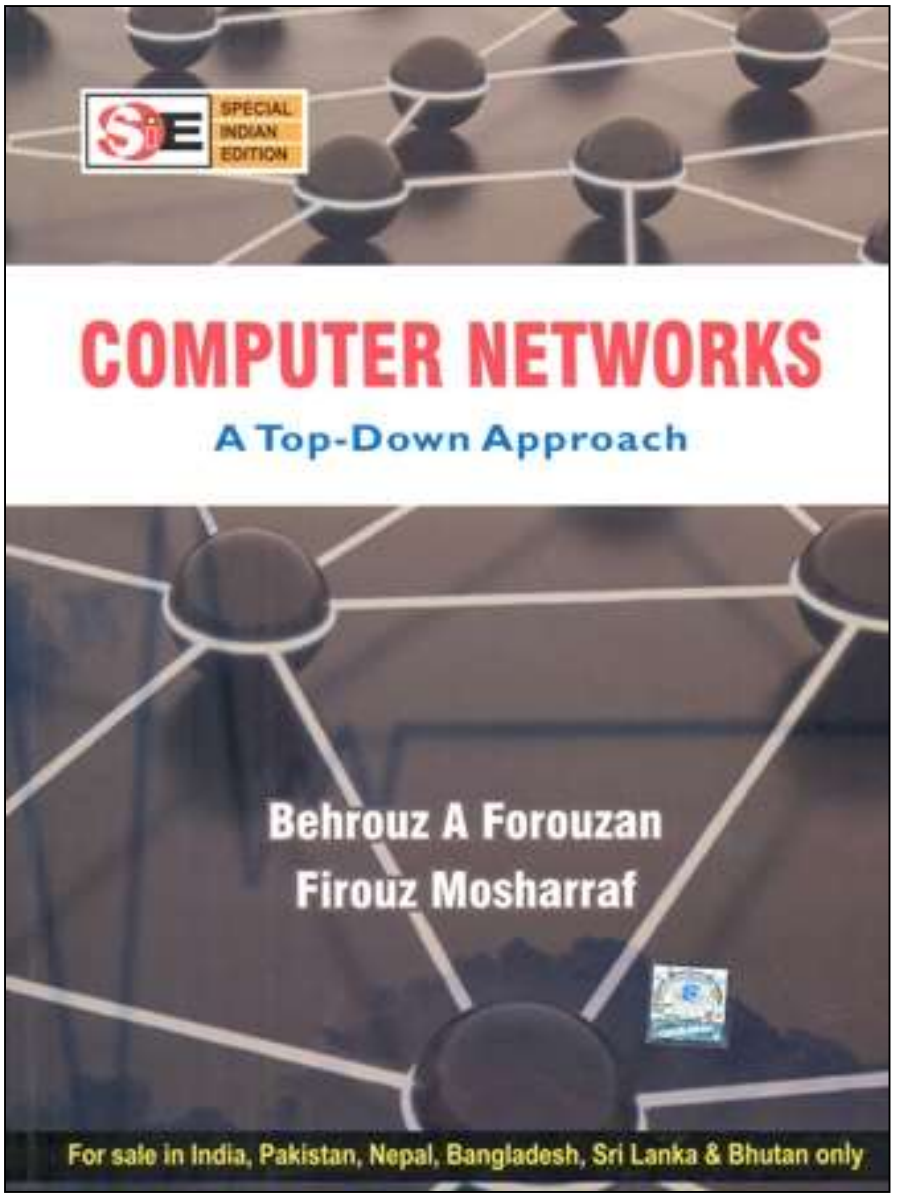
Chapter 3

Transport Layer



Chapter 3: Outline

3.1 INTRODUCTION

3.2 TRANSPORT-LAYER PROTOCOLS

3.3 USER DATAGRAM PROTOCOL

3.4 TRANSMISSION CONTROL PROTOCOL

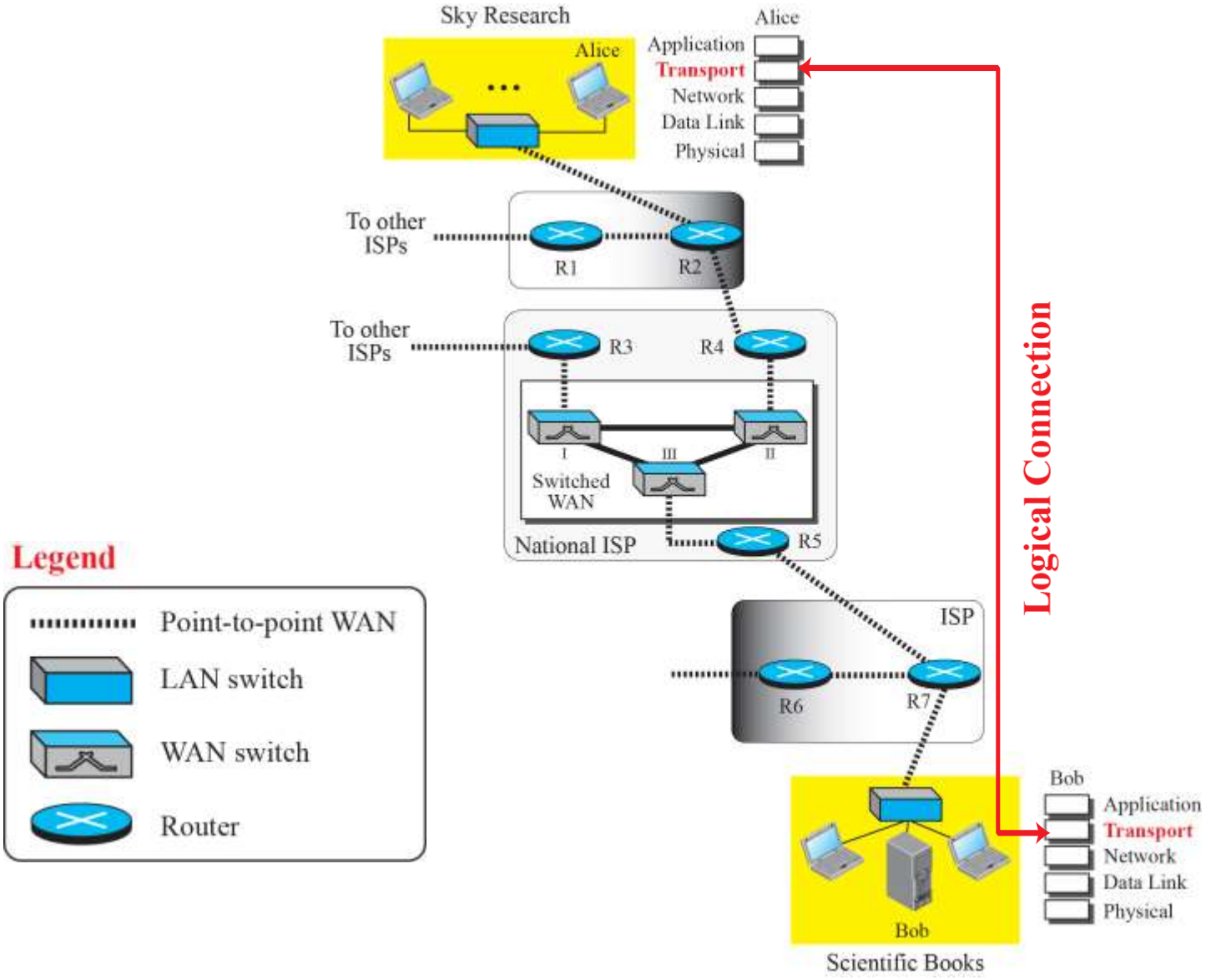
Chapter 3: Objective

- *We introduce general services we normally require from the transport layer, such as process-to-process communication, addressing, multiplexing, error, flow, and congestion control.*
- *We discuss general transport-layer protocols such as Stop-and-Wait, Go-Back-N, and Selective-Repeat.*
- *We discuss UDP, which is the simpler of the two protocols we discuss in this chapter.*
- *We discuss TCP services and features. We then show how TCP provides a connection-oriented service using a transition diagram. Finally, we discuss flow and error control, and congestion control in TCP.*

3-1 INTRODUCTION

The transport layer provides a process-to-process communication between two application layers. Communication is provided using a logical connection, which means that the two application layers assume that there is an imaginary direct connection through which they can send and receive messages.

Figure 3.1: Logical connection at the transport layer



3.1.1 Transport-Layer Services

As we discussed in Chapter 1, 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 the transport layer; in the next section, we discuss several transport-layer protocols.

3.1.1 (continued)

- *Process-to-Process Communication*
- *Addressing: Port Numbers*
- *ICANN Ranges*
  - *Well-known ports*
  - *Registered ports*
  - *Dynamic ports*
- *Encapsulation and Decapsulation*
- *Multiplexing and Demultiplexing*

3.1.1 (continued)

- **Flow Control**
  - Pushing or Pulling
  - Flow Control at Transport Layer
  - Buffers
- **Error Control**
  - Sequence Numbers
  - Acknowledgment
- **Combination of Flow and Error Control**
  - Sliding Window

3.1.1 (continued)

- **Congestion Control**
- **Connectionless and Connection-Oriented**
  - Connectionless Service
  - Connection-Oriented Service
  - Finite State Machine
- **Multiplexing and Demultiplexing**

Figure 3.2: Network layer versus transport layer

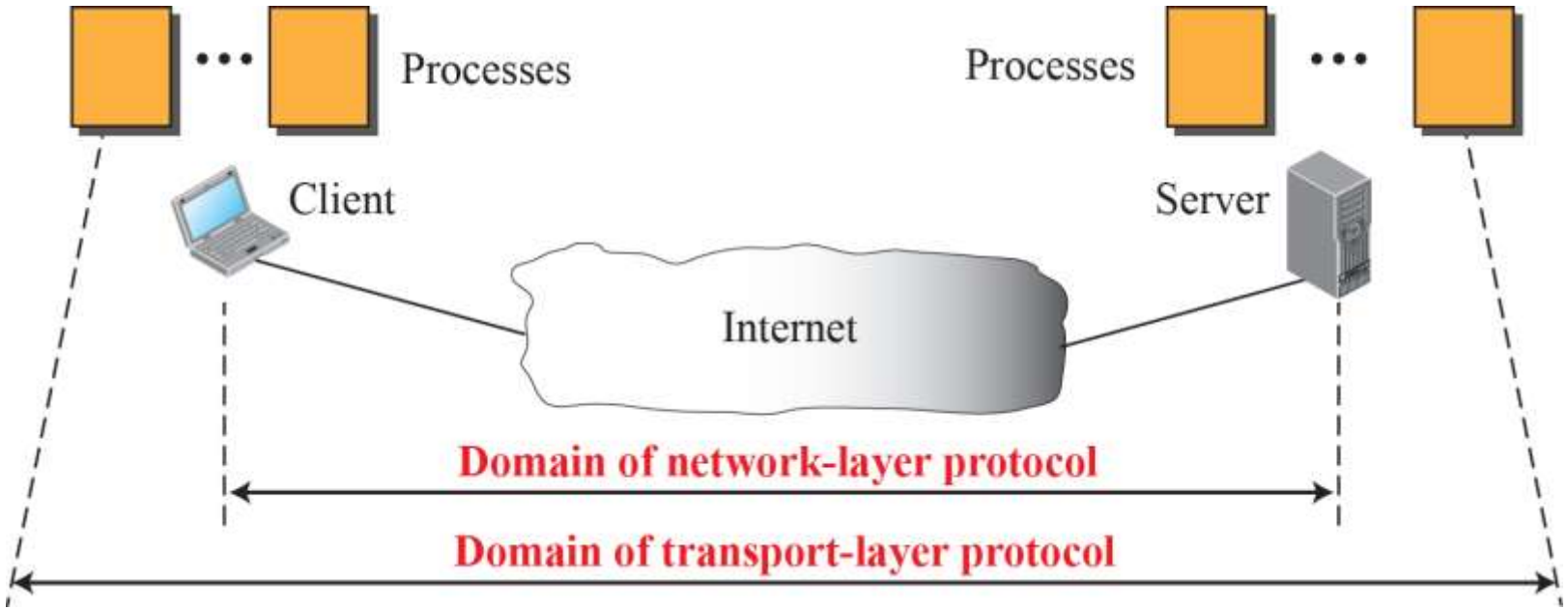


Figure 3.3: Port numbers

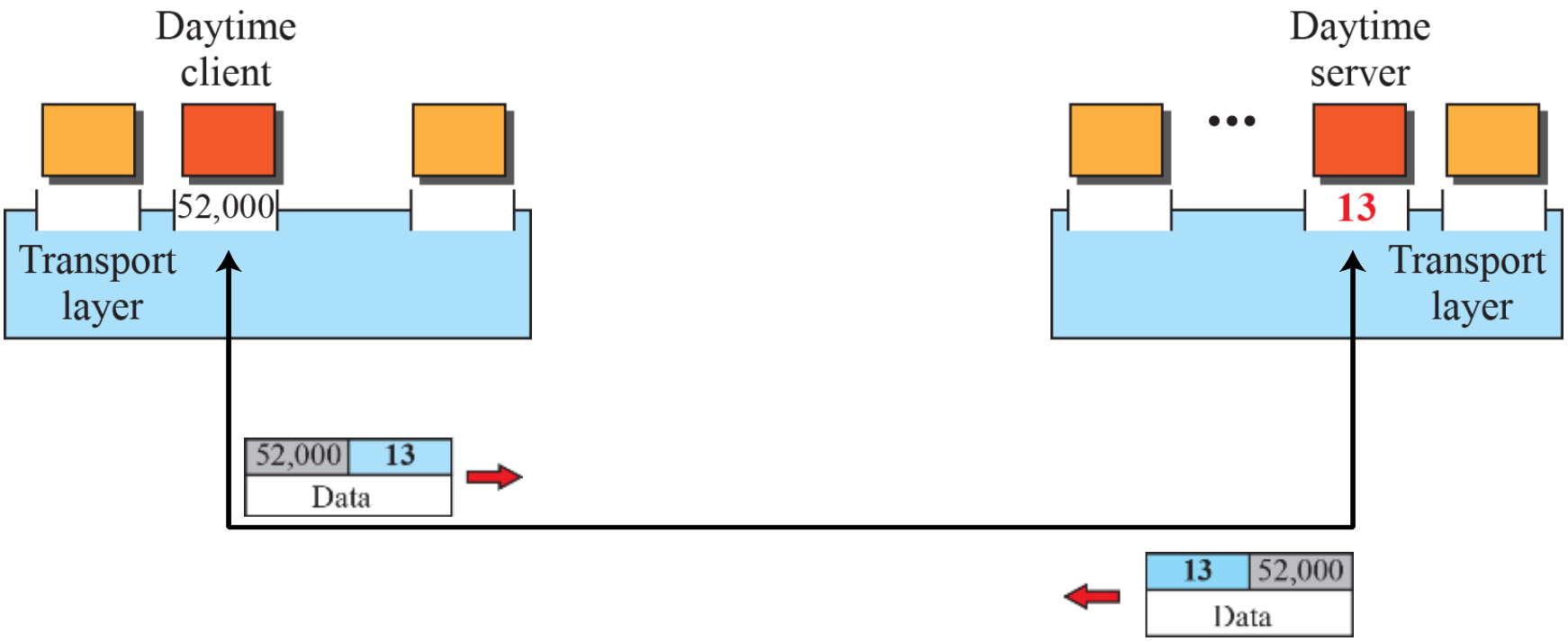


Figure 3.4: IP addresses versus port numbers

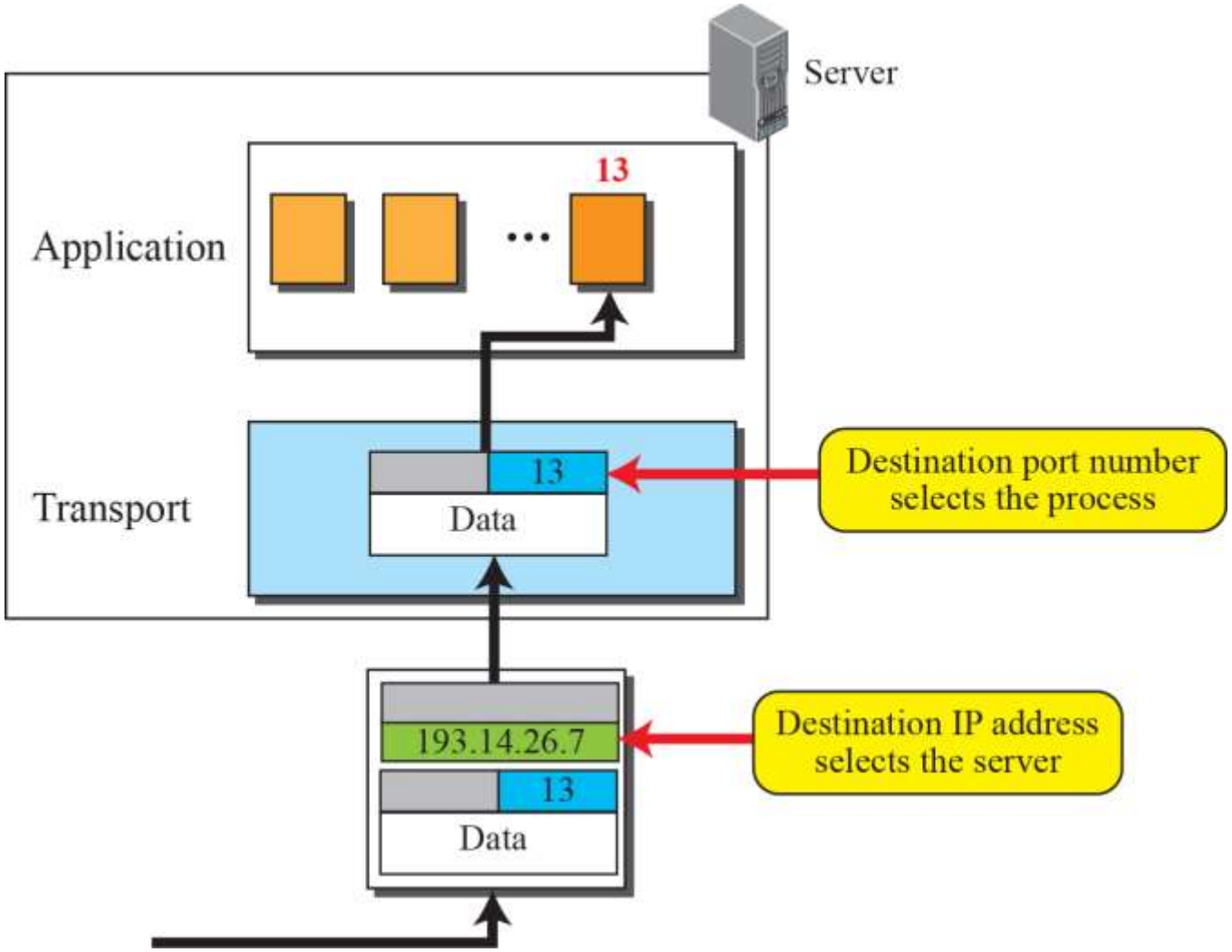
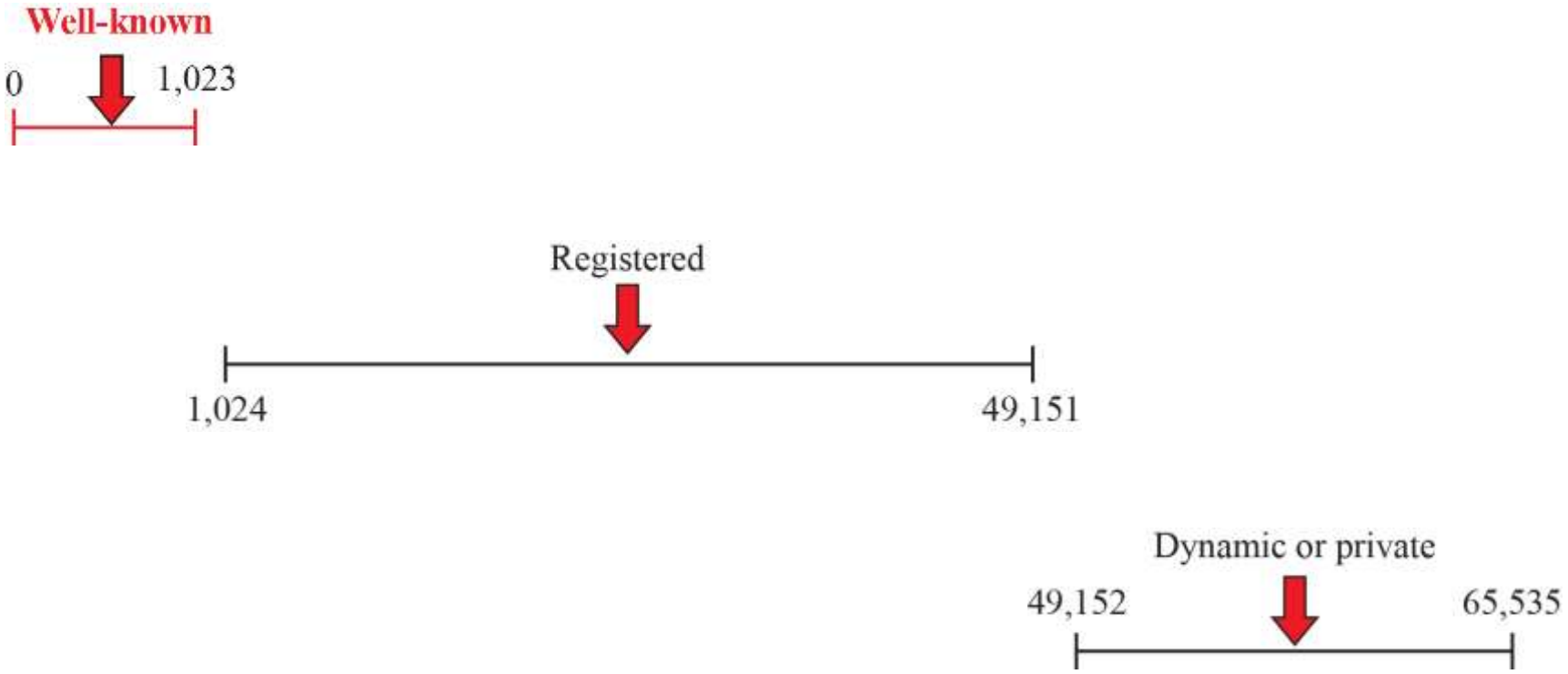


Figure 3.5: ICANN ranges



Example 3.1

In UNIX, the well-known ports are stored in a file called /etc/services. We can use the *grep* utility to extract the line corresponding to the desired application.

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

SNMP (see Chapter 9) uses two port numbers (161 and 162), each for a different purpose.

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

Figure 3.6: *Socket address*

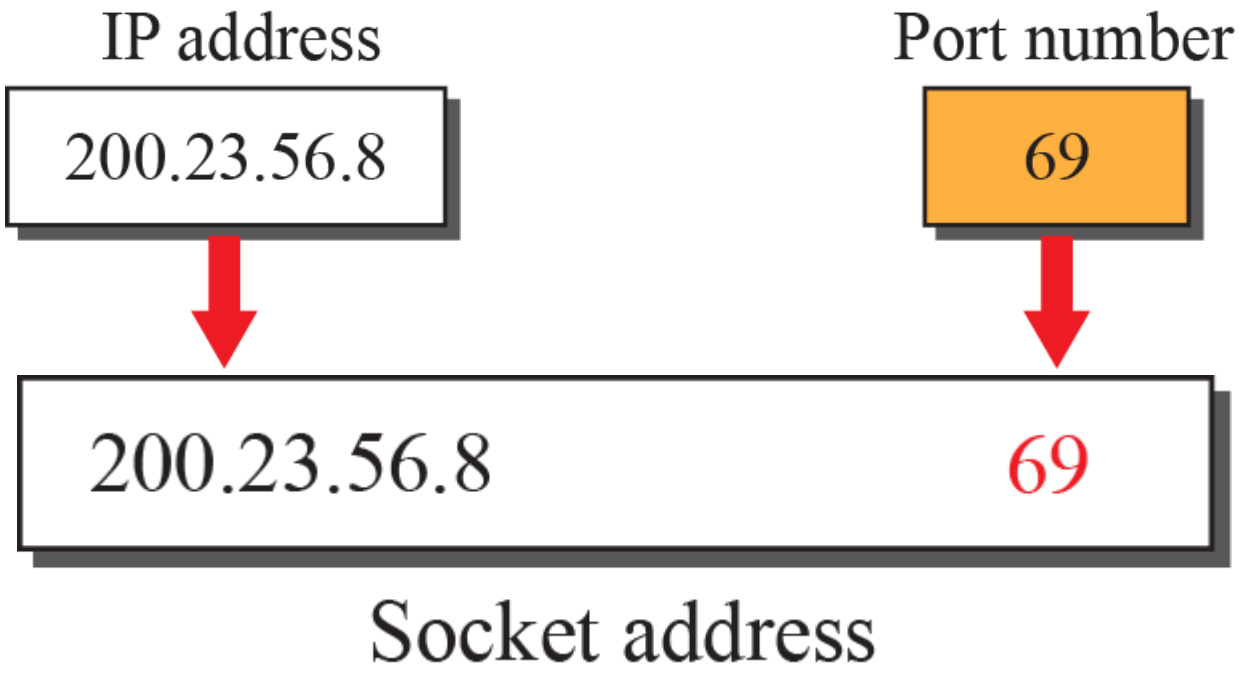


Figure 3.7: Encapsulation and decapsulation

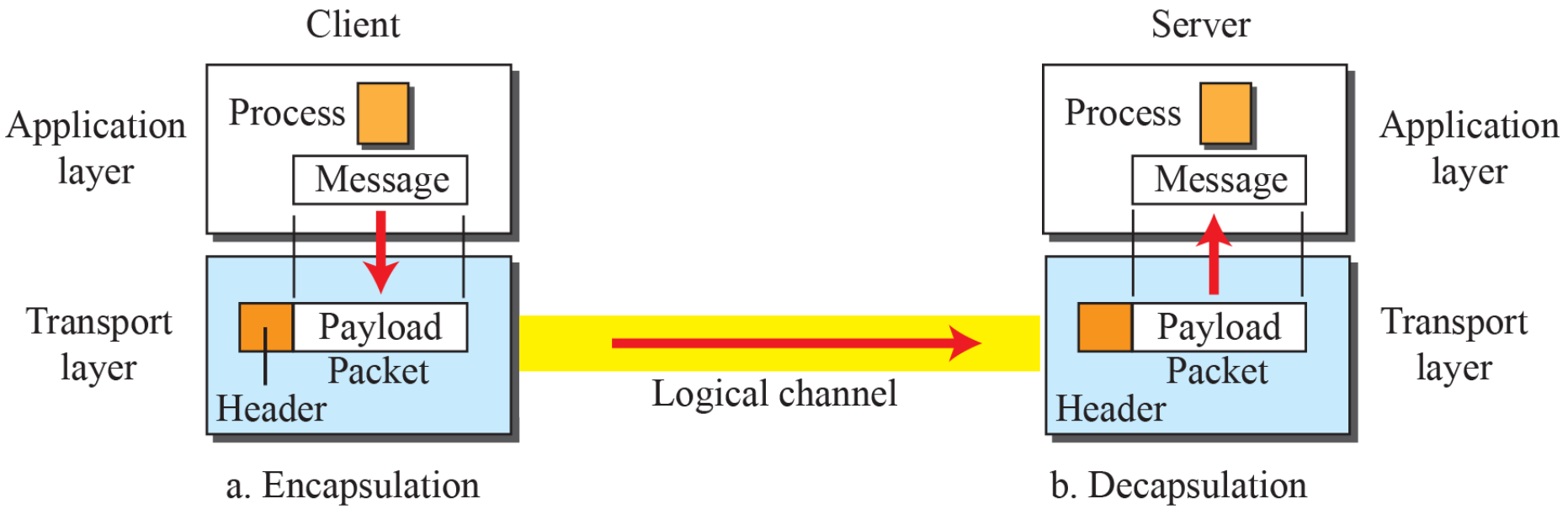


Figure 3.8: Multiplexing and demultiplexing

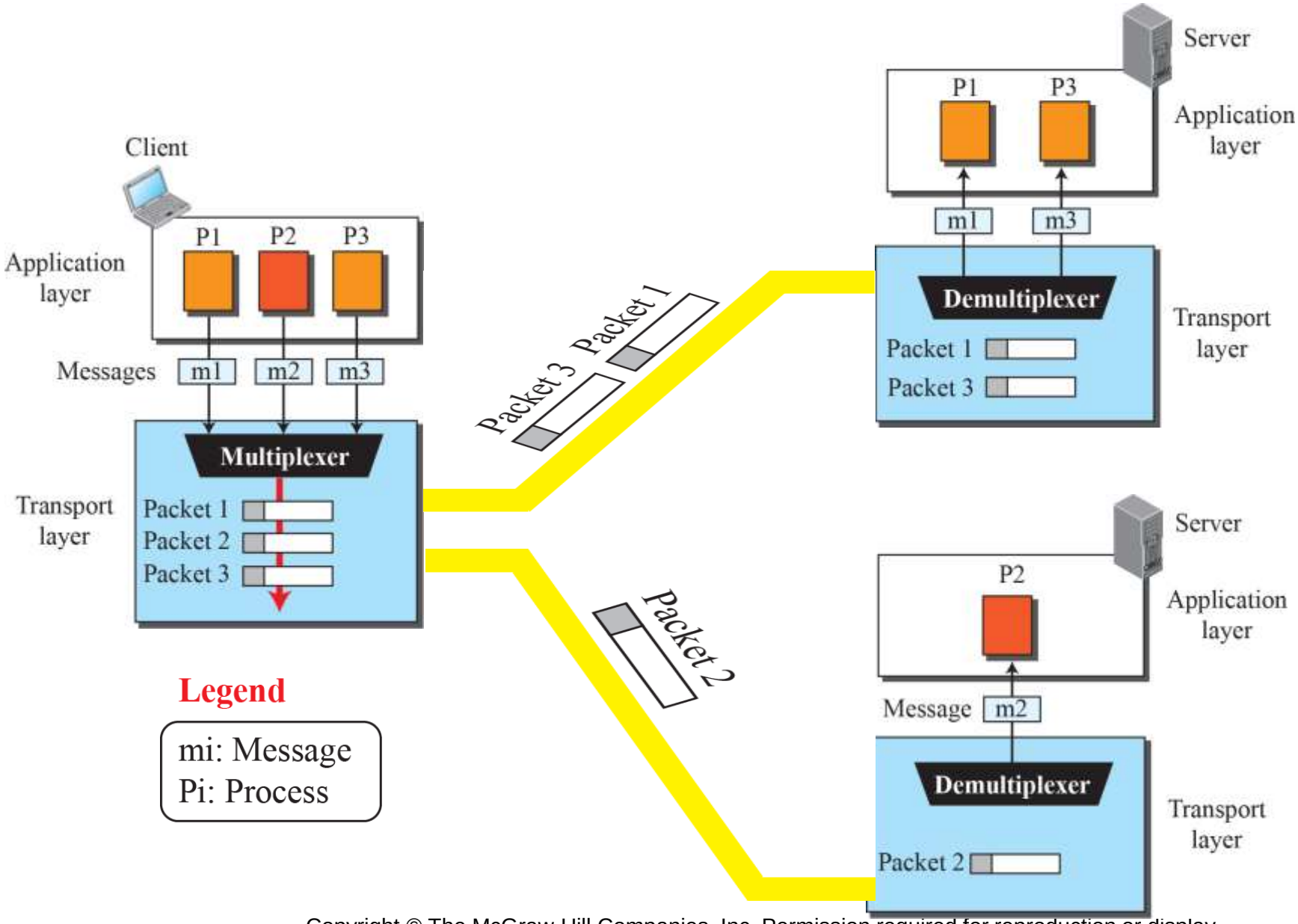
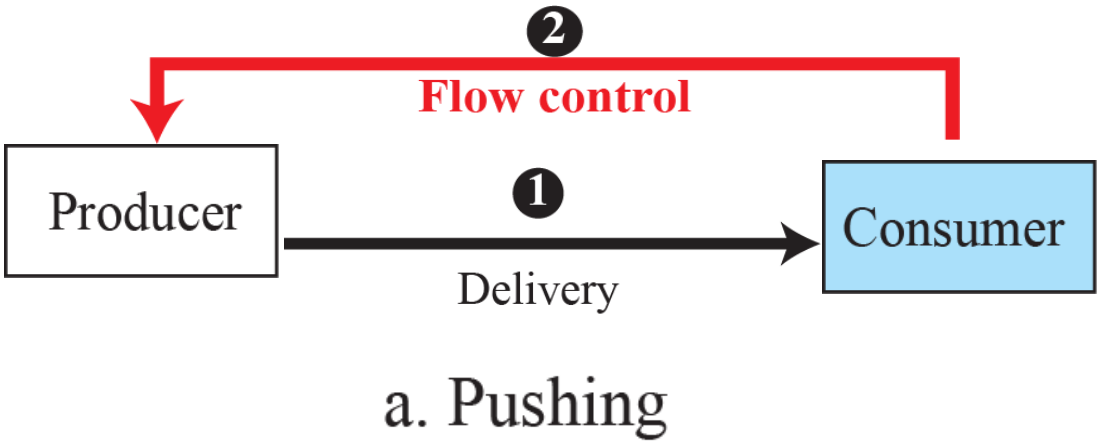
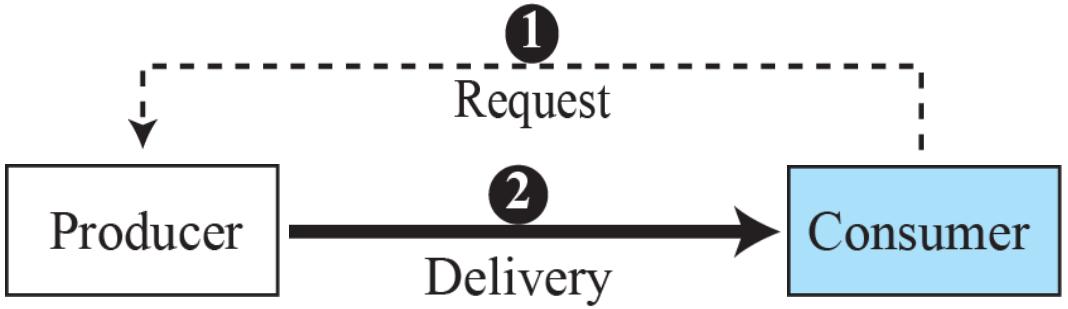


Figure 3.9: *Pushing or pulling*

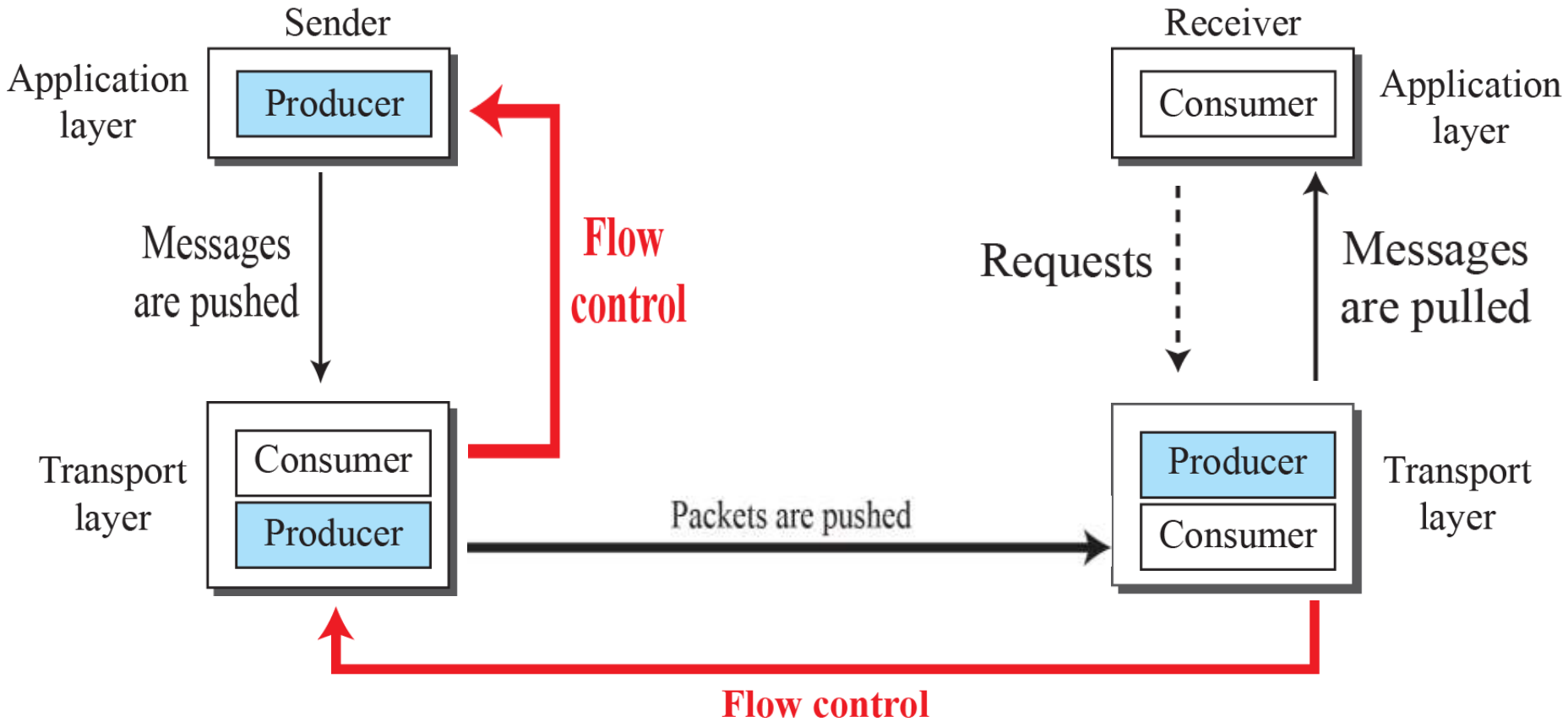


a. Pushing



b. Pulling

Figure 3.10: Flow control at the transport layer



Example 3.2

The above discussion requires that the consumers communicate with the producers on two occasions: when the buffer is full and when there are vacancies. If the two parties use a buffer with 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, however, this type of flow control, using a single-slot buffer at the sender and the receiver, is inefficient.

Figure 3.11: Error control at the transport layer

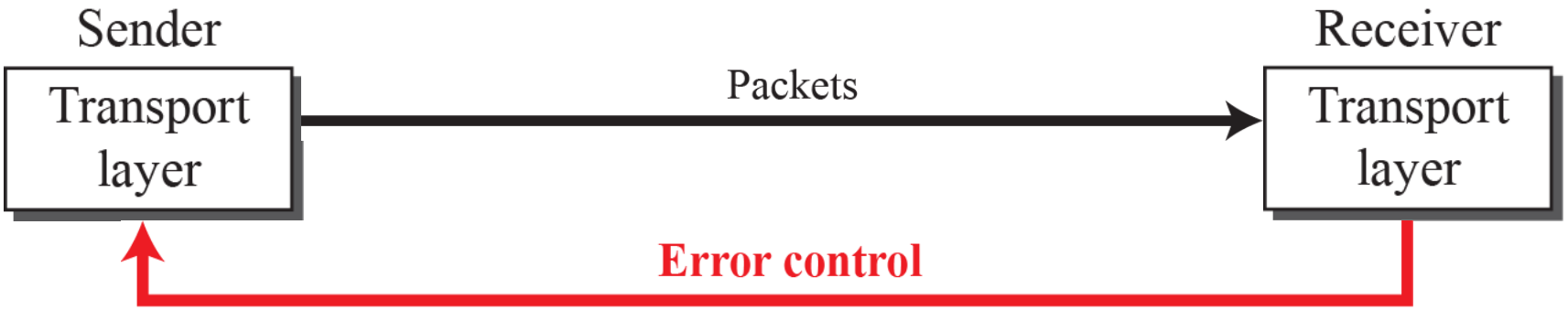
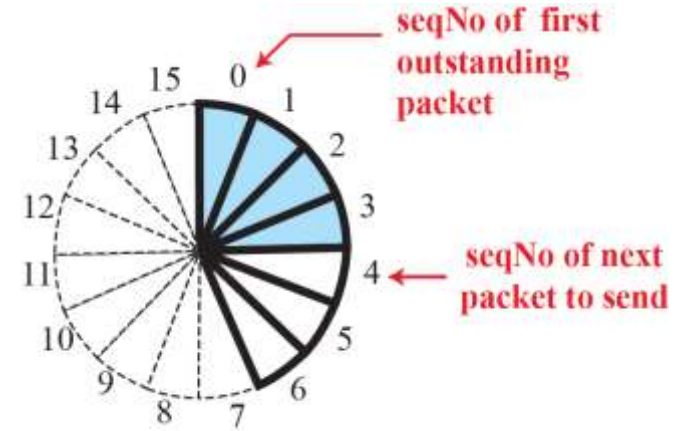
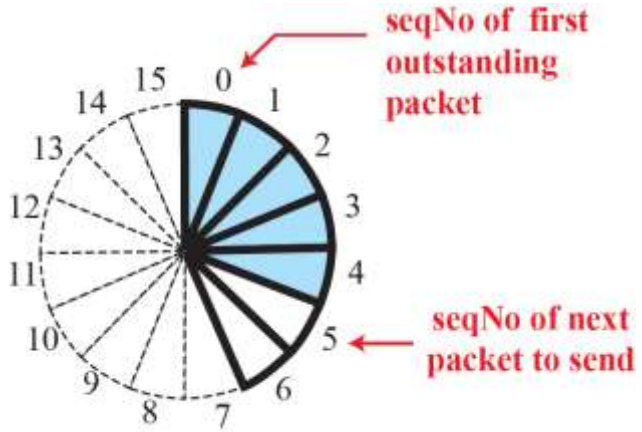


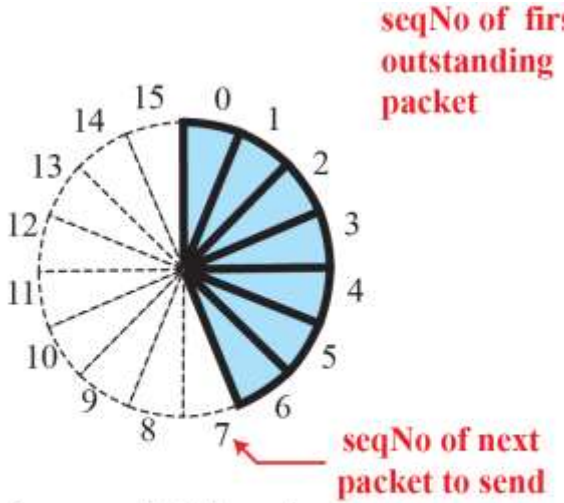
Figure 3.12: Sliding window in circular format



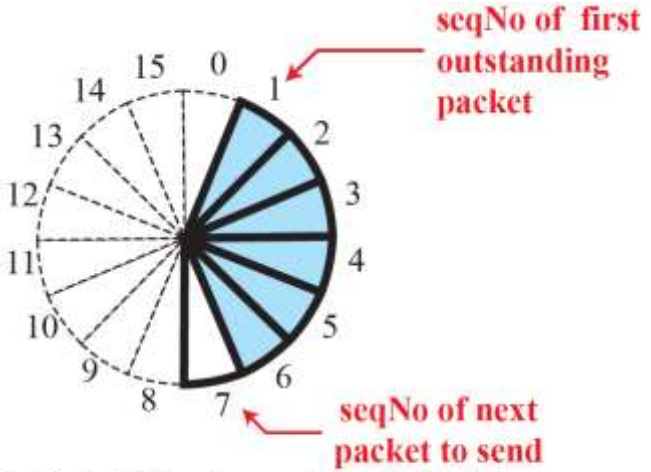
a. Four packets have been sent.



b. Five packets have been sent.

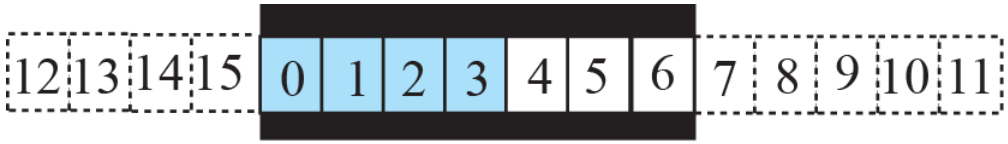


c. Seven packets have been sent; window is full.

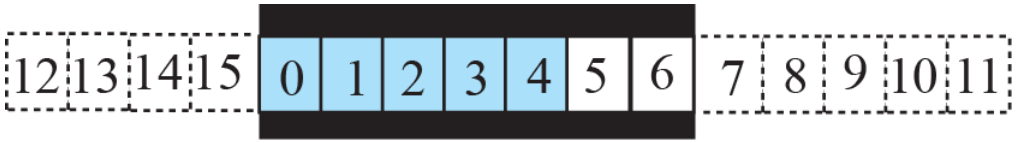


d. Packet 0 has been acknowledged; window slides.

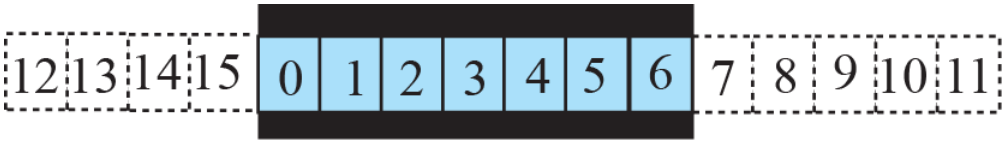
Figure 3.13: Sliding window in linear format



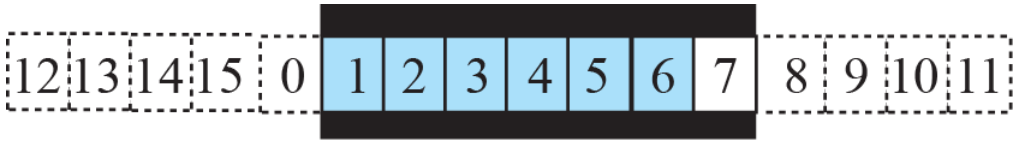
a. Four packets have been sent.



b. Five packets have been sent.



c. Seven packets have been sent;
window is full.



d. Packet 0 has been acknowledged;
window slides.

Figure 3.14: *Connectionless service*

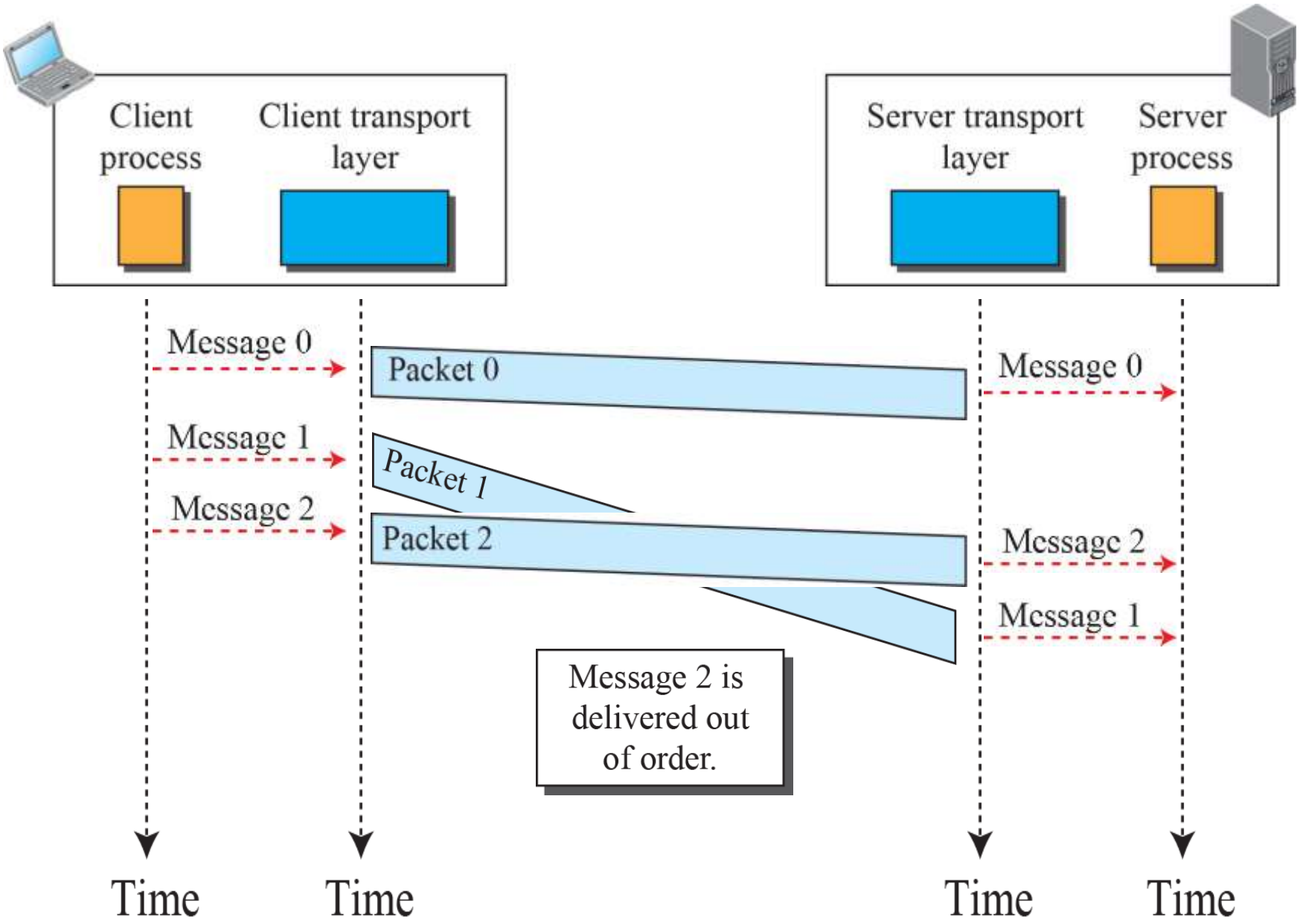


Figure 3.15: Connection-oriented service

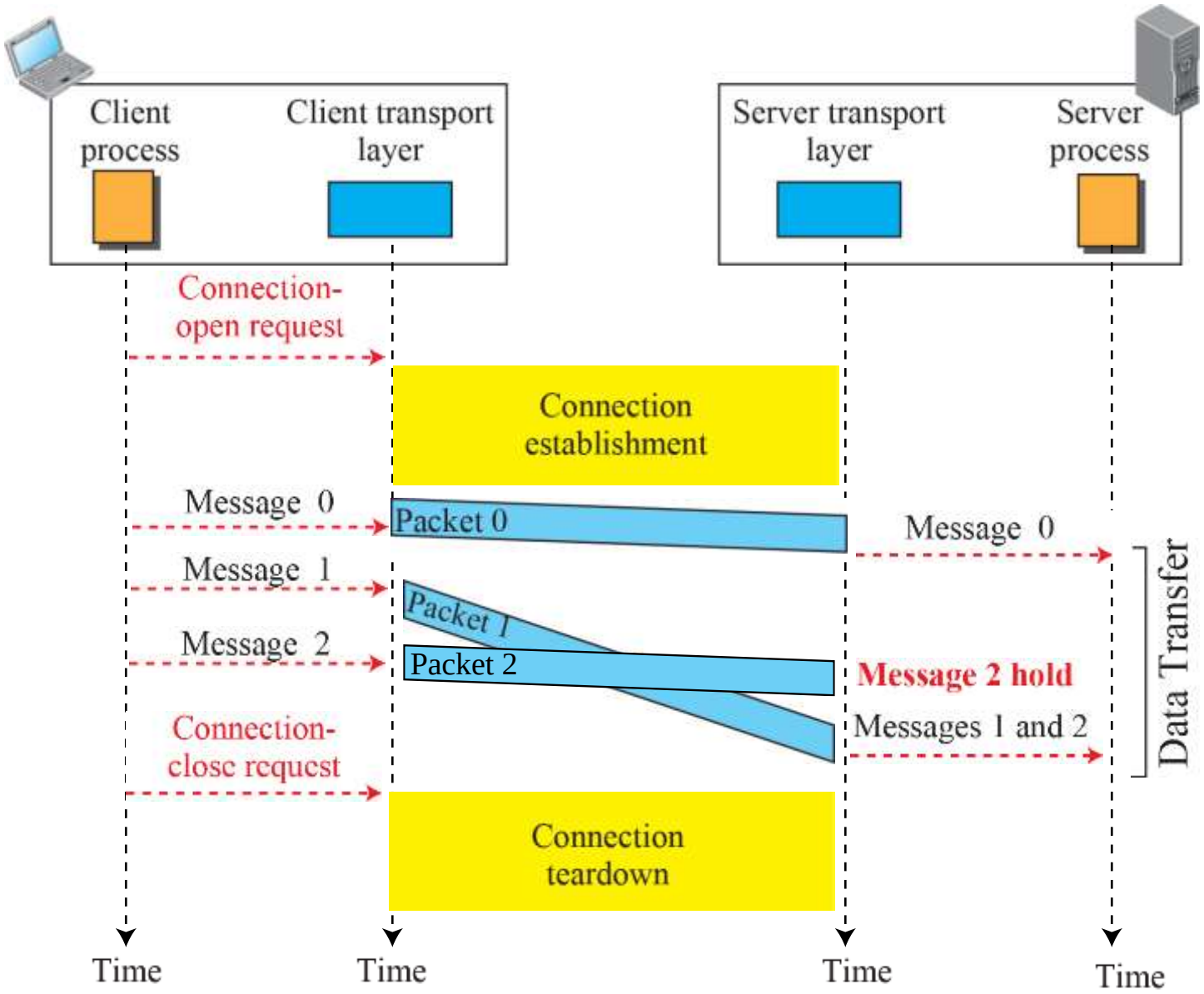
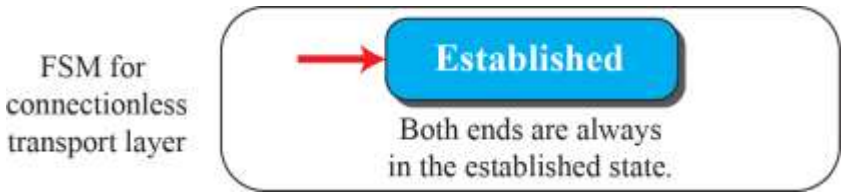
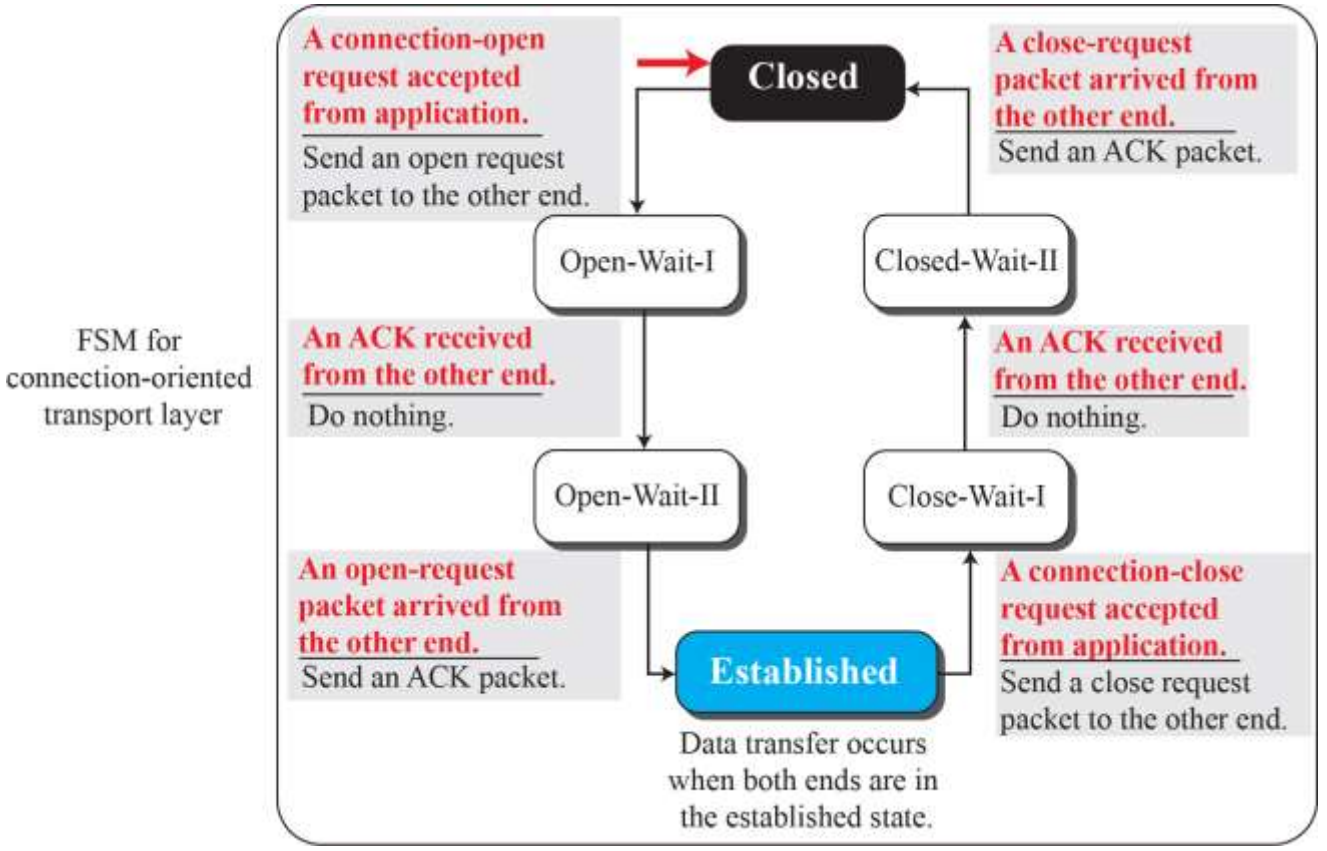


Figure 3.16: Connectionless and connection-oriented service represented as FSMs



Note:
The colored arrow shows the starting state.



3-2 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.

3.2.1 Simple Protocol

Our first protocol is a simple connectionless protocol with neither flow nor error control. We assume that the receiver can immediately handle any packet it receives. In other words, the receiver can never be overwhelmed with incoming packets. Figure 3.17 shows the layout for this protocol.

- ***FSMs***

Figure 3.17: Simple protocol

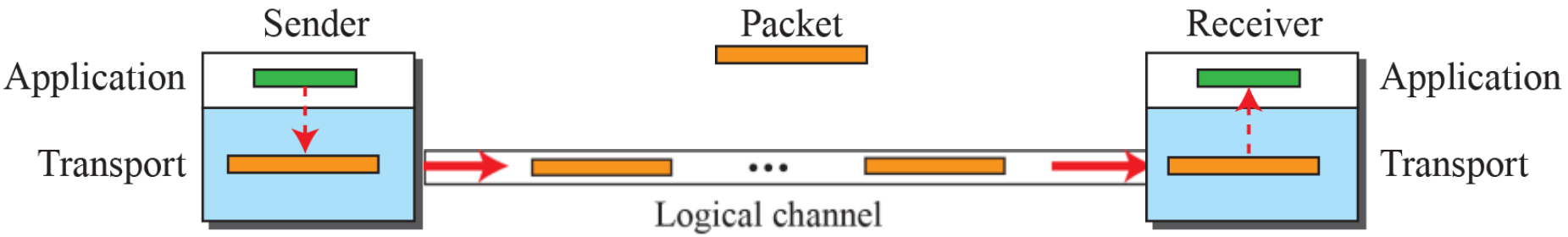
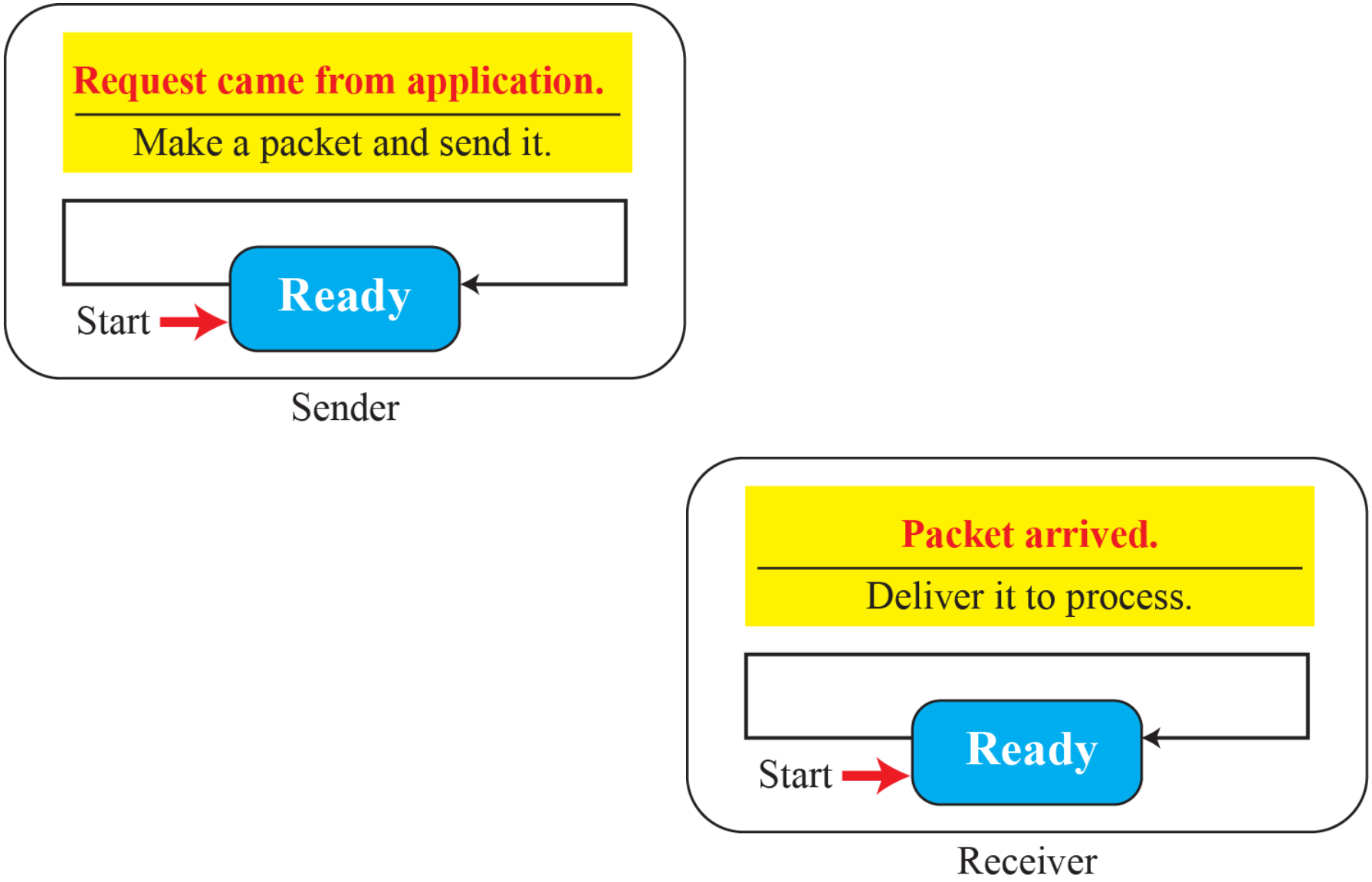


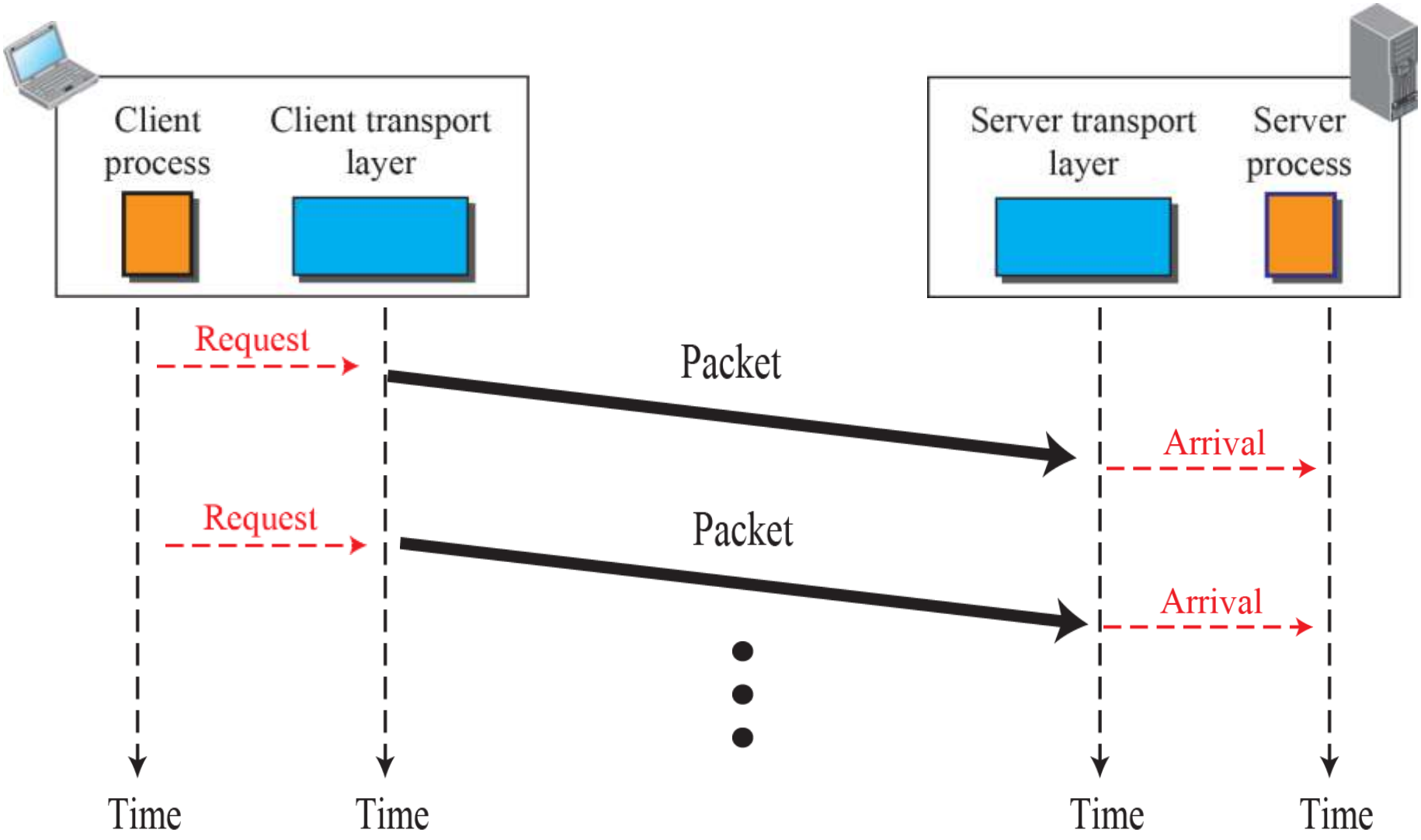
Figure 3.18: FSMs for the simple protocol



Example 3.3

Figure 3.19 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 3.19: Flow diagram for Example 3.3



3.2.2 Stop-and-Wait Protocol

Our second protocol is a connection-oriented protocol called the Stop-and-Wait 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. To detect corrupted packets, we need to add a checksum to each data packet. When a packet arrives at the receiver site,

3.2.2 (continued)

- *Sequence Numbers*
- *Acknowledgment Numbers*
- *FSMs*
  - *Sender*
  - *Receiver*
- *Efficiency*
- *Pipelining*

Figure 3.20: Stop-and-Wait protocol

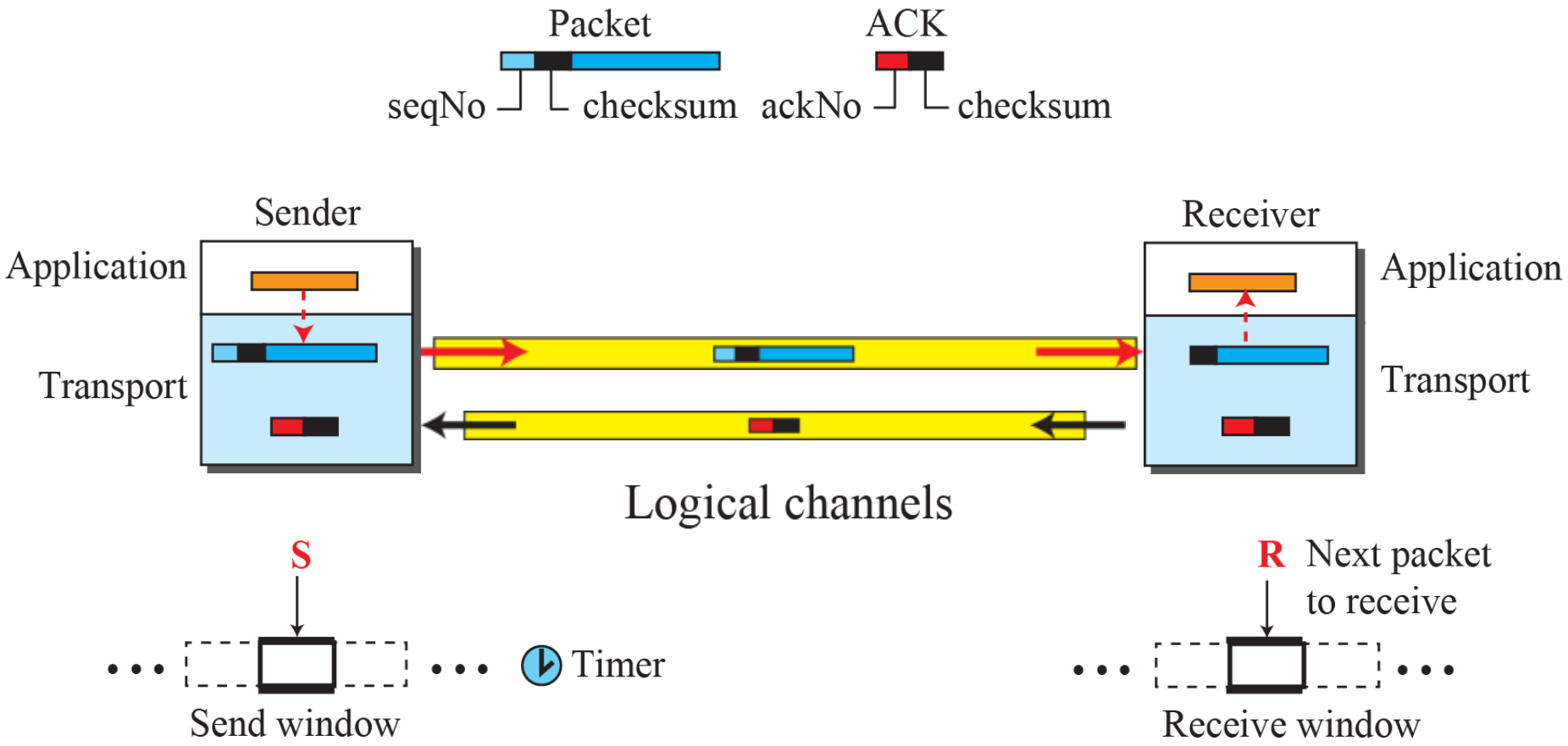
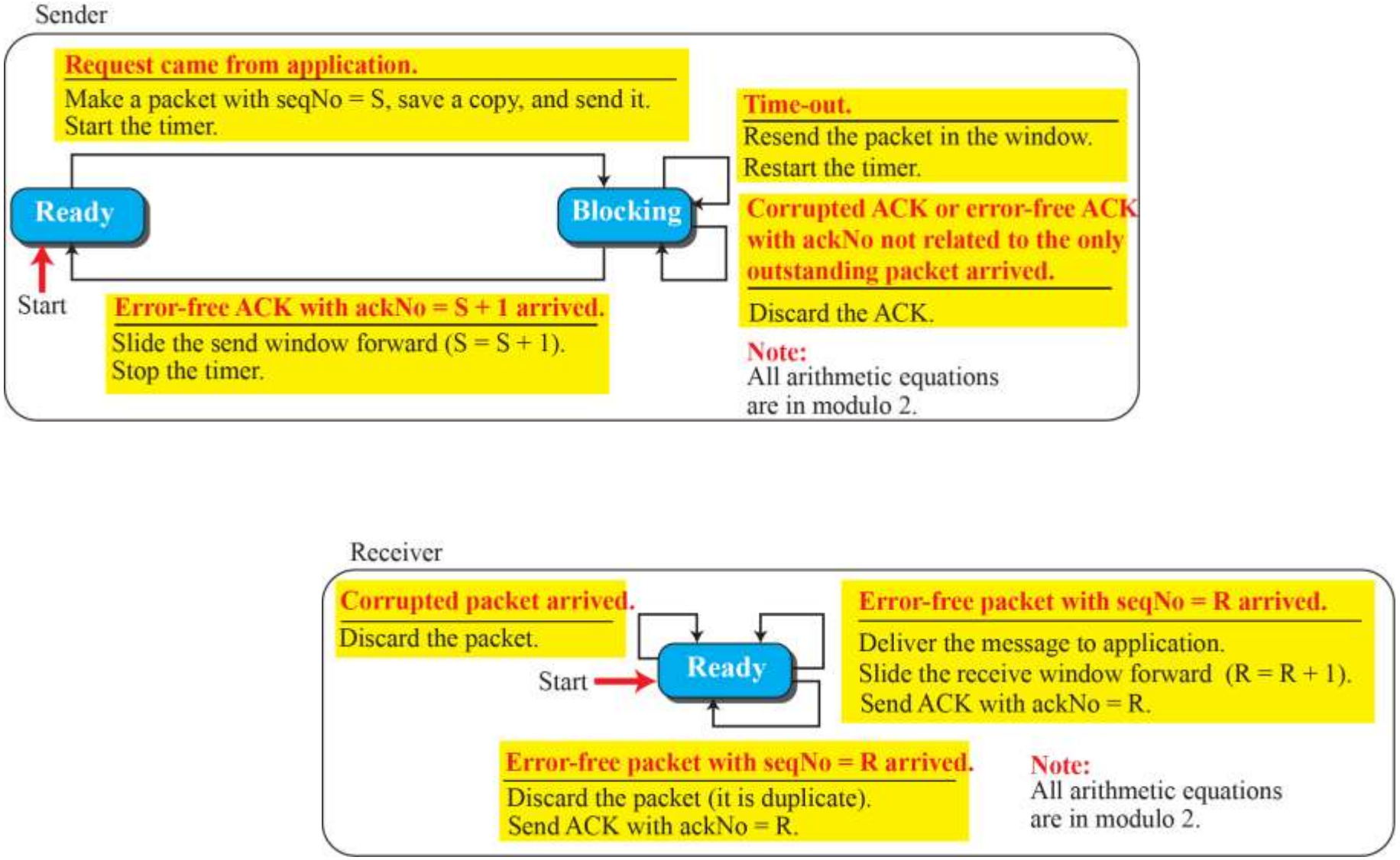


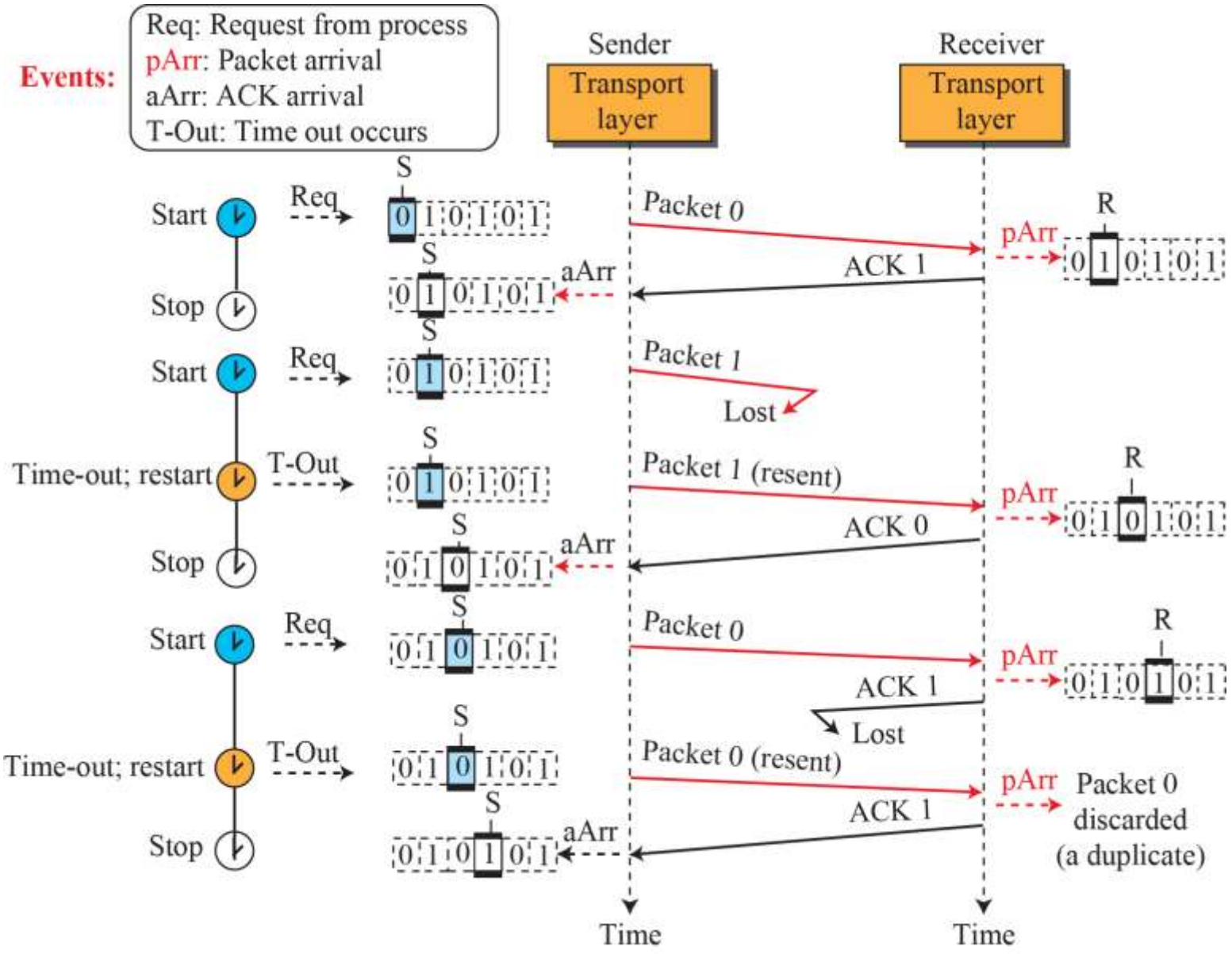
Figure 3.21: FSM for the Stop-and-Wait protocol



Example 3.4

Figure 3.22 shows an example of the 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 3.22: Flow diagram for Example 3.4



Example 3.5

Assume that, 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 $(1 \times 10^6) \times (20 \times 10^{-3}) = 20{,}000$ 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. The link utilization is only 1,000/20,000, or 5 percent.

Example 3.6

What is the utilization percentage of the link in Example 3.5 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 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.

3.2.3 Go-Back-N Protocol

To improve the efficiency of transmission multiple packets must be in transition while the sender is waiting for acknowledgment. In this section, we discuss one protocol that can achieve this goal; in the next section, we discuss a second. The first is called Go-Back-N (GBN) (the rationale for the name will become clear later).

3.2.3 (continued)

- Sequence Numbers
- Acknowledgment Numbers
- Send Window
- Receive Window
- Timers
- Resending packets

3.2.3 (continued)

- *FSMs*
  - *Sender*
  - *Receiver*
- *Send Window Size*
- *Go-Back-N versus Stop-and-Wait*

Figure 3.23: Go-Back-N protocol

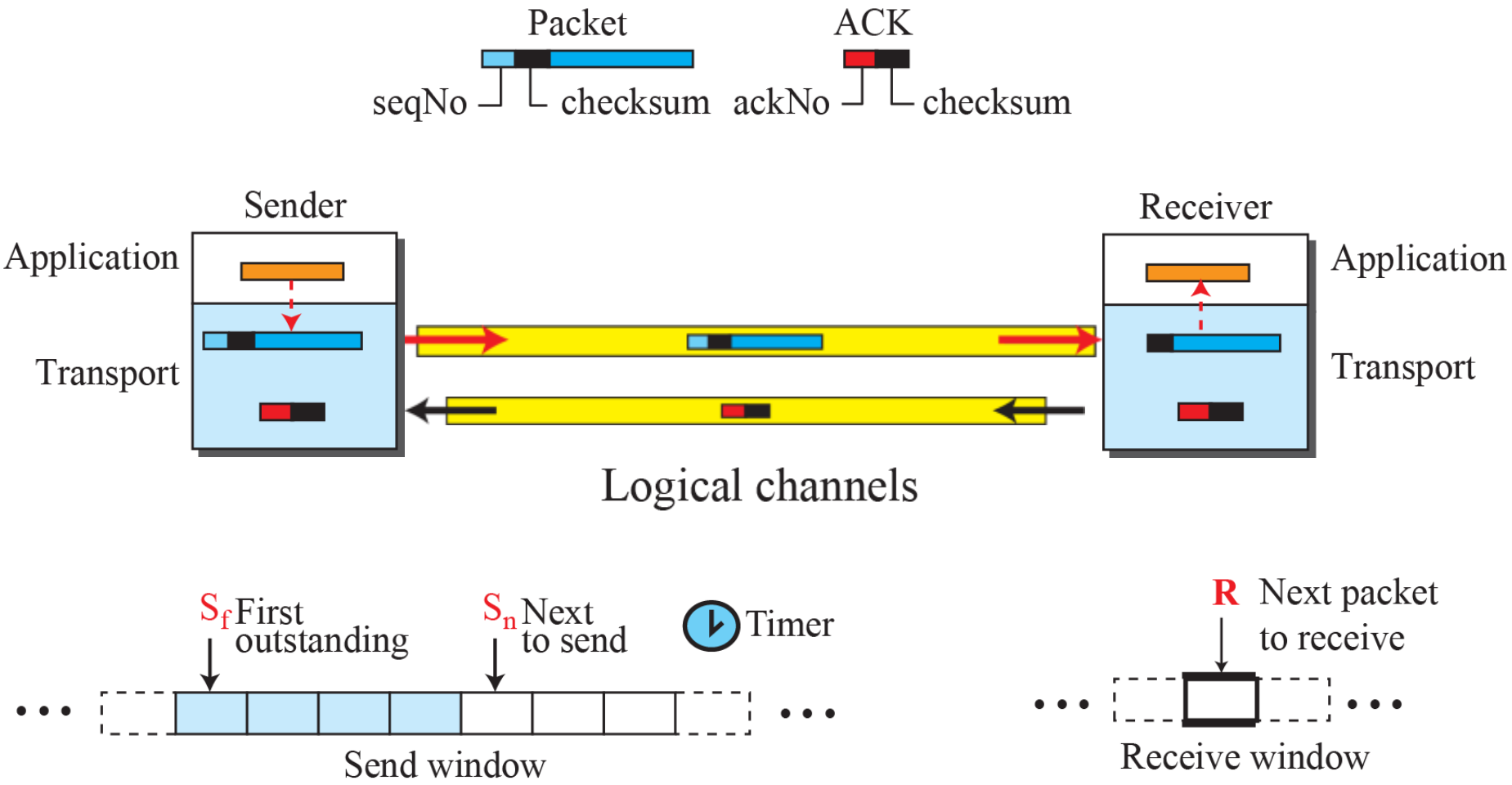


Figure 3.24: Send window for Go-Back-N

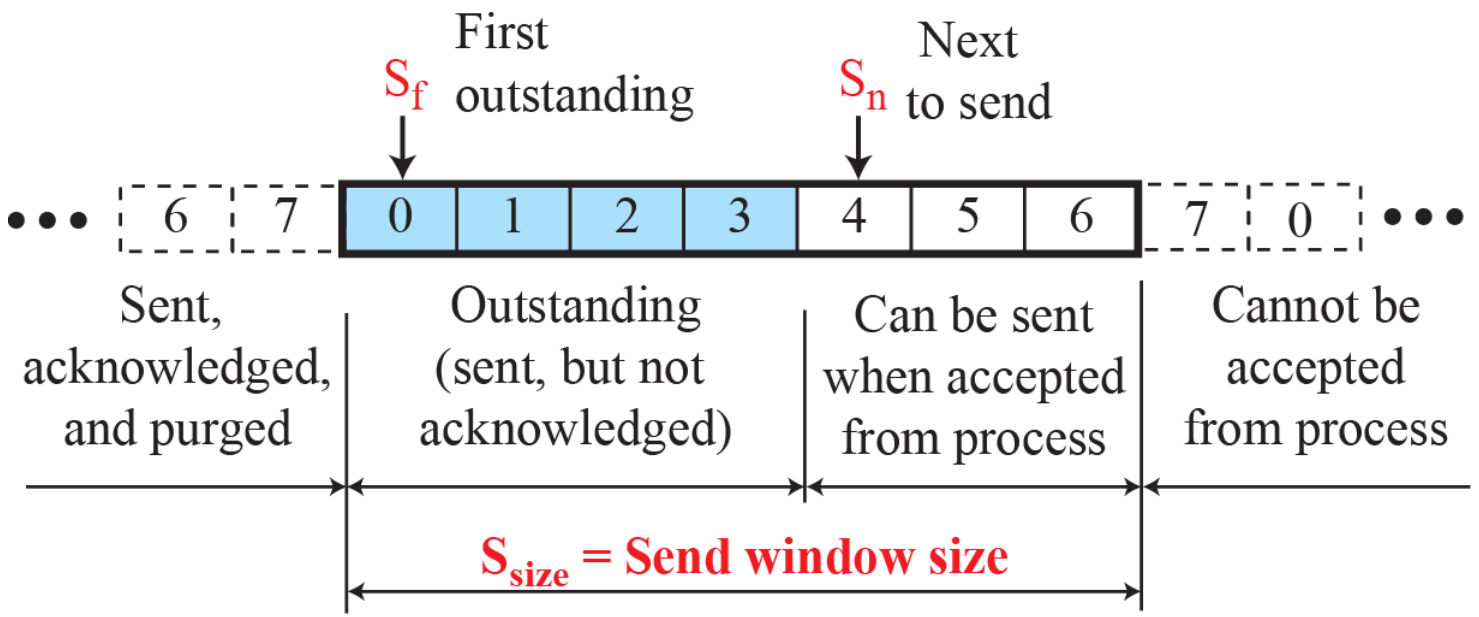


Figure 3.25: Sliding the send window

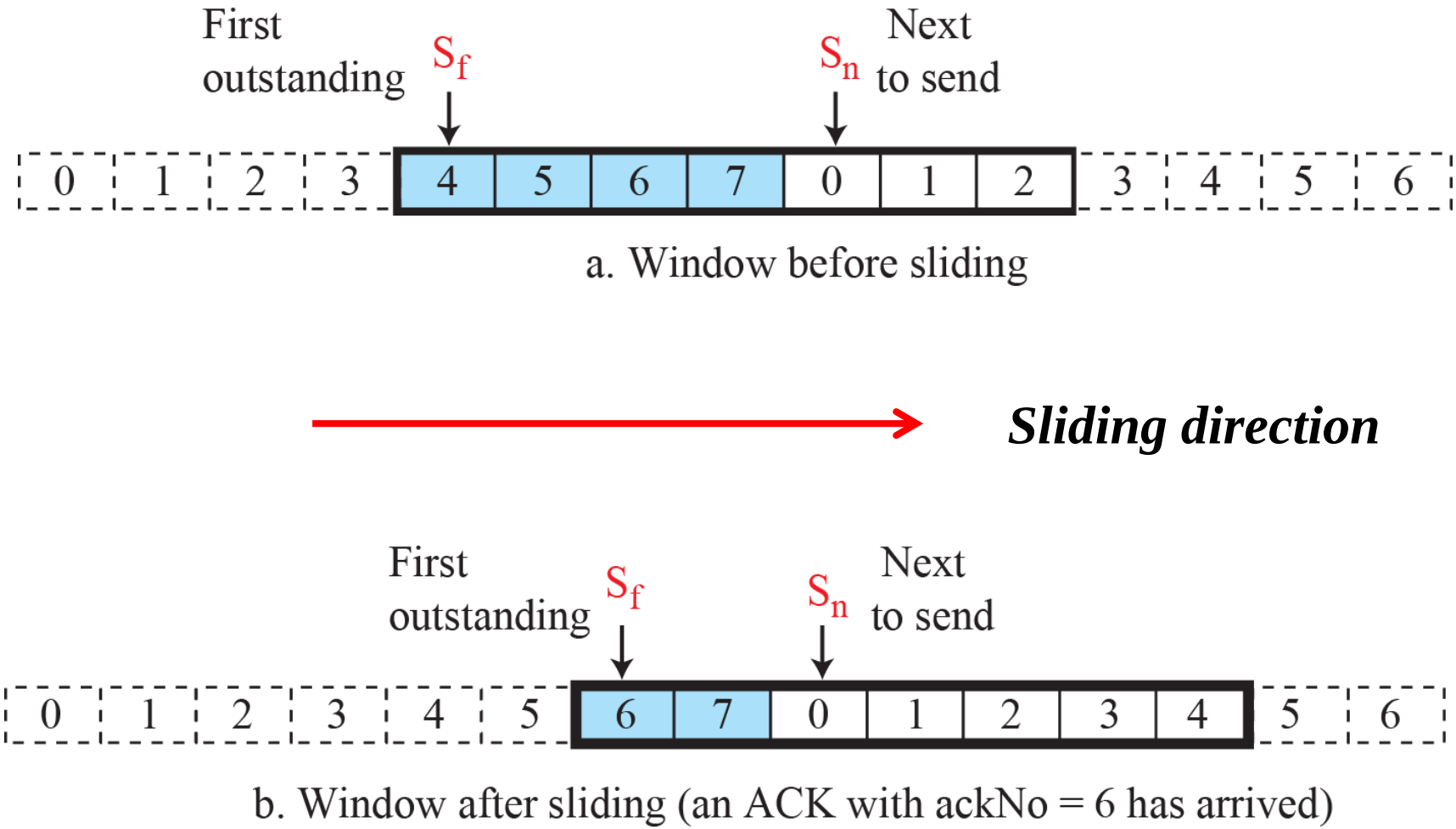


Figure 3.26: Receive window for Go-Back-N

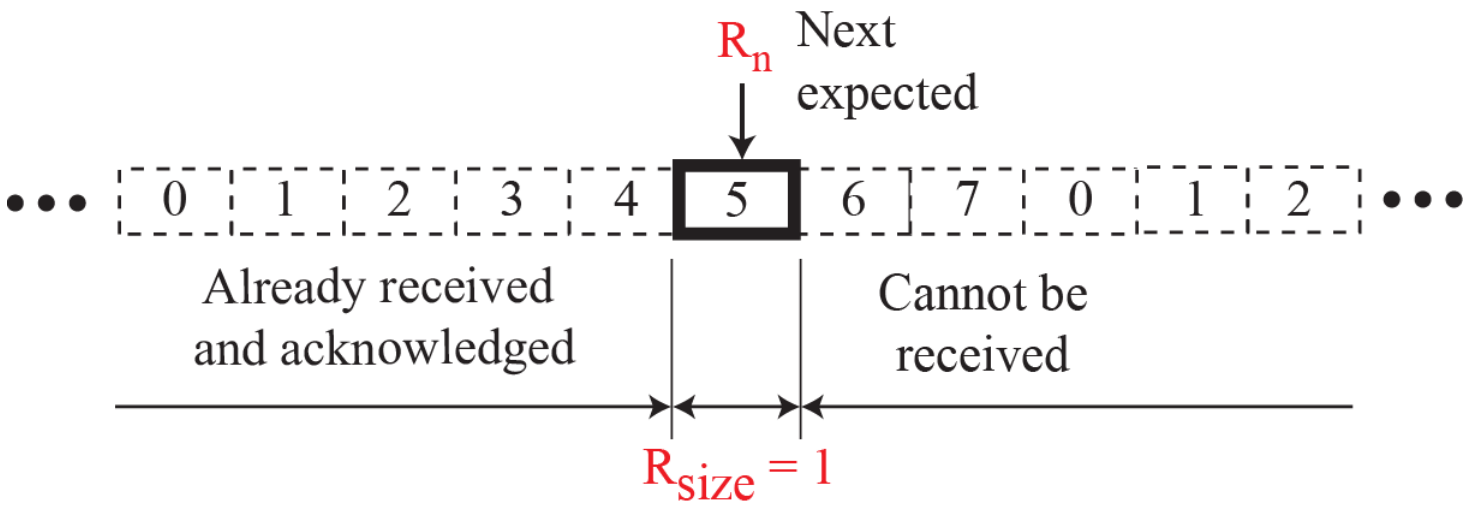


Figure 3.27: FSMs for the Go-Back-N protocol

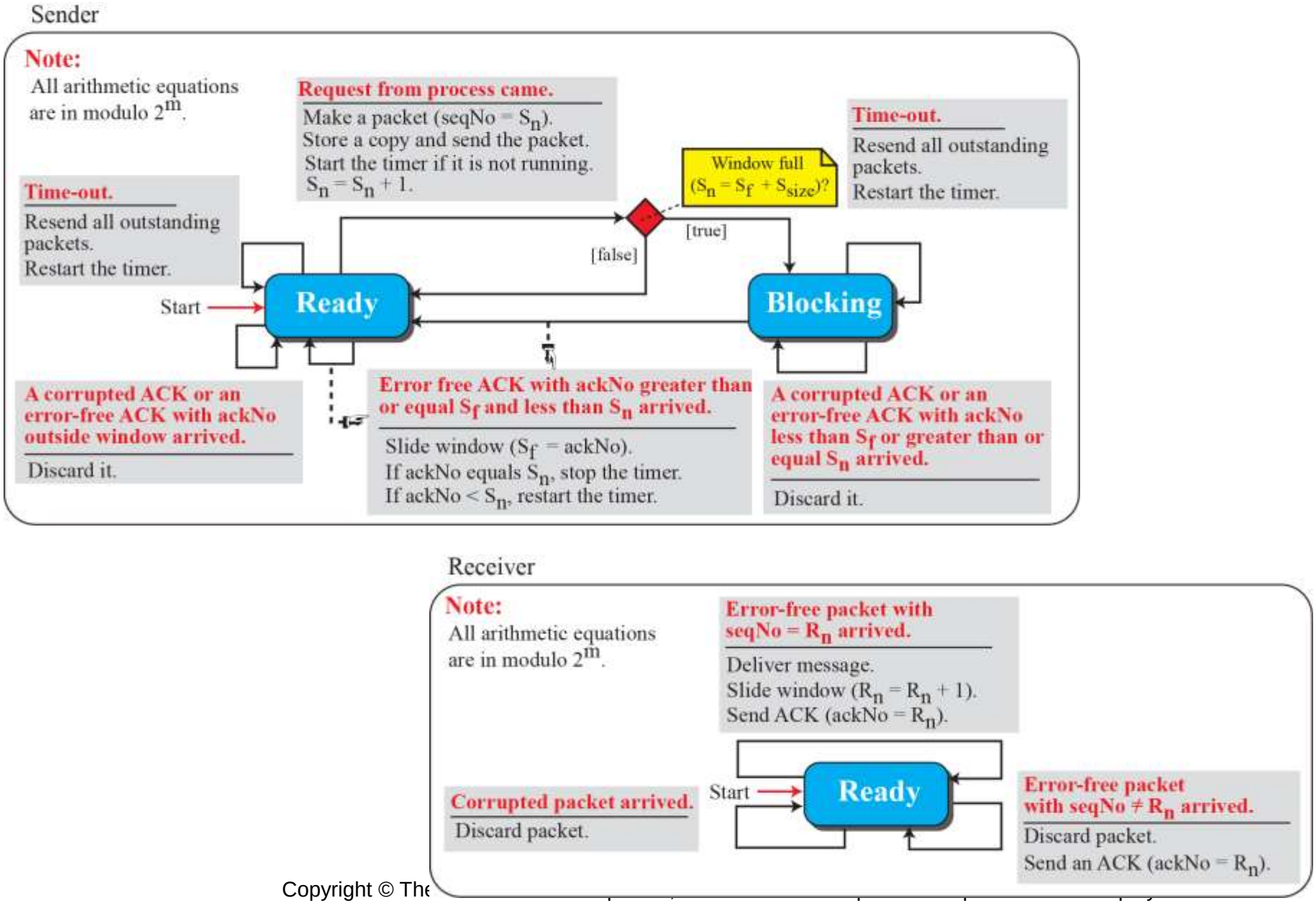
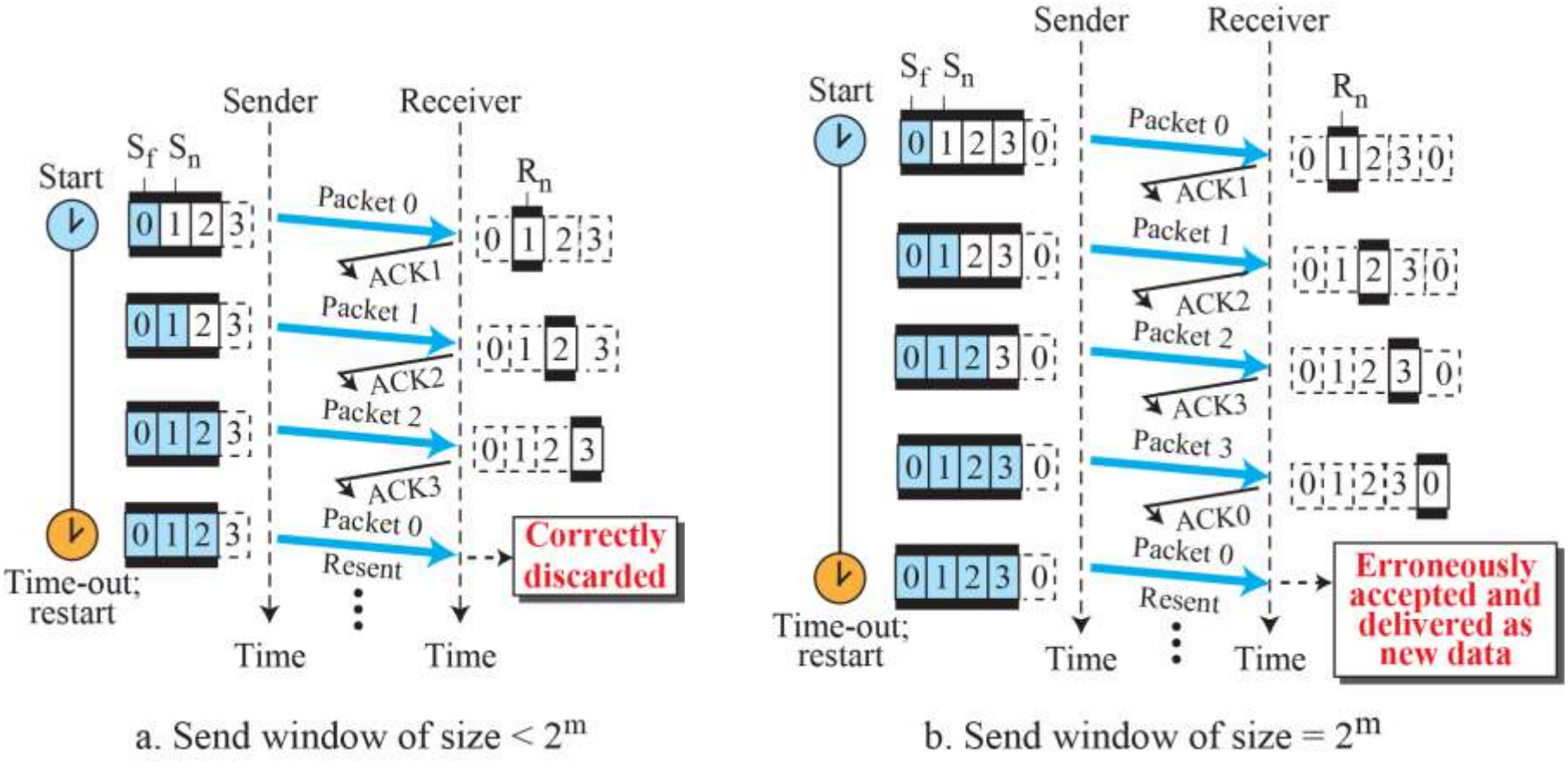


Figure 3.28: Send window size for Go-Back-N



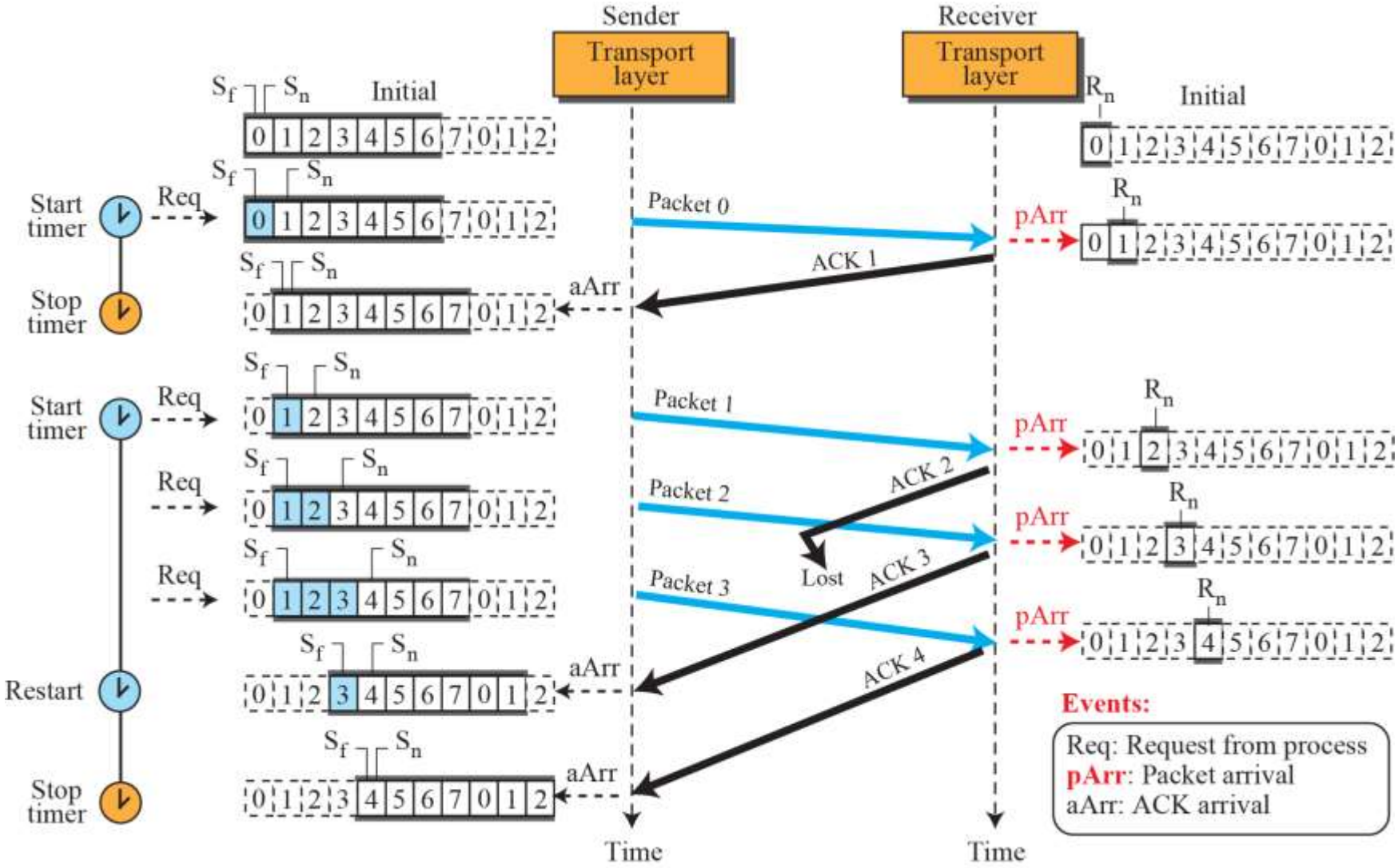
a. Send window of size < 2^m

b. Send window of size = 2^m

Example 3.7

Figure 3.29 shows an example of Go-Back-*N*. This is an example of a case where the forward channel is reliable, but the reverse is not. 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.

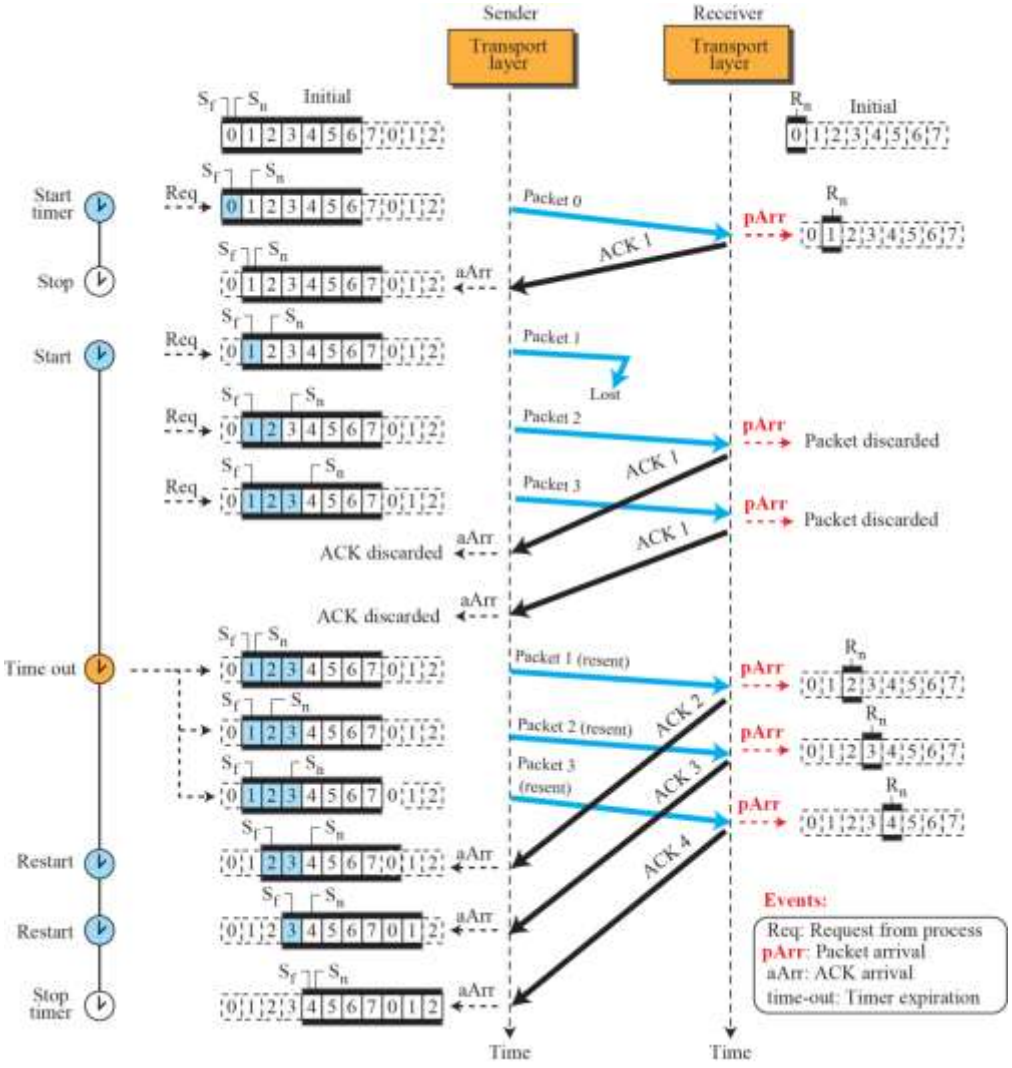
Figure 3.29: Flow diagram for Example 3.7



Example 3.8

Figure 3.30 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 to S_f, not greater that S_f . So the sender discards them. When the time-out occurs, the sender resends packets 1, 2, and 3, which are acknowledged.

Figure 3.30: Flow diagram for Example 3.8



3.2.4 Selective-Repeat Protocol

The Go-Back-N protocol simplifies the process at the receiver. The receiver keeps track of only one variable, and there is no need to buffer out-of-order packets; they are simply discarded. Another protocol, called the Selective-Repeat (SR) protocol, has been devised, which, as the name implies, resends only selective packets, those that are actually lost. The outline of this protocol is shown in Figure 3.31.

3.2.4 (continued)

- *Windows*
- *Timer*
- *Acknowledgments*
- *FSMs*
  - *Sender*
  - *Receiver*

Figure 3.31: Outline of Selective-Repeat

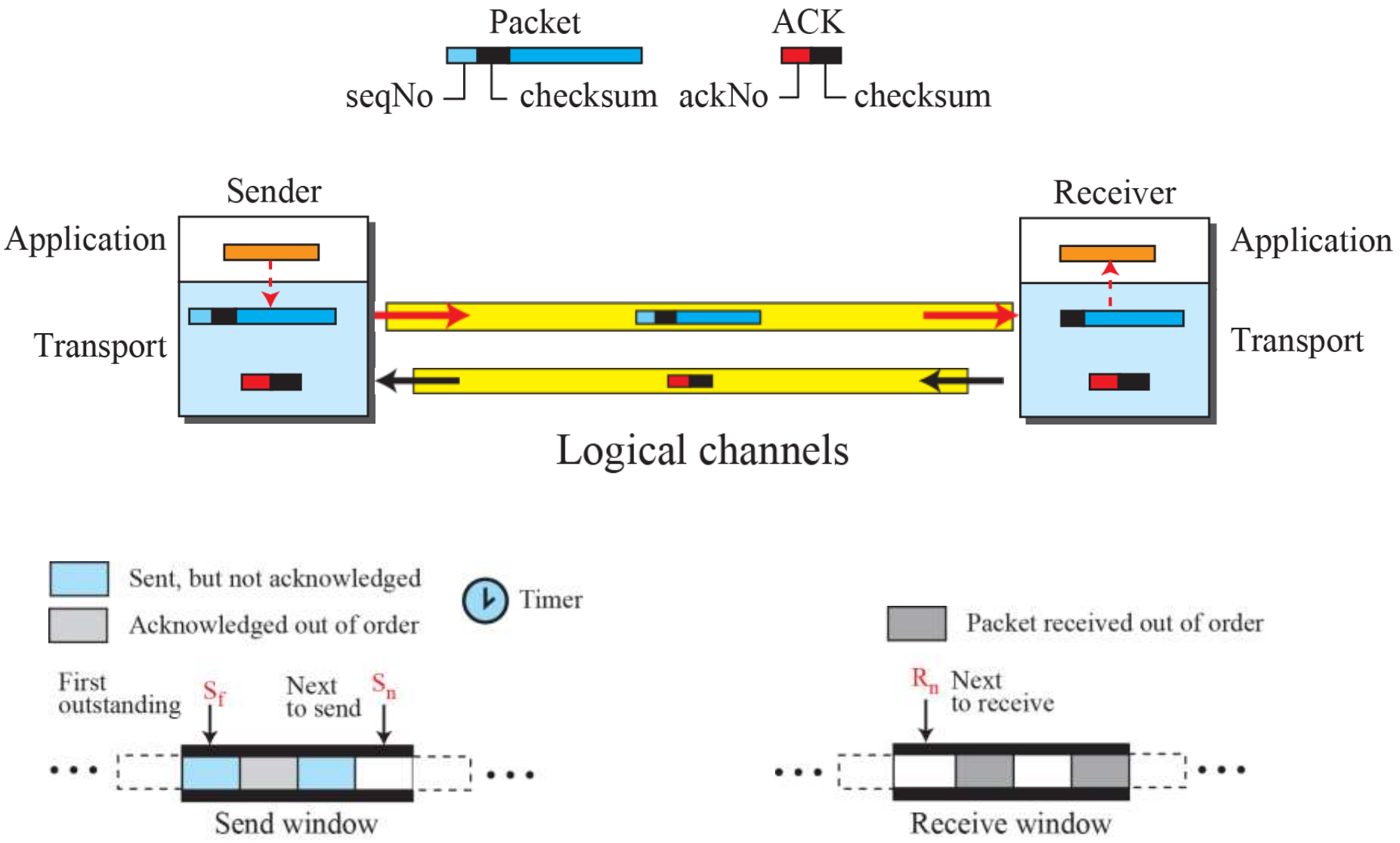


Figure 3.32: Send window for Selective-Repeat protocol

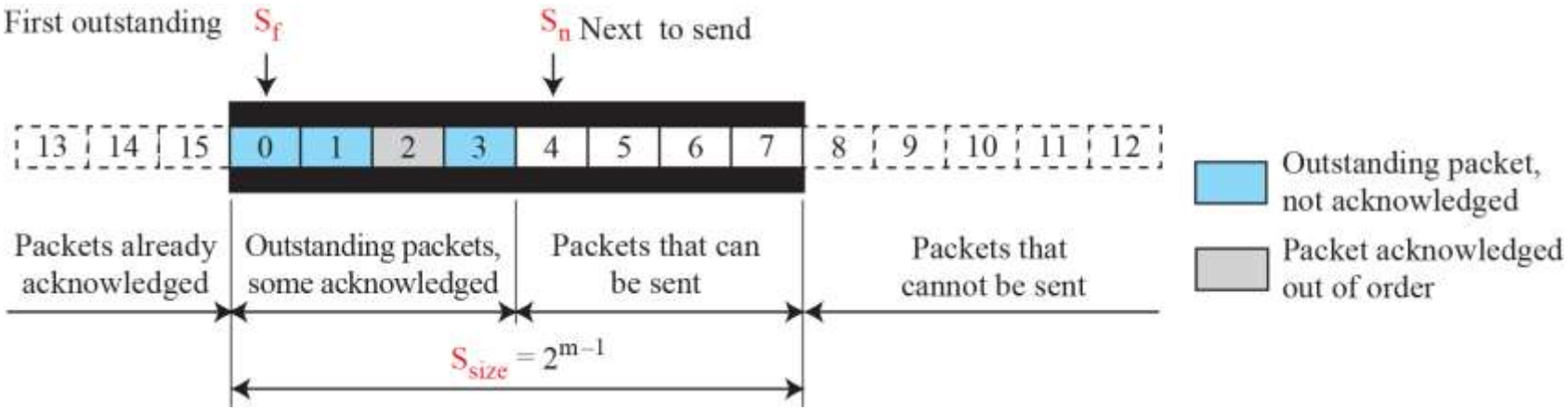
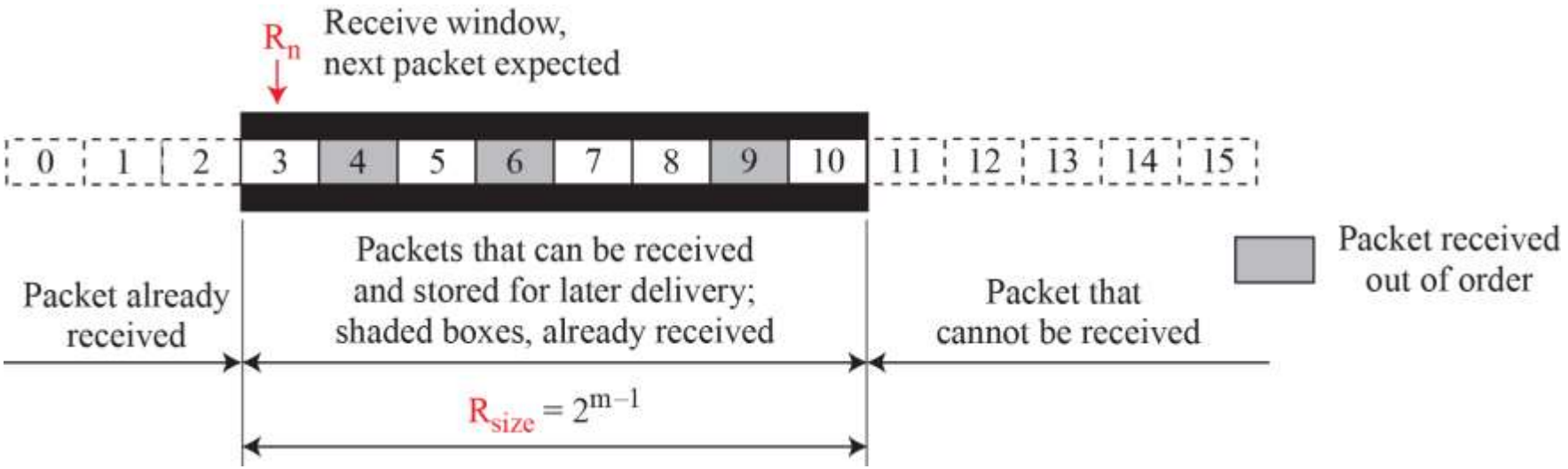


Figure 3.33: *Receive window for Selective-Repeat protocol*



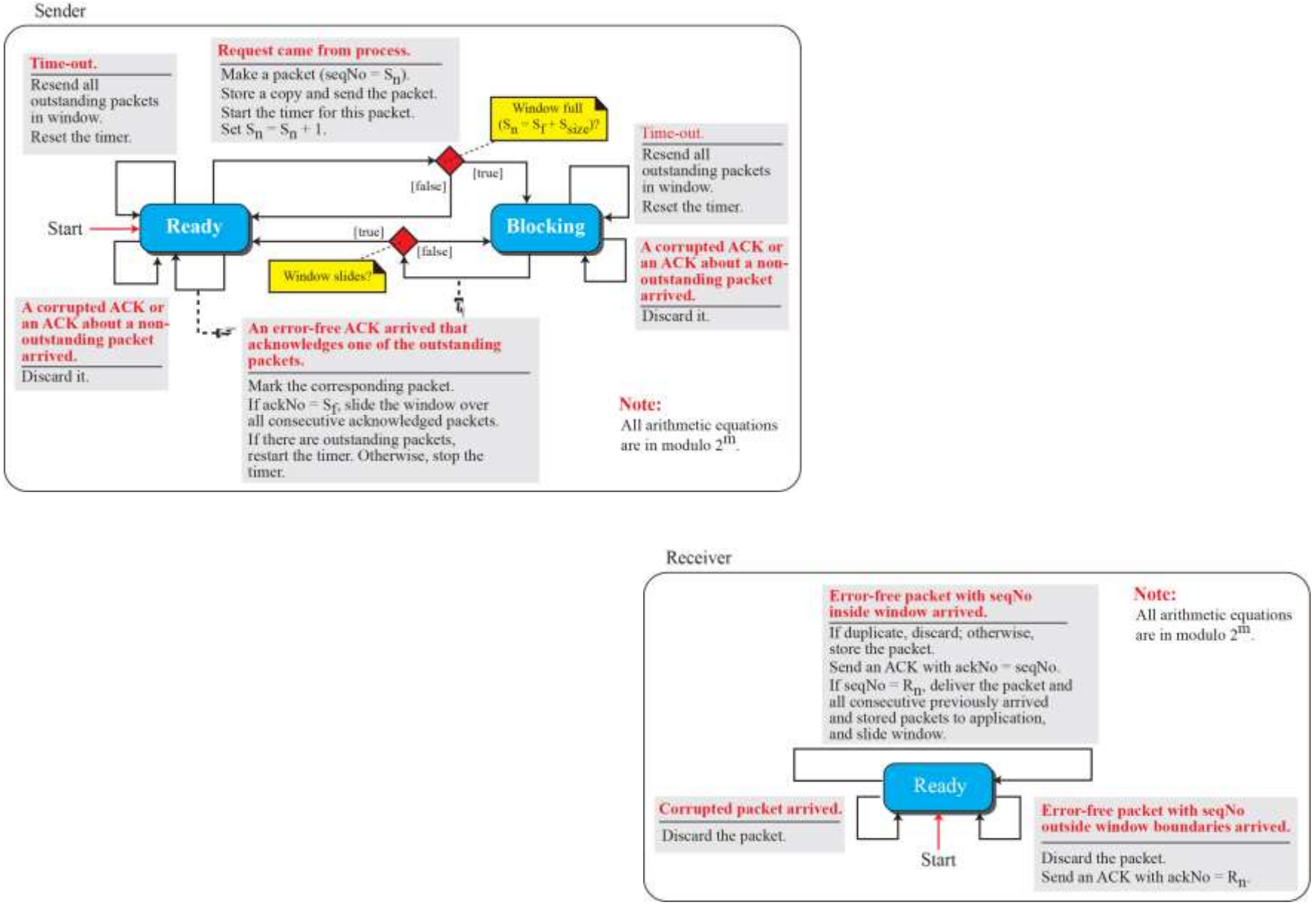
Example 3.9

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.

Figure 3.34: FSMs for SR protocol



Example 3.10

This example is similar to Example 3.8 (Figure 3.30) in which packet 1 is lost. We show how Selective-Repeat behaves in this case. Figure 3.35 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.

Example 3.10 (continued)

At the receiver site we need to distinguish between the acceptance of a packet and its delivery to the application layer. At the second arrival, packet 2 arrives and is stored and marked (shaded slot), but it cannot be delivered because packet 1 is missing. At the next arrival, packet 3 arrives and is marked and stored, but still none of the packets can be delivered. Only at the last arrival, when finally a copy of packet 1 arrives, can packets 1, 2, and 3 be delivered to the application layer. There are two conditions for the delivery of packets to the application layer: First, a set of consecutive packets must have arrived. Second, the set starts from the beginning of the window.

Figure 3.35: Flow diagram for Example 3.10

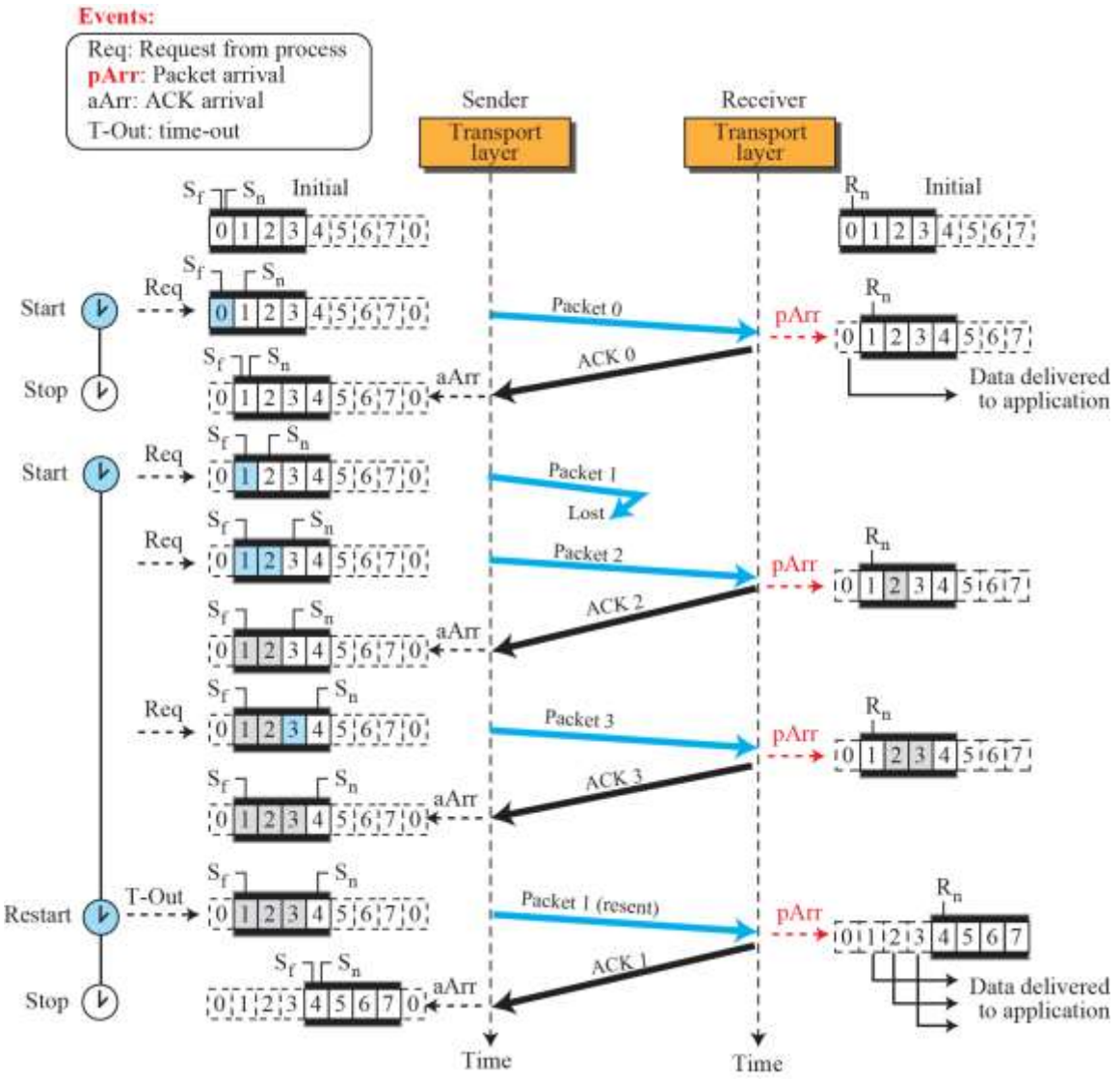
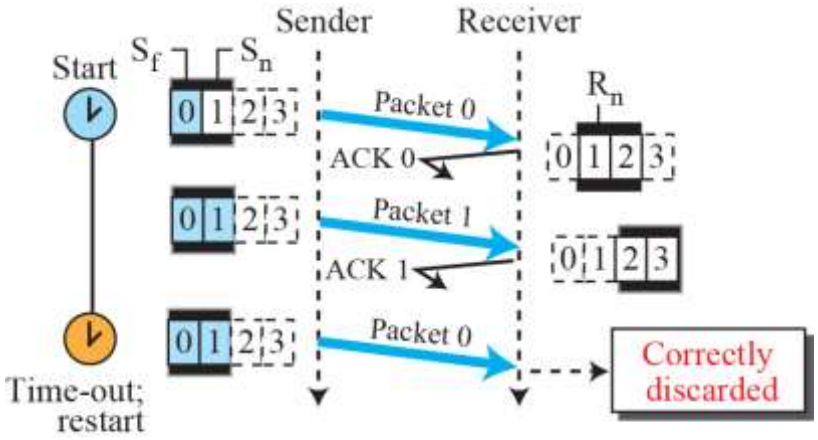
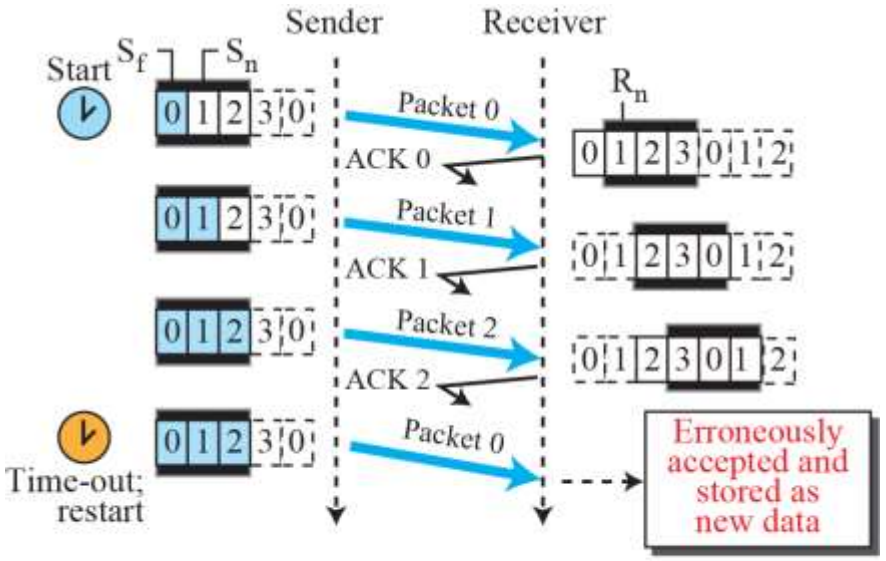


Figure 3.36: Selective-Repeat, window size



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

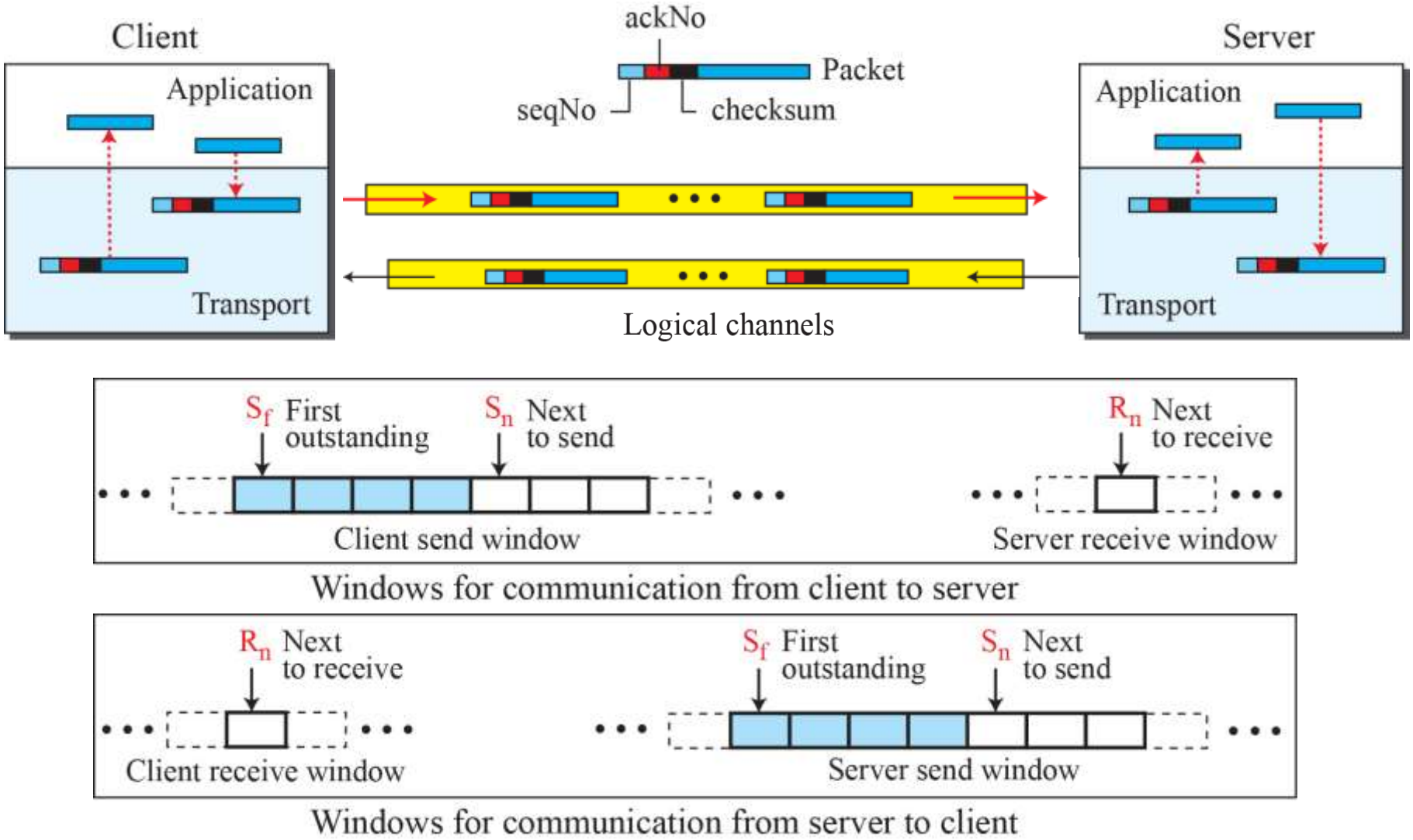


b. Send and receive windows of size > 2^{m-1}

3.2.5 Bidirectional Protocols

The four protocols we discussed earlier in this section are all unidirectional: data packets flow in only one direction and acknowledgments travel in the other direction. In real life, data packets are normally flowing in both directions: from client to server and from server to client. This means that acknowledgments also need to flow in both directions. A technique called piggybacking is used to improve the efficiency of the bidirectional protocols.

Figure 3.37: Design of piggybacking in Go-Back-N



3.2.6 Internet Transport-Layer Protocols

A network is the interconnection of a set of devices capable of communication. In this definition, a device can be a host such as a large computer, desktop, laptop, workstation, cellular phone, or security system. A device in this definition can also be a connecting device such as a router a switch, a modem that changes the form of data, and so on.

Figure 3.38: Position of transport-layer protocols in the TCP/IP protocol suite

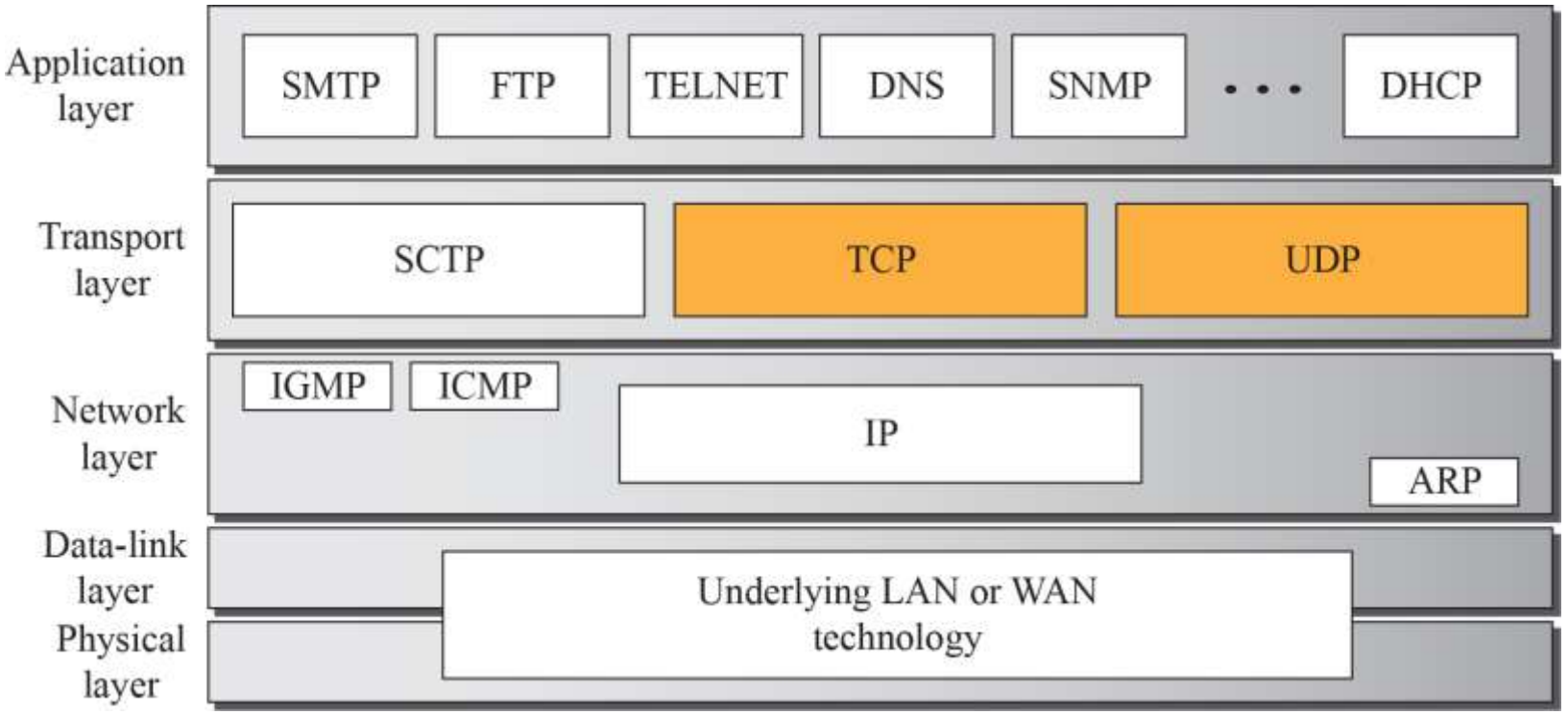


Table 3.1: Some well-known ports used with UDP and TCP

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

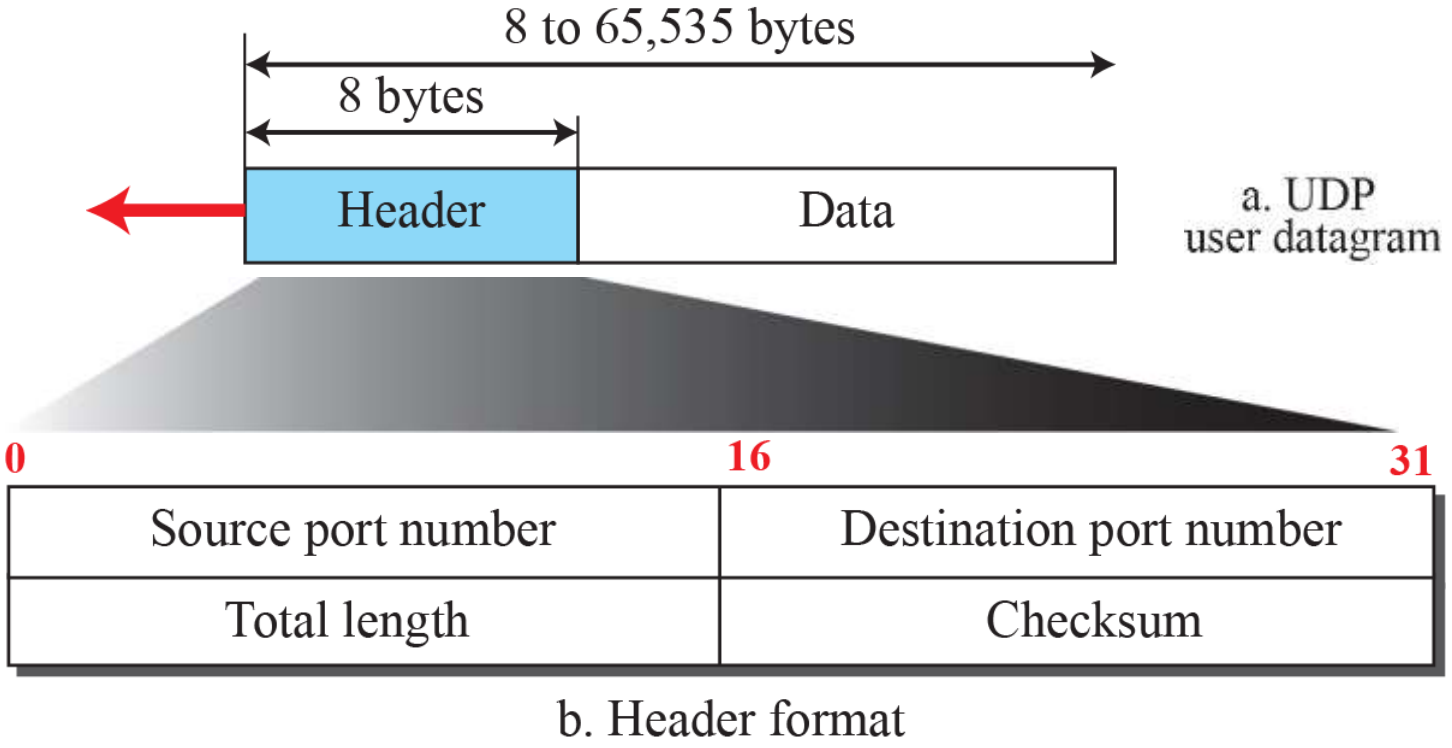
3-3 USER DATAGRAM PROTOCOL (UDP)

The User Datagram Protocol (UDP) is a connectionless, unreliable transport protocol. It does not add anything to the services of IP except for providing process-to-process instead of host-to-host communication. UDP is a very simple protocol using a minimum of overhead.

3.3.1 User Datagram

UDP packets, called user datagrams, have a fixed size header of 8 bytes made of four fields, each of 2 bytes (16 bits). Figure 3.39 shows the format of a user datagram. The first two fields define the source and destination port numbers. The third field defines the total length of the user datagram, header plus data. The 16 bits can define a total length of 0 to 65,535 bytes.

Figure 3.39: User datagram packet format



Example 3.11

The following is the contents 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 3.11 (continue)

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 3.1).

3.3.2 UDP Services

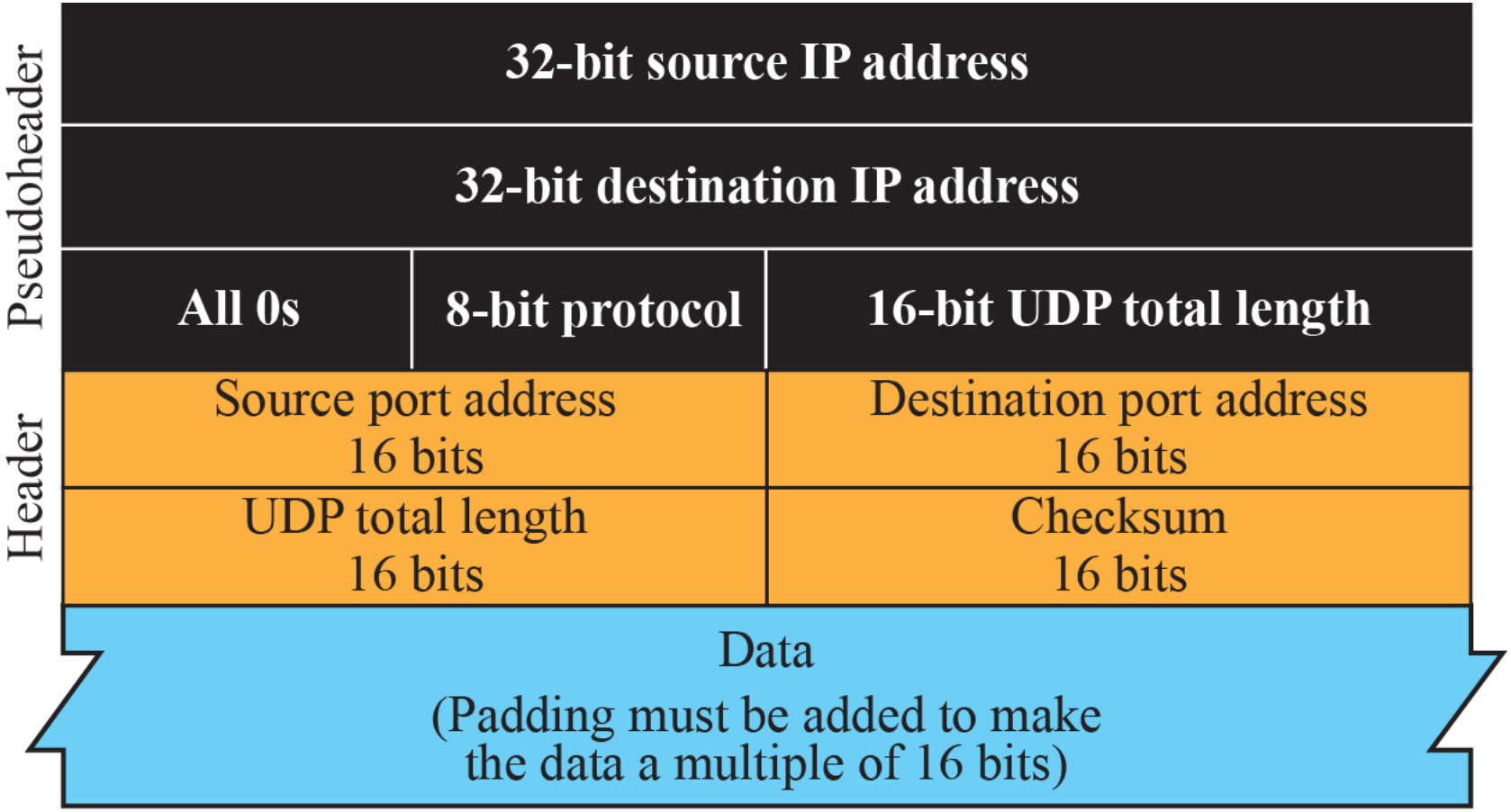
Earlier we discussed the general services provided by a transport-layer protocol. In this section, we discuss what portions of those general services are provided by UDP.

- ***Process-to-Process Communication***
- ***Connectionless Services***
- ***Flow Control***
- ***Error Control***

3.3.2 (continued)

- **Checksum**
  - **Optional Inclusion of Checksum**
- **Congestion Control**
- **Encapsulation and Decapsulation**
- **Queuing**
- **Multiplexing and Demultiplexing**
- **Comparison : UDP and Simple Protocol**

Figure 3.40: Pseudoheader for checksum calculation



Example 3.12

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 3.12 (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.

3.3.3 UDP Applications

Although UDP meets almost none of the criteria we mentioned earlier for a reliable transport-layer protocol, UDP is preferable for some applications. The reason is that some services may have some side effects that are either unacceptable or not preferable. An application designer sometimes needs to compromise to get the optimum.

In this section, we first discuss some features of UDP that may need to be considered when one designs an application program and then show some typical applications.

3.3.3 (continued)

- **UDP Features**
  - Connectionless Service
  - Lack of Error Control
  - Lack of Congestion Control
- **Typical Applications**

Example 3.13

A client-server application such as DNS (see Chapter 2) 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 3.14

A client-server application such as SMTP (see Chapter 2), 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 3.15

Assume we are downloading a very large text file from the Internet. We definitely need to use a transport layer that provides reliable service. We don't want part of the file to be missing or corrupted when we open the file. The delay created between the deliveries of the parts is not 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 3.16

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

3-4 TRANSMISSION CONTROL PROTOCOL

Transmission Control Protocol (TCP) is a connection-oriented, reliable protocol. TCP explicitly defines connection establishment, data transfer, and connection teardown phases to provide a connection-oriented service. TCP uses a combination of GBN and SR protocols to provide reliability.

3.4.1 TCP Services

A Before discussing TCP in detail, let us explain the services offered by TCP to the processes at the application layer.

- **Process-to-Process Communication**
- **Stream Delivery Service**
  - Sending and Receiving Buffers
  - Segments

3.4.1 (continued)

- ***Full-Duplex Communication***
- ***Multiplexing and Demultiplexing***
- ***Connection-Oriented Service***
- ***Reliable Service***

Figure 3.41: *Stream delivery*

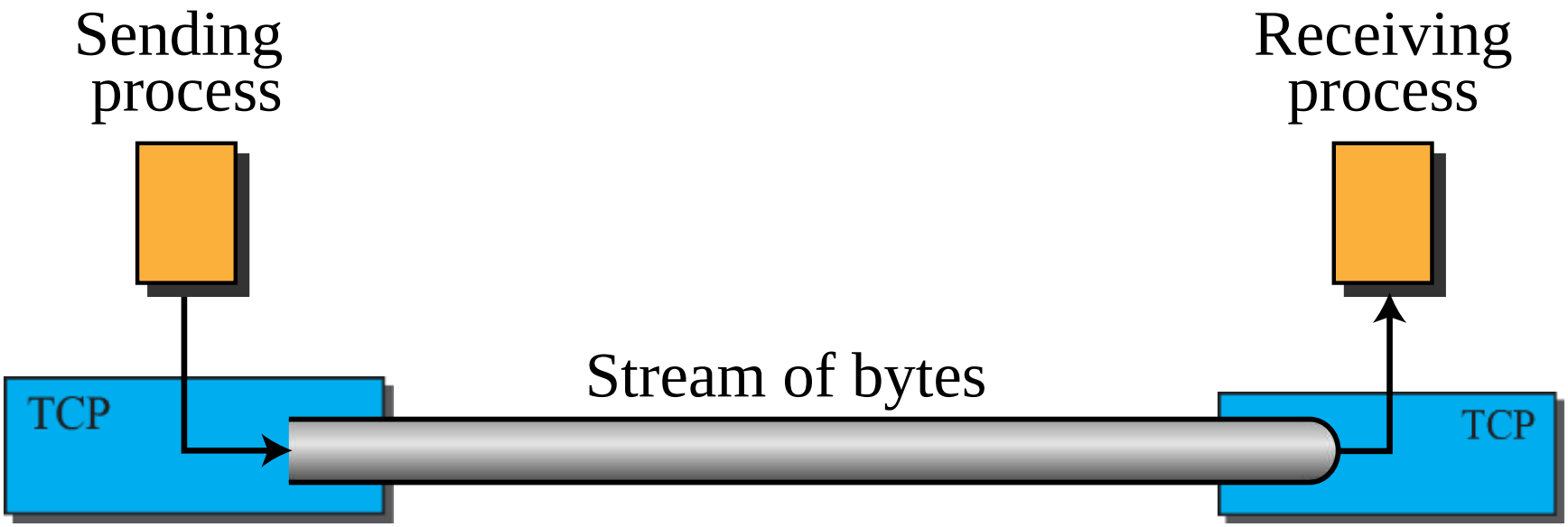


Figure 3.42: Sending and receiving buffers

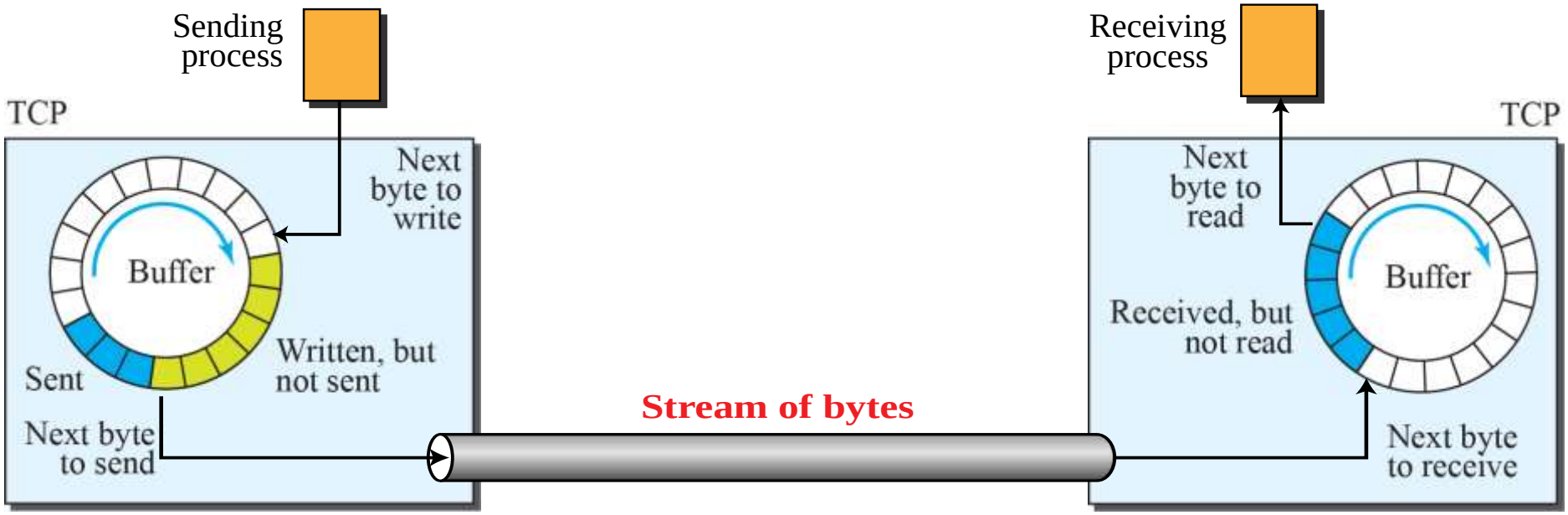
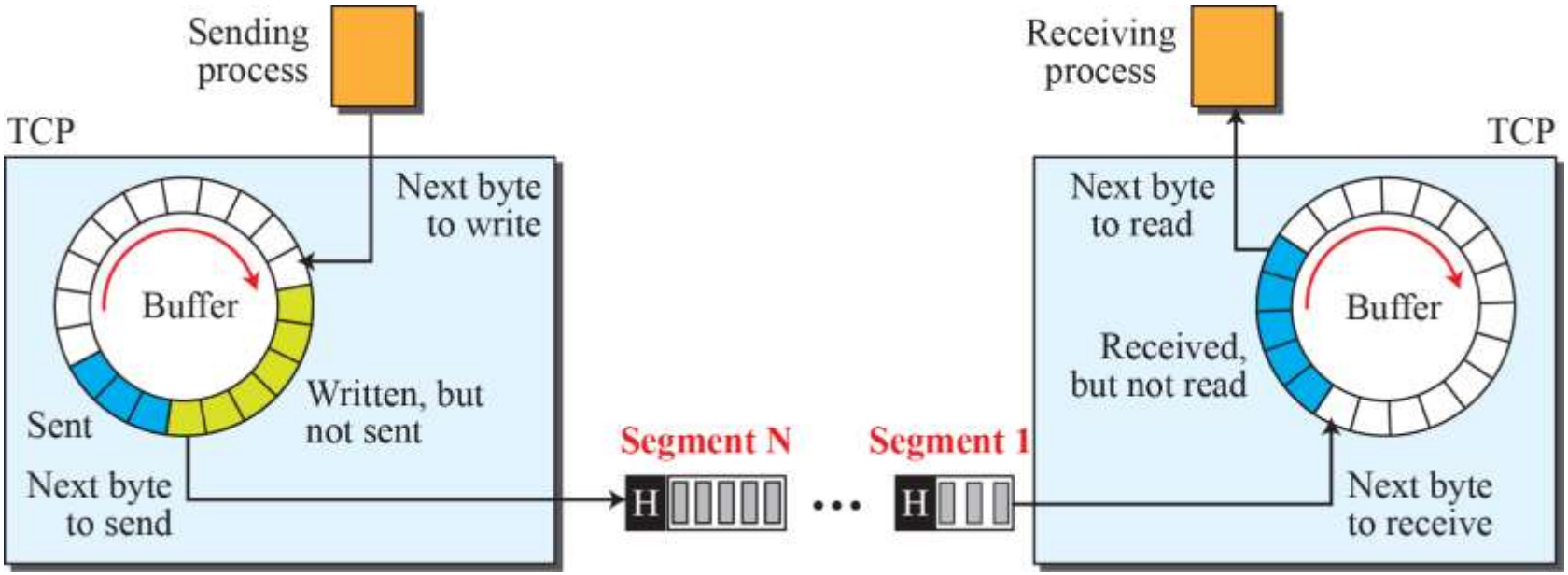


Figure 3.43: TCP segments



3.4.2 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.

- **Numbering System**
  - Byte Number
  - Sequence Number
  - Acknowledgment Number

Example 3.17

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

3.4.3 Segment

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

- ***Format***
- ***Encapsulation***

Figure 3.44: TCP segment format

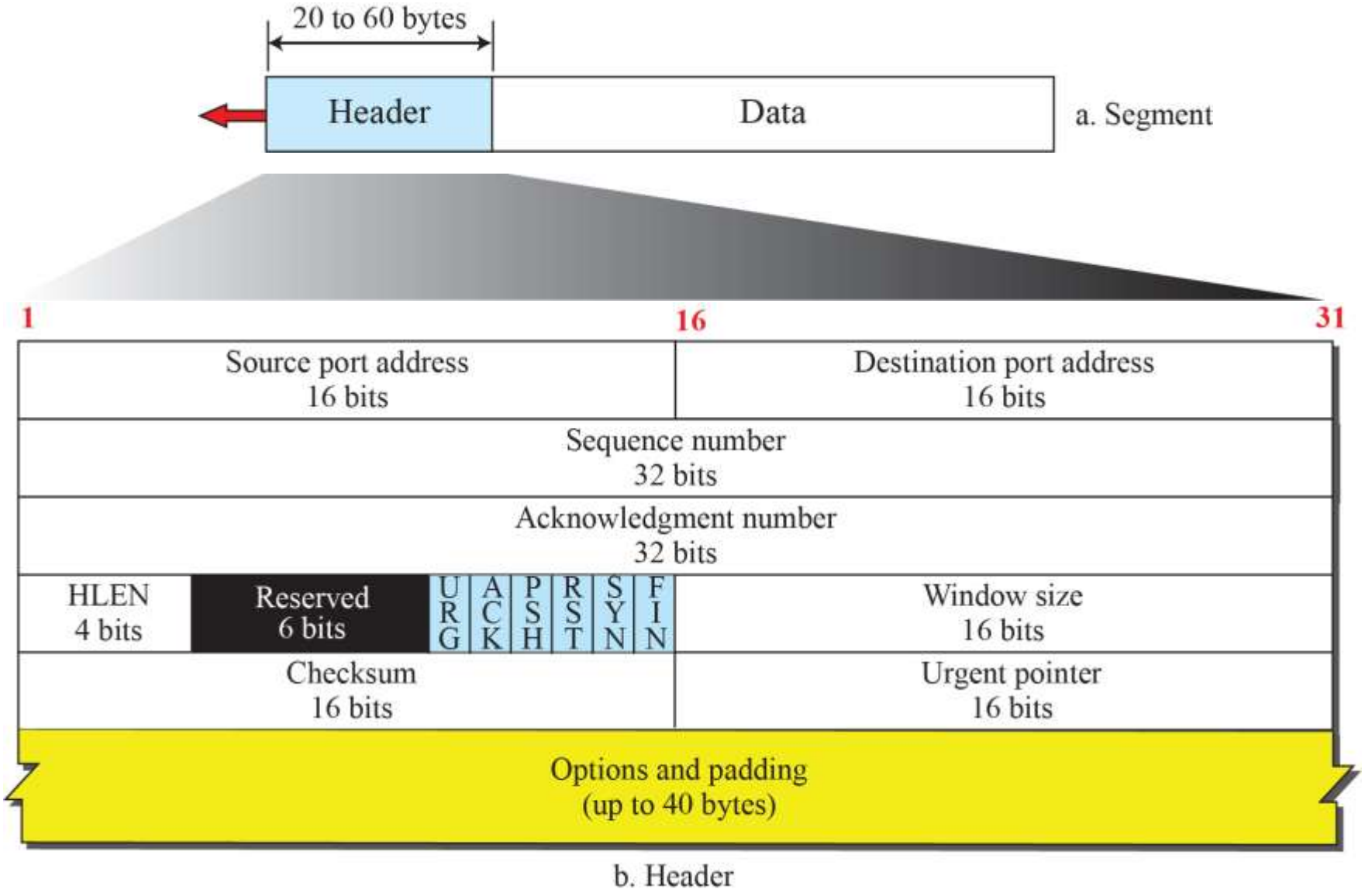
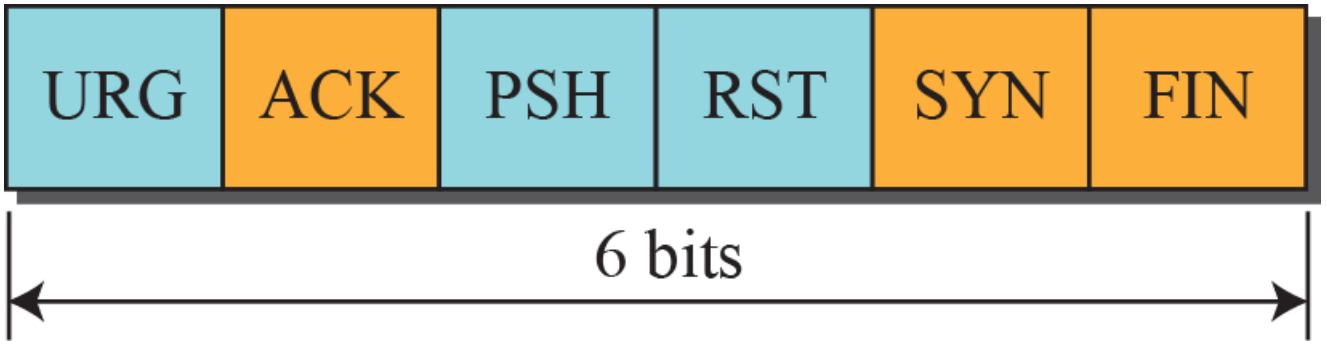
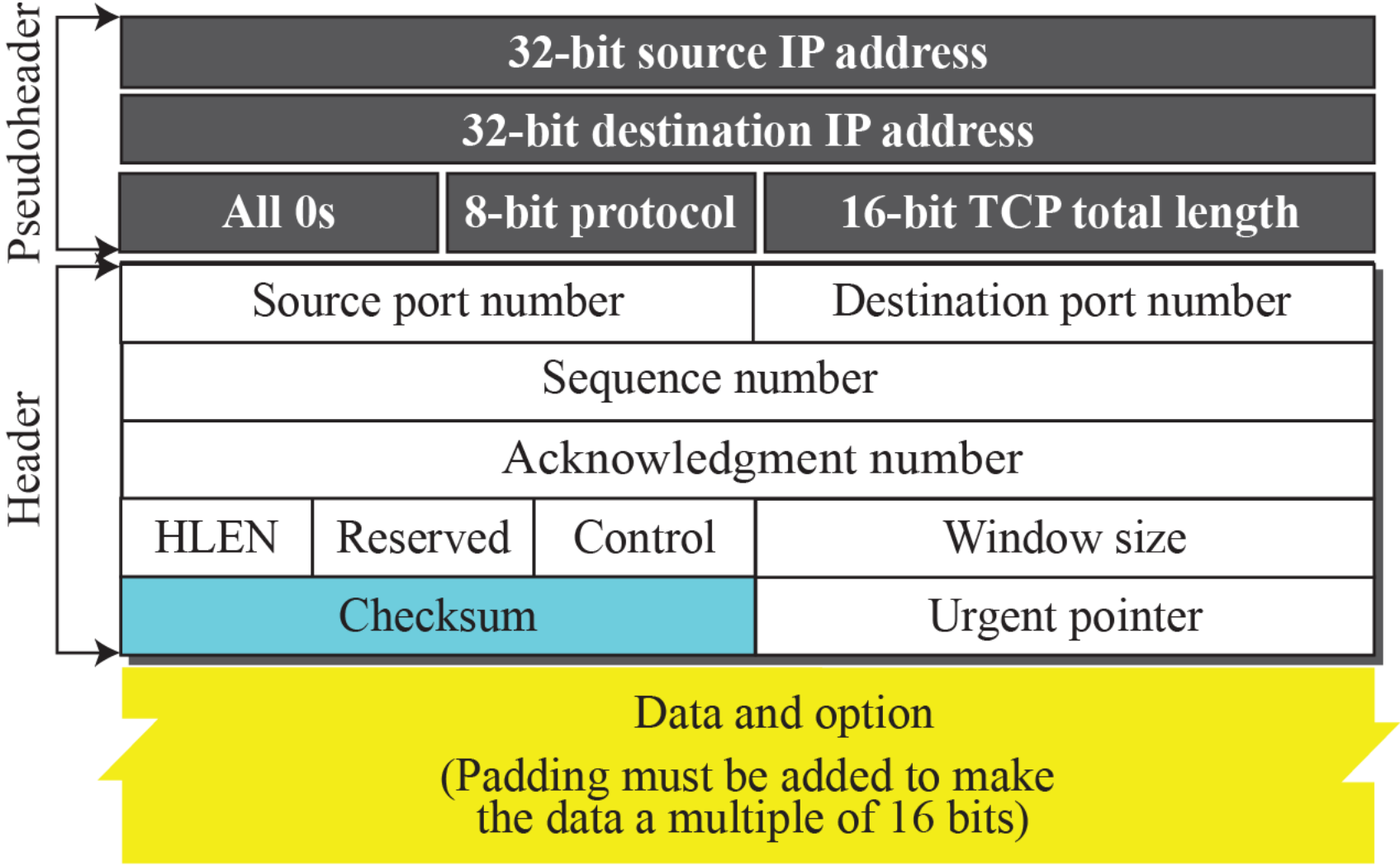


Figure 3.45: Control field



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

Figure 3.46: Pseudoheader added to the TCP datagram



3.4.4 A TCP Connection

TCP is connection-oriented. As discussed before, a connection-oriented transport protocol establishes a logical path between the source and destination. All of the segments belonging to a message are then sent over this logical path. Using a single logical pathway for the entire message facilitates the acknowledgment process as well as retransmission of damaged or lost frames.

3.4.4 (continued)

- **Connection Establishment**
  - Three-Way Handshaking
  - SYN Flooding Attack
- **Data Transfer**
  - Pushing Data
  - Urgent Data
- **Connection Termination**
  - Three-Way Handshaking
  - Half-Close
- **Connection Reset**

Figure 3.47: Connection establishment using three-way handshaking

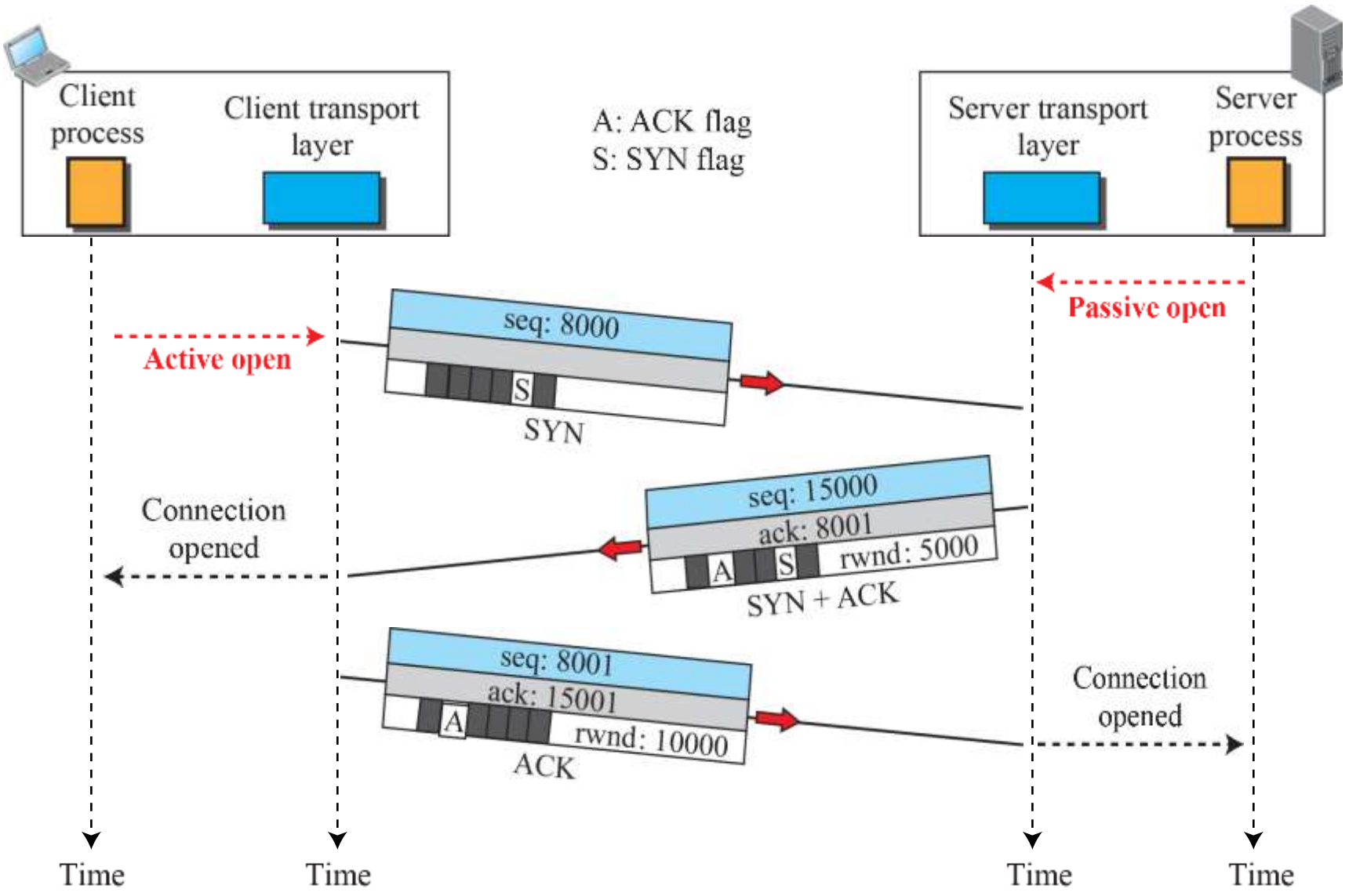


Figure 3.48: Data transfer

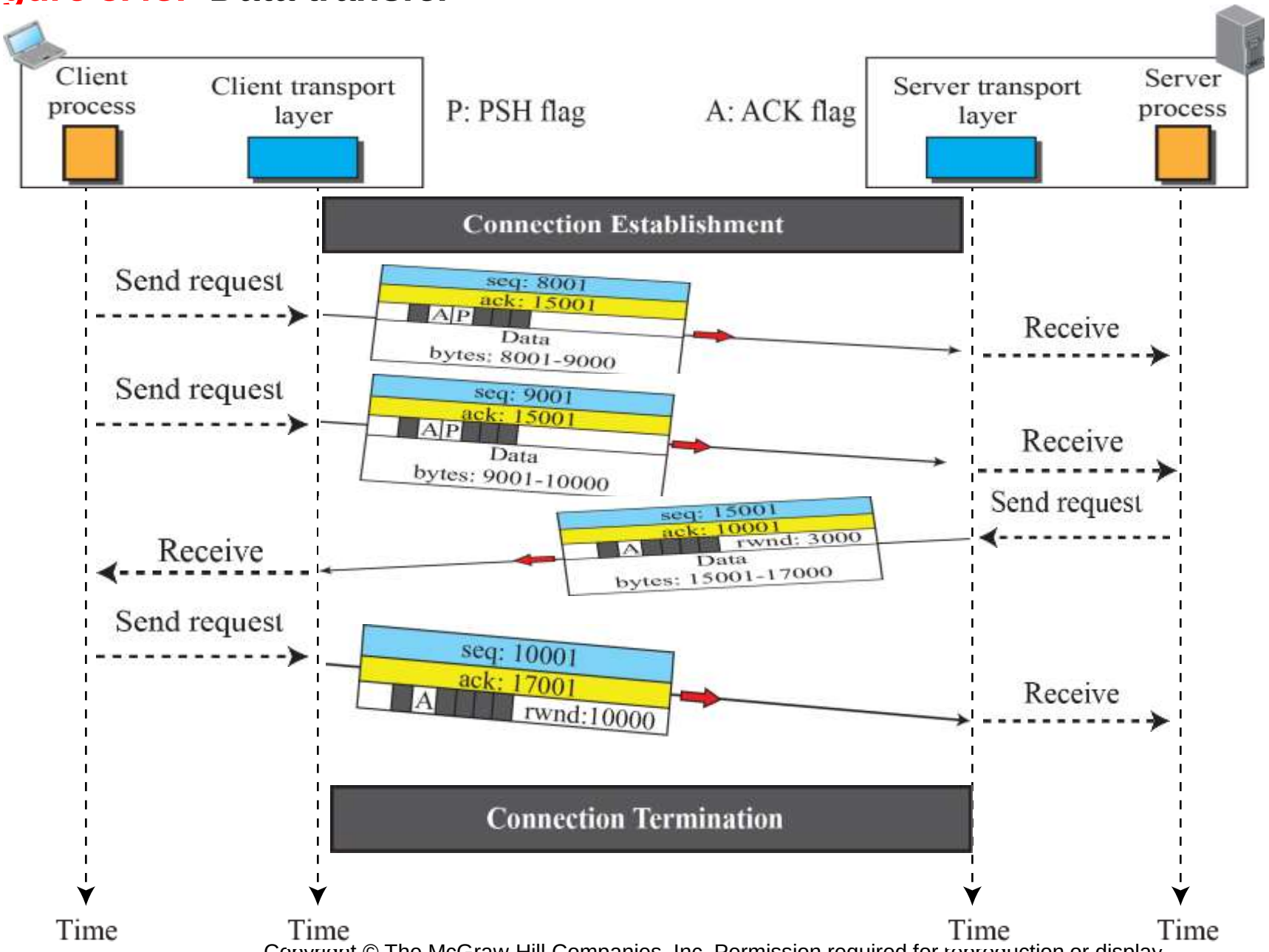


Figure 3.49: Connection termination using three-way handshaking

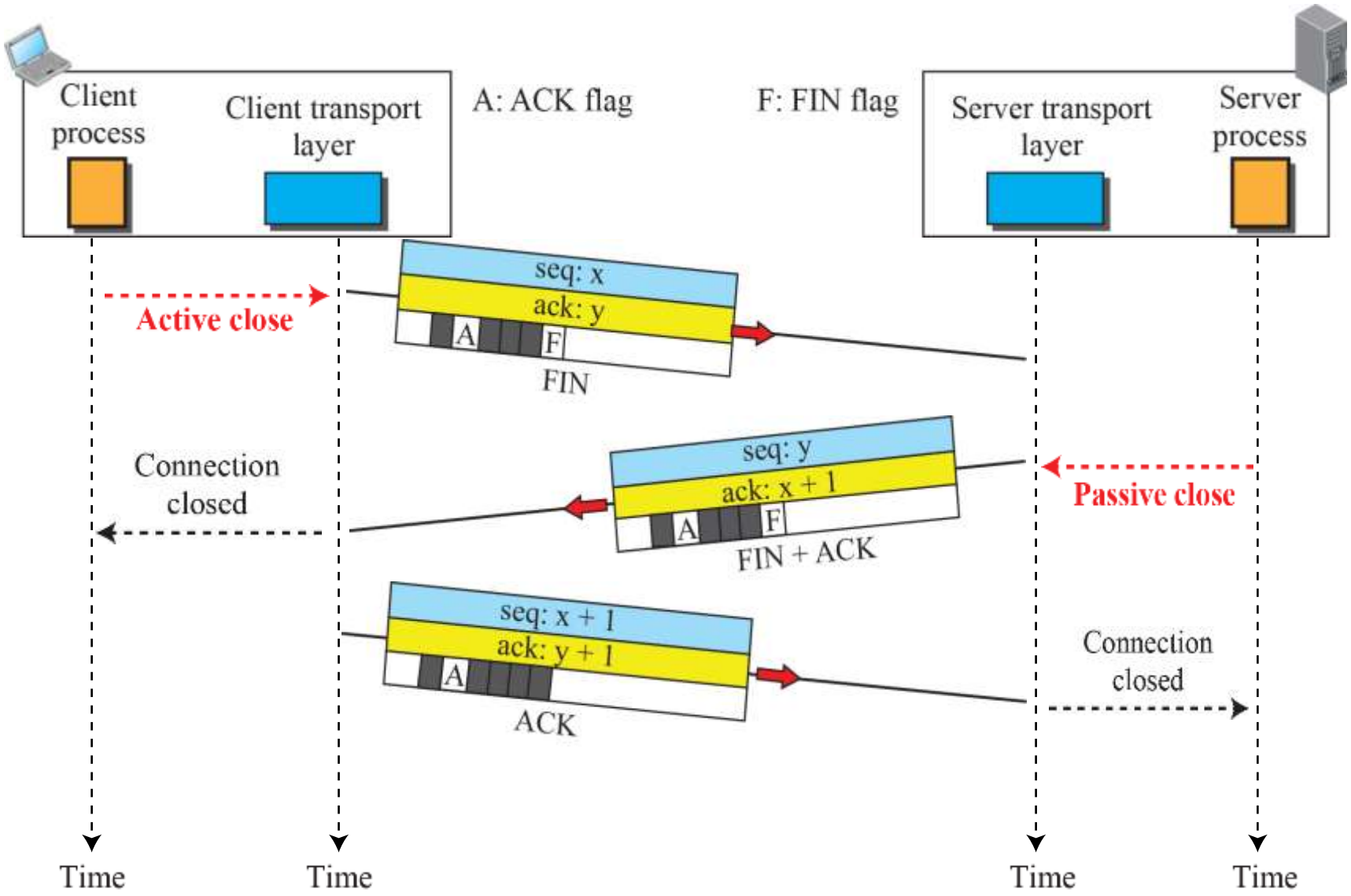
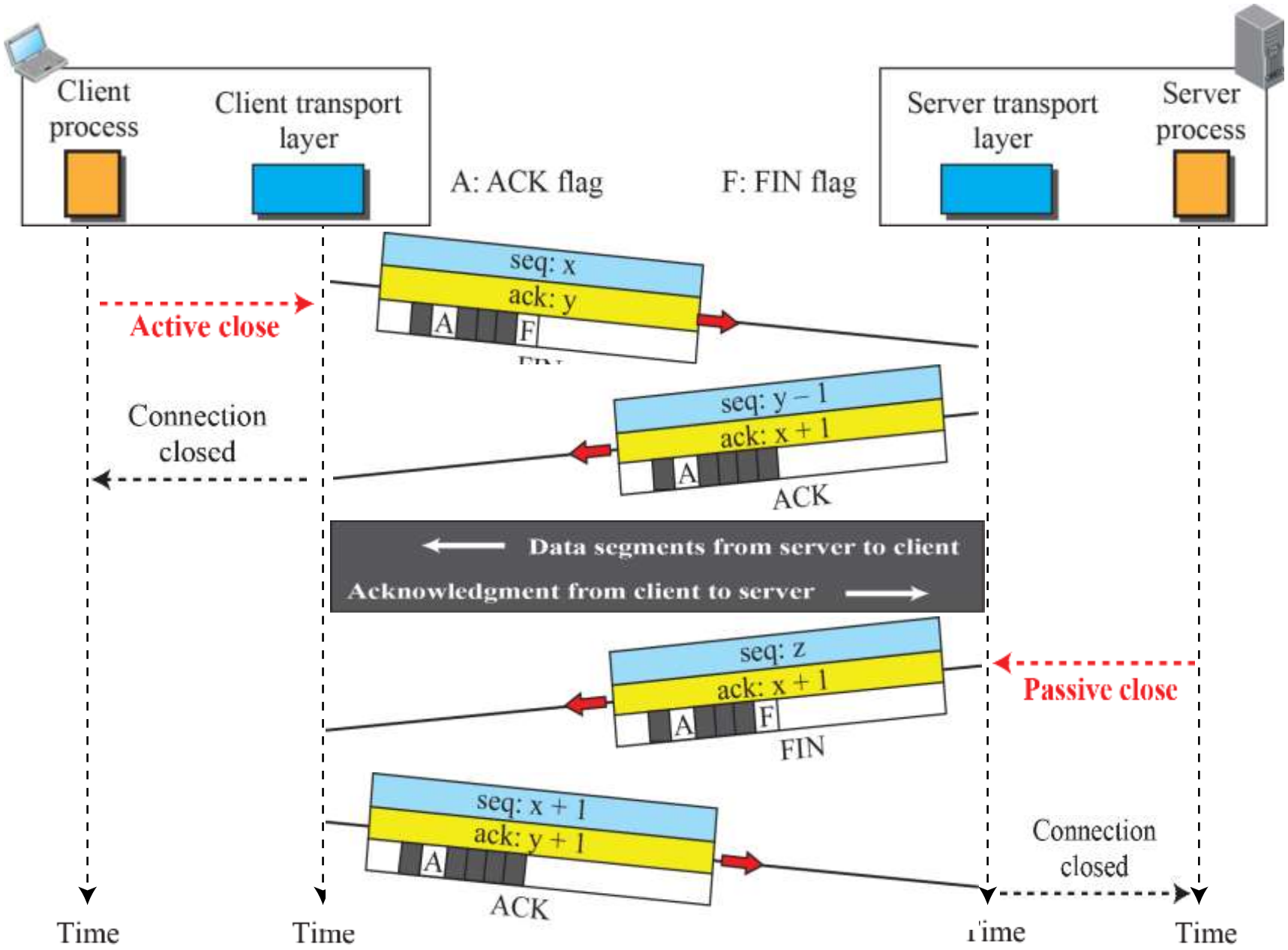


Figure 3.50: Half-close



3.4.5 State Transmission Diagram

To keep track of all the different events happening during connection establishment, connection termination, and data transfer, TCP is specified as the finite state machine (FSM) as shown in Figure 3.51.

- **Scenarios**
  - *A Half-Close Scenario*

Figure 3.51: State transition diagram

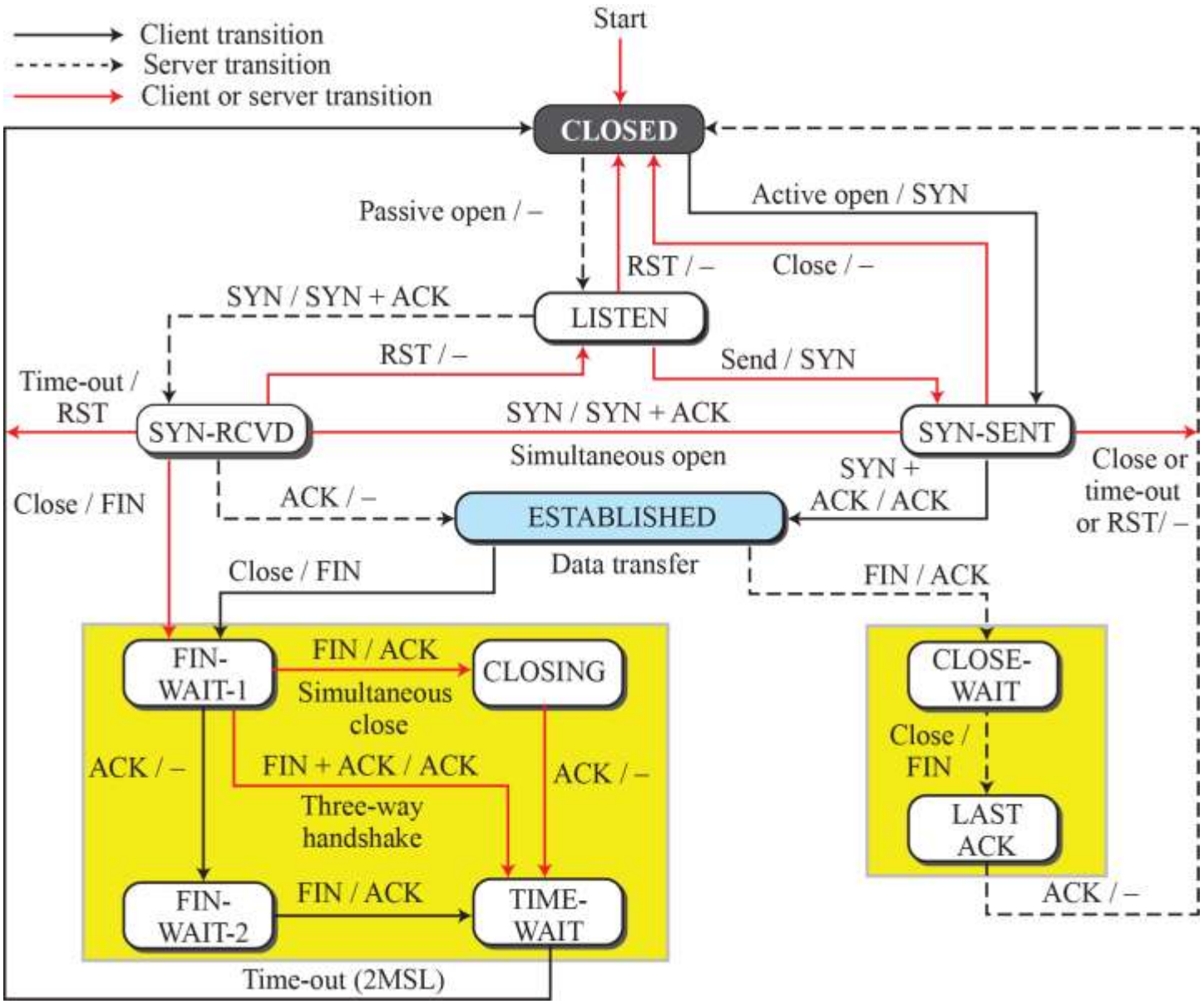


Table 3.2: States for TCP

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

Figure 3.52: Transition diagram with half-close connection termination

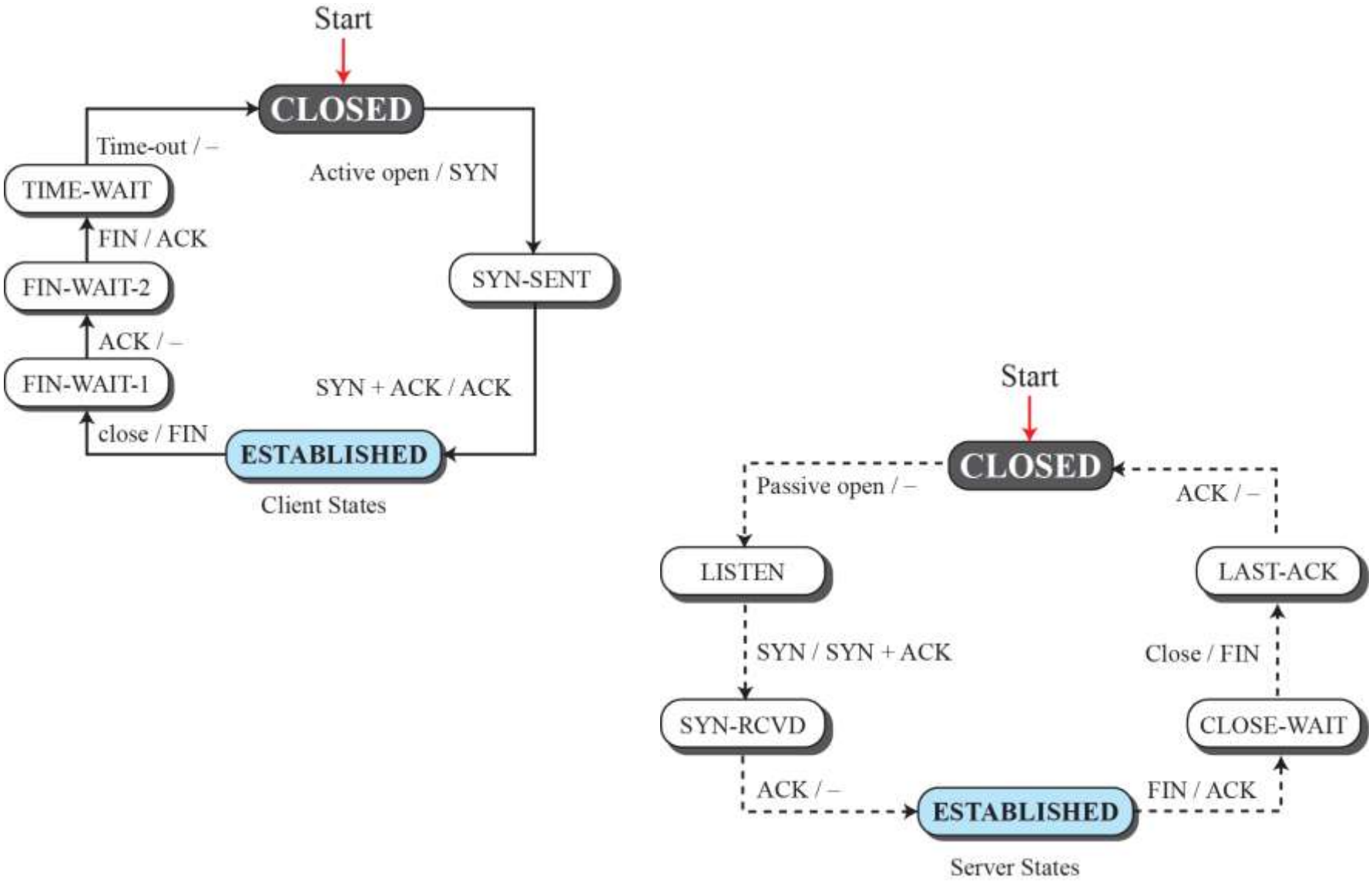
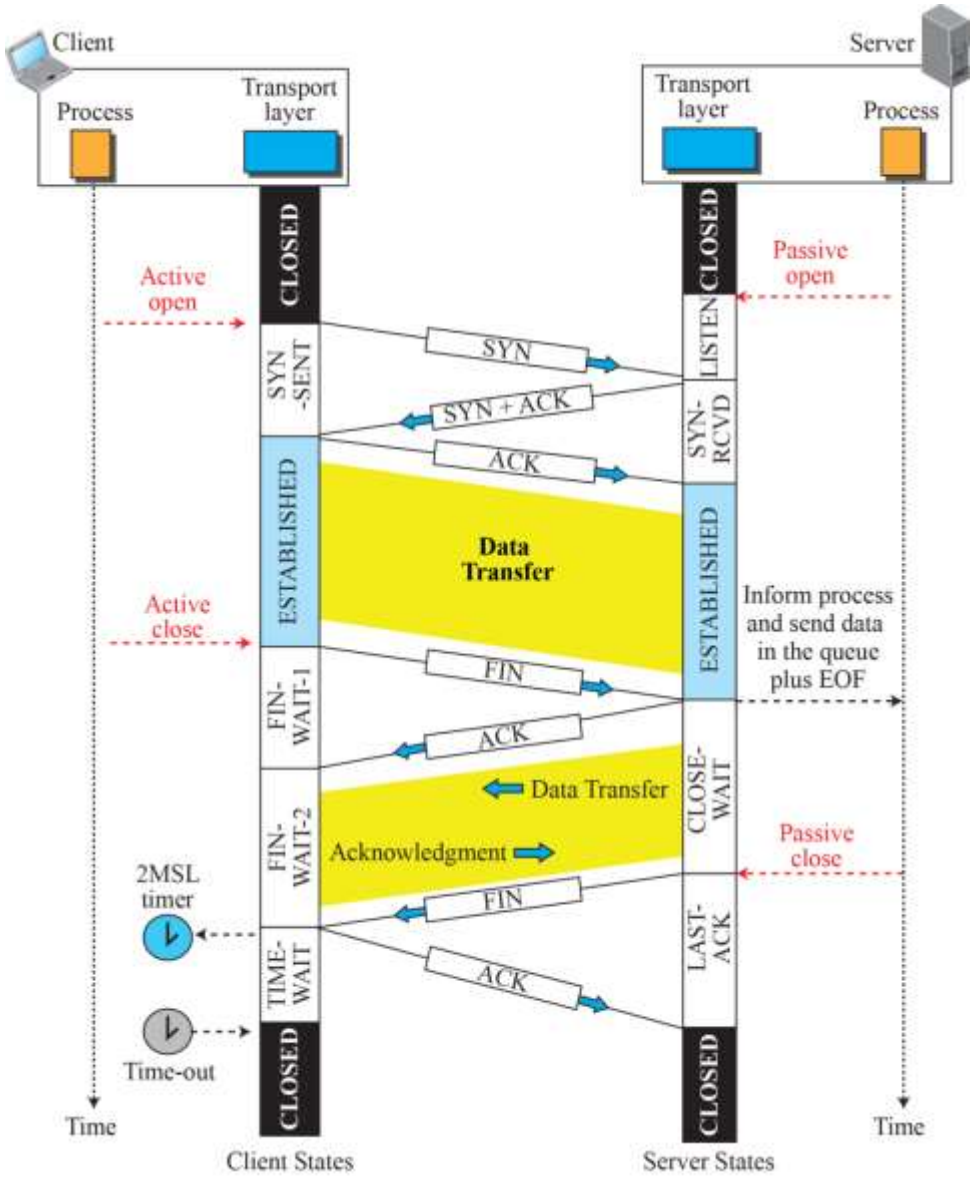


Figure 3.53: Time-line diagram for a common scenario



3.4.6 Windows 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 unrealistic unidirectional; the bidirectional communication can be inferred using two unidirectional communications with piggybacking.

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

Figure 3.54: Send window in TCP

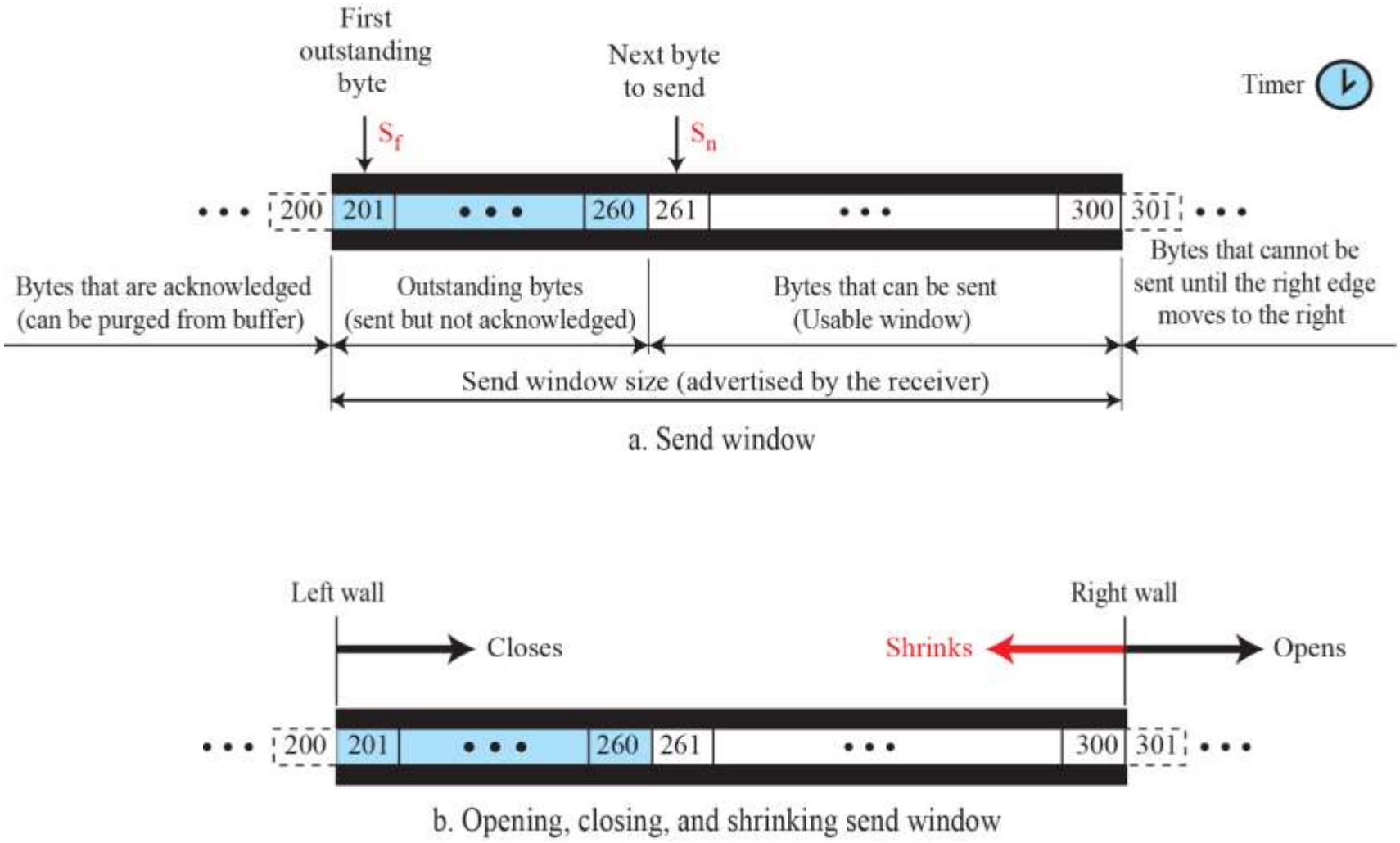
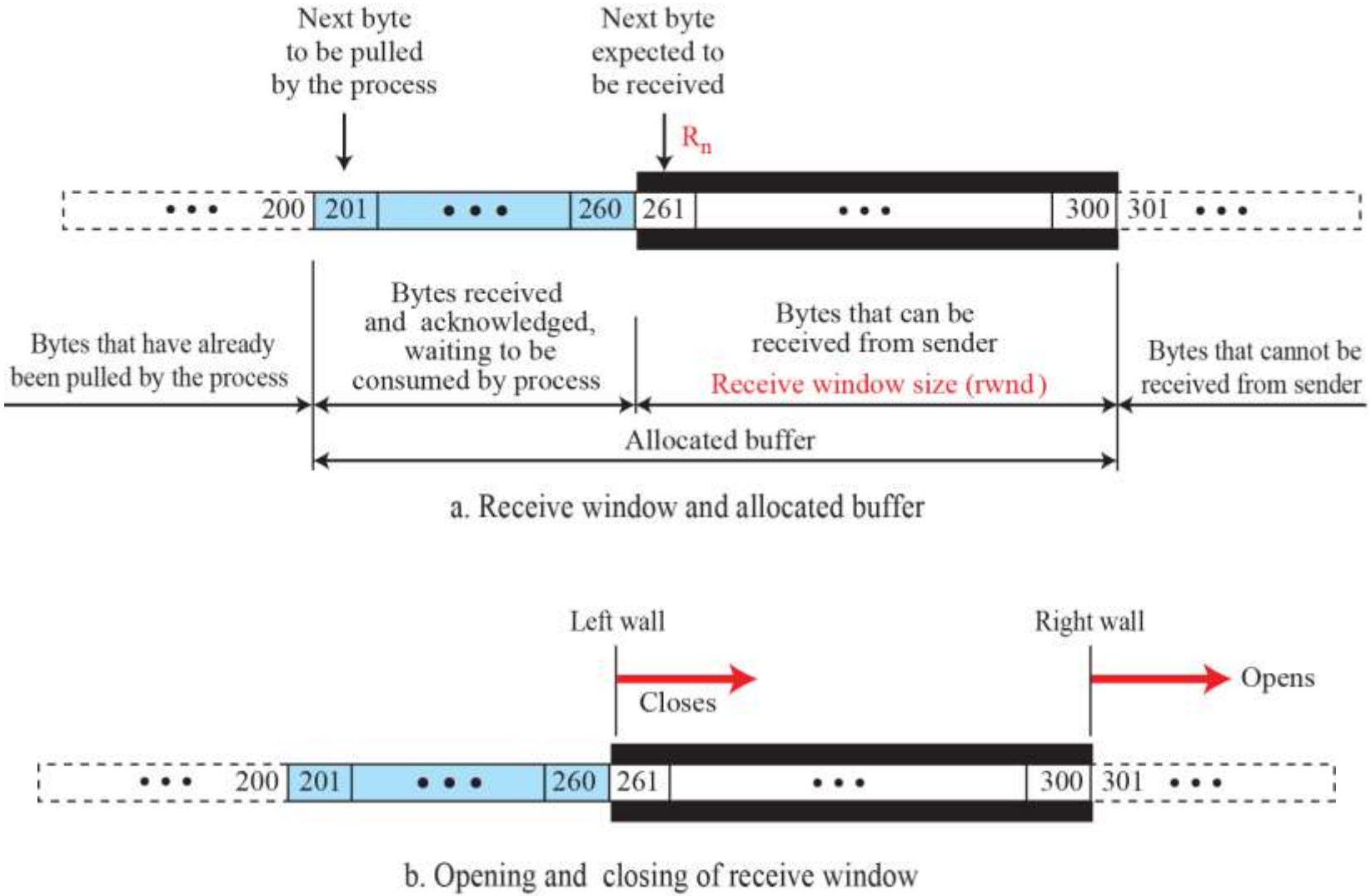


Figure 3.55: Receive window in TCP



3.4.7 Flow Control

As discussed before, 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 assume that the logical channel between the sending and receiving TCP is error-free.

3.4.7 (continued)

- **Opening and Closing Windows**
  - A Scenario
- **Shrinking of Windows**
  - Window Shutdown
- **Silly Window Syndrome**
  - Syndrome Created by the Sender
  - Syndrome Created by the Receiver

Figure 3.56: Data flow and flow control feedbacks in TCP

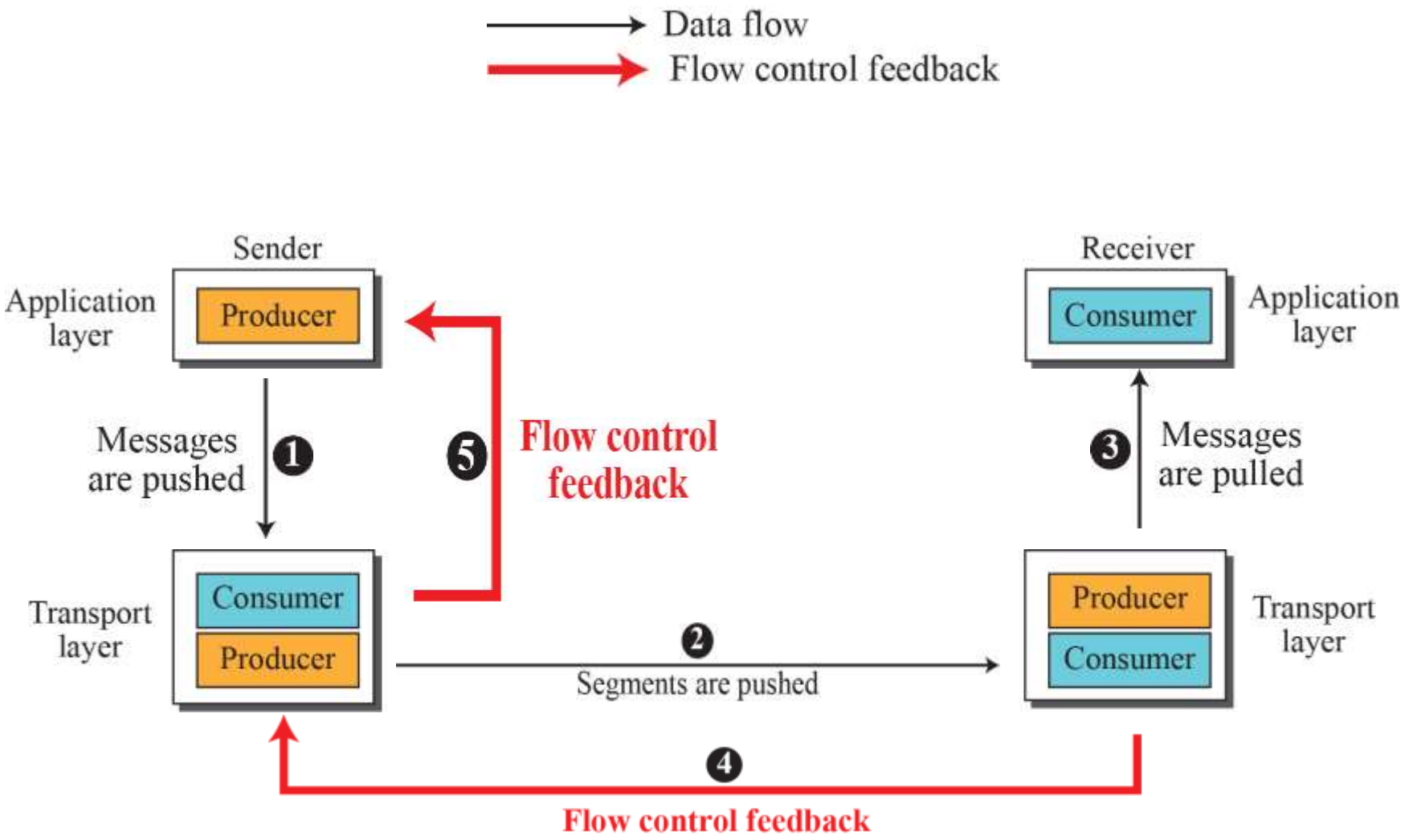
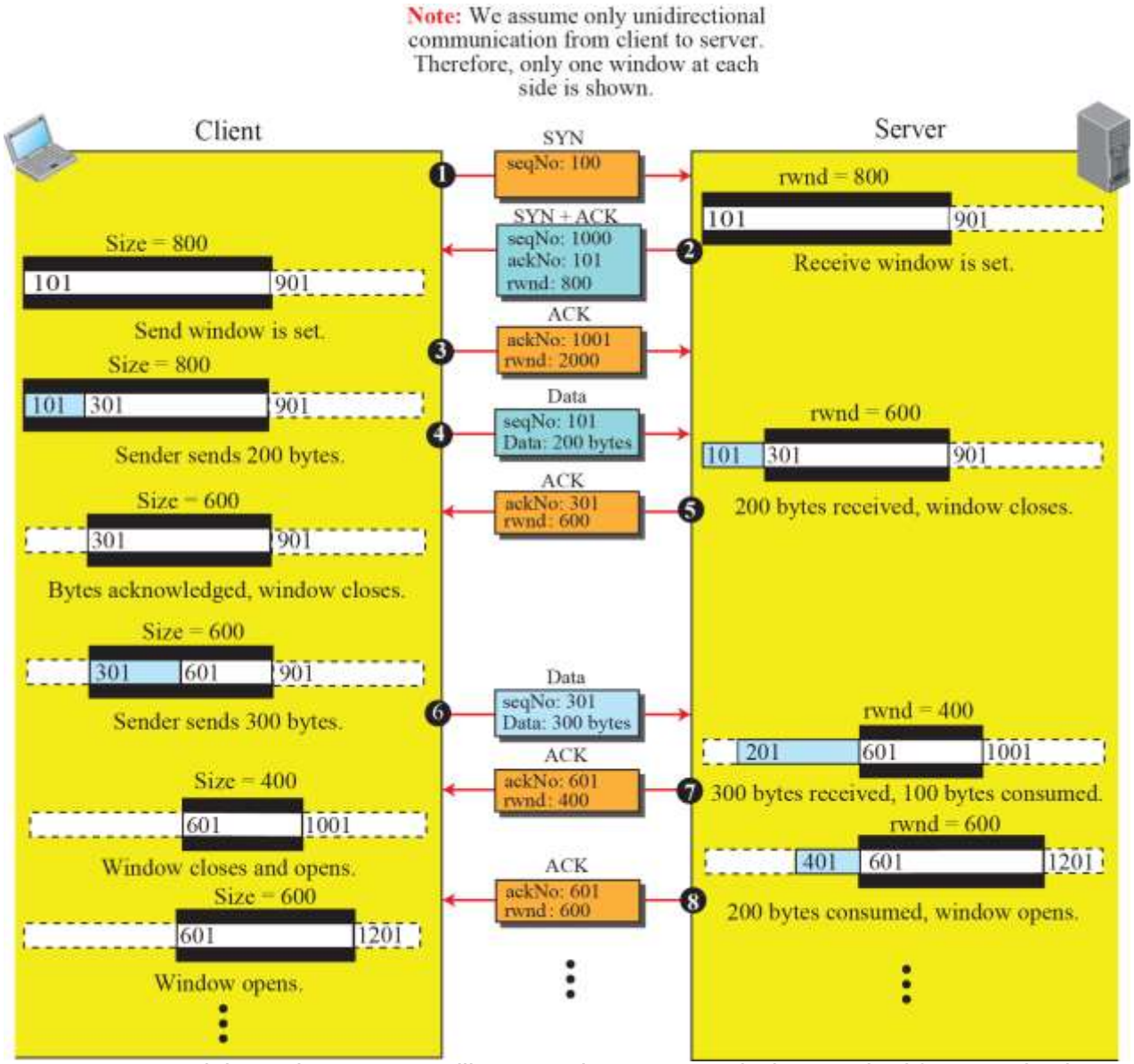


Figure 3.57: An example of flow control

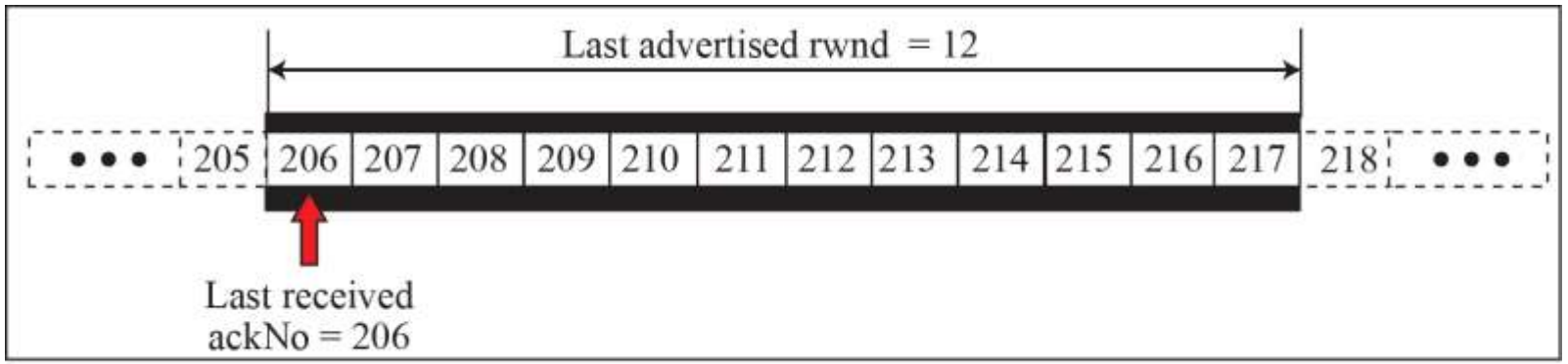


Example 3.18

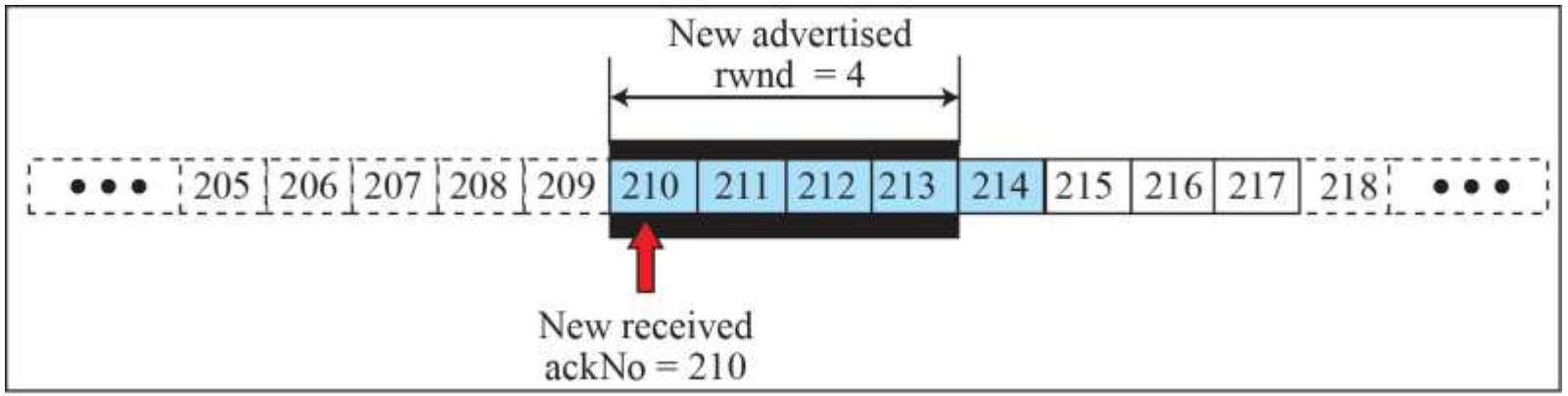
Figure 3.58 shows the reason for this mandate.

Part a of the figure shows the values of the 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. 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 already been 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. described above.

Figure 3.58: Example 3.18



a. The window after the last advertisement



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

3.4.8 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 order, without error, and without any part lost or duplicated.

3.4.8 (continued)

- Checksum
- Acknowledgment
  - Cumulative Acknowledgment (ACK)
  - Selective Acknowledgment (SACK)
- Generating Acknowledgments
- Retransmission
  - Retransmission after RTO
  - Retransmission after Three Duplicate ACK
- Out-of-Order Segments

3.4.8 (continued)

- **FSMs for Data Transfer in TCP**
  - Sender-Side FSM
  - Receiver-Side FSM
- **Some Scenarios**
  - Normal Operation
  - Lost Segment
  - Fast Retransmission
  - Delayed Segment
  - Duplicate Segment
  - Automatically Corrected Lost ACK
  - Correction by Resending a Segment
  - Deadlock Created by Lost Acknowledgment

Figure 3.59: Simplified FSM for the TCP sender side

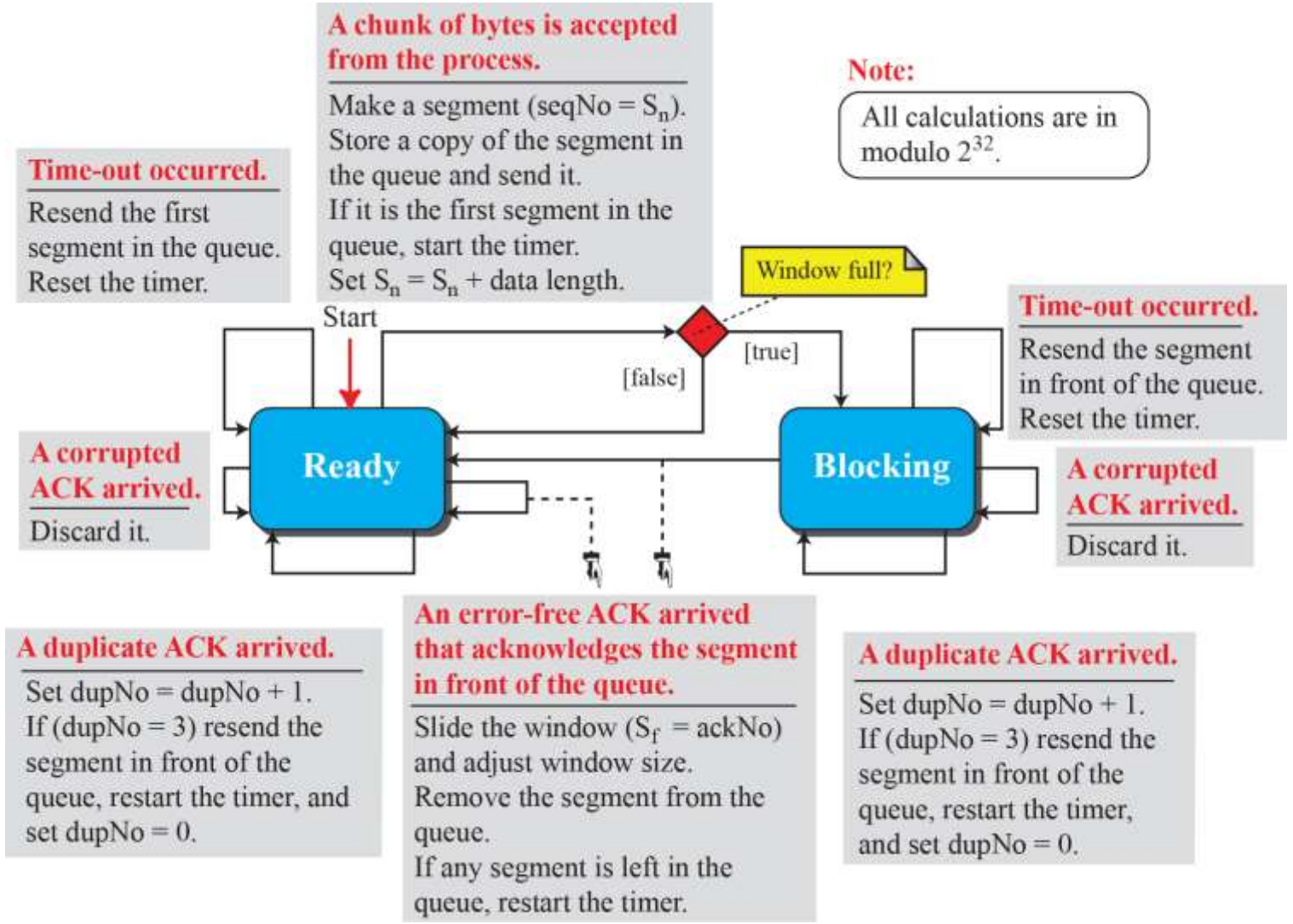


Figure 3.60: Simplified FSM for the TCP receiver side

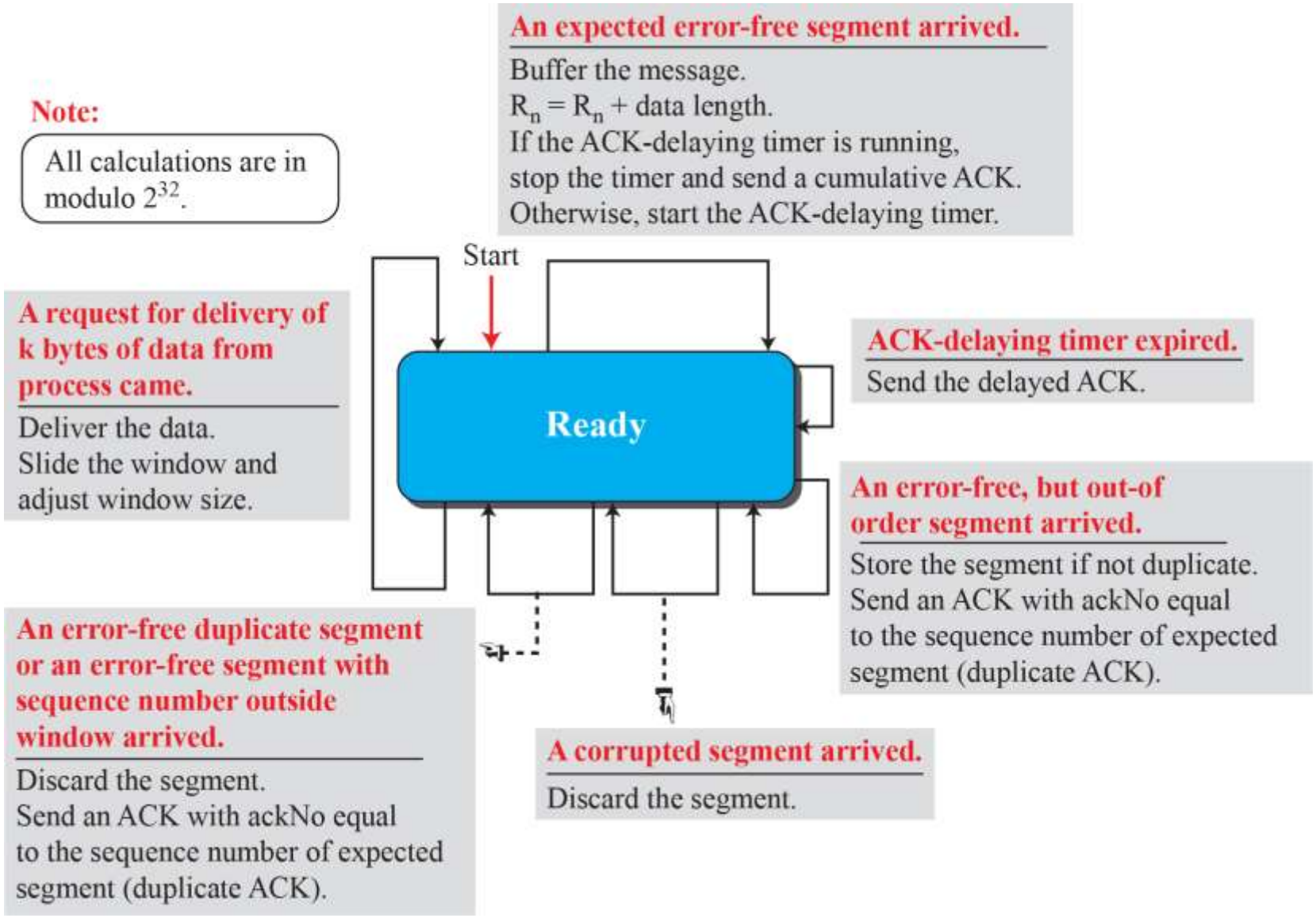


Figure 3.61: Normal operation

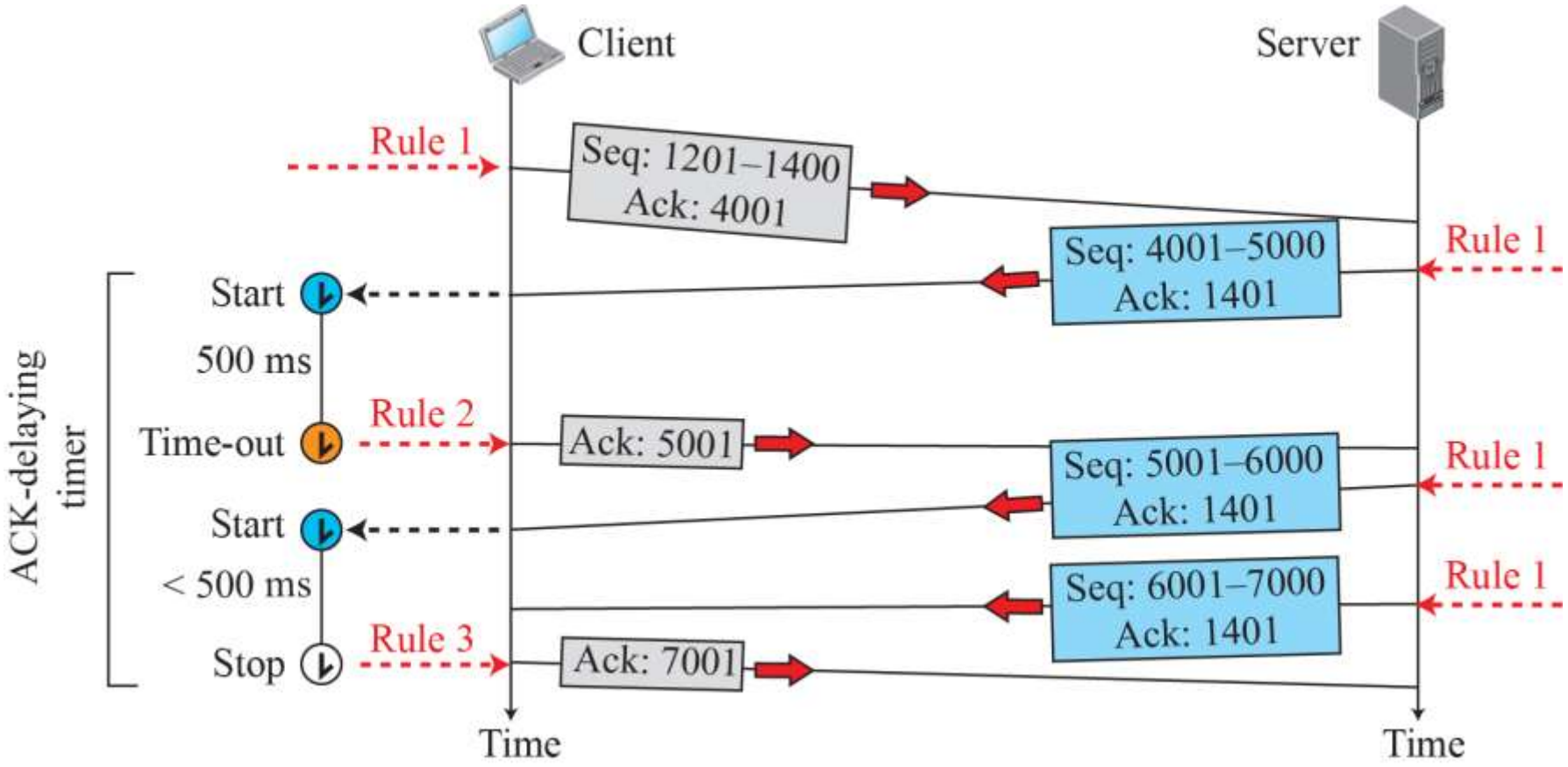


Figure 3.62: Lost segment

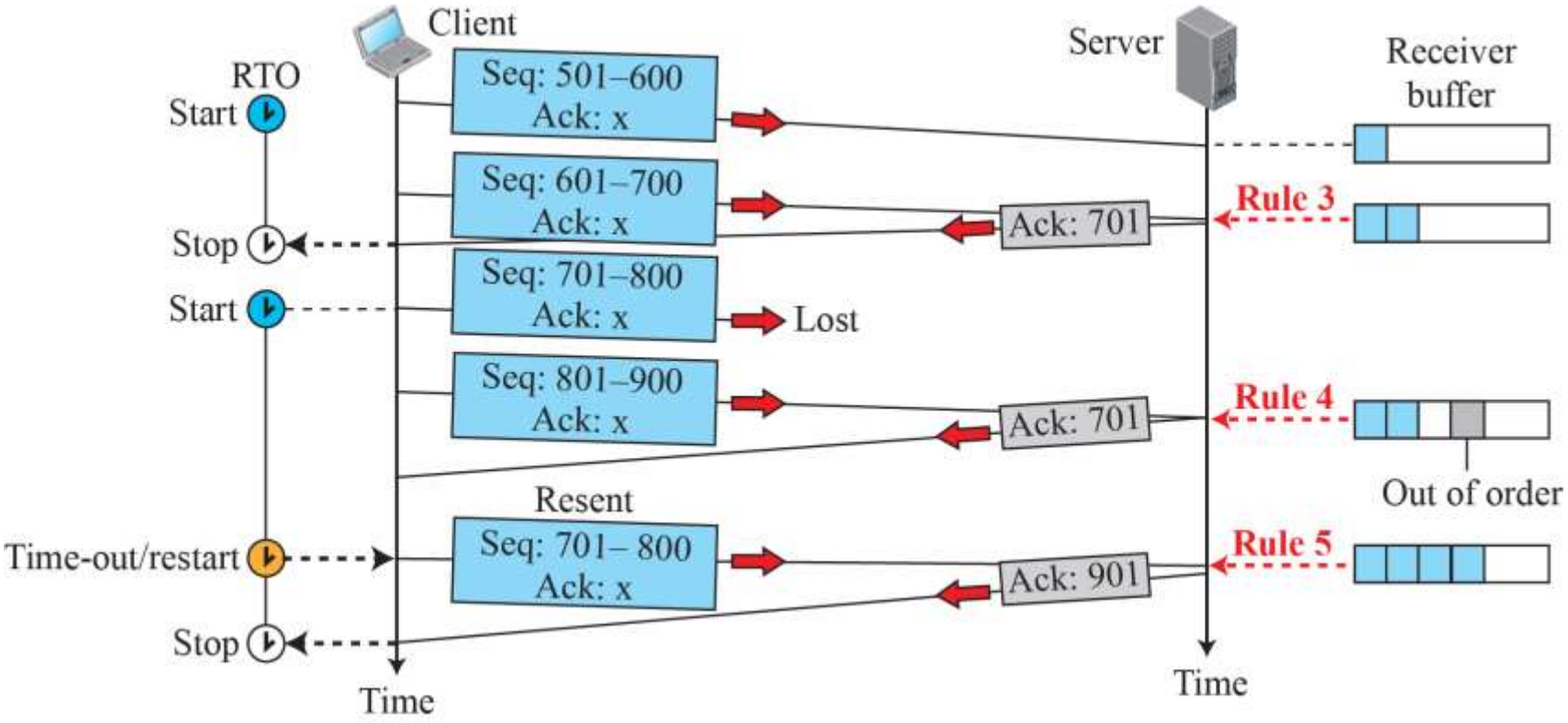


Figure 3.63: *Fast retransmission*

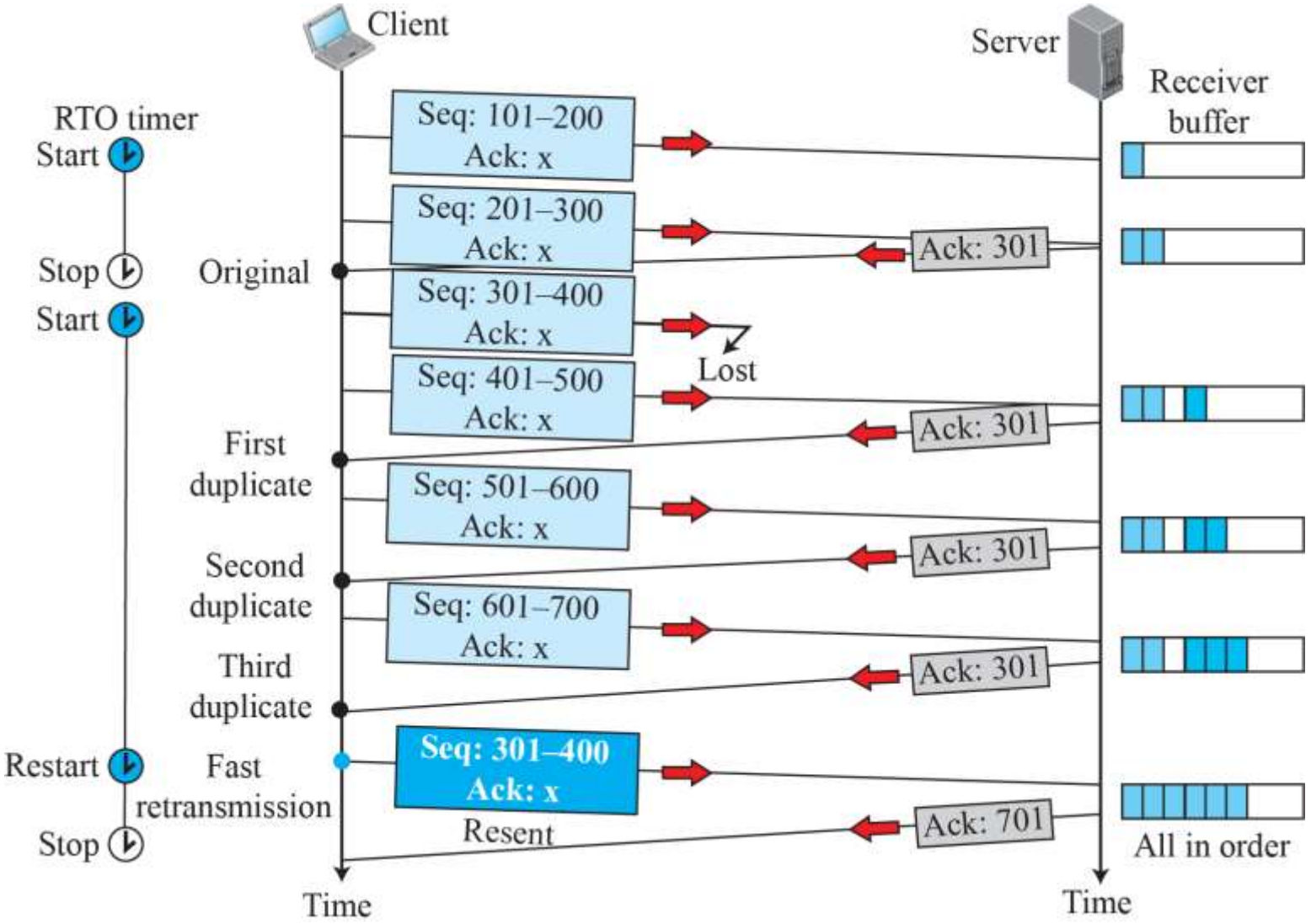


Figure 3.64: Lost acknowledgment

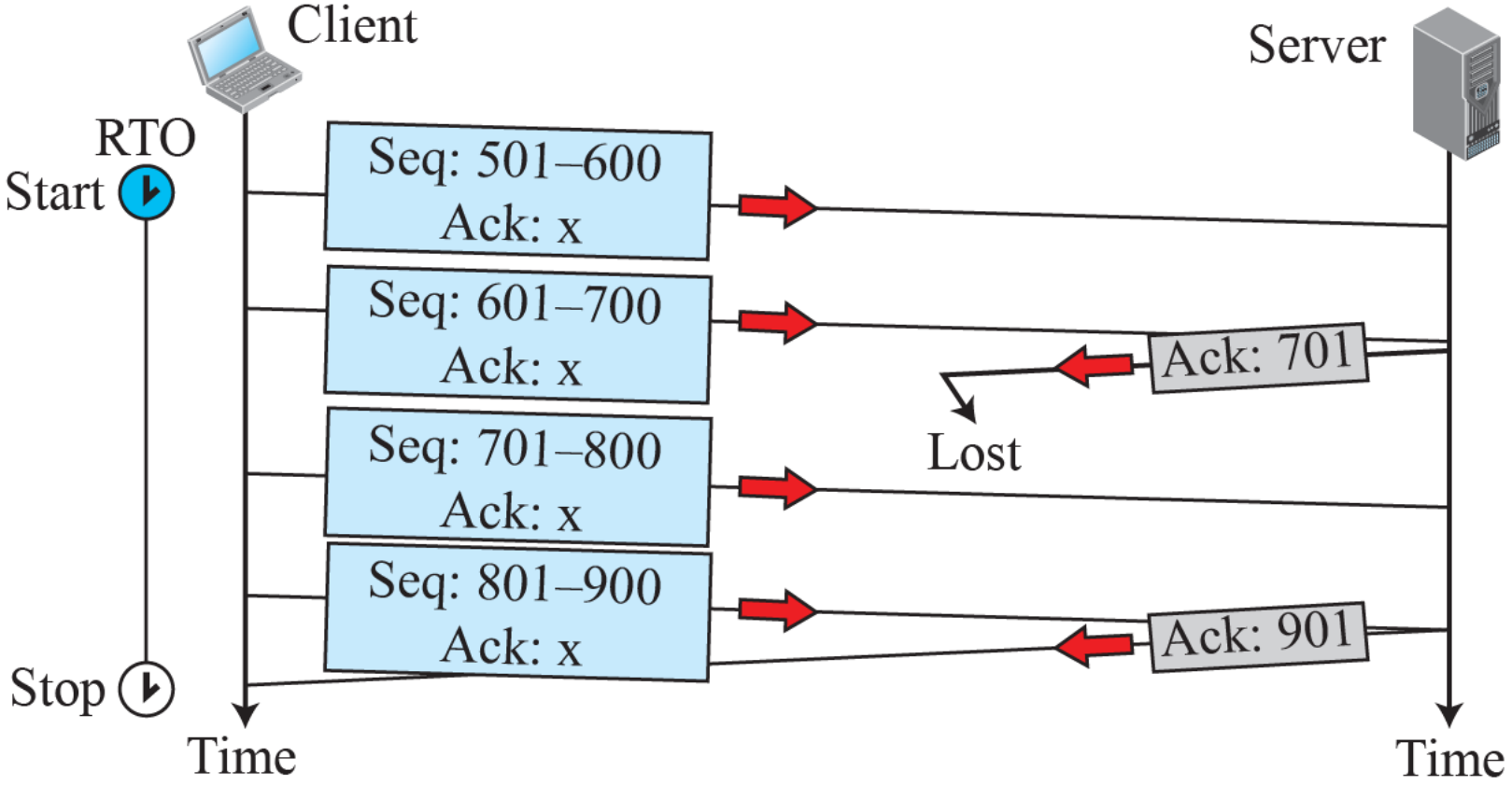
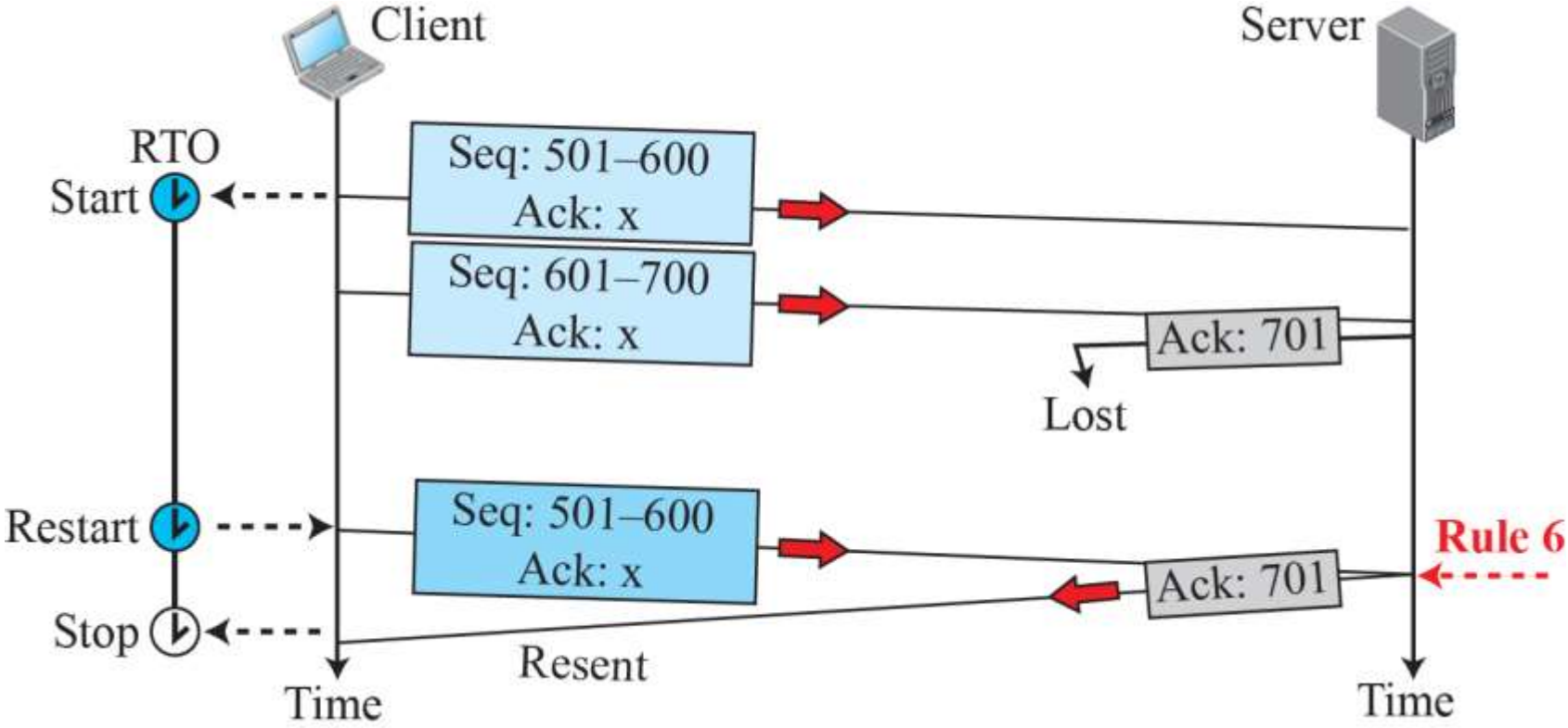


Figure 3.65: *Lost acknowledgment corrected by resending a segment*



3.4.9 TCP Congestion Control

TCP uses different policies to handle the congestion in the network. We describe these policies in this section.

- ***Congestion Window***
- ***Congestion Detection***
- ***Congestion Policies***
  - ***Slow Start: Exponential Increase***
  - ***Congestion Avoidance: Additive Increase***

3.4.9 (continued)

- **Policy Transition**
  - Taho TCP
  - Reno TCP
  - NewReno TCP
- **Additive Increase, Multiplicative Decrease**
- **TCP Throughput**

Figure 3.66: Slow start, exponential increase

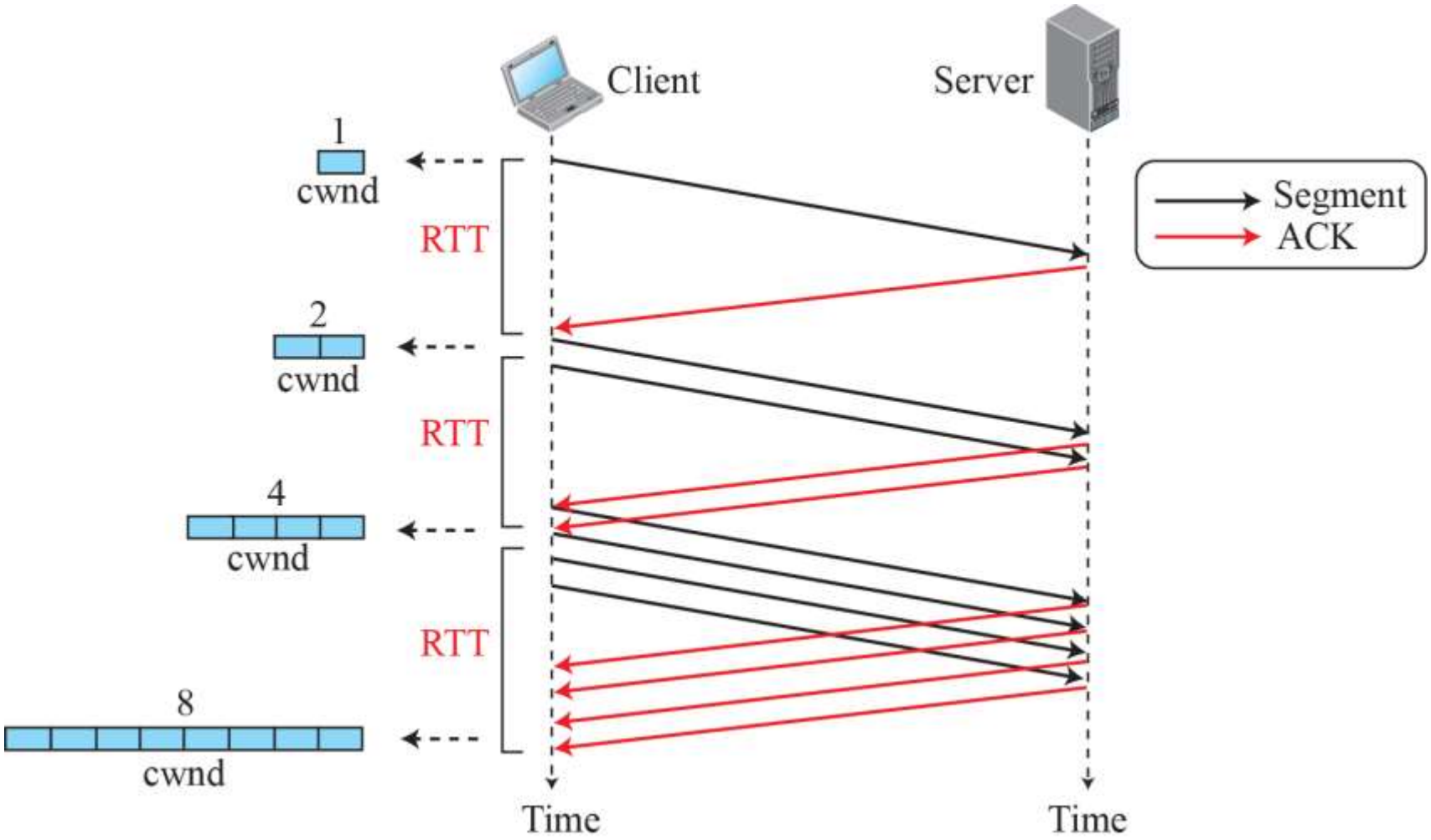


Figure 3.67: Congestion avoidance, additive increase

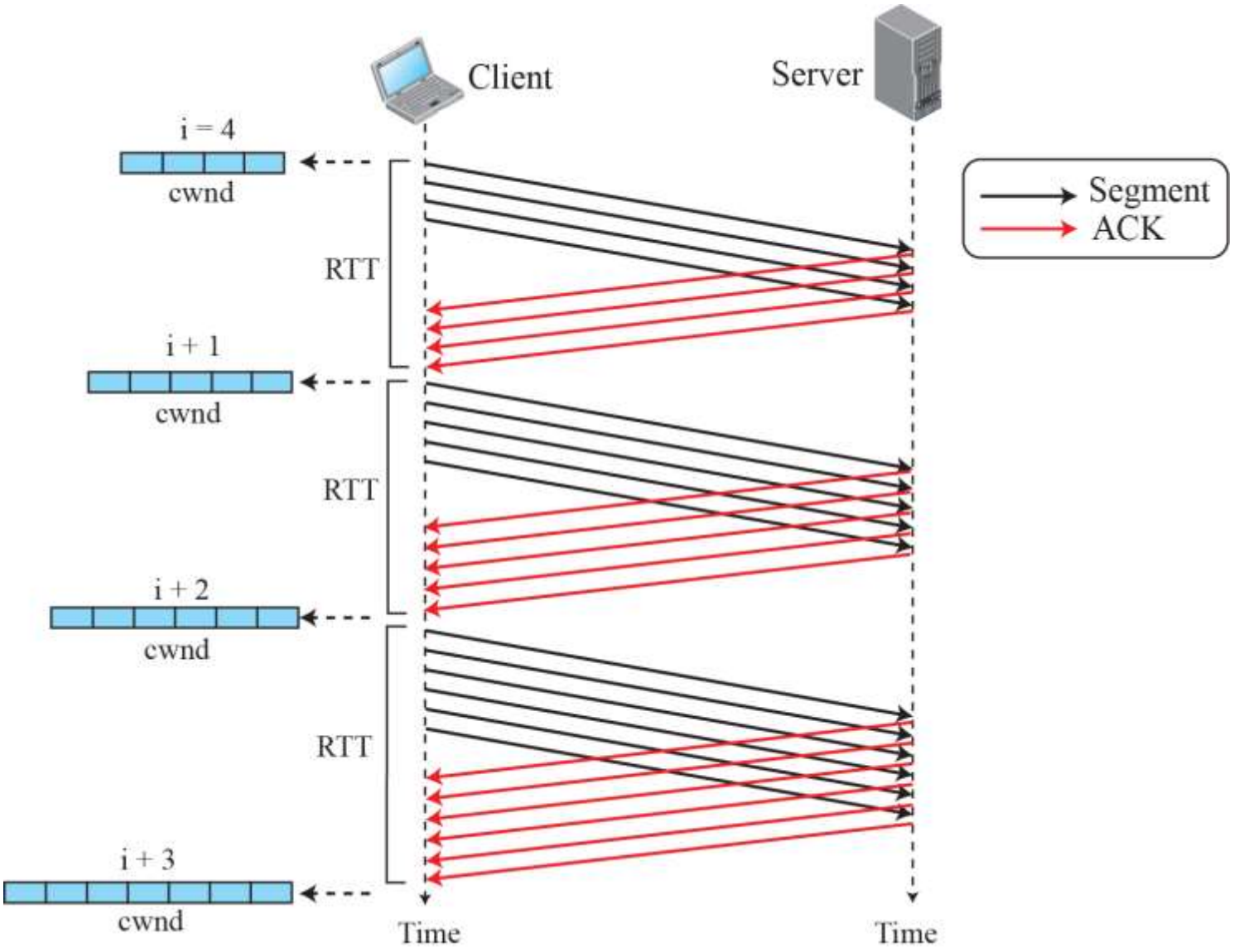
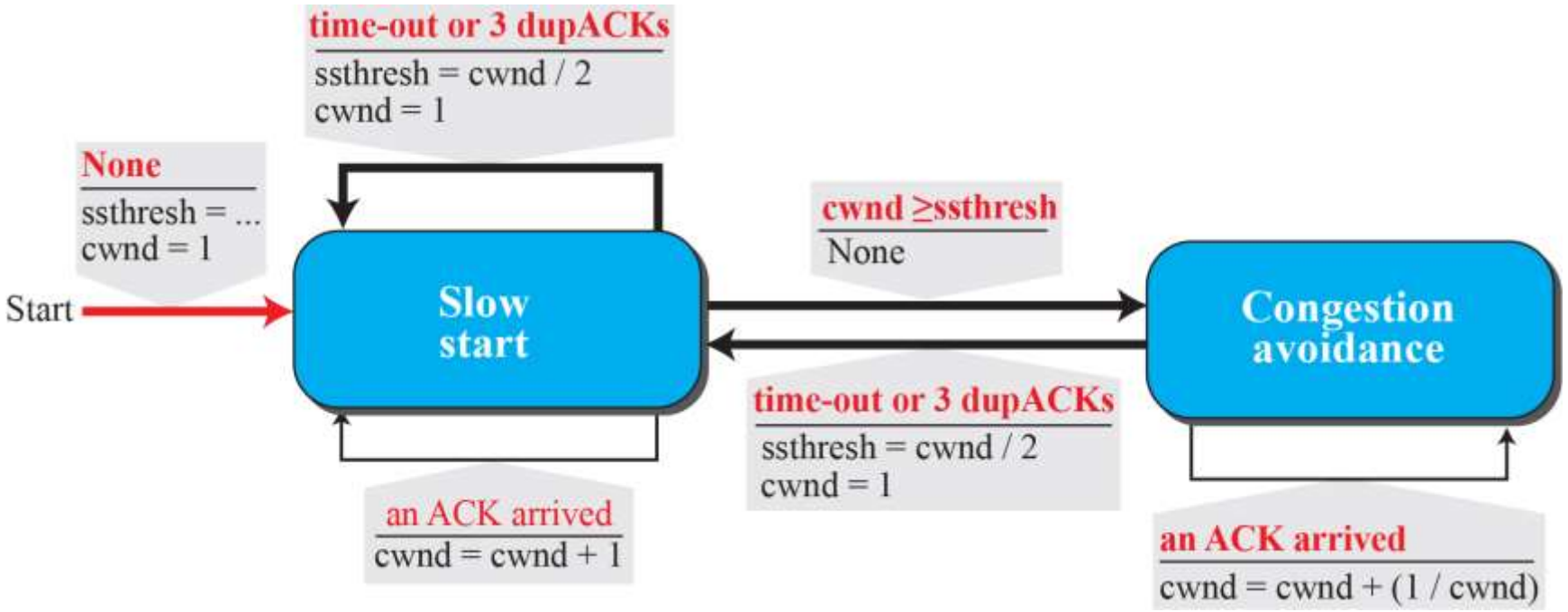


Figure 3.68: FSM for Taho TCP



Example 3.19

Figure 3.69 shows an example of congestion control in a Taho TCP. TCP starts data transfer and sets the *ssthresh* variable to an ambitious value of 16 MSS. TCP begins at the slow-start (SS) state with the *cwnd* = 1. The congestion window grows exponentially, but a time-out occurs after the third RTT (before reaching the threshold). TCP assumes that there is congestion in the network. It immediately sets the new *ssthresh* = 4 MSS (half of the current cwnd, which is 8) and begins a new slow start (SA) state with *cwnd* = 1 MSS. The congestion grows exponentially until it reaches the newly set threshold. TCP now moves to the congestion avoidance (CA) state and the congestion window grows additively until it reaches *cwnd* = 12 MSS.

Example 3.19 (continued)

At this moment, three duplicate ACKs arrive, another indication of the congestion in the network. TCP again halves the value of ssthresh to 6 MSS and begins a new slow-start (SS) state. The exponential growth of the cwnd continues. After RTT 15, the size of cwnd is 4 MSS. After sending four segments and receiving only two ACKs, the size of the window reaches the *ssthresh* (6) and the TCP moves to the congestion avoidance state. The data transfer now continues in the congestion avoidance (CA) state until the connection is terminated after RTT 20.

Figure 3.69: Example of Taho TCP

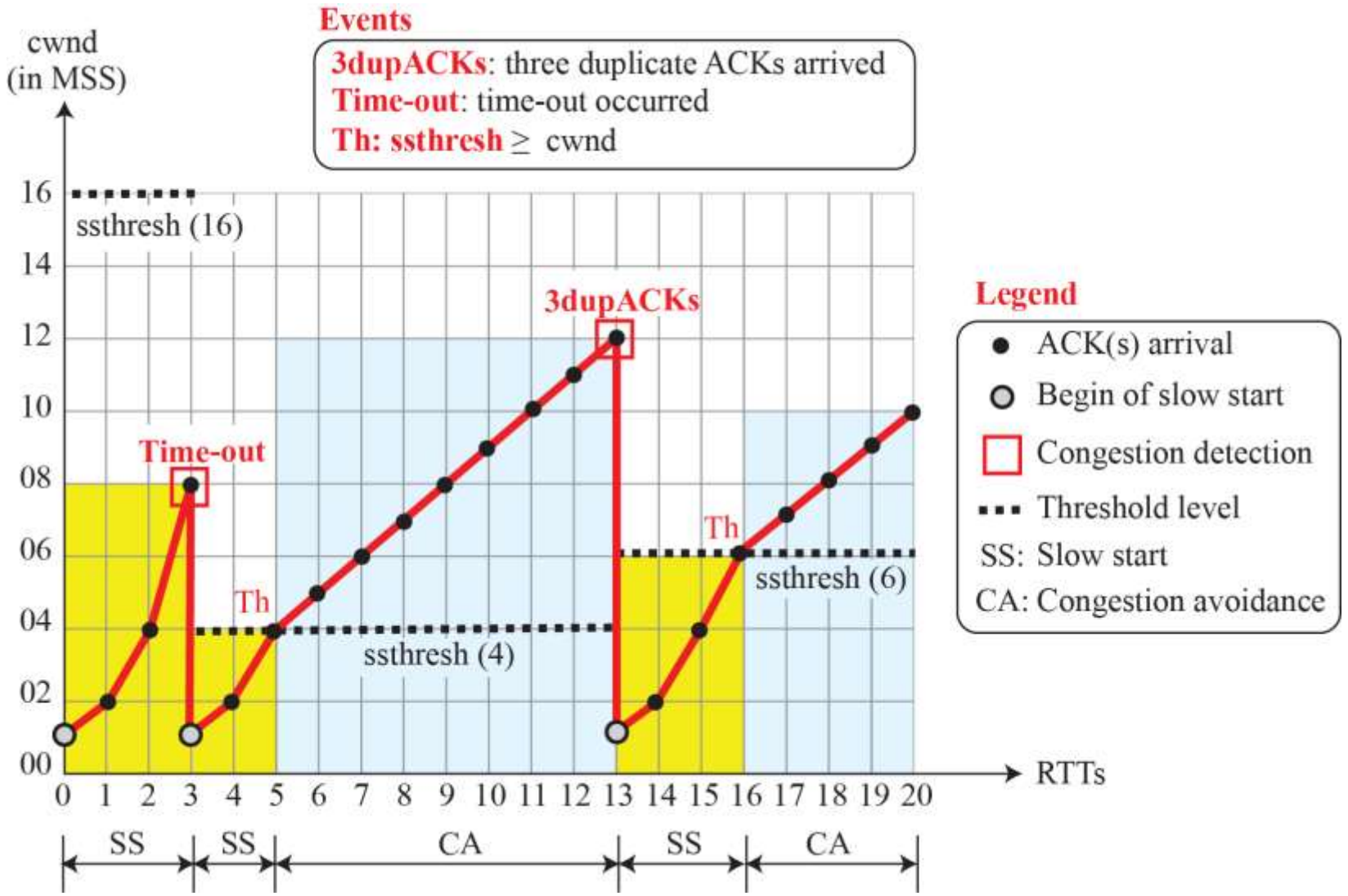
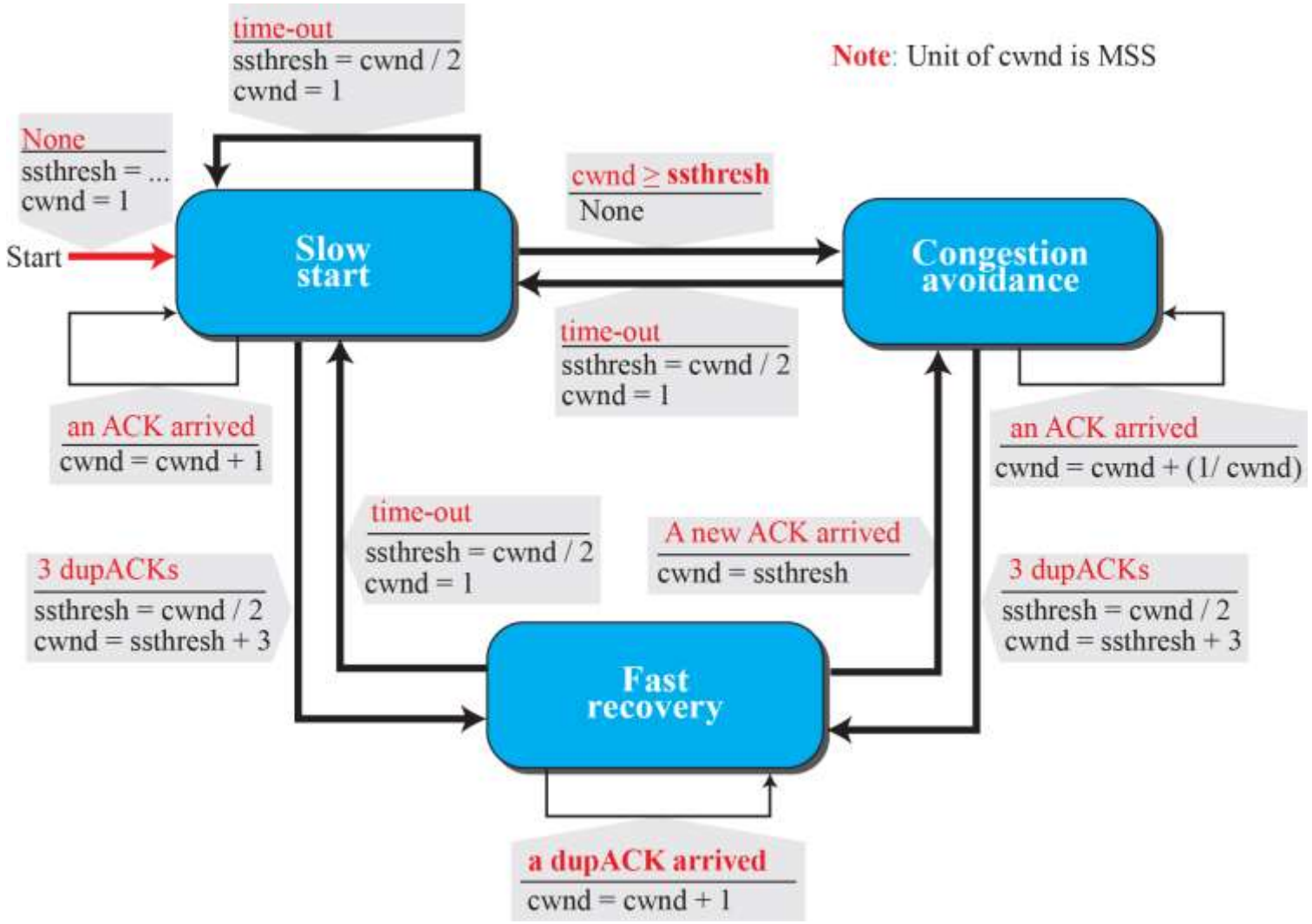


Figure 3.70: FSM for Reno TCP



Example 3.20

Figure 3.71 shows the same situation as Figure 3.69, but in Reno TCP. The changes in the congestion window are the same until RTT 13 when three duplicate ACKs arrive. At this moment, Reno TCP drops the ssthresh to 6 MSS, but it sets the cwnd to a much higher value (*ssthresh* + 3 = 9 MSS) instead of 1 MSS. It now moves to the fast recovery state. We assume that two more duplicate ACKs arrive until RTT 15, where *cwnd*grows exponentially. In this moment, a new ACK (not duplicate) arrives that announces the receipt of the lost segment. It now moves to the congestion avoidance state, but first deflates the congestion window to 6 MSS as though ignoring the whole fast-recovery state and moving back to the previous track.

Figure 3.71: Example of a Reno TCP

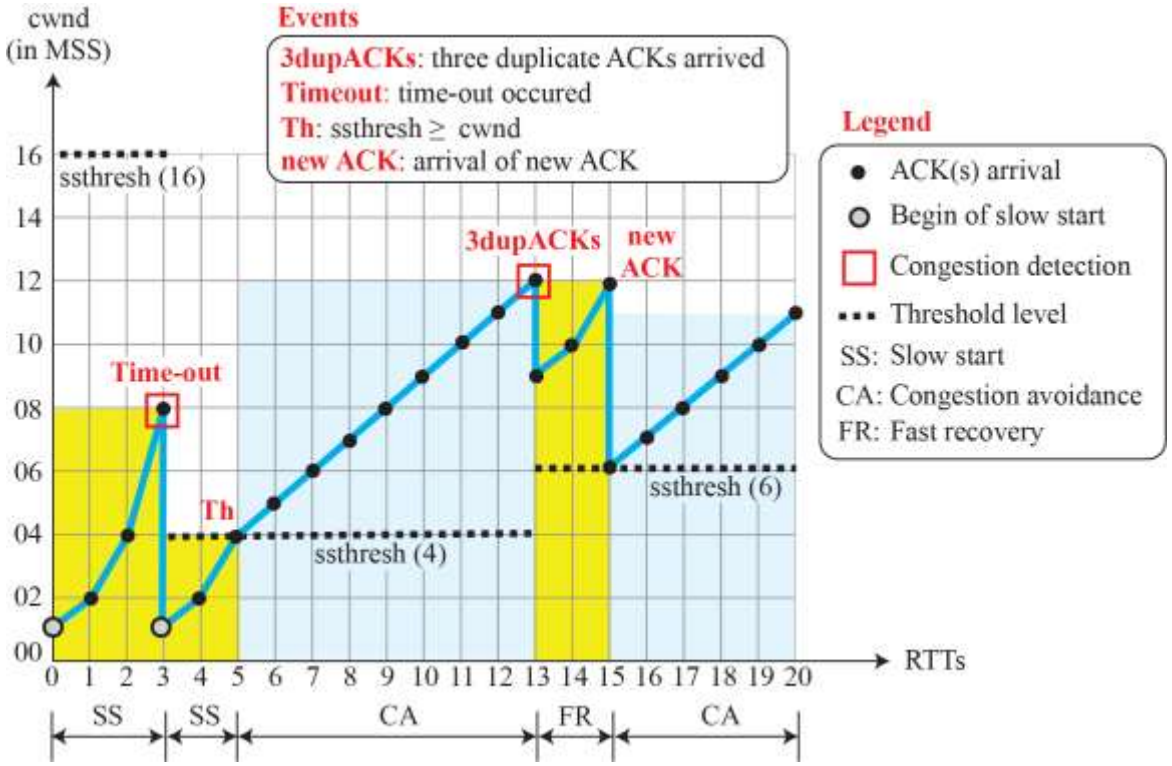
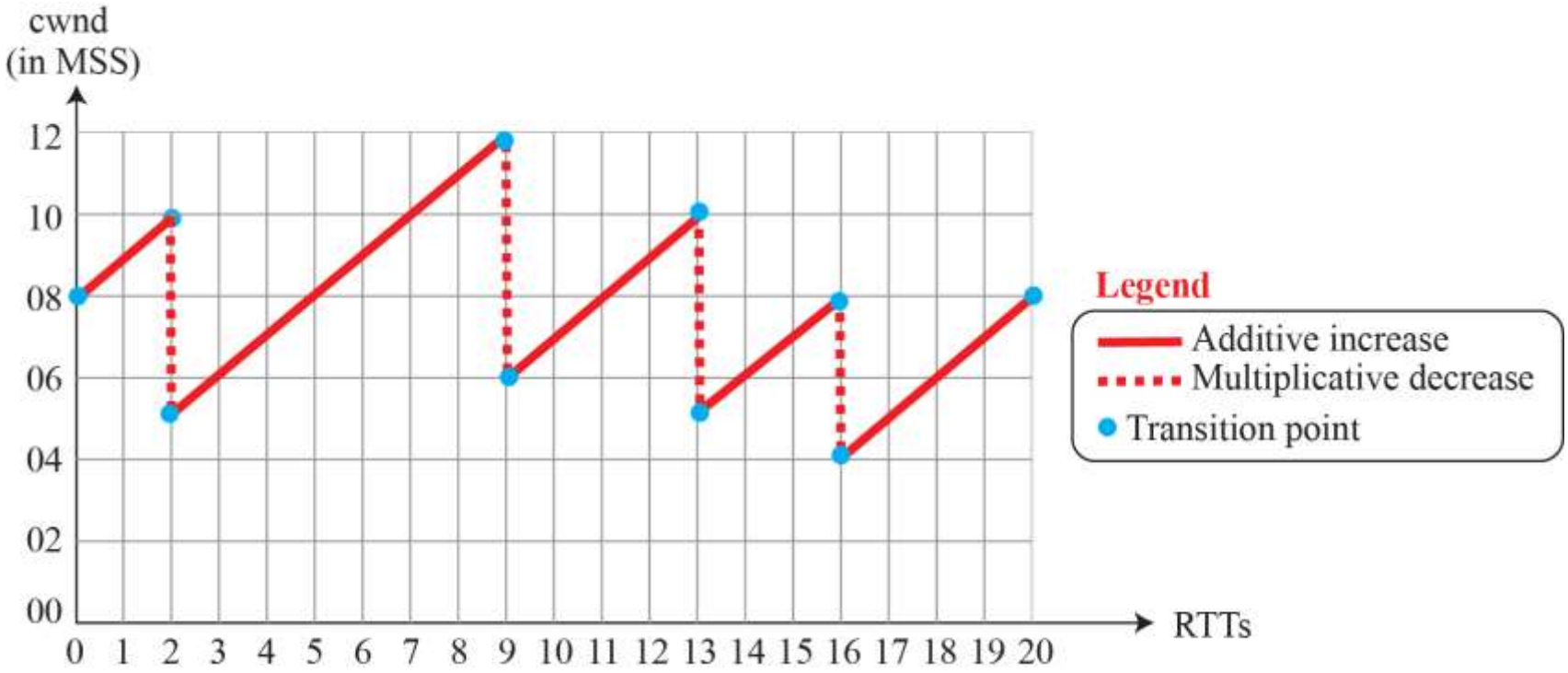


Figure 3.72: Additive increase, multiplicative decrease (AIMD)



Example 3.21

If MSS = 10 KB (kilobytes) and RTT = 100 ms in Figure 3.72, we can calculate the throughput as shown below.

$$W_{max} = (10 + 12 + 10 + 8 + 8) / 5 = 9.6 \text{ MSS}$$

$$\text{Throughput} = (0.75\ W_{max} / \text{RTT}) = 0.75 \times 960 \text{ kbps} / 100 \text{ ms} = 7.2 \text{ Mbps}$$

3.4.10 TCP Timers

To perform their operations smoothly, most TCP implementations use at least four timers.

- *Retransmission Timer*
  - *Round-Trip Time (RTT)*
  - *Karn's Algorithm*
  - *Exponential Backoff*
- *Persistence Timer*
- *Keepalive Timer*
- *TIME-WAIT Timer*

Example 3.22

Let us give a hypothetical example. Figure 3.73 shows part of a connection. The figure shows the connection establishment and part of the data transfer phases.

1. When the SYN segment is sent, there is no value for RTTM, RTTS, or RTTD. The value of RTO is set to 6.00 seconds. The following shows the value of these variables at this moment:

$$\text{RTO} = 6$$

Example 3.22 (continued)

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

$$RTT_M = 1.5$$
$$RTT_S = 1.5$$
$$RTT_D = (1.5)/2 = 0.75$$
$$RTO = 1.5 + 4 \times 0.75 = 4.5$$

Example 3.22 (continued)

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

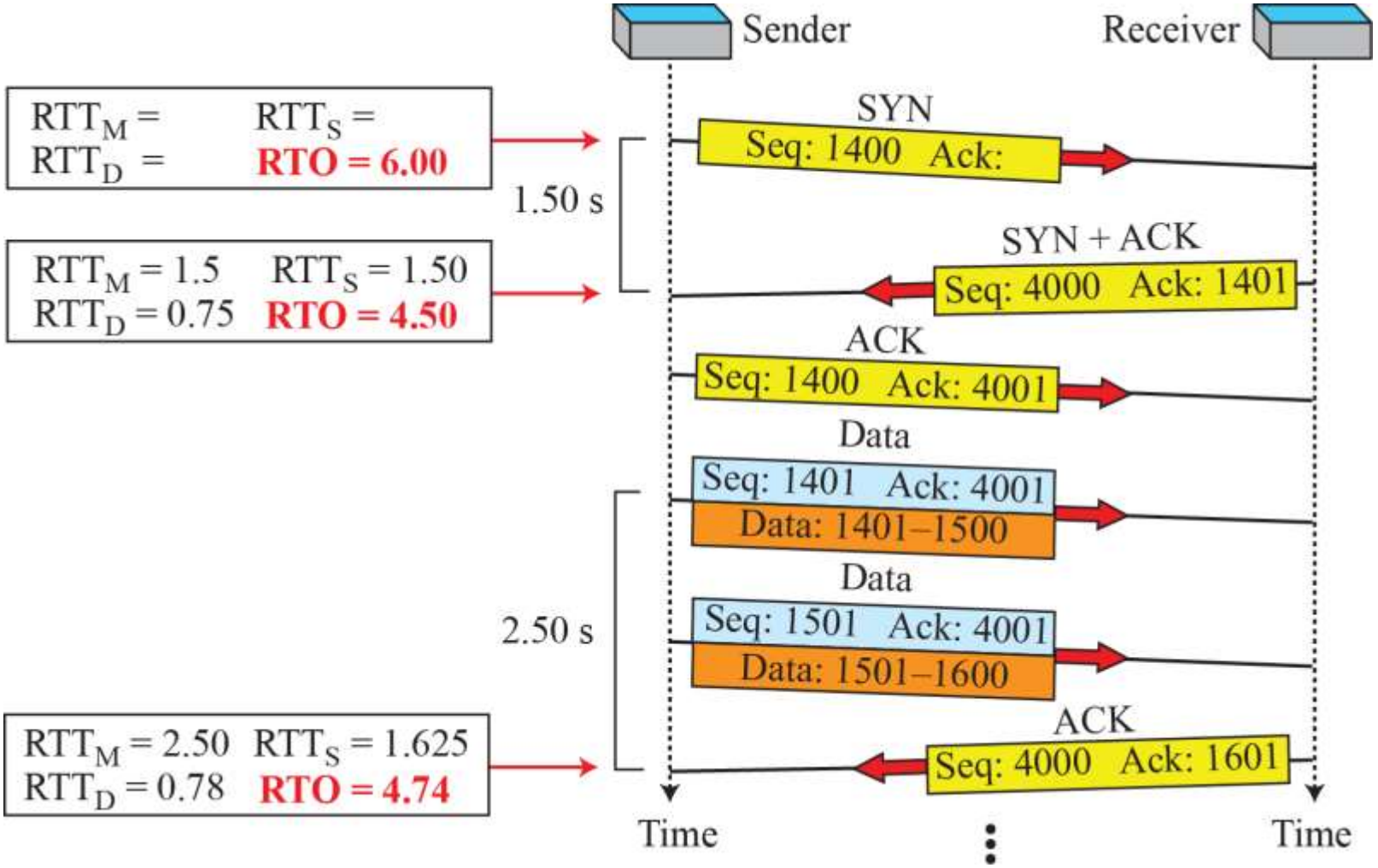
$$RTT_M = 2.5$$

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

$$RTT_D = (3/4) \times (0.75) + (1/4) \times |1.625 - 2.5| = 0.78$$

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

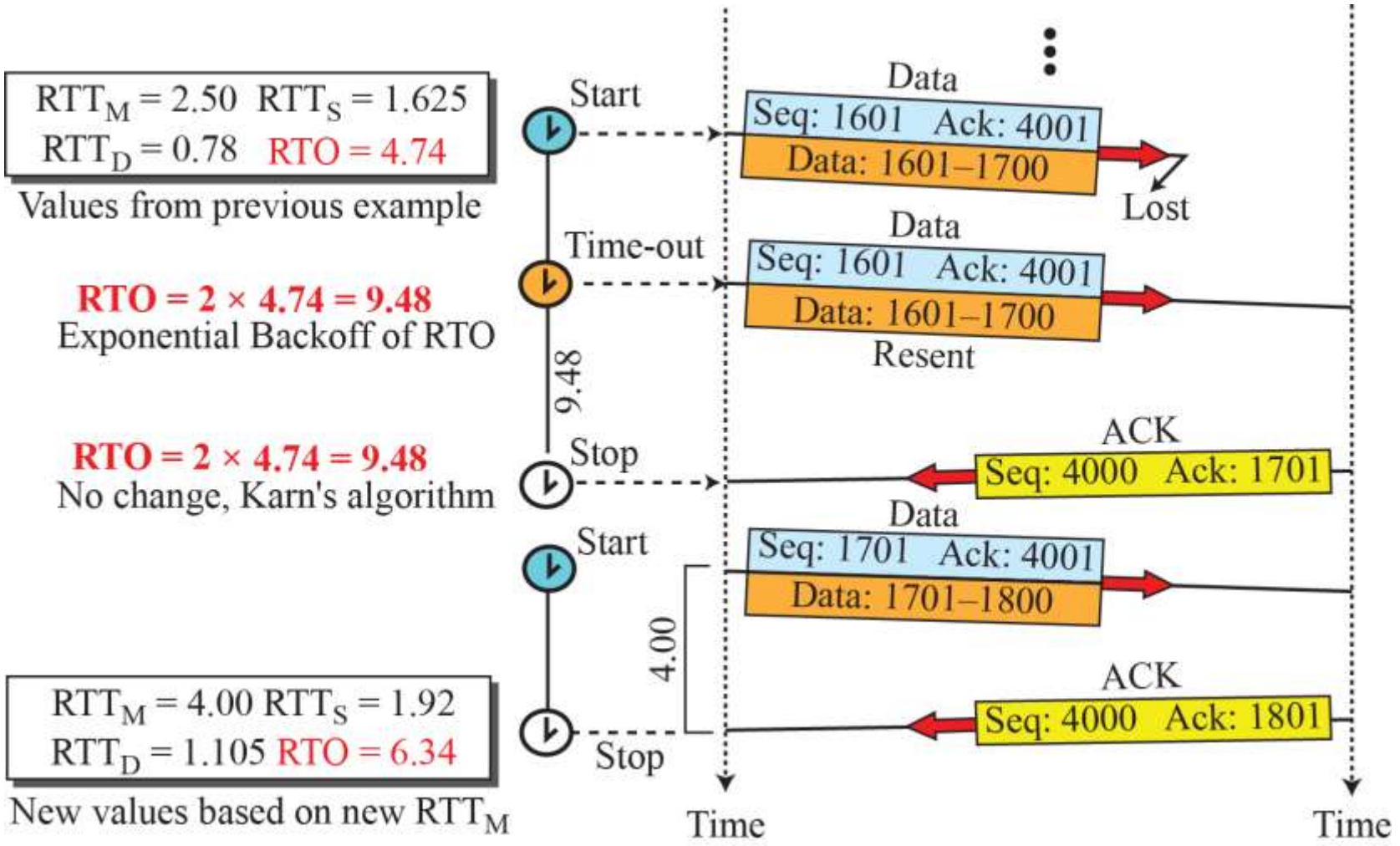
Figure 3.73: Example 3.22



Example 3.23

Figure 3.74 is a continuation of the previous example. There is retransmission and Karn's algorithm is applied. The first segment in the figure is sent, but lost. The RTO timer expires after 4.74 seconds. The segment is retransmitted and the timer is set to 9.48, twice the previous value of RTO. This time an ACK is received before the time-out. We wait until we send a new segment and receive the ACK for it before recalculating the RTO (Karn's algorithm).

Figure 3.74: Example 3.23



3.4.11 Options

The TCP header can have up to 40 bytes of optional information. Options convey additional information to the destination or align other options. These option are included on the book website for further reference.

Chapter 3: Summary

- ***The main duty of a transport-layer protocol is to provide process-to-process communication. To define the processes, we need port numbers. The client program defines itself with an ephemeral port number. The server defines itself with a well-known port number. To send a message from one process to another, the transport-layer protocol encapsulates and decapsulates messages. Flow control balances the exchange of data items between a producer and a consumer. A transport-layer protocol can provide two types of services: connectionless and connection-oriented. In a connectionless service, the sender sends packets to the receiver without any connection establishment. In a connection-oriented service, the client and the server first need to establish a connection between themselves.***

Chapter 3: Summary (continued)

- *We have discussed several common transport-layer protocols in this chapter. The Stop-and-Wait protocol provides both flow and error control, but is inefficient. The Go-Back-N protocol is the more efficient version of the Stop-and-Wait protocol and takes advantage of pipelining. The Selective-Repeat protocol, a modification of the Go-Back-N protocol, is better suited to handle packet loss. All of these protocols can be implemented bidirectionally using piggybacking.*

- *UDP is a transport protocol that creates a process-to-process communication. UDP is a (mostly) unreliable and connectionless protocol that requires little overhead and offers fast delivery. The UDP packet is called a user datagram.*

Chapter 3: Summary (continued)

- ***Transmission Control Protocol (TCP) is another transport-layer protocol in the TCP/IP protocol suite. TCP provides process-to-process, full-duplex, and connection-oriented service. The unit of data transfer between two devices using TCP software is called a segment. A TCP connection consists of three phases: connection establishment, data transfer, and connection termination. TCP software is normally implemented as a finite state machine (FSM).***