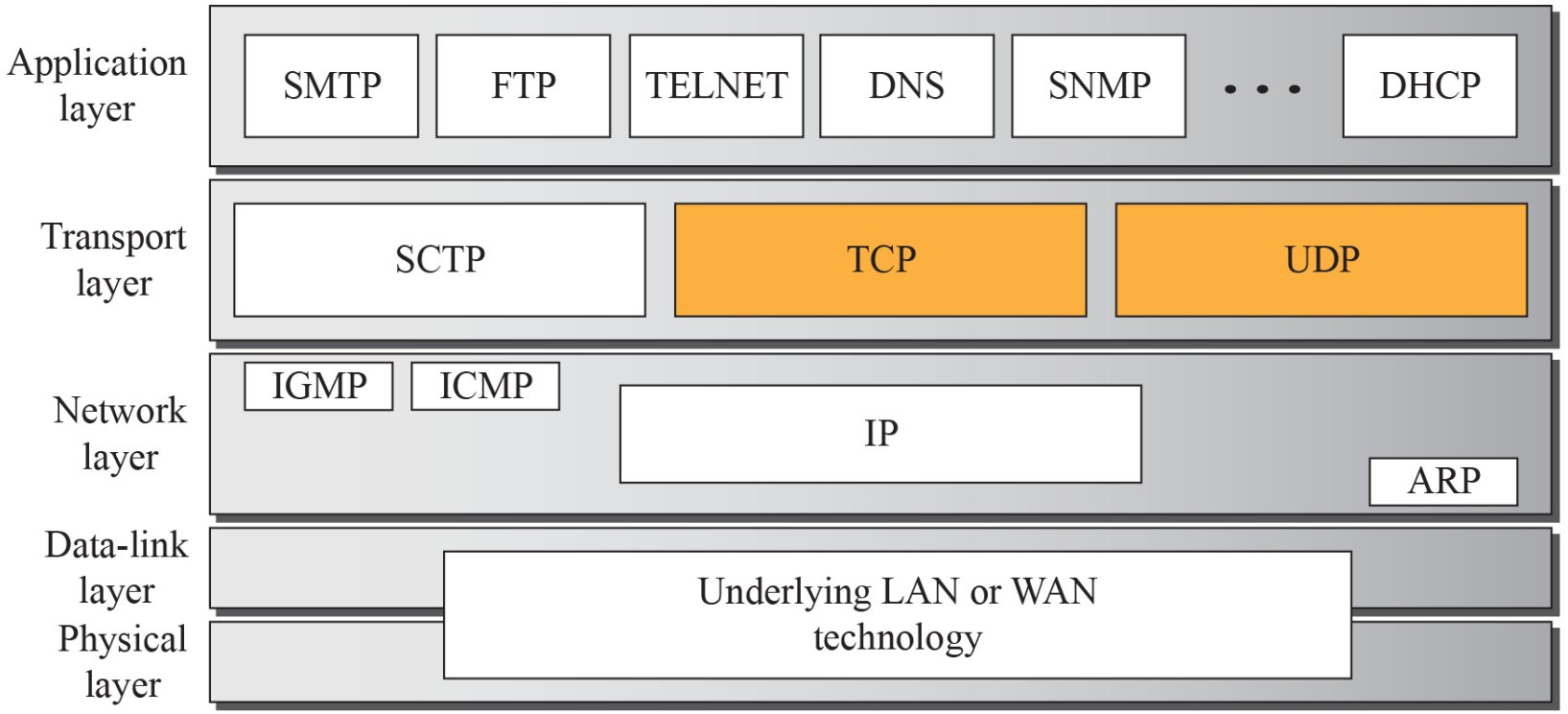
UNIT-5

Transport Layer Protocols

24-1 INTRODUCTION

After discussing the general principle behind the transport layer in the previous chapter, we concentrate on the transport protocols in the Internet in this chapter. Figure 24.1 shows the position of these three protocols in the TCP/IP protocol suite.

Figure 24.1: ***Position of transport-layer protocols in the TCP/IP protocol suite***



24.24.2 Port Numbers

As discussed in the previous chapter, a transport-layer protocol usually has several responsibilities. One is to create a process-to-process communication; these protocols use port numbers to accomplish this. Port numbers provide end-to-end addresses at the transport layer and allow multiplexing and demultiplexing at this layer, just as IP addresses do at the network layer. Table 24.1 gives some common port numbers for all three protocols we discuss in this chapter.

Table 24.1: 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

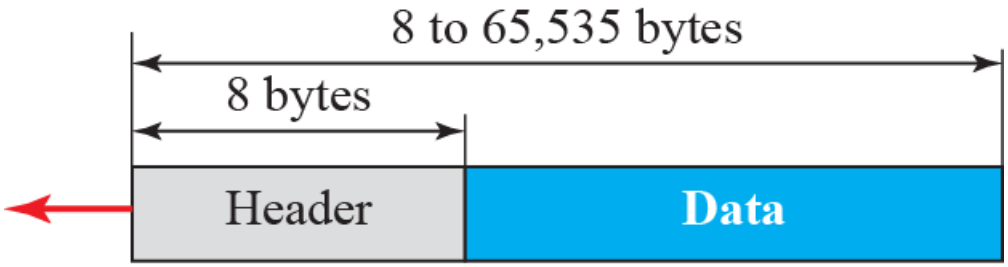
24-2 UDP

The User Datagram Protocol (UDP) is a connectionless, unreliable transport protocol. If UDP is so powerless, why would a process want to use it? With the disadvantages come some advantages. UDP is a very simple protocol using a minimum of overhead.

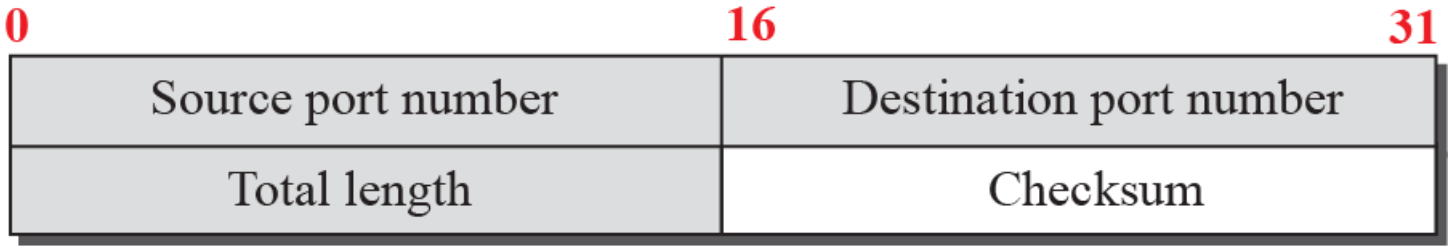
24.2.1 User Datagram

UDP packets, called user datagrams, have a fixed-size header of 8 bytes made of four fields, each of 2 bytes (16 bits). Figure 24.2 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. However, the total length needs to be less because a UDP user datagram is stored in an IP datagram with the total length of 65,535 bytes. The last field can carry the optional checksum (explained later).

Figure 24.2: User datagram packet format



a. UDP user datagram



b. Header format

Example 24.1

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 24.1 (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 3.1).

24.2.3 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. For example, in our daily life, we all know that a one-day delivery of a package by a carrier is more expensive than a three-day delivery. Although high speed and low cost are both desirable features in delivery of a parcel, they are in conflict with each other.

24-3 TCP

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.

24.3.2 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.

Example 24.7

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

24.3.3 Segment

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

Figure 24.7: TCP segment format

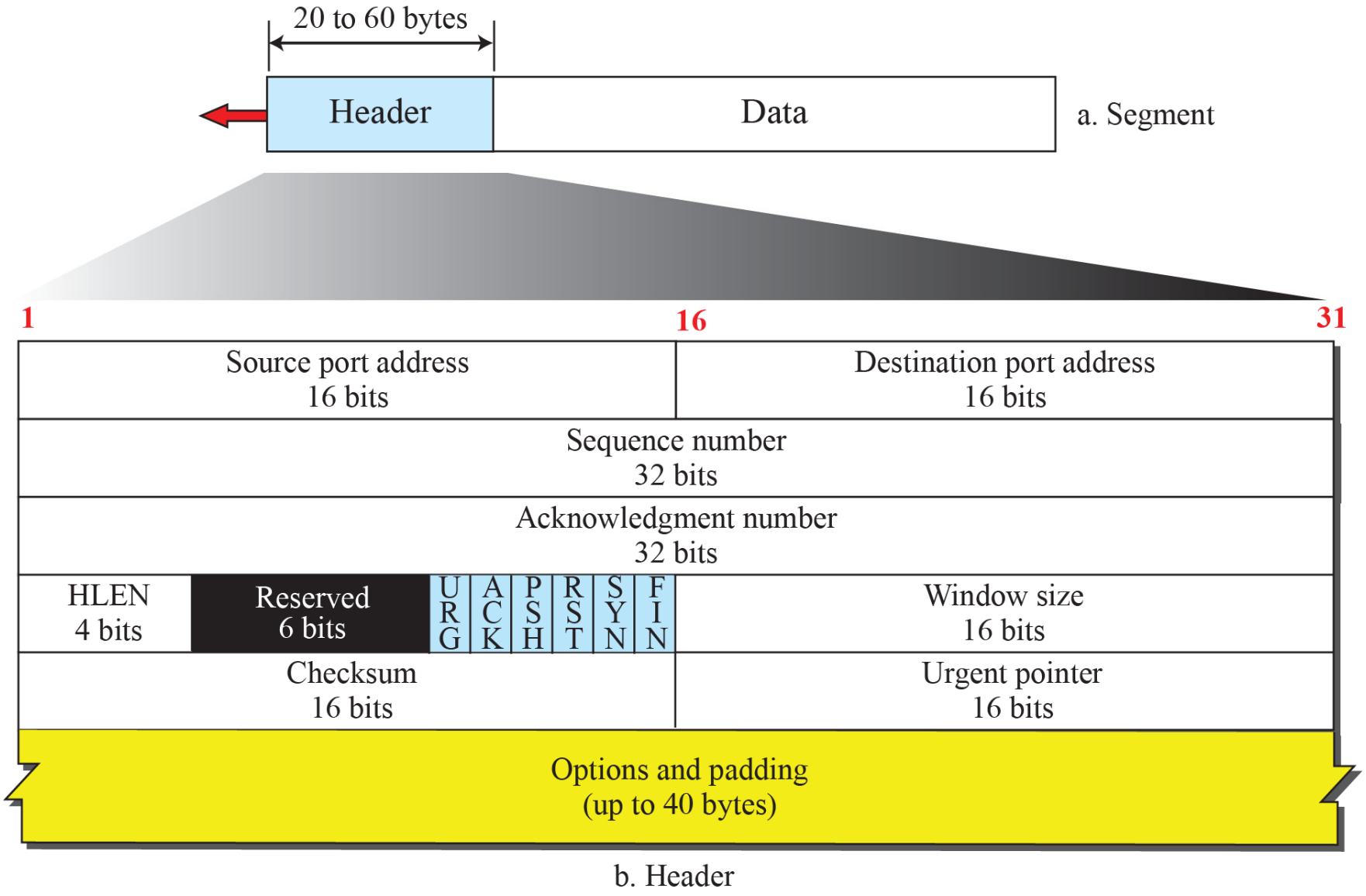
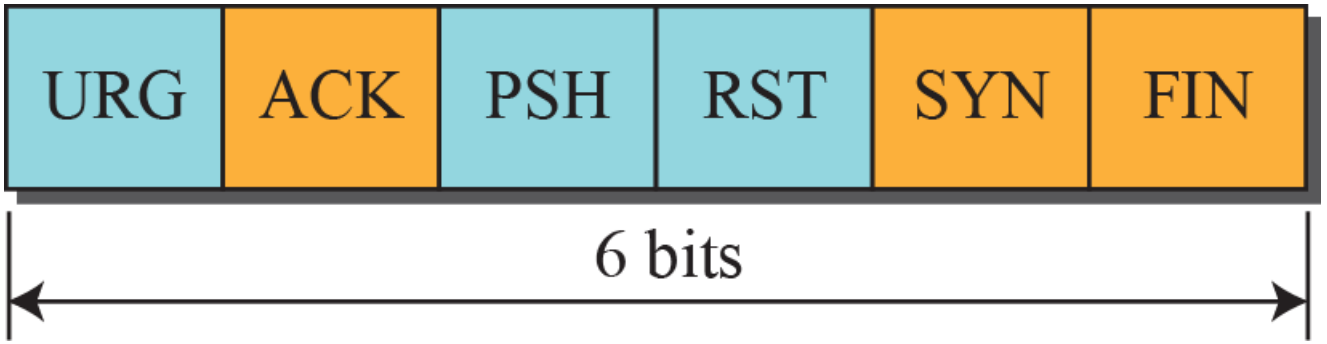


Figure 24.8: Control field



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

Example 23.2.4

The following is a dump of a TCP header in hexadecimal format

05320017 00000001 00000000 500207FF 00000000

What is the source port number?
What is the destination port number?
What is sequence number?
What is the acknowledgment number?
What is the length of the header?
What is the type of the segment?
What is the window size?

24.3.4 A TCP Connection

TCP is connection-oriented. 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. 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 logical, 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.

Figure 24.10: Connection establishment using three-way handshaking

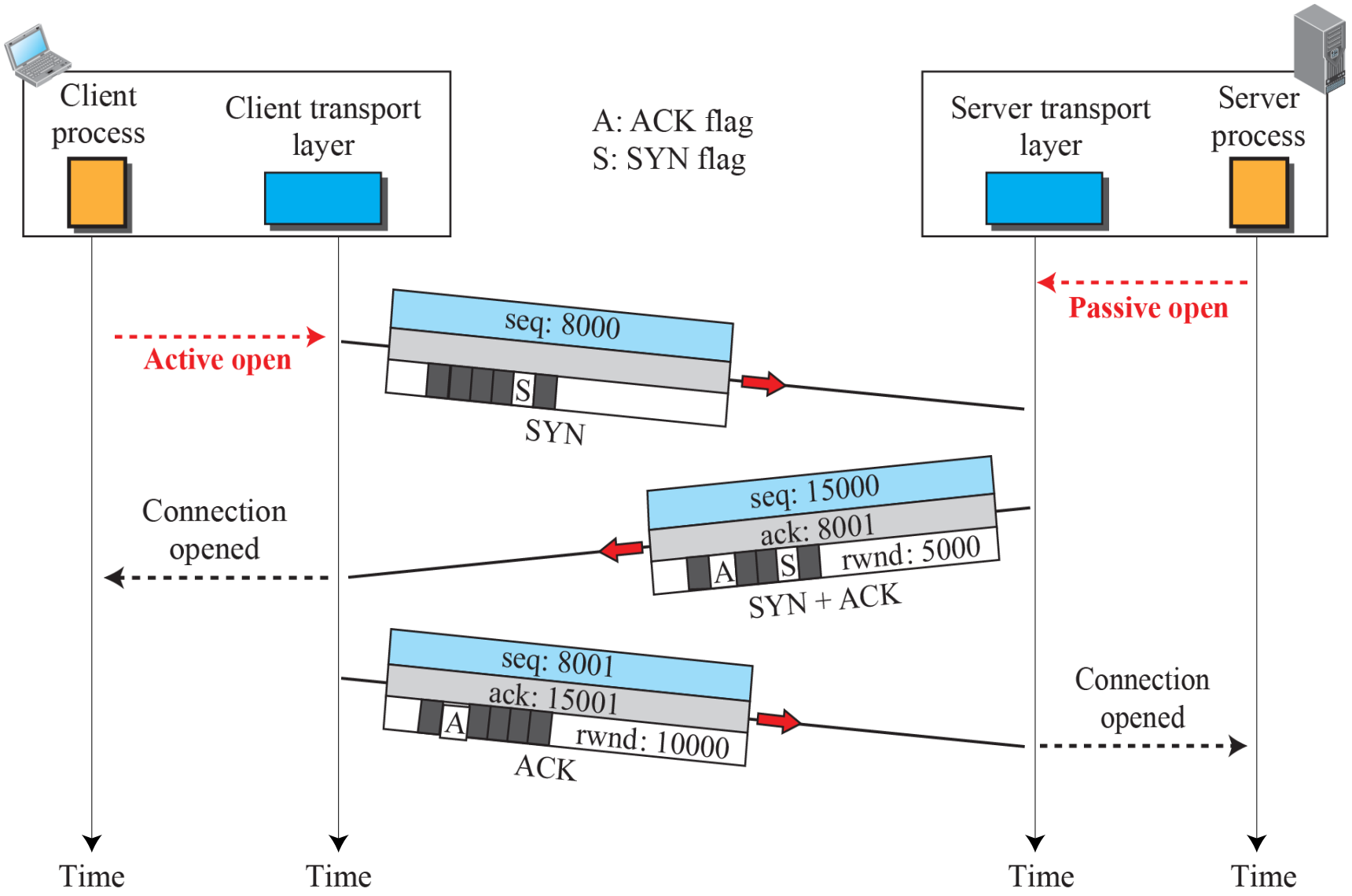


Figure 24.11: Data transfer

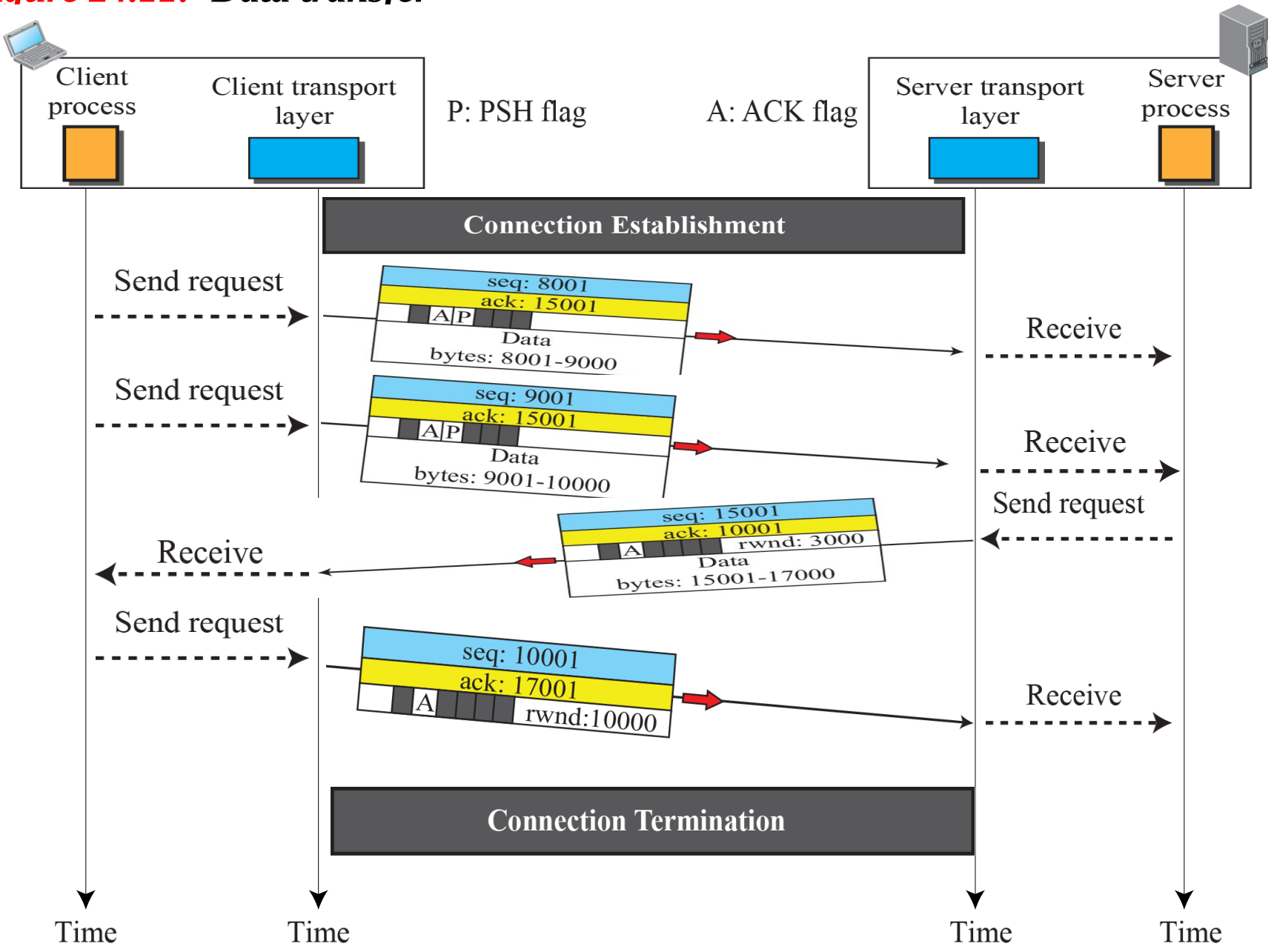


Figure 24.12: Connection termination using three-way handshaking

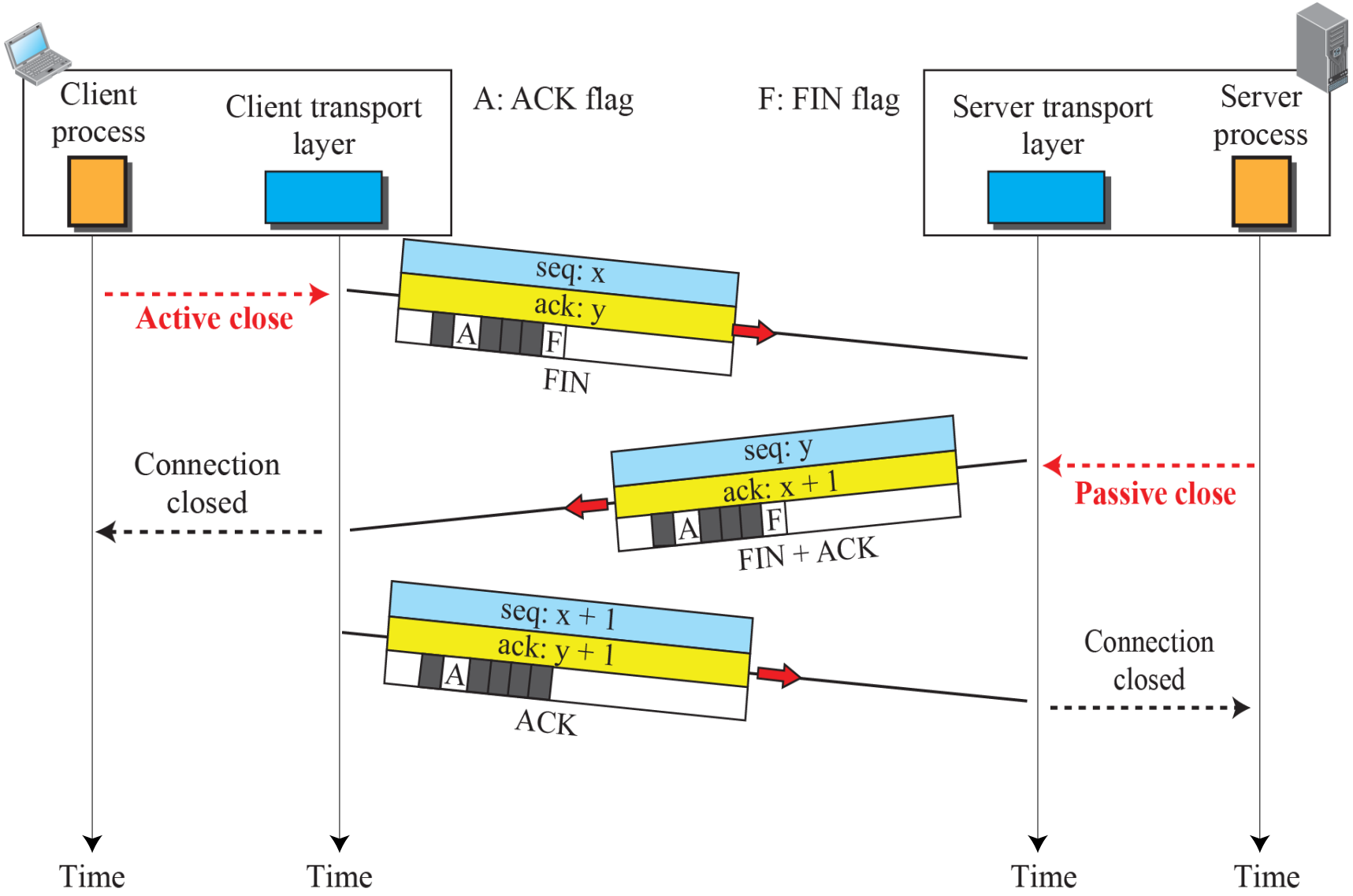
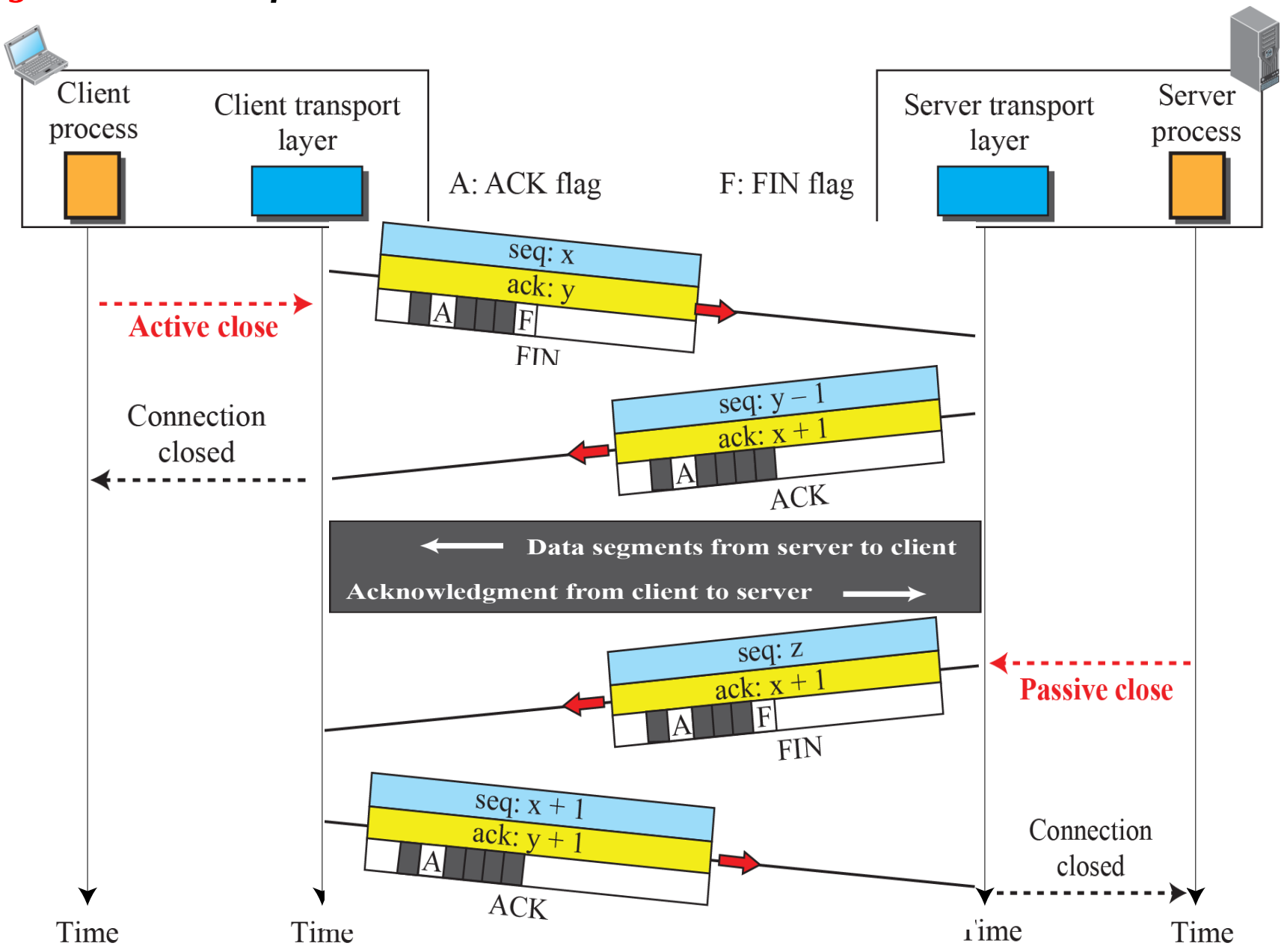


Figure 24.13: *Half-close*



24.3.5 State Transition 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 24.14.

Figure 24.16: Time-line diagram for a common scenario

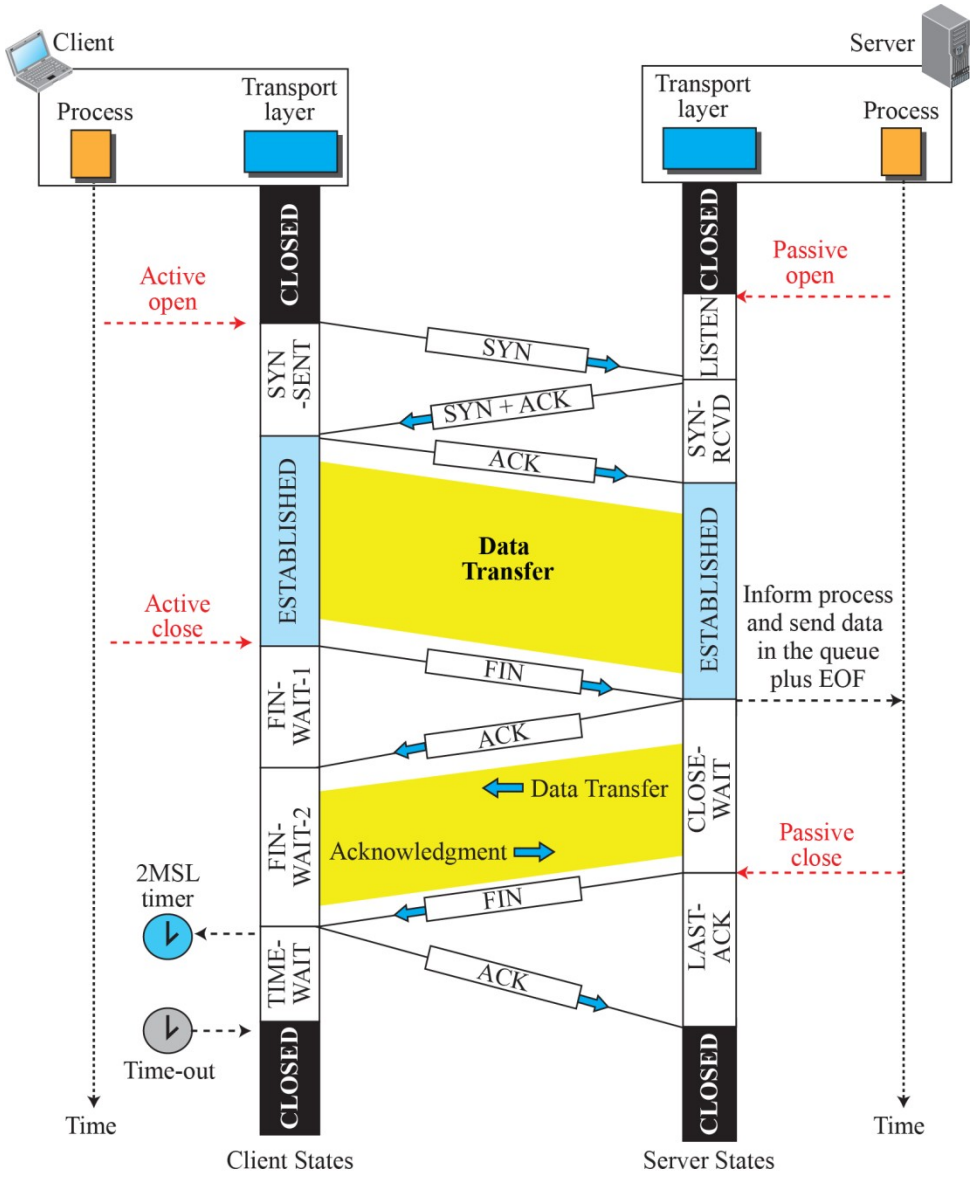


Figure 24.14: State transition diagram

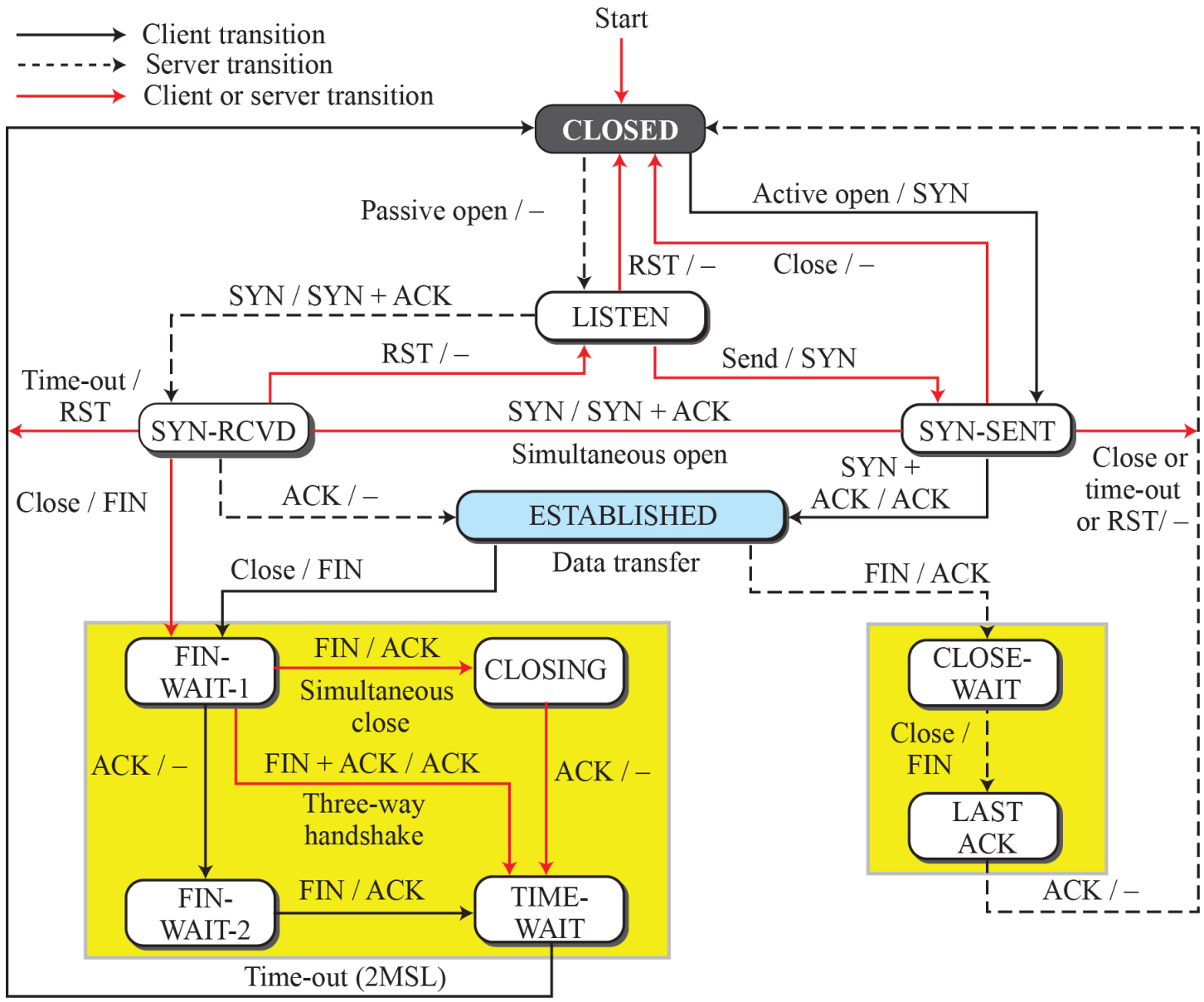
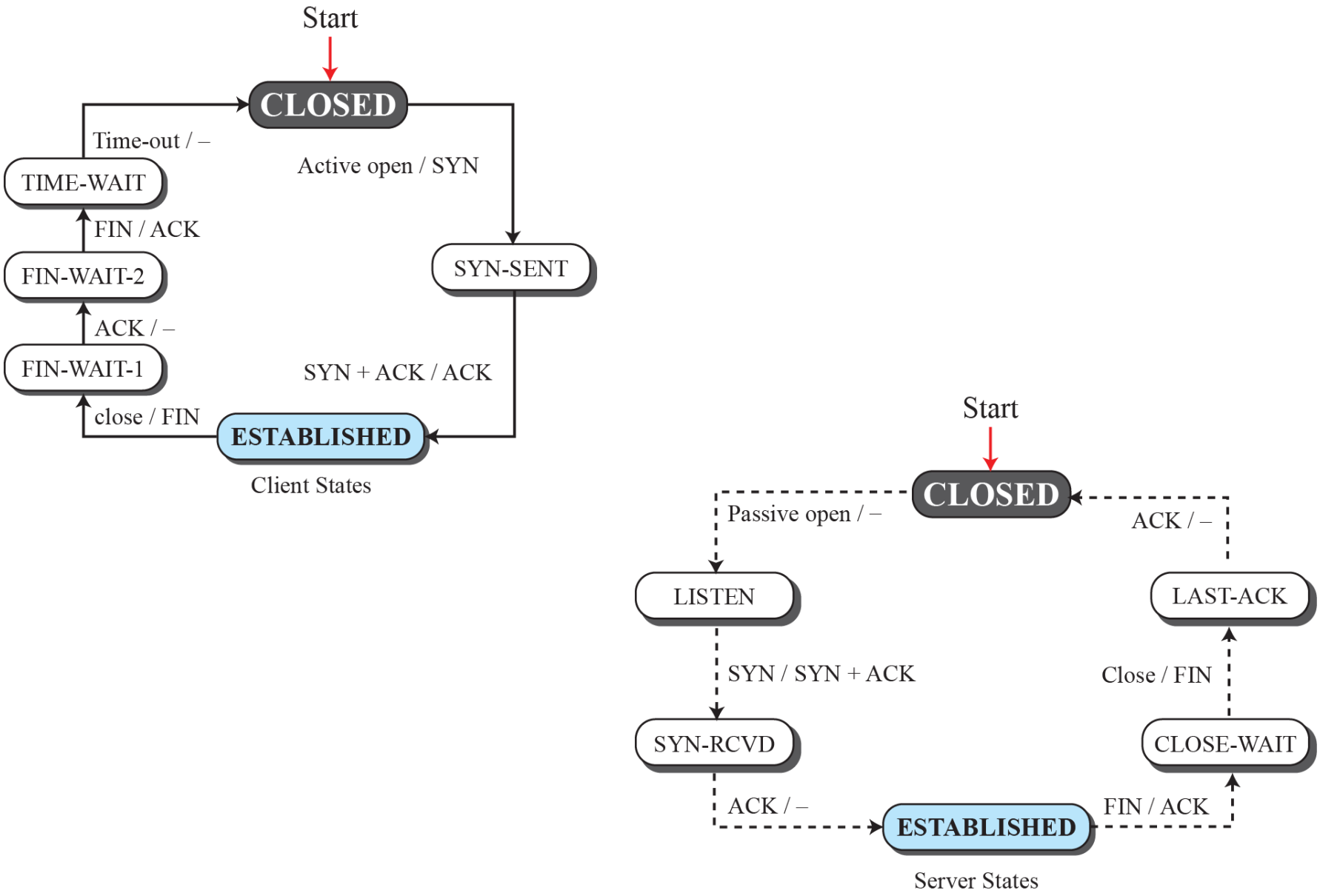


Table 24.2: States for TCP

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

Figure 24.15: Transition diagram with half-close connection termination



24.3.6 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 unrealistic assumption that communication is only unidirectional. The bidirectional communication can be inferred using two unidirectional communications with piggybacking.

Figure 24.17: Send window in TCP

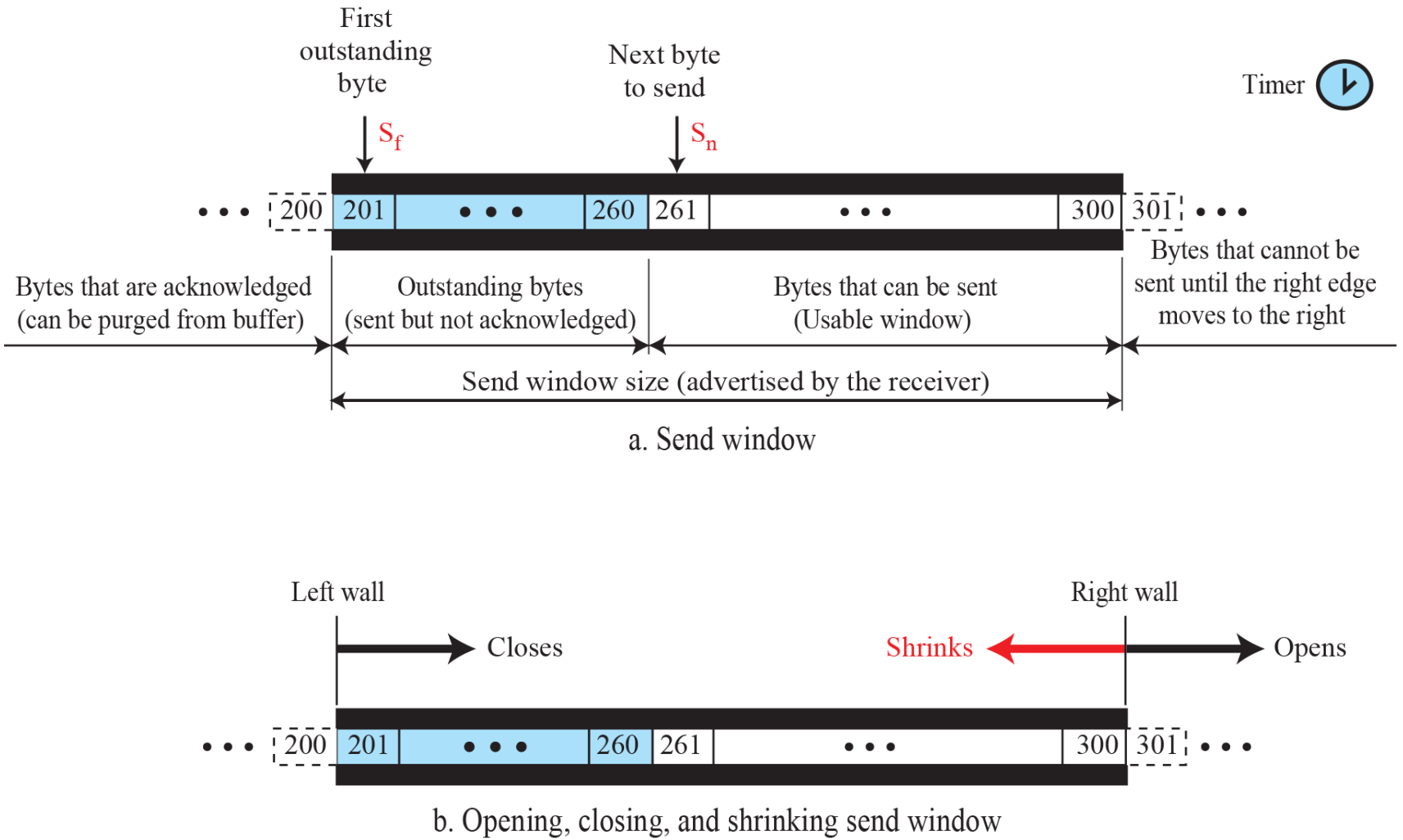
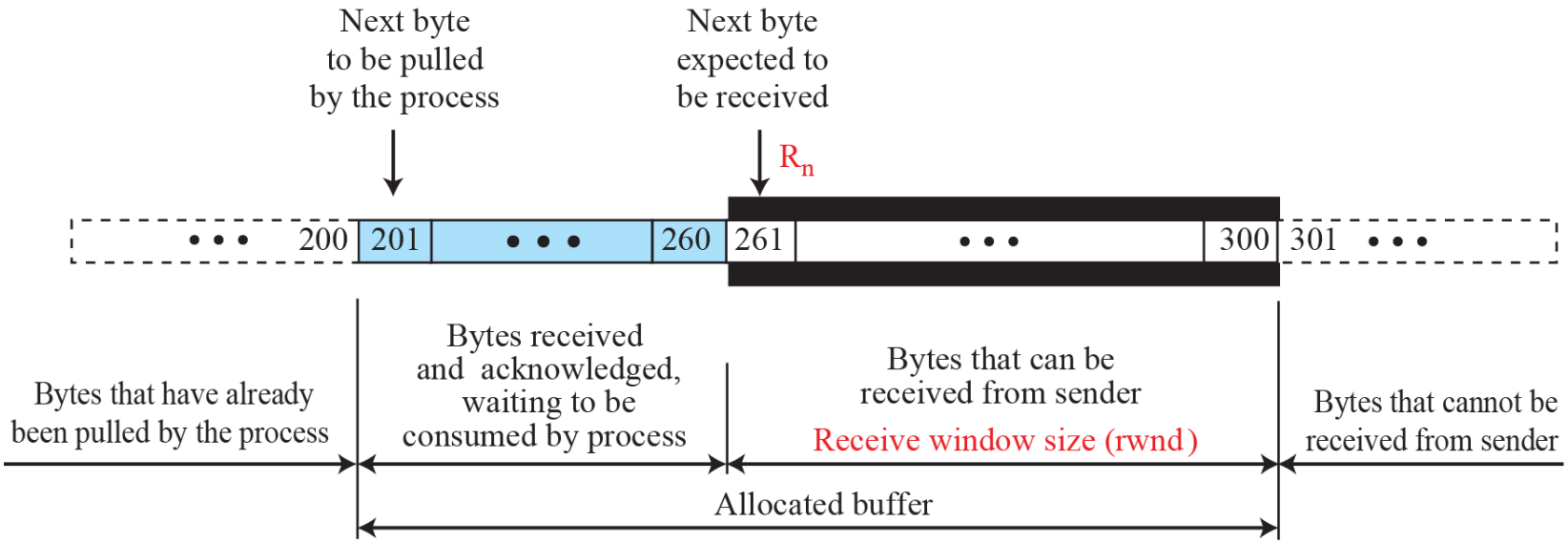
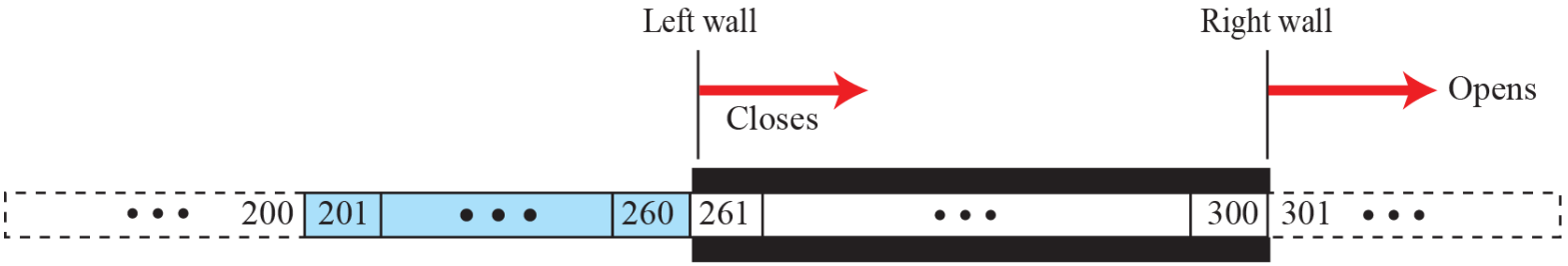


Figure 24.18: Receive window in TCP



a. Receive window and allocated buffer



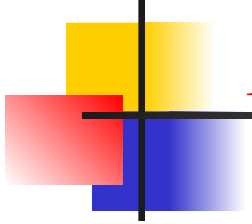
b. Opening and closing of receive window

Example

What is the value of the receiver window (rwnd) for host A if the receiver, host B, has a buffer size of 5000 bytes and 1000 bytes of received and unprocessed data?

Solution

The value of rwnd = 5000 − 1000 = 4000. Host B can receive only 4000 bytes of data before overflowing its buffer. Host B advertises this value in its next segment to A.



Example

A TCP connection is using a window size of 10,000 bytes, and the previous acknowledgement number was 22,001. It receives a segment with acknowledgment number 24,001 and window size advertisement of 12,000. Draw a diagram to show the situation of the window before and after.

Example

A window holds bytes 2001 and 5000. The next byte to be sent is 3001. Draw a figure to show the situation of the window after the following two events:

1.An ACK segment with the acknowledgement number 25, 000 and window size advertisement 4,000 is received.
2.A segment carrying 1000 bytes is sent.

24.3.7 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.

Figure 24.19: Data flow and flow control feedbacks in TCP

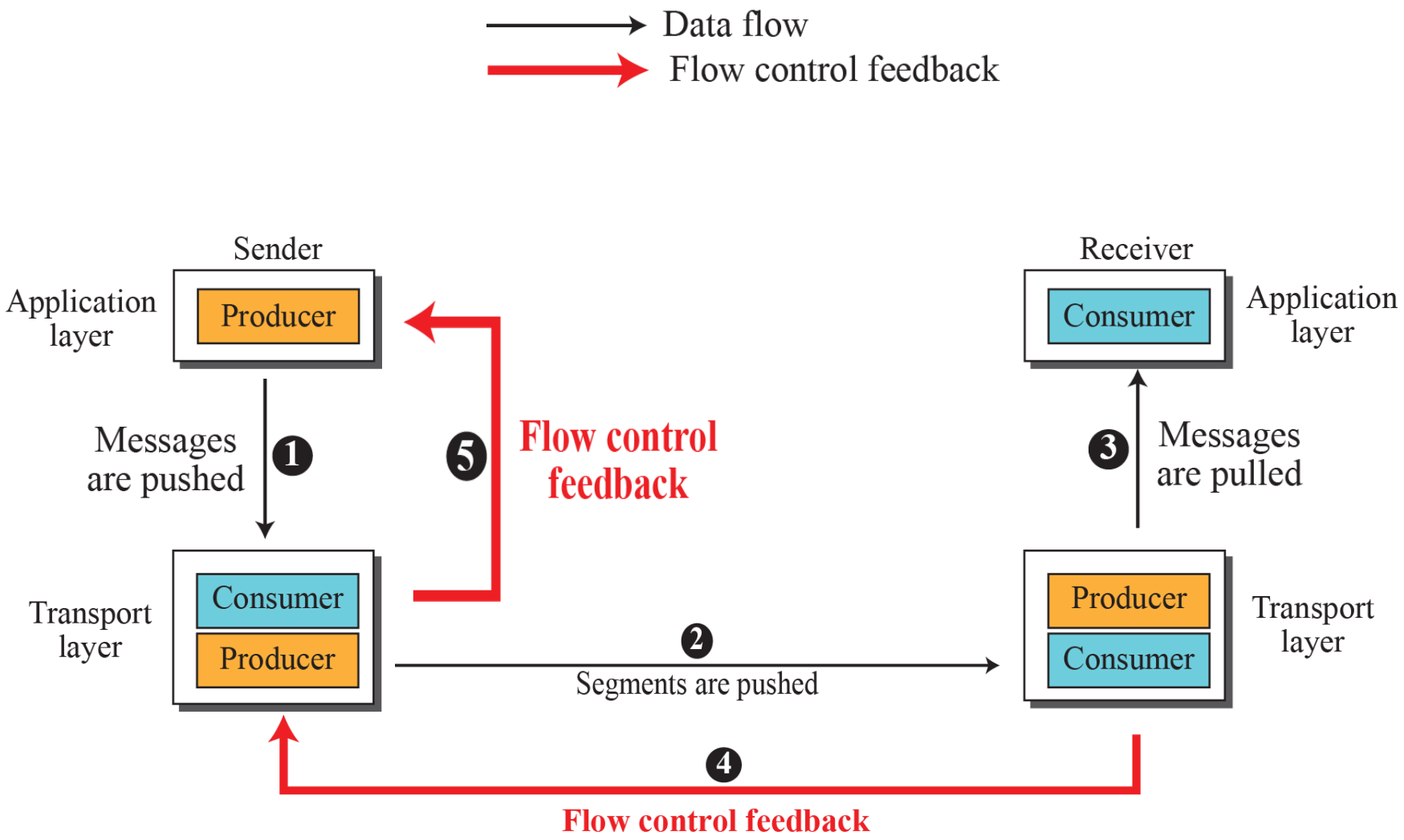
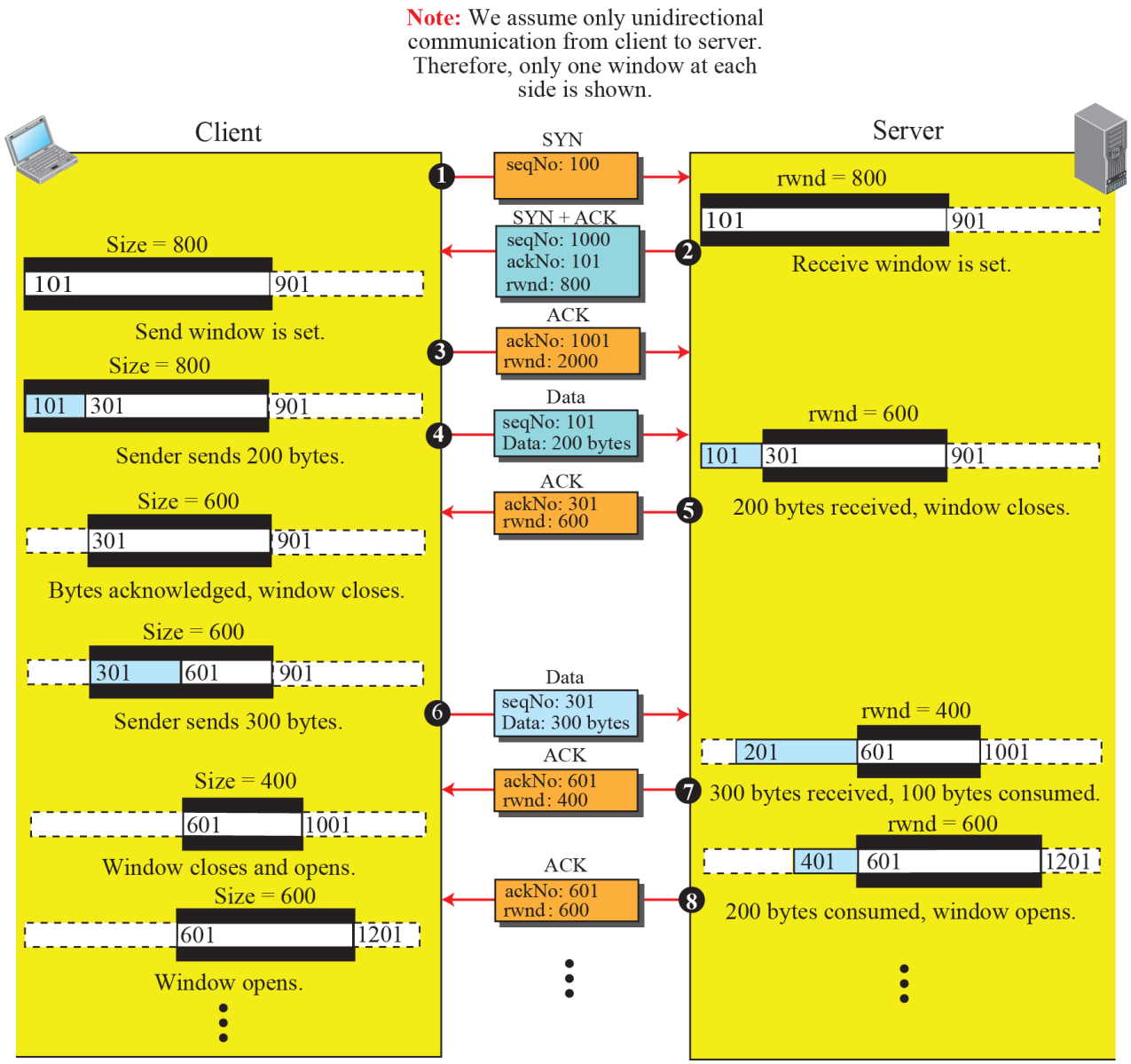


Figure 24.20: An example of flow control



24.3.8 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.

Figure 24.24: *Normal operation*

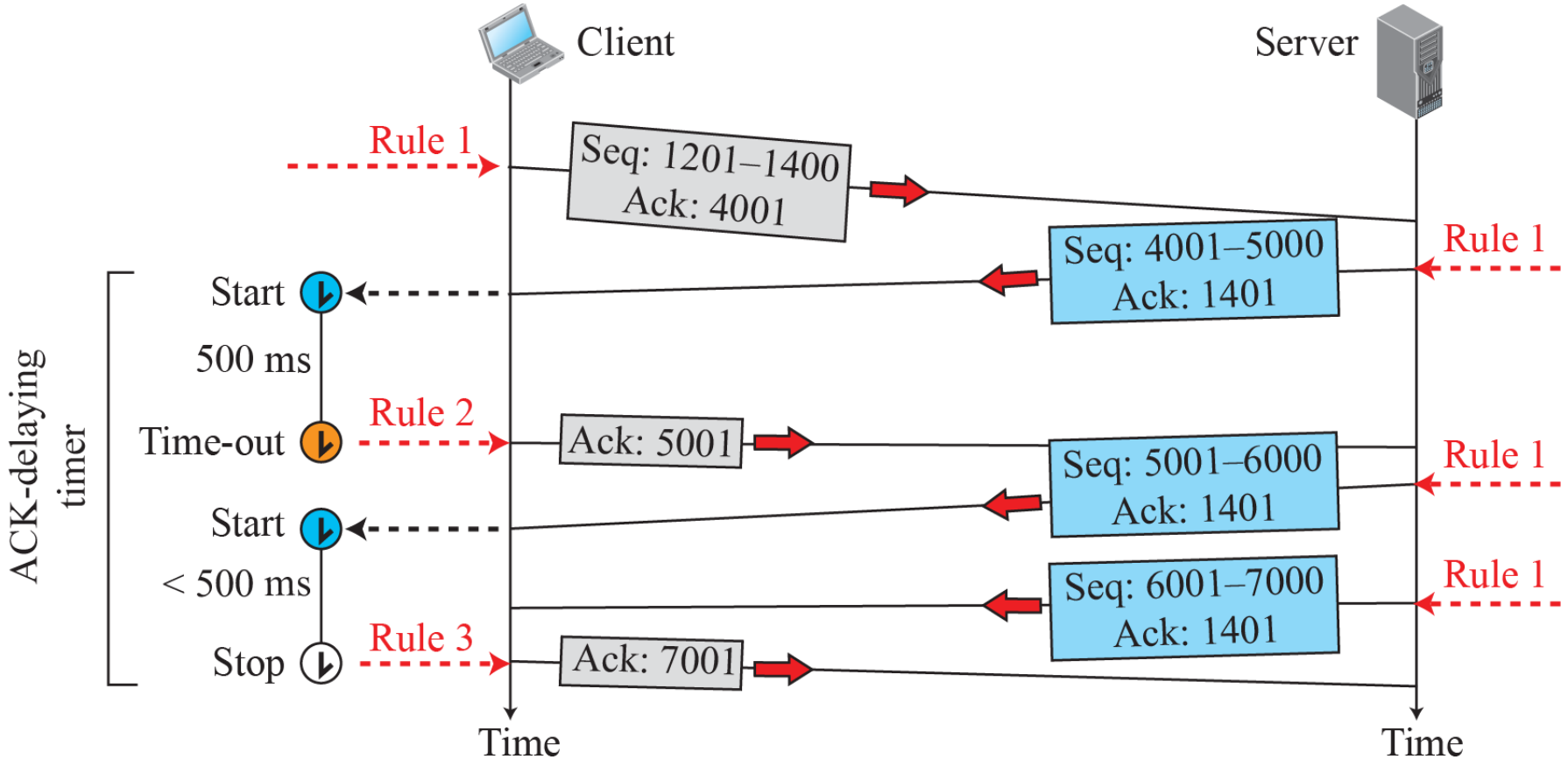


Figure 24.25: Lost segment

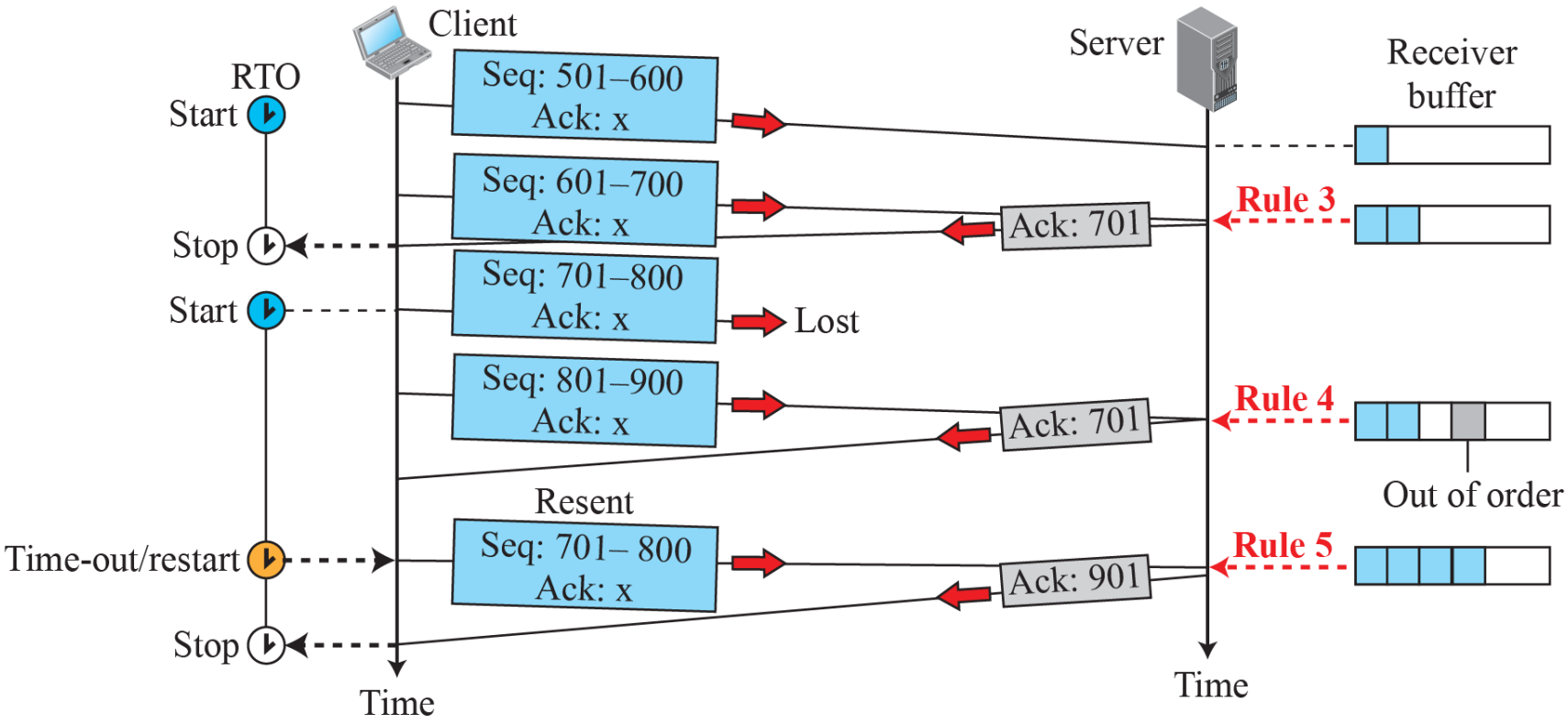


Figure 24.26: Fast retransmission

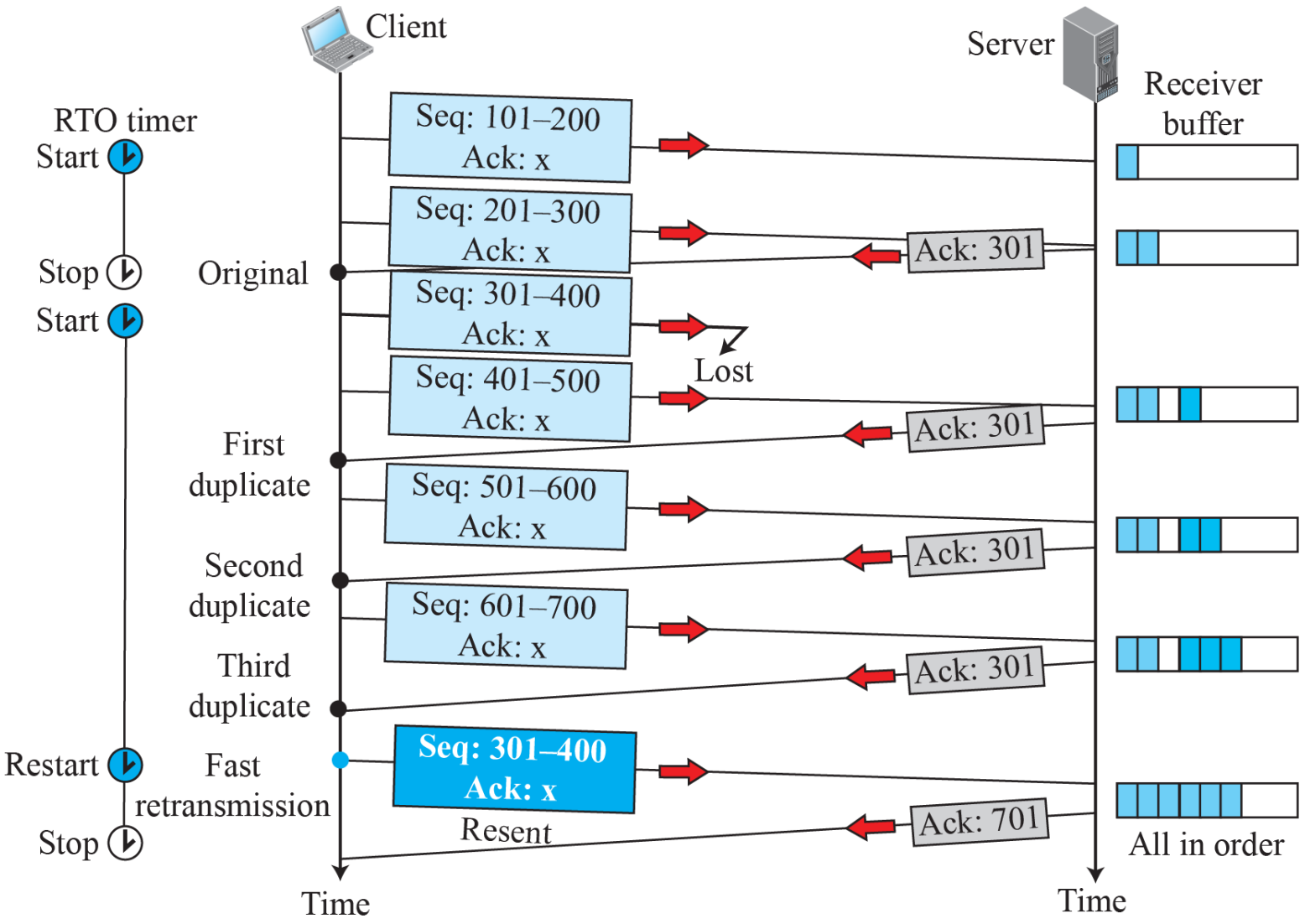


Figure 24.27: ***Lost acknowledgment***

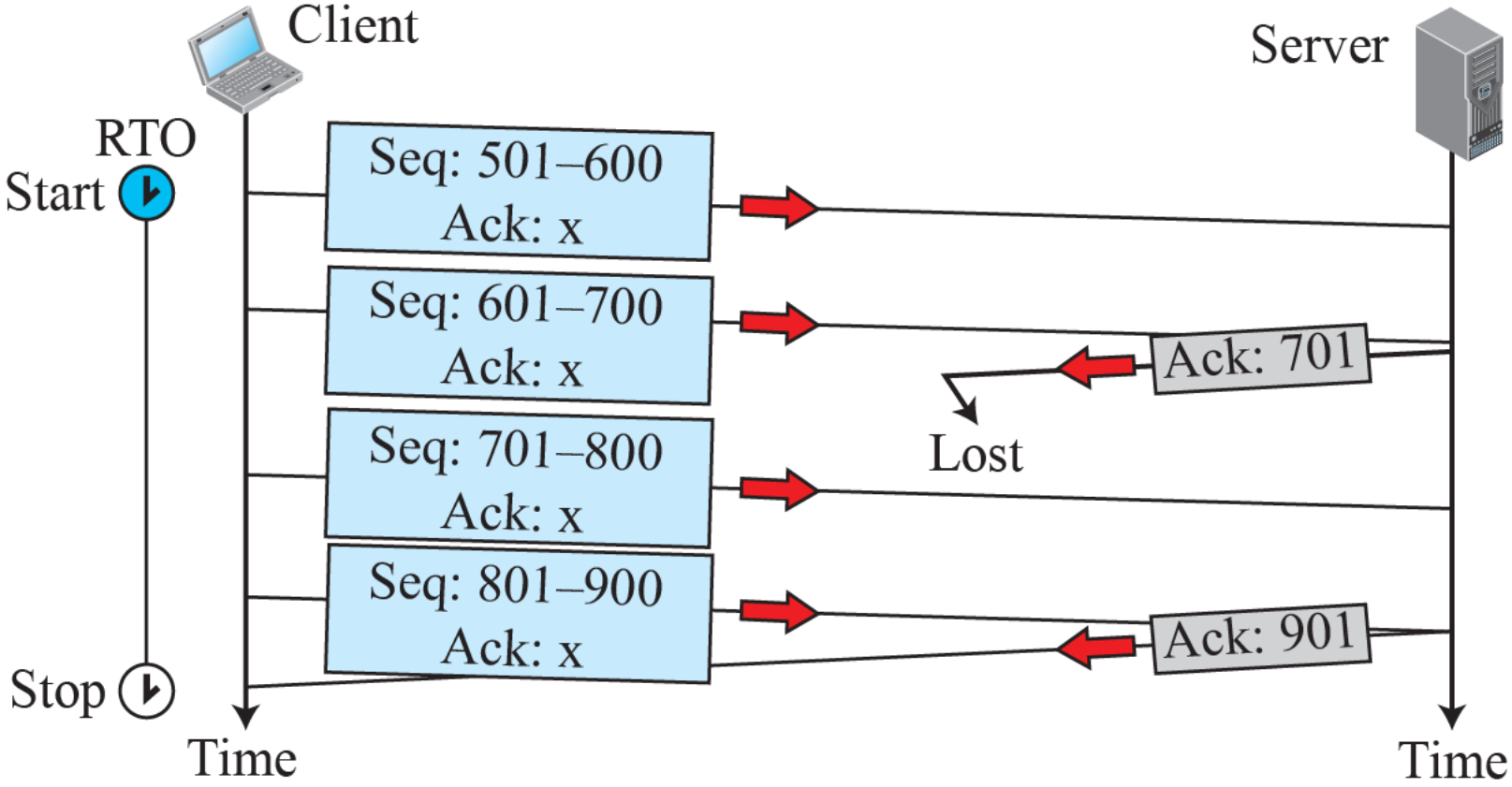


Figure 24.28: ***Lost acknowledgment corrected by resending a segment***

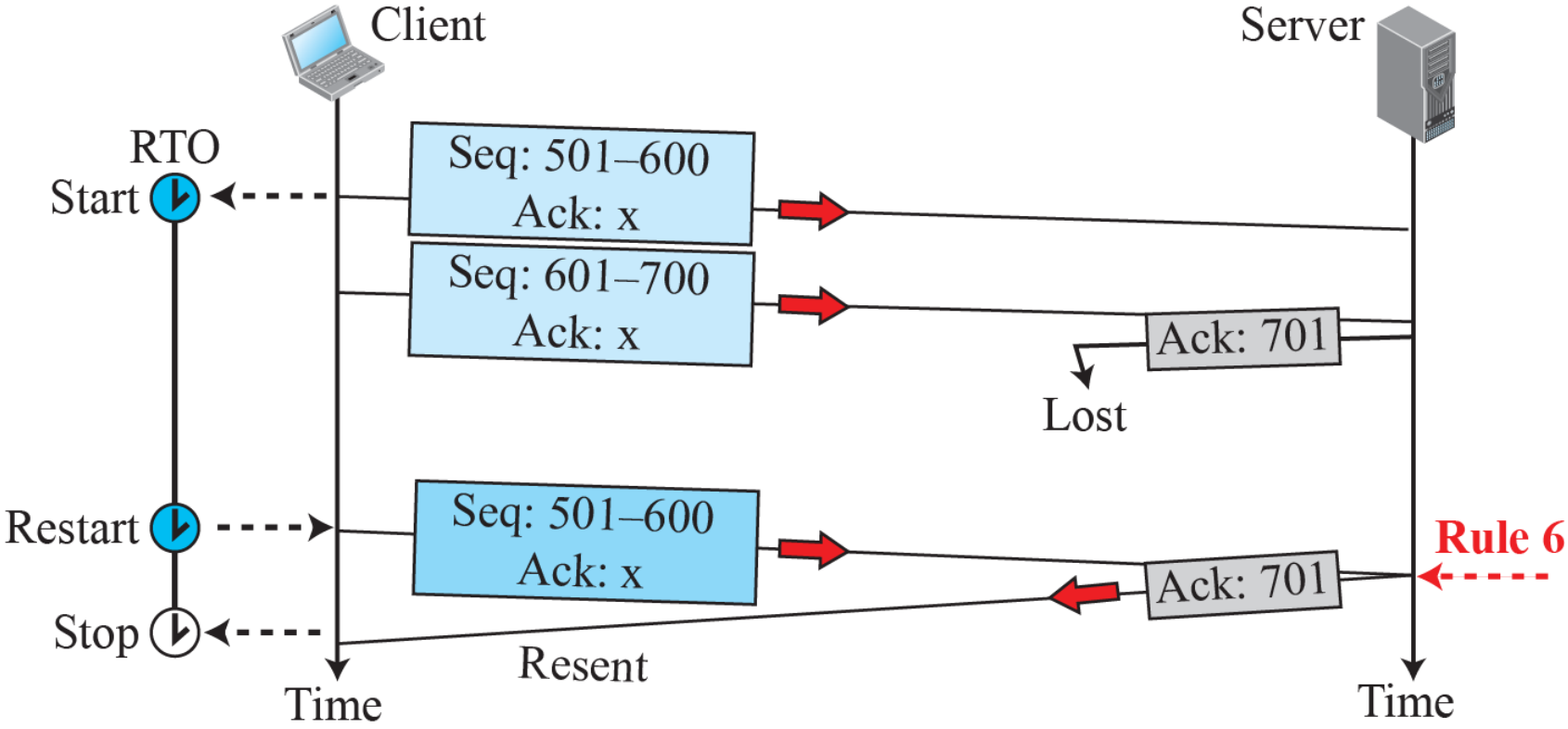


Figure 24.22: *Simplified FSM for the TCP sender side*

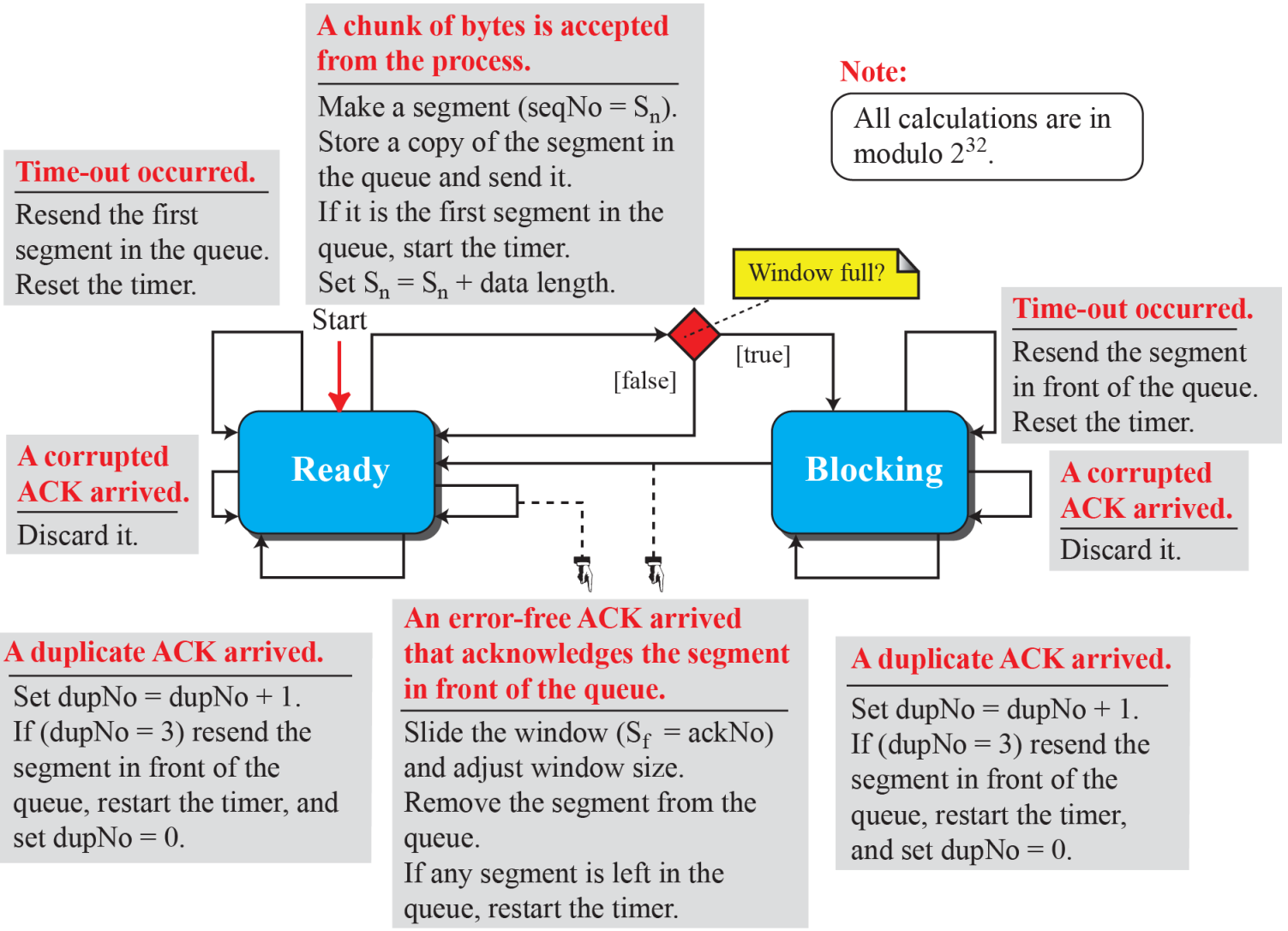
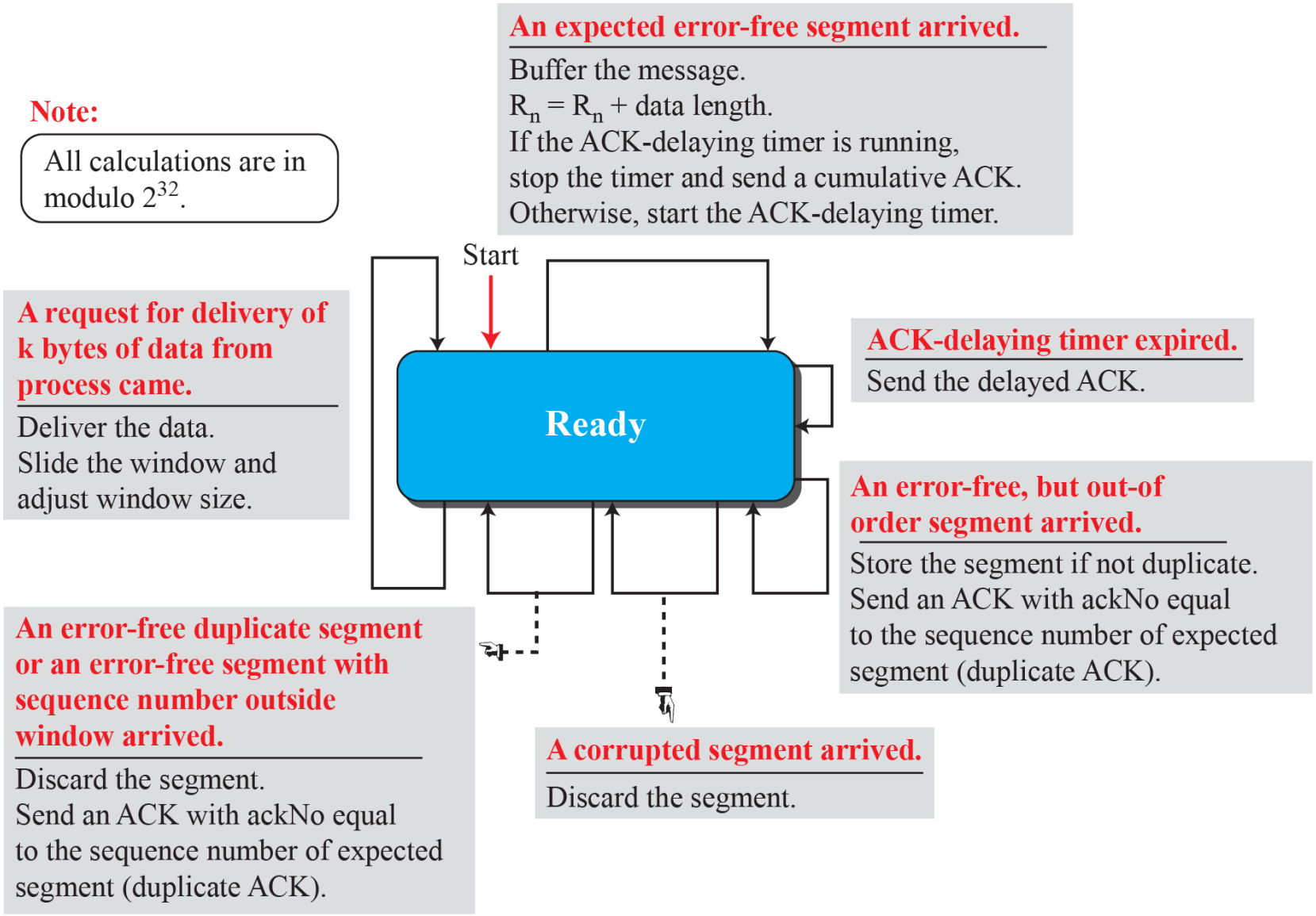


Figure 24.23: Simplified FSM for the TCP receiver side



24.3.9 TCP Congestion Control

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

Figure 24.29: *Slow start, exponential increase*

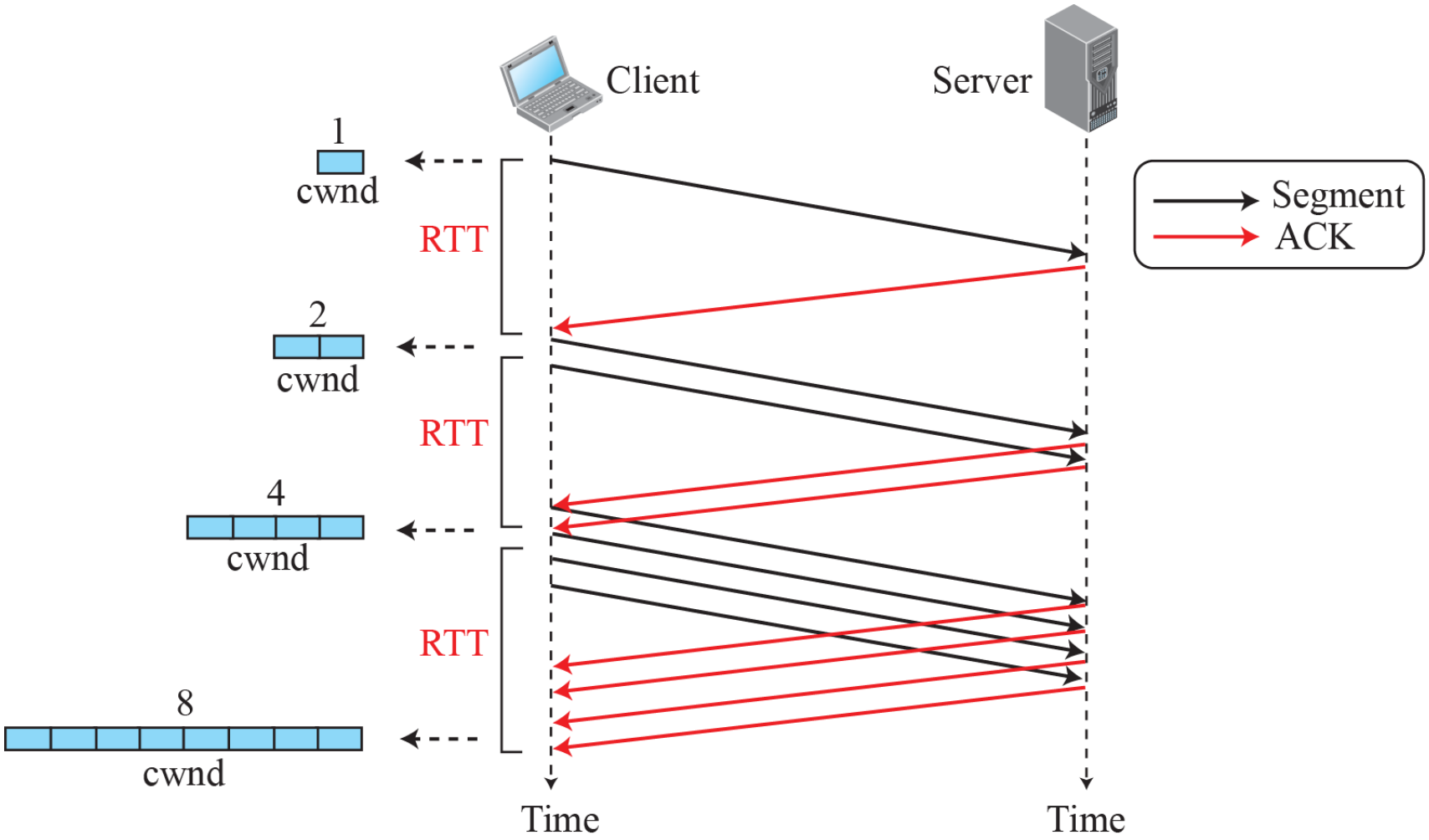


Figure 24.30: Congestion avoidance, additive increase

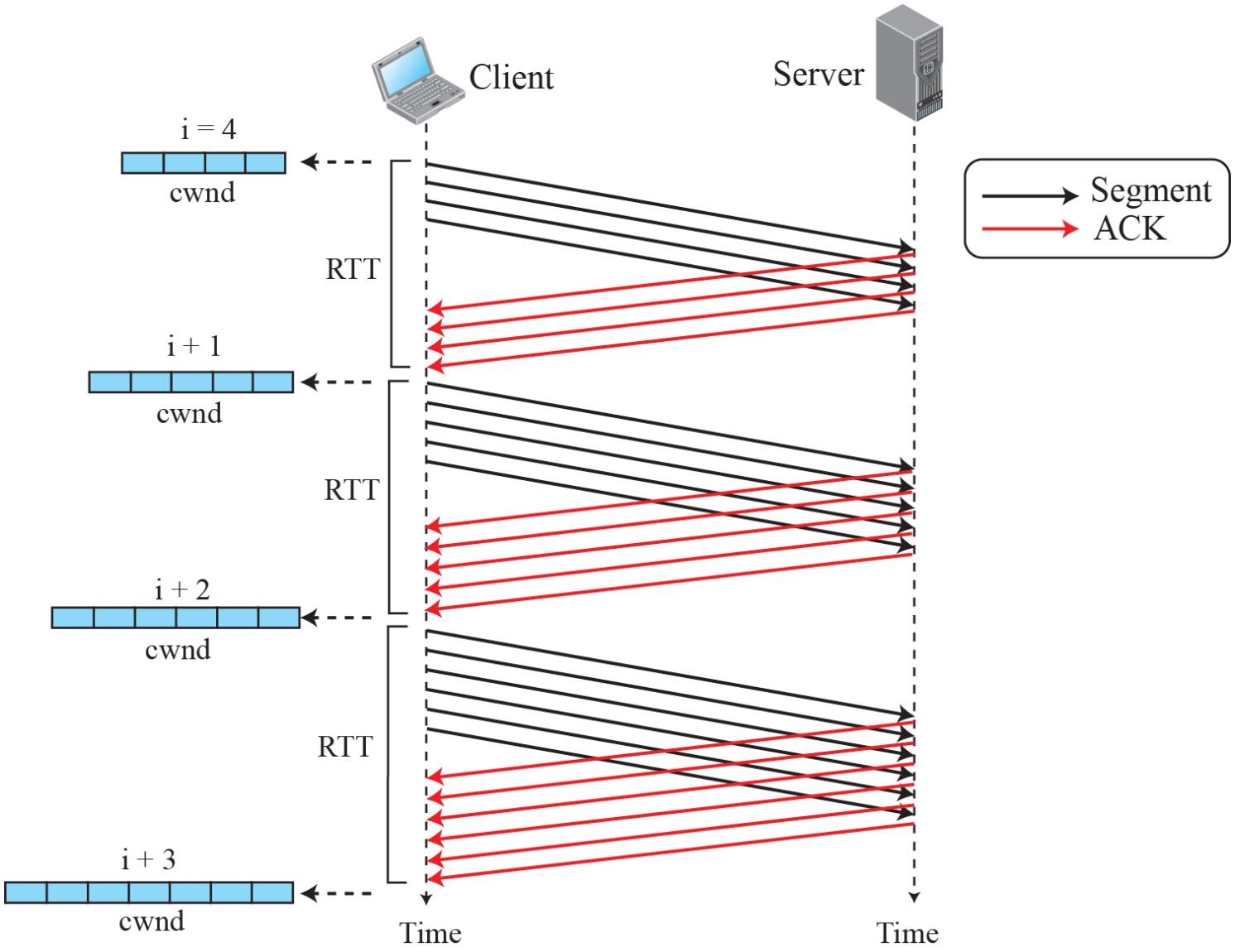
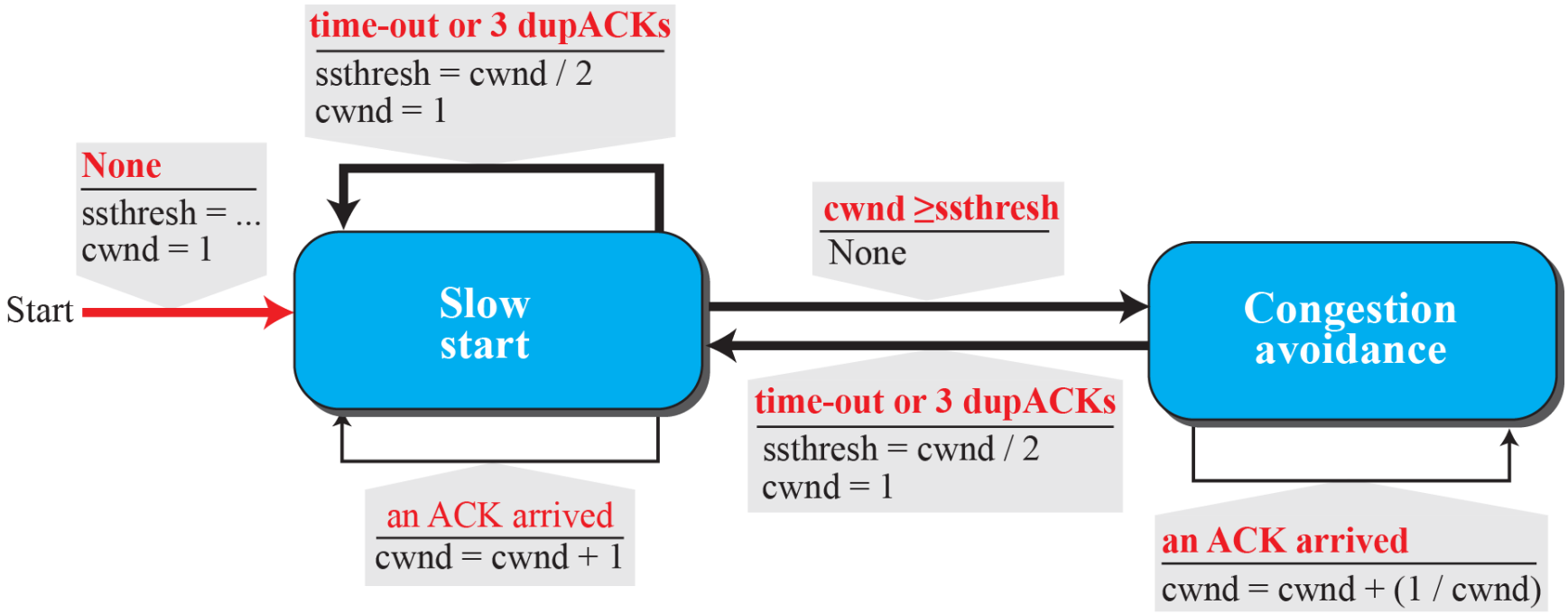


Figure 24.31: FSM for Taho TCP



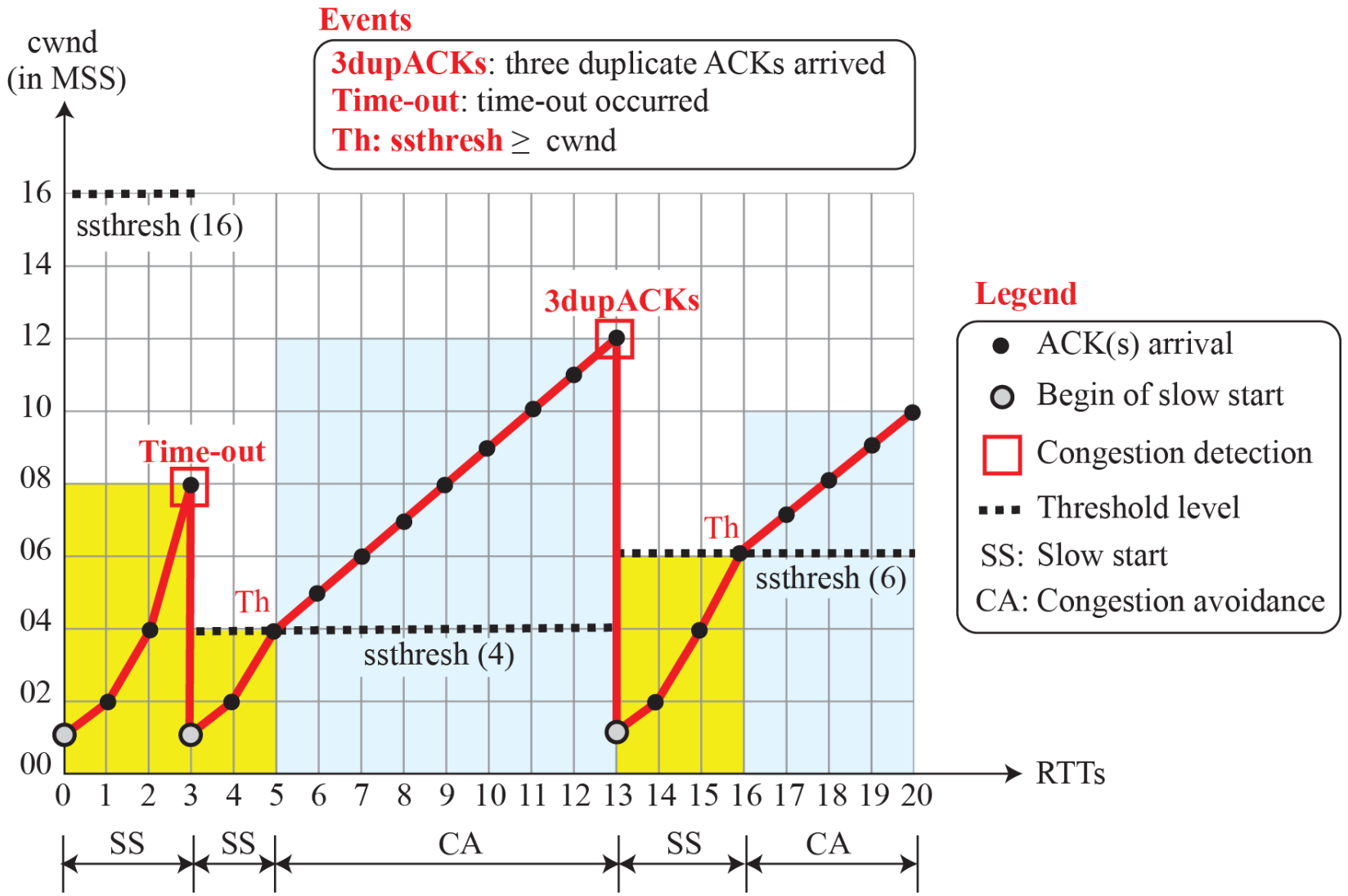
Example 24.9

Figure 24.32 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* = 24. 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 24.9 (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.

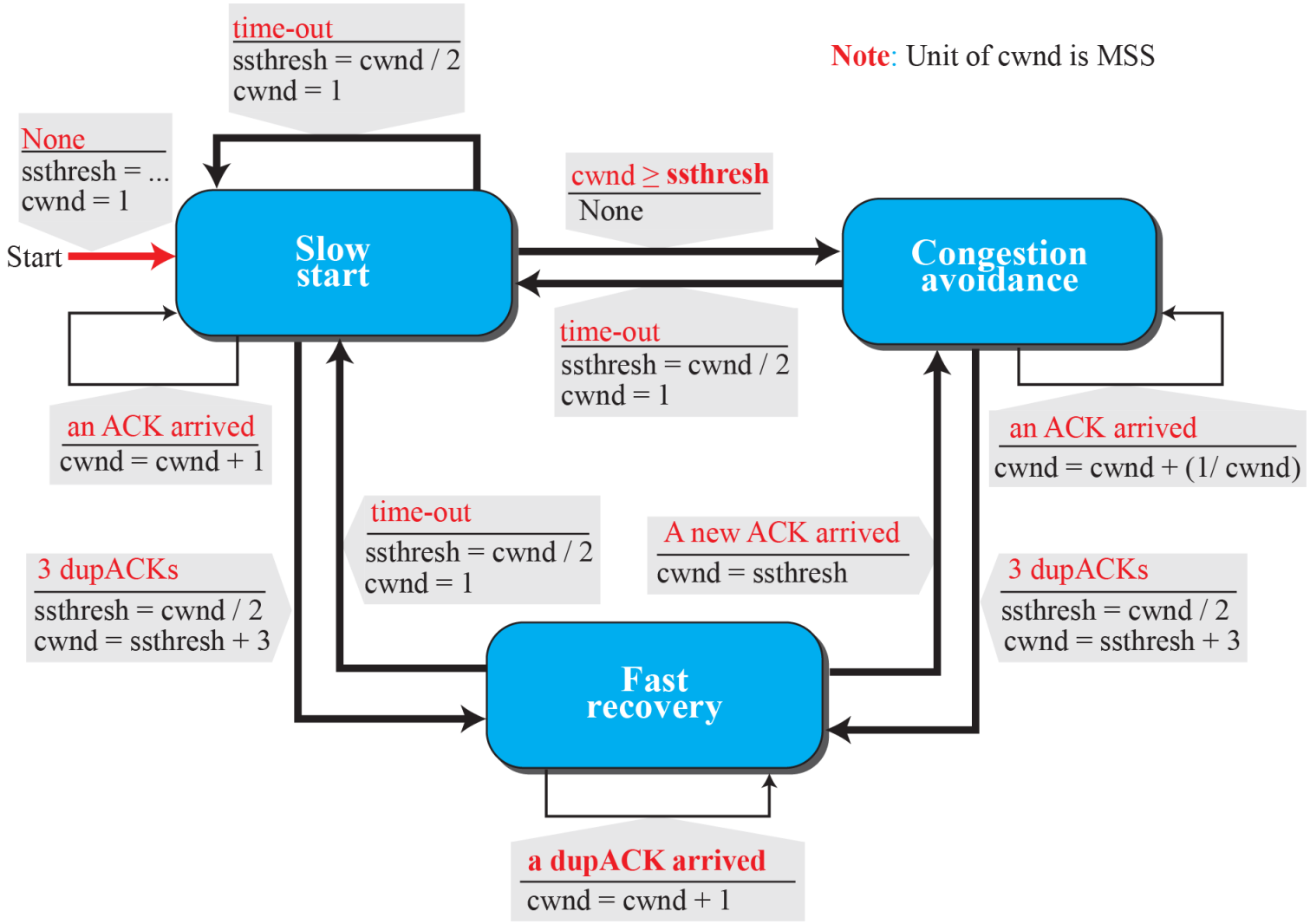
Figure 24.32: Example of Taho TCP



Exercise

The *ssthresh* value for a Taho TCP station is set to 6 MSS. The station now is in the slow-start state with *cwnd* = 4 MSS. Show the values of *cwnd*, *sstresh*, and the state of the station before and after each of following events: four con- secutive nonduplicate ACKs arrived followed by a time-out, and followed by three nonduplicate ACKs.

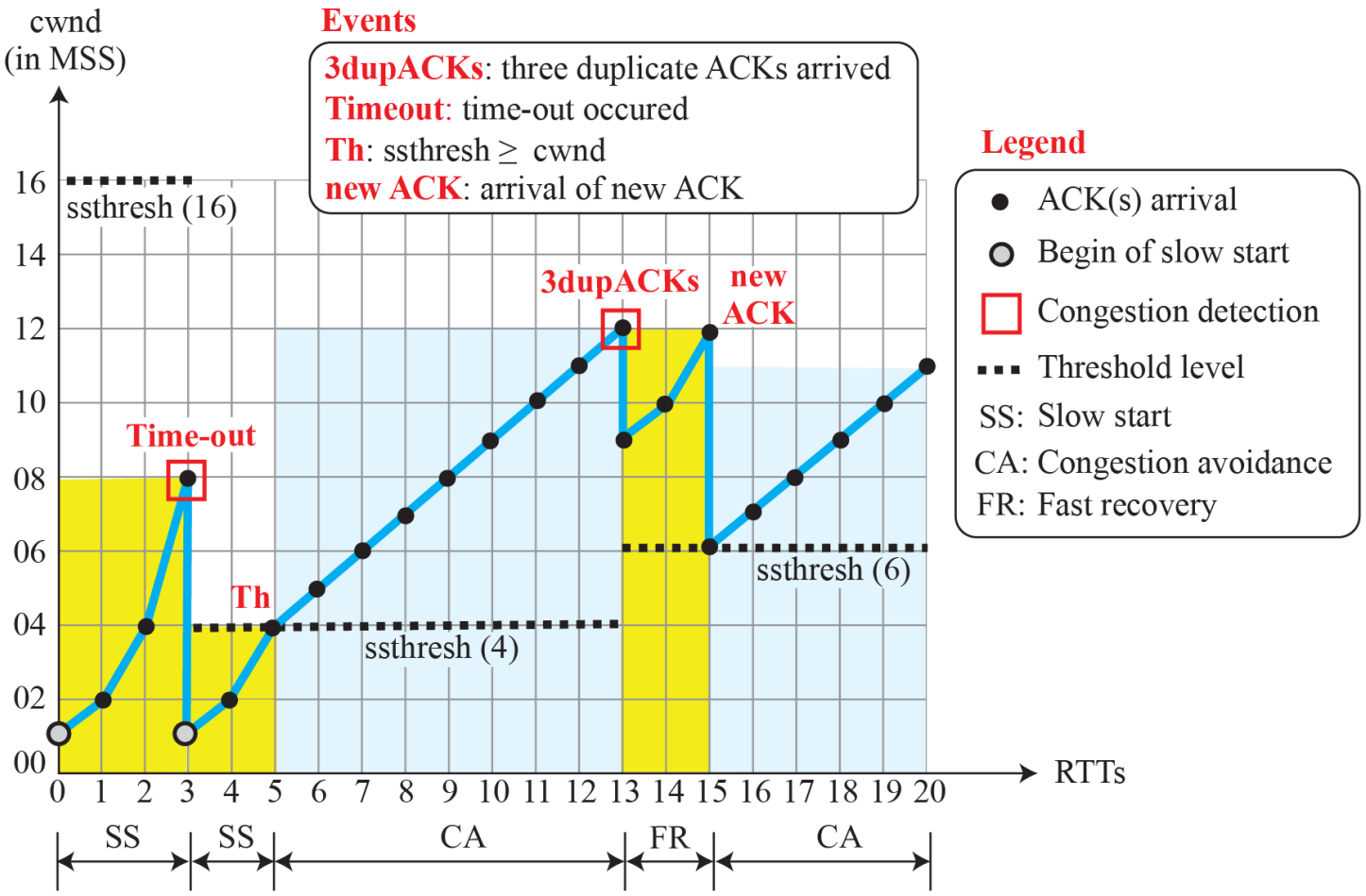
Figure 24.33: FSM for Reno TCP



Example 24.10

Figure 24.34 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.

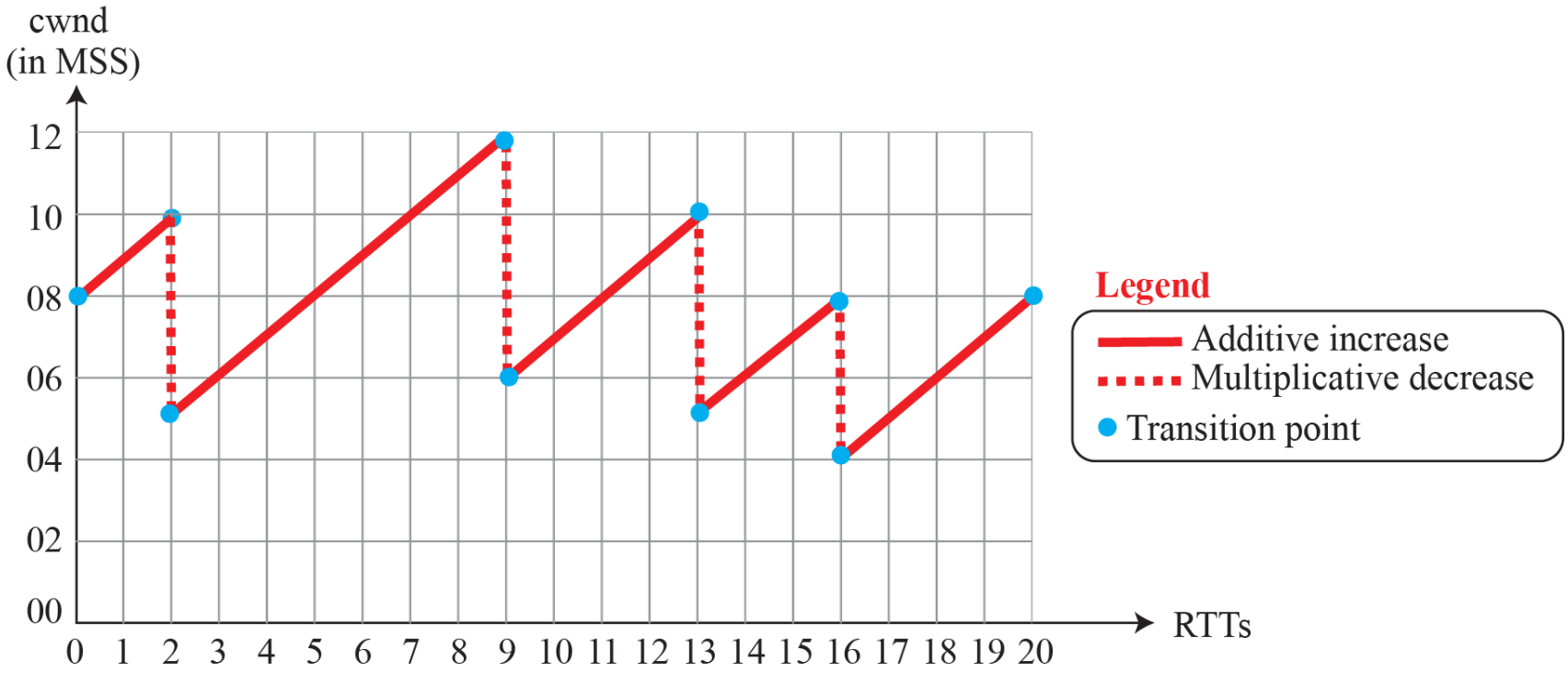
Figure 24.34: Example of a Reno TCP



Exercise

The ssthresh value for a Reno TCP station is set to 8 MSS. The station is now in the slow-start state with cwnd = 5 MSS and ssthresh = 8 MSS. Show the values of cwnd, ssthresh, and the current and the next state of the station after the following events: three consecutive nonduplicate ACKs arrived, followed by five duplicate ACKs, followed by two nonduplicate ACKs, and followed by a time-out.

Figure 24.35: Additive increase, multiplicative decrease (AIMD)



Example 24.11

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

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

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

Exercise

In a TCP connection, the window size fluctuates between 60,000 bytes and 30,000 bytes.

If the average RTT is 30 ms, what is the throughput of the connection?

Retransmission Timer Management

- static timer likely too long or too short
- estimate round trip delay by observing pattern of delay for recent segments
- set time to value a bit greater than estimate
- simple average over a number of segments
- exponential average using time series (RFC793)
- RTT Variance Estimation (Jacobson's algorithm)

Retransmission Timer

- **Simple Average**
  - **RTT(i): round-trip time observed for the i^{th} transmitted segment**
  - **ARTT(K): average round-trip time for the first *K* segments**

$$ARTT(K+1) = \frac{1}{K+1}\sum_{i=1}^{K+1} RTT(i)$$

$$ARTT(K+1) = \frac{K}{K+1} ARTT(K) + \frac{1}{K+1} RTT(K+1)$$

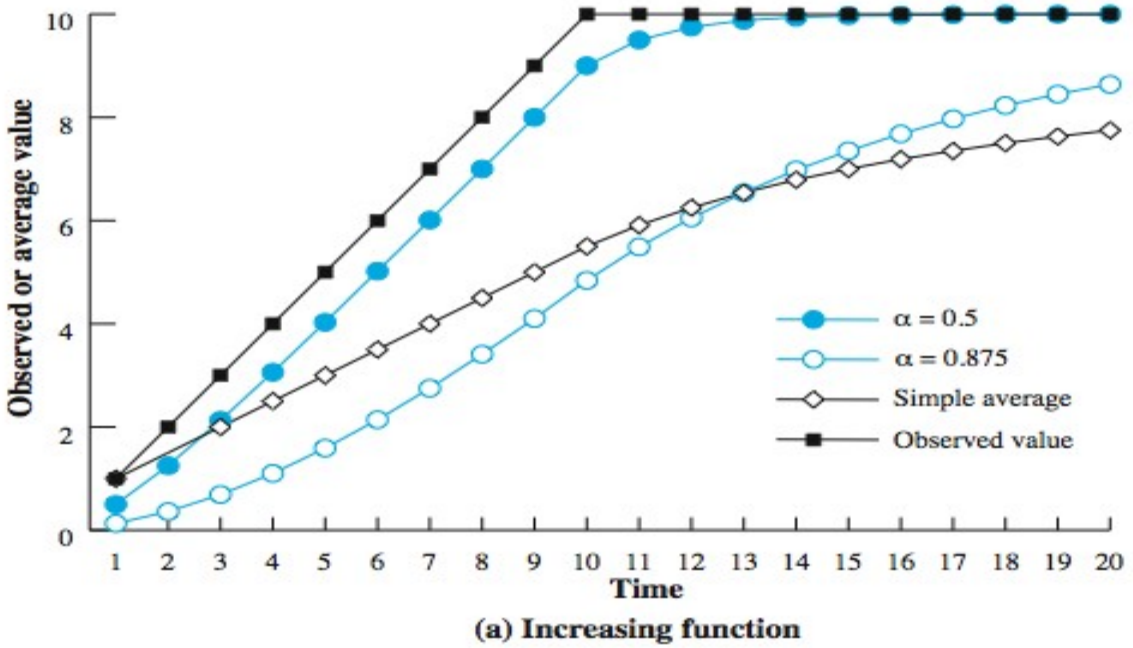
Retransmission Timer

- **Exponential Average**
  - **SRTT: smoothed round-trip time estimate**
  - **RTO: retransmission timer**

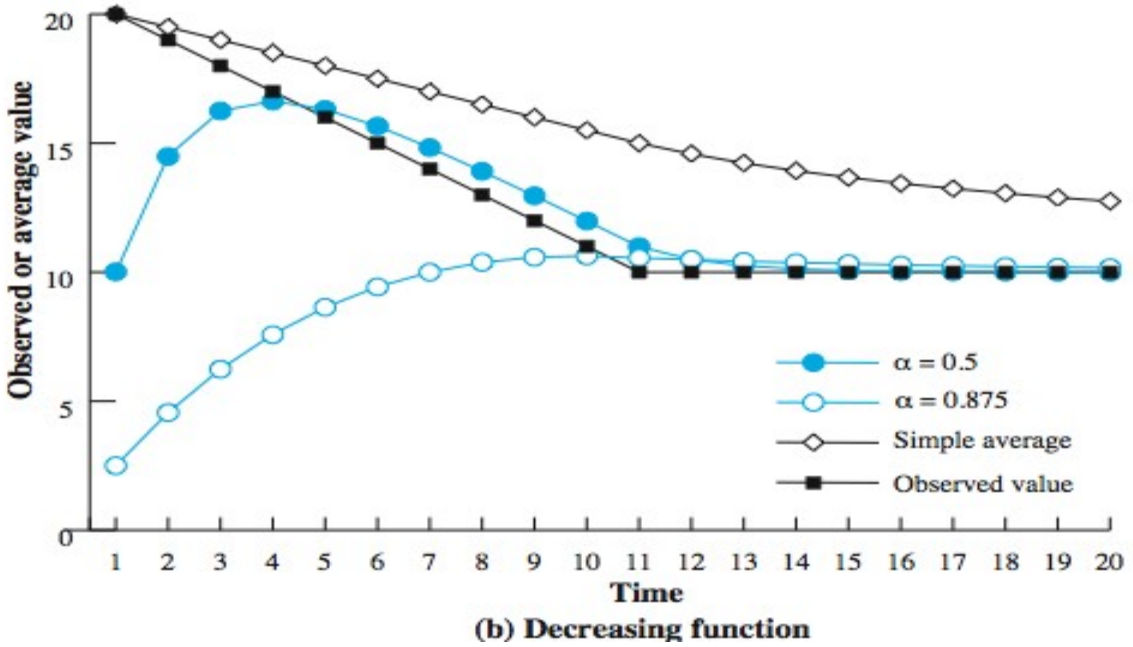
$$SRTT(K+1) = \alpha \times SRTT(K) + (1-\alpha) \times RTT(K+1)$$

$$RTO(K+1) = SRTT(K+1) + \Delta$$

Use of Exponential Averaging



(a) Increasing function



(b) Decreasing function

RTT Variance Estimation

- **AERR(K): sample mean deviation measured at time K**

$$AERR(K+1) = RTT(K+1) - ARTT(K)$$

$$ADEV(K+1) = \frac{1}{K+1}\sum_{i=1}^{K+1}\left|AERR(i)\right|$$

$$= \frac{K}{K+1}ADEV(K) + \frac{1}{K+1}\left|AERR(K+1)\right|$$

RTT Variance Estimation

- **Jacobson's Algorithm**

$$SRTT(K+1) = (1-g) \times SRTT(K) + g \times RTT(K+1)$$

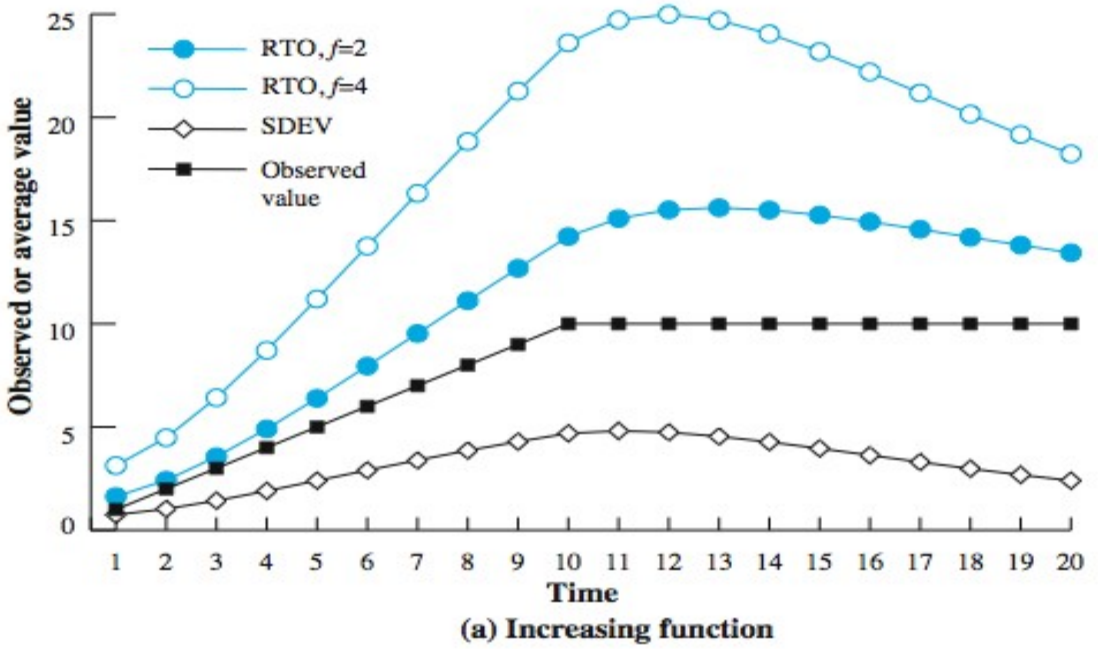
$$SERR(K+1) = RTT(K+1) - SRTT(K)$$

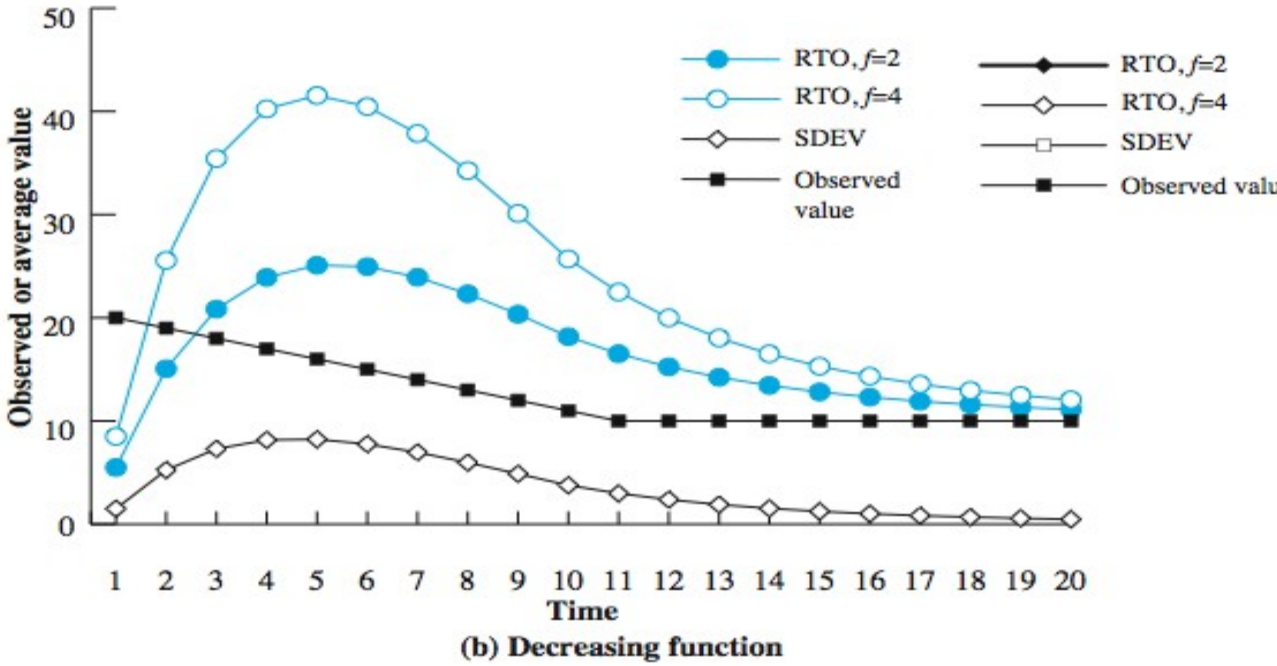
$$SDEV(K+1) = (1-h) \times SDEV(K) + h \times |SERR(K+1)|$$

$$RTO(K+1) = SRTT(K+1) + f \times SDEV(K+1)$$

g = 1/8 = 0.125, h = ¼ = 0.25, f = 2

Jacobson's RTO Calculation





Jacobson's RTO

- **Smoothed RTT**

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

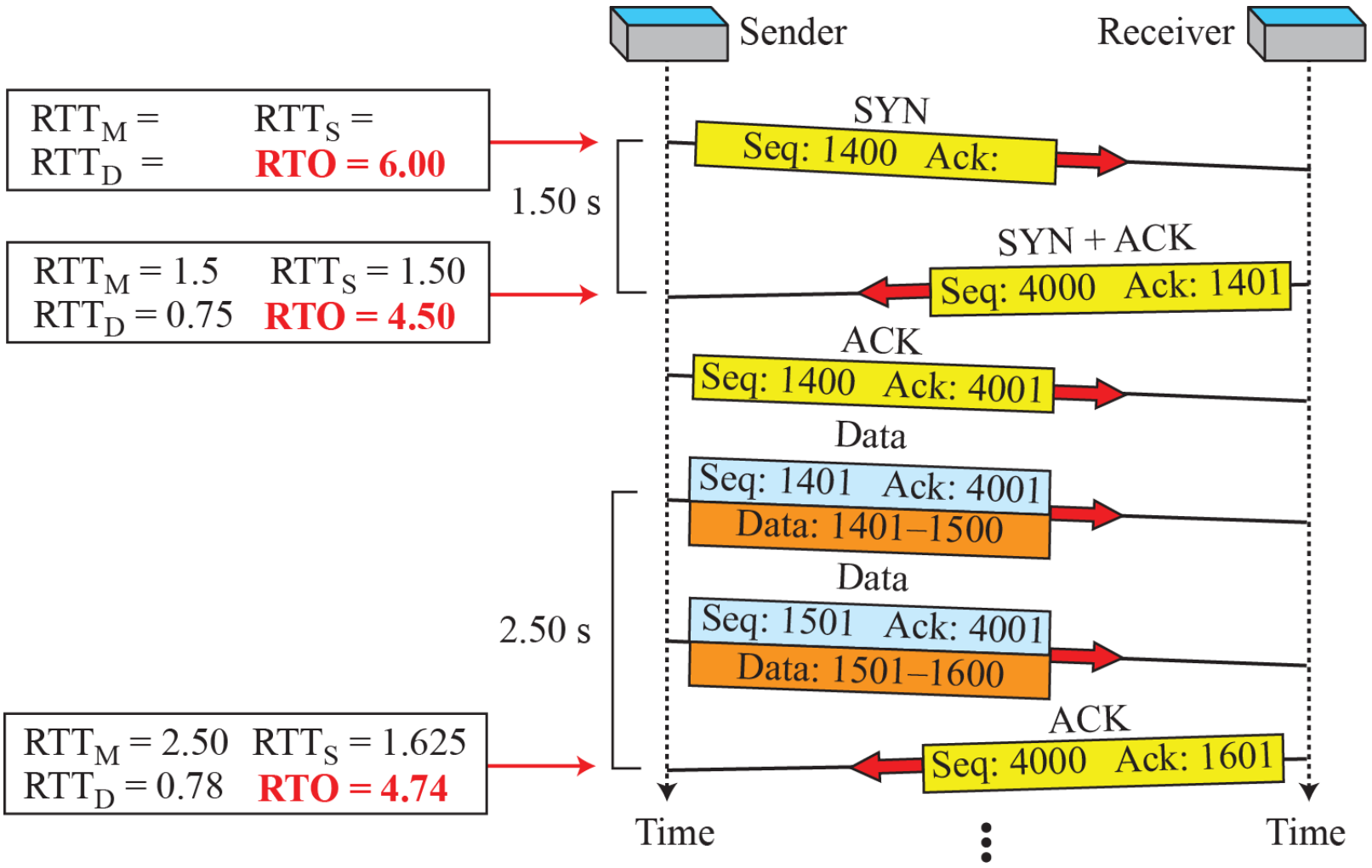
- **RTT Deviation**

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

- **RTO**

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

Figure 24.36: Example 3.22



Example 24.12

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.

24. 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:

RTO = 6

Example 24.12 (continued)

2. When the SYN+ACK segment arrives, RTTM is measured and is equal to 24.5 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 24.12 (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.

$$RTT_M = 2.5$$

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

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

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

Exponential RTO Backoff

- timeout probably due to congestion
  - dropped packet or long round trip time
- hence maintaining RTO is not good idea
- better to increase RTO each time a segment is re-transmitted
  - RTO = q*RTO
  - commonly q=2 (binary exponential backoff)
  - as in ethernet CSMA/CD

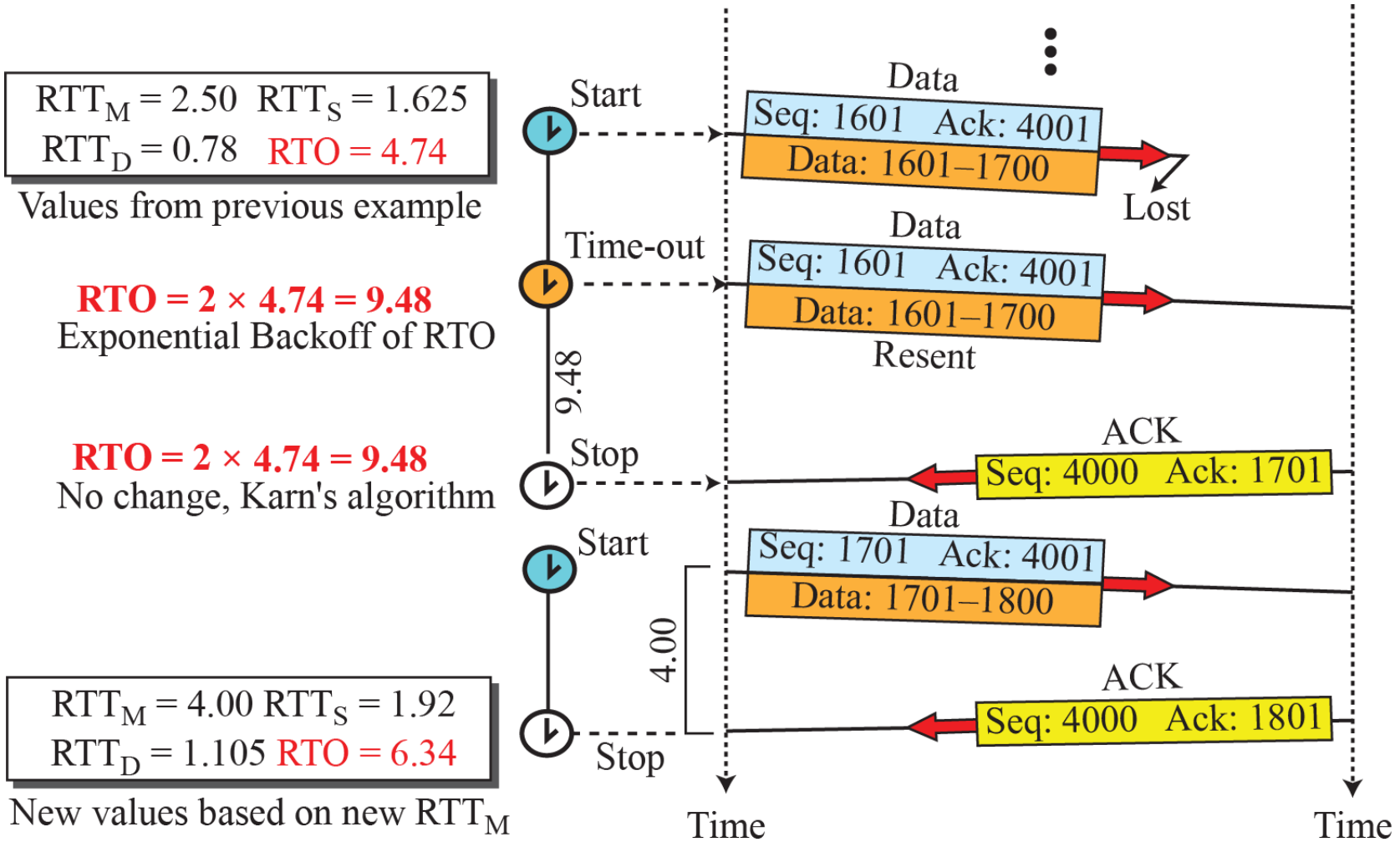
Example 24.13

Figure 24.37 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).

Karn's Algorithm

- if segment is re-transmitted, ACK may be for:
  - first copy of the segment (longer RTT than expected)
  - second copy
- no way to tell
- don't measure RTT for re-transmitted segments
- calculate backoff when re-transmission occurs
- use backoff RTO until ACK arrives for segment that has not been re-transmitted

Figure 24.37: *Example 3.23*



Exercise

If originally RTTS = 14 ms and α is set to 0.2, calculate the new RTTS after the following events (times are relative to event 1):

Event 1: 00 ms Segment 1 was sent.
Event 2: 06 ms Segment 2 was sent.
Event 3: 16 ms Segment 1 was timed-out and resent.
Event 4: 21 ms Segment 1 was acknowledged.
Event 5: 23 ms Segment 2 was acknowledged.

24-4 SCTP

Stream Control Transmission Protocol (SCTP) is a new transport-layer protocol designed to combine some features of UDP and TCP in an effort to create a protocol for multimedia communication.

24.4.1 SCTP Services

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

Figure 24.38 : Multiple-stream concept

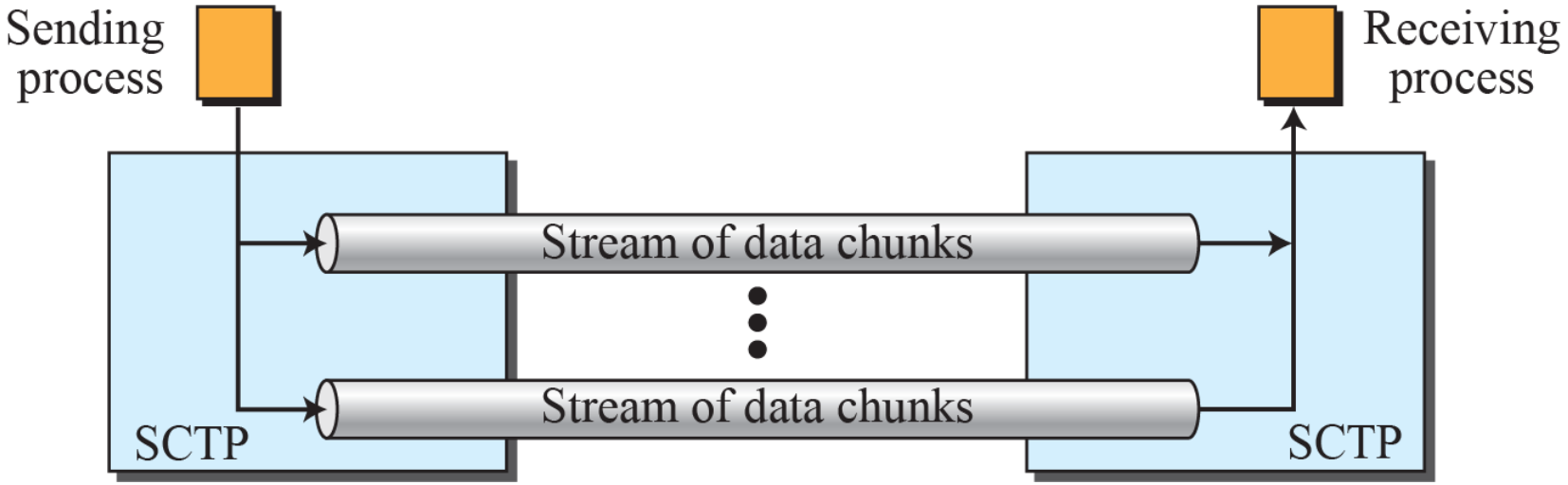
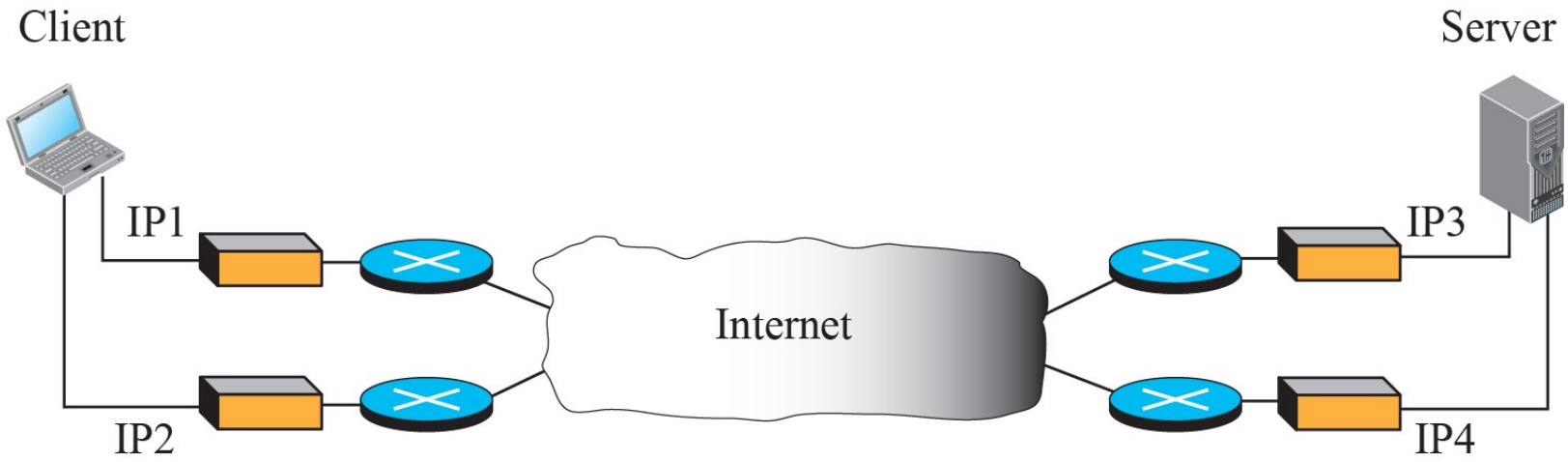


Figure 24.39 : Multihoming concept



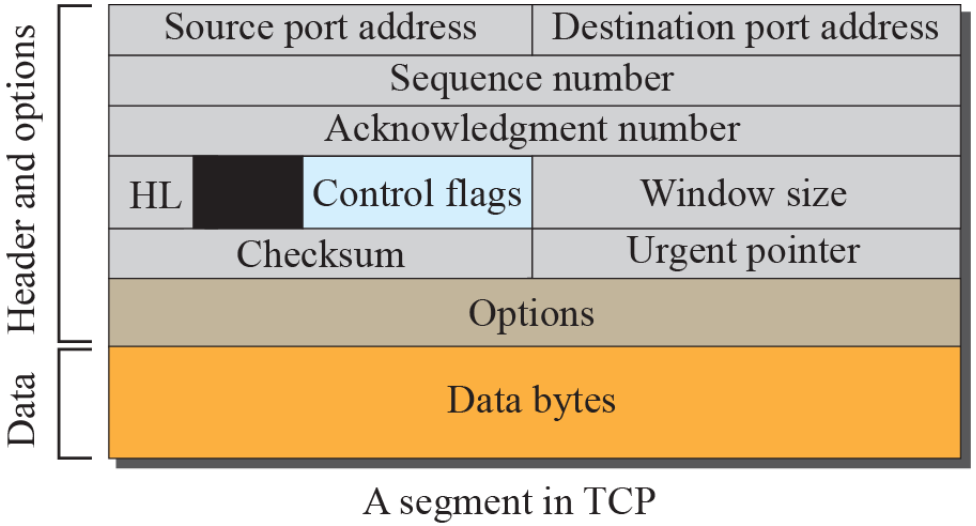
24.4.2 SCTP Features

The following shows the general features of SCTP.

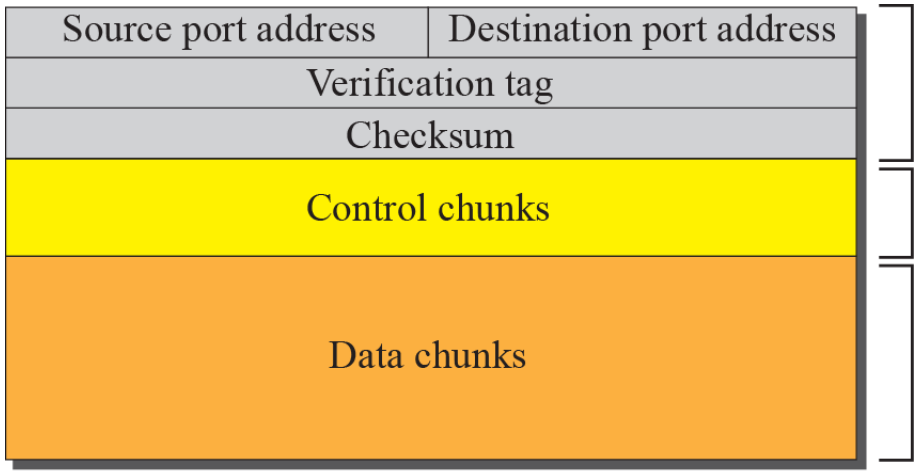
Transmission Sequence Number (TSN)
Stream Identifier (SI)

Stream Sequence Number (SSN)

Figure 24.40 : Comparison between a TCP segment and an SCTP packet

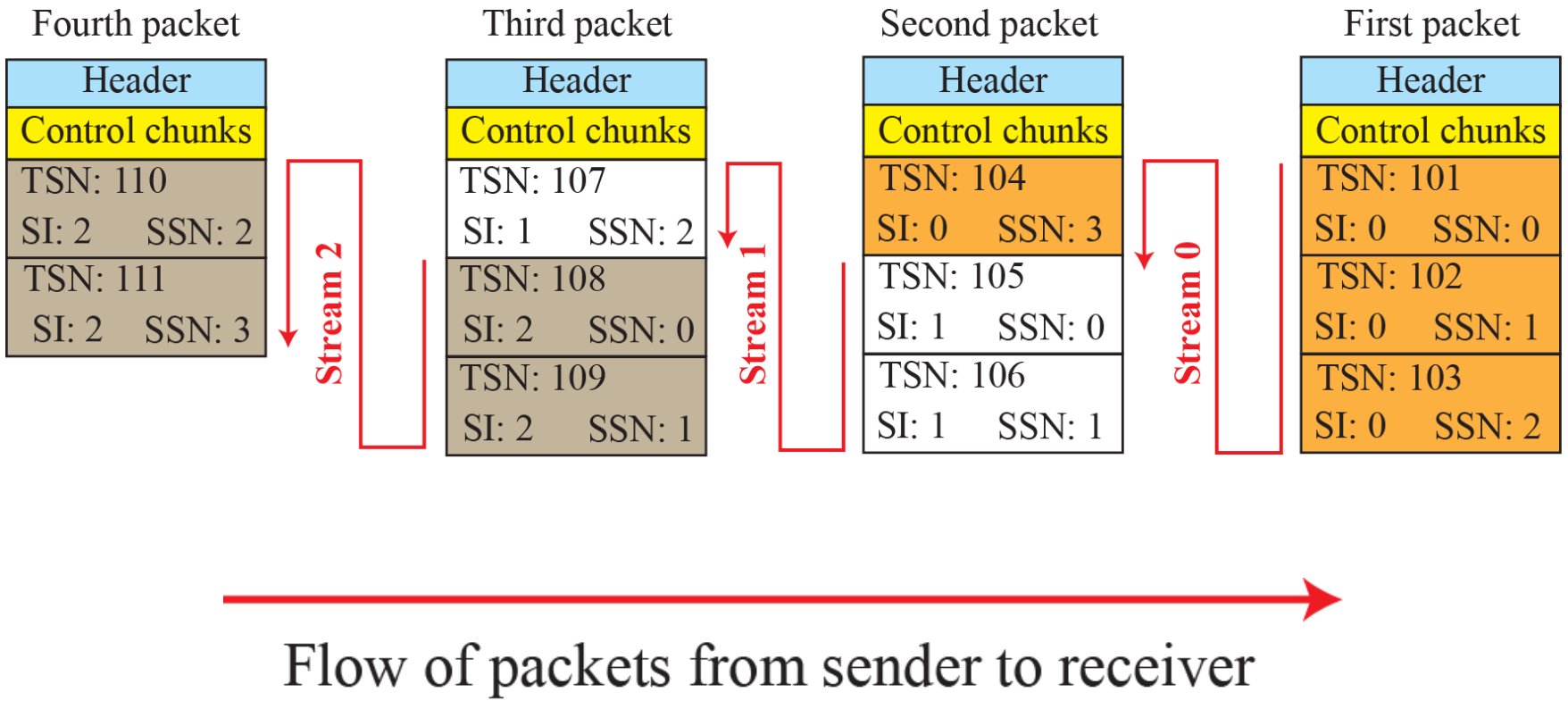


A segment in TCP



A packet in SCTP

Figure 24.41 : Packets, data chunks, and streams



24.4.3 Packet Format

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. A control chunk controls and maintains the association; a data chunk carries user data. In a packet, the control chunks come before the data chunks. Figure 24.42 shows the general format of an SCTP packet.

Figure 24.43 : SCTP packet format

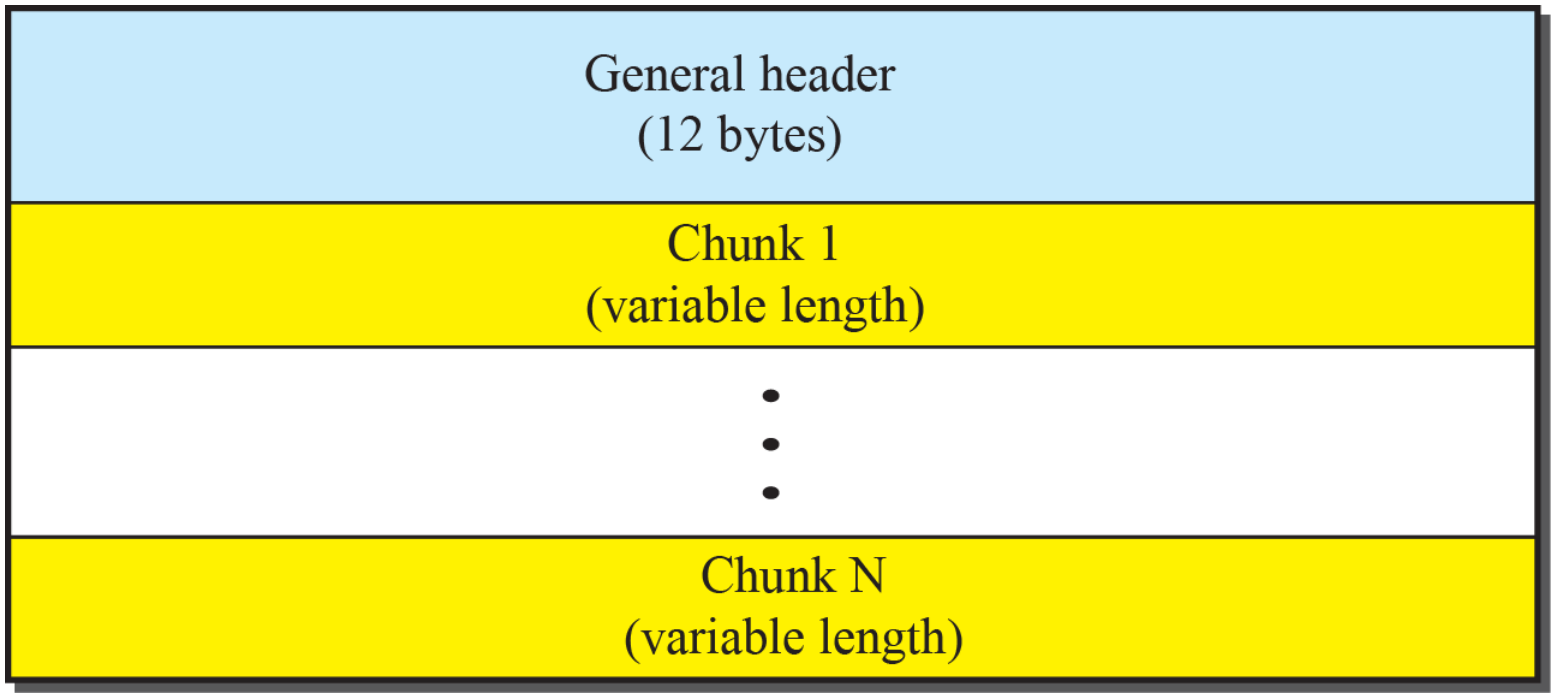


Figure 8.50 : General header

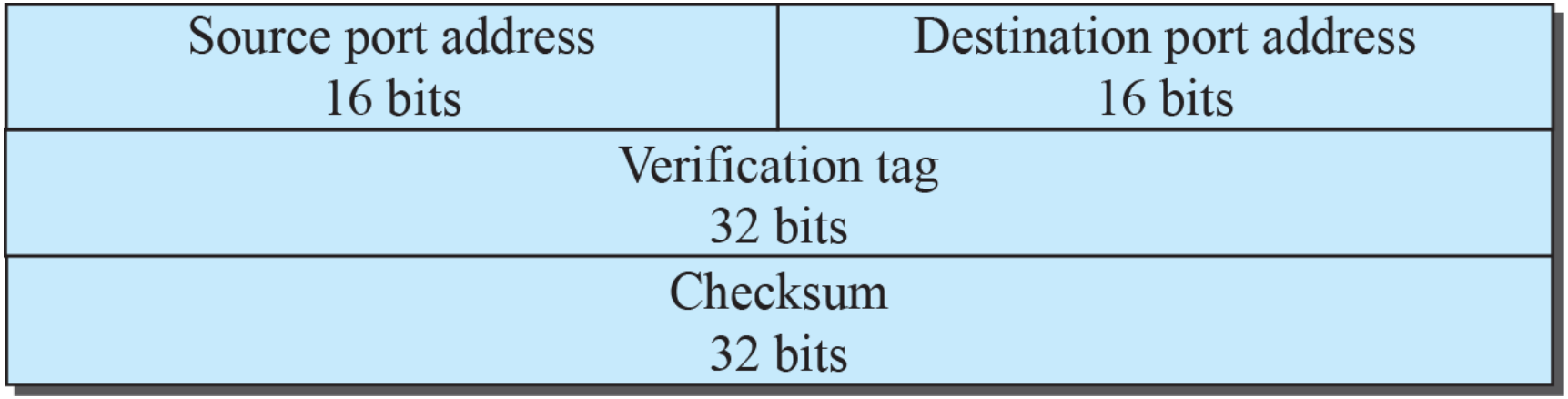


Figure 24.44 : Common layout of a chunk

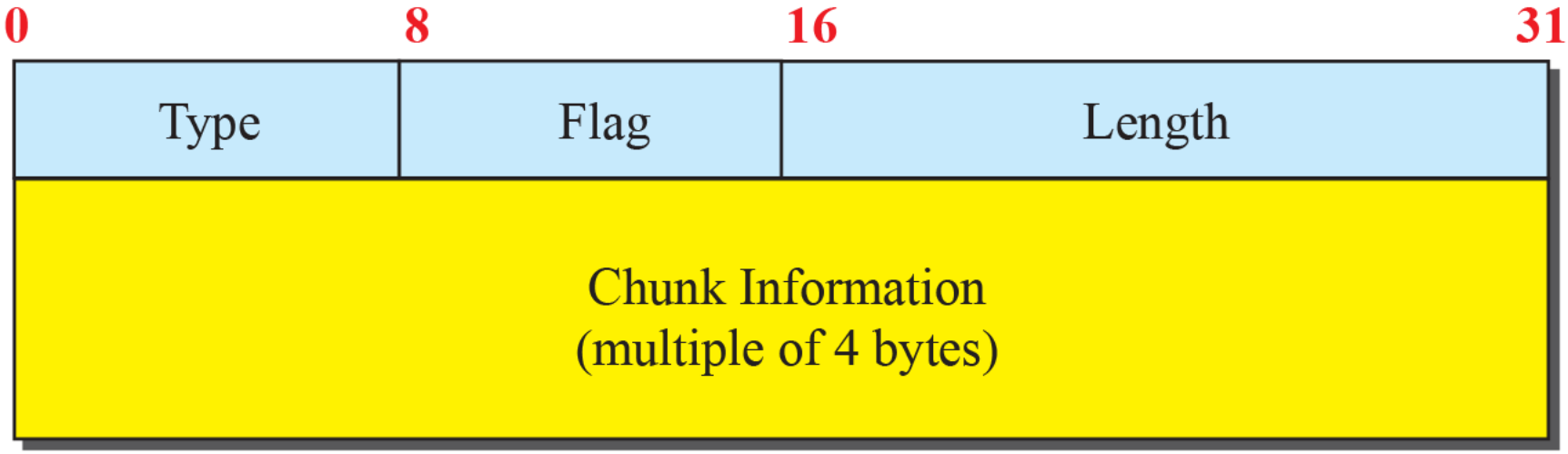


Table 24.3: Chunks

Type	*Chunk*	*Description*
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 liveliness
5	HEARTBEAT ACK	Acknowledges HEARTBEAT chunk
6	ABORT	Aborts 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

24.4.4 An SCTP Association

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

Figure 24.45: Four-way handshaking

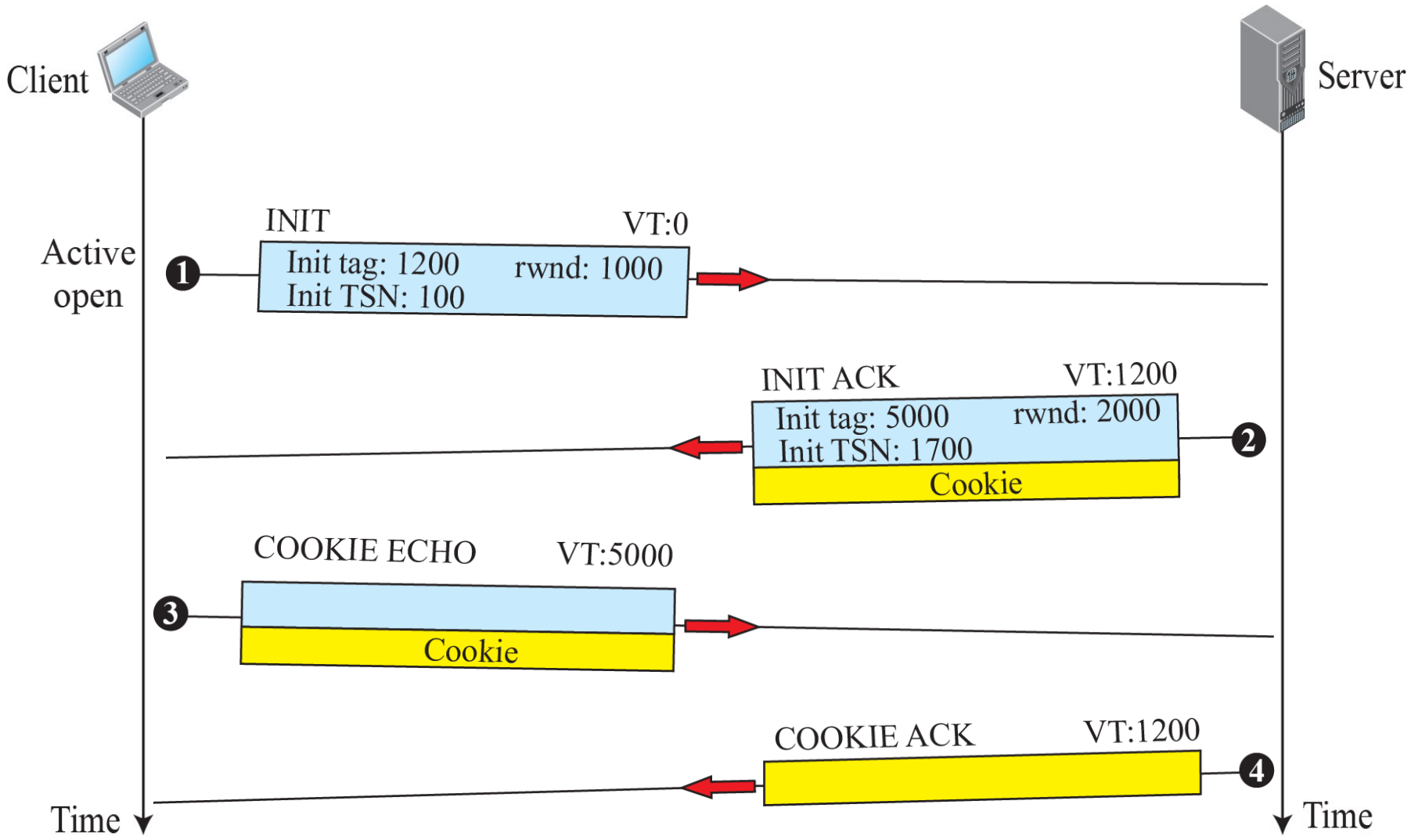
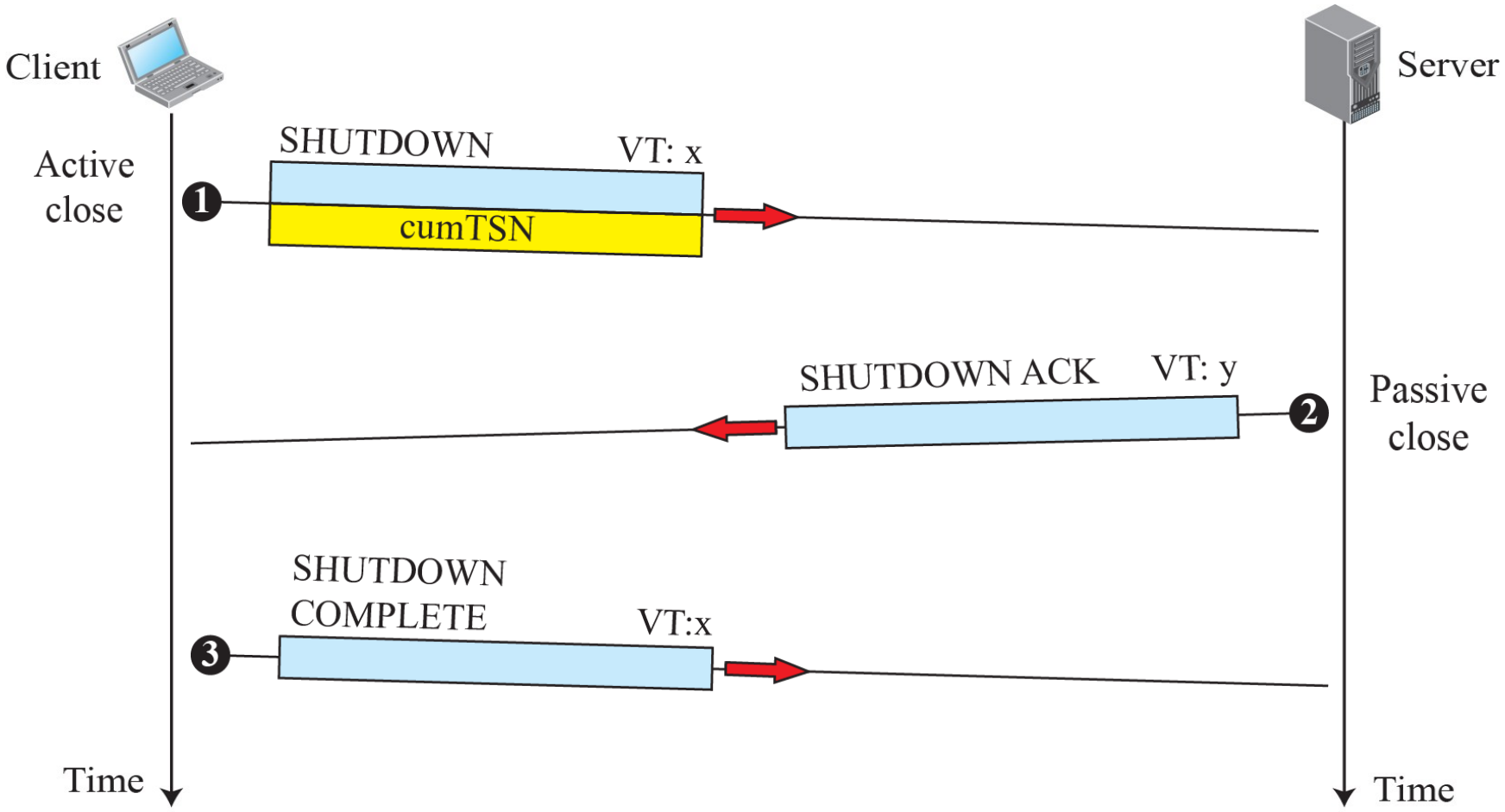


Figure 24.46 : Association termination



24.4.5 Flow Control

Flow control in SCTP is similar to that in TCP. 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. To show the concept, we make some unrealistic assumptions. We assume that there is never congestion in the network and that the network is error free.

Figure 24.47: ***Flow control, receiver site***

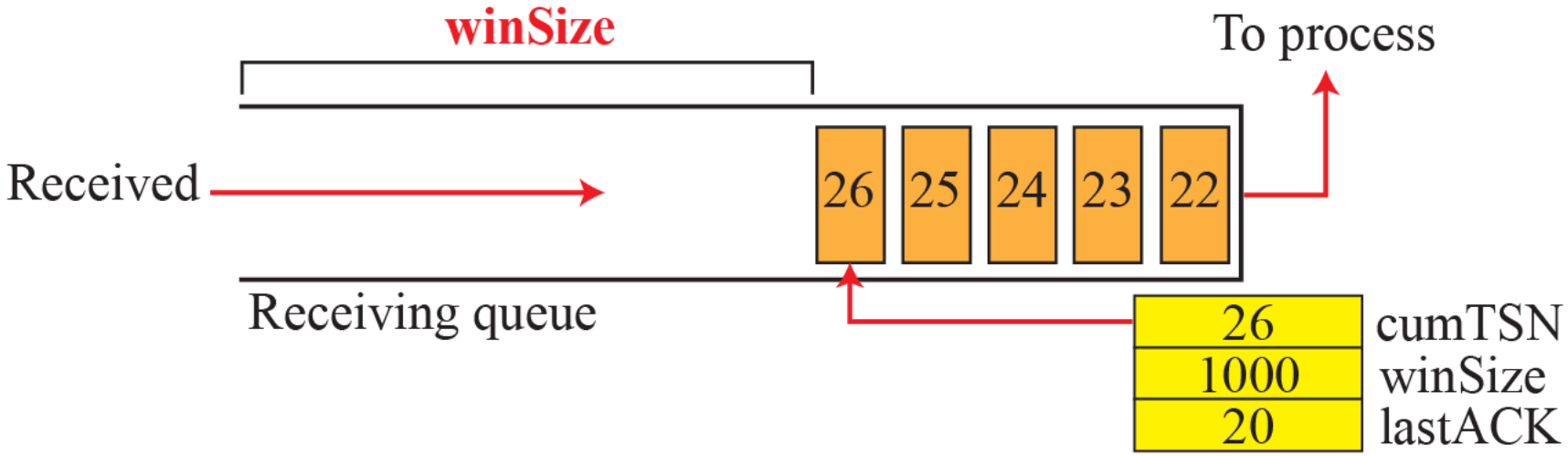
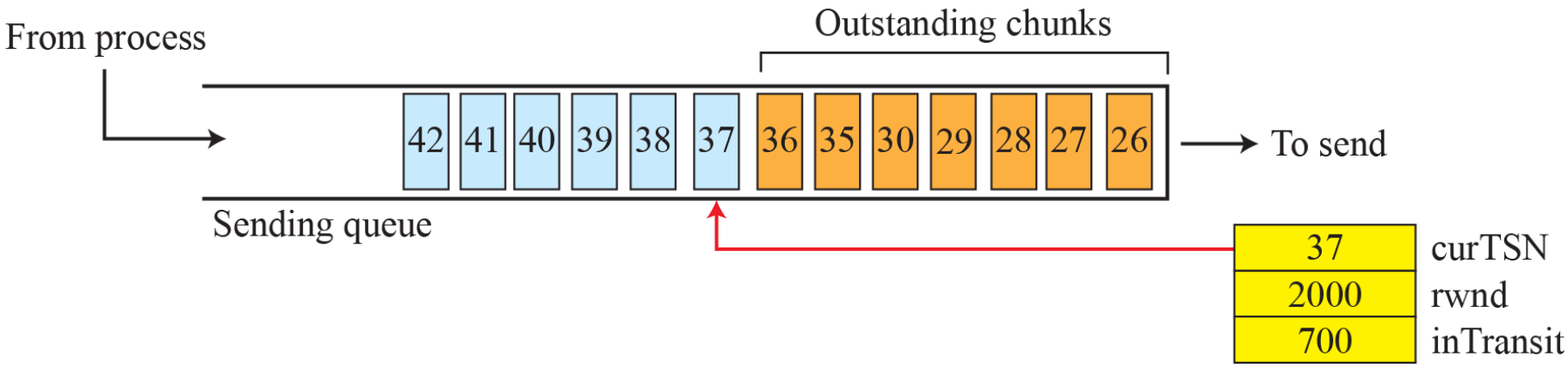


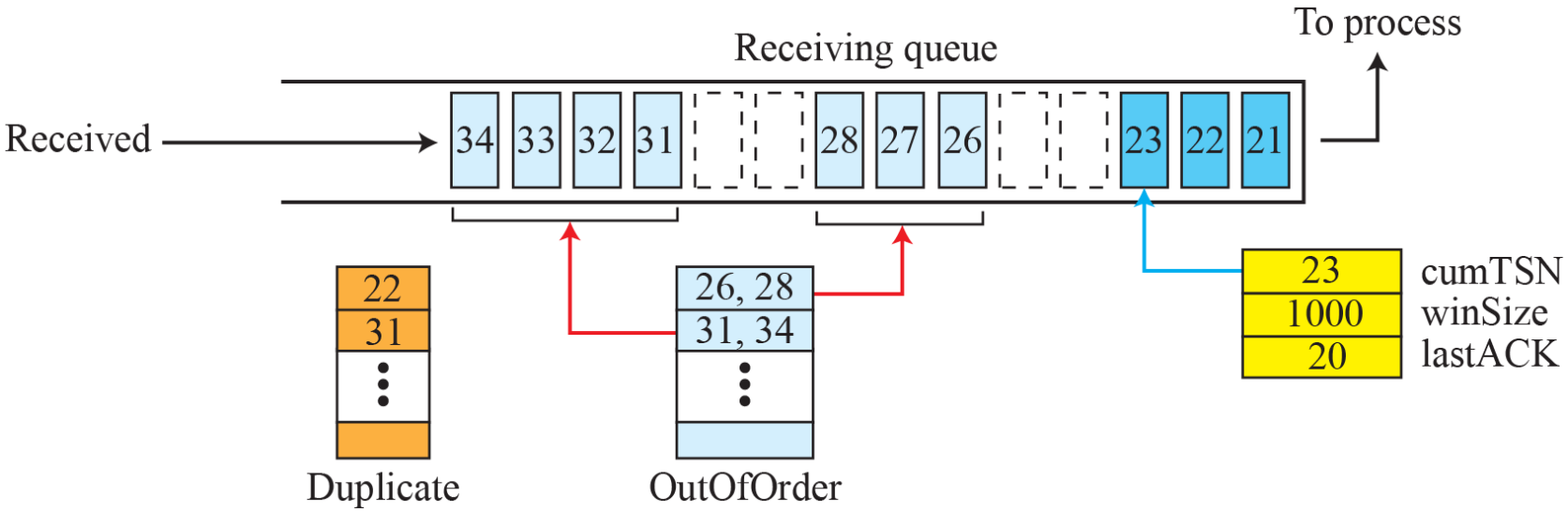
Figure 24.48: Flow control, sender site



24.4.6 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.

Figure 24.49 : Error control, receiver site



SACK chunk

Type: 3	Flag: 0	Length: 32
Cumulative TSN: 23		
Advertised receiver window credit: 1000		
Number of gap ACK blocks: 2		Number of duplicates: 2
Gap ACK block #1 start: 3		Gap ACK block #1 end: 5
Gap ACK block #2 start: 8		Gap ACK block #2 end: 11
Duplicate TSN: 22		
Duplicate TSN: 31		

Numbers are relative to cumTSN

Figure 24.50 : Error control, sender site

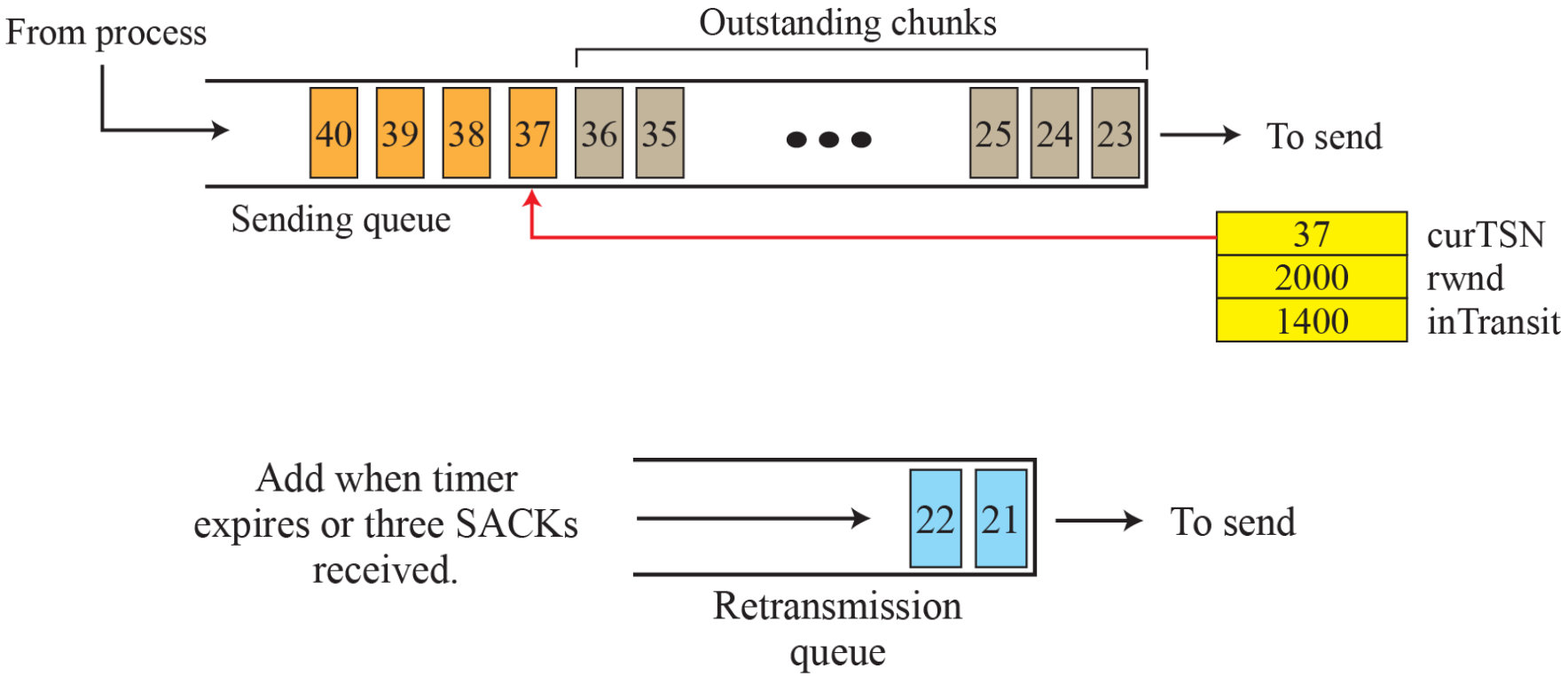


Figure 24.51: New state at the sender site after receiving a SACK chunk