Chapter 15

Transmission Control Protocol (TCP)

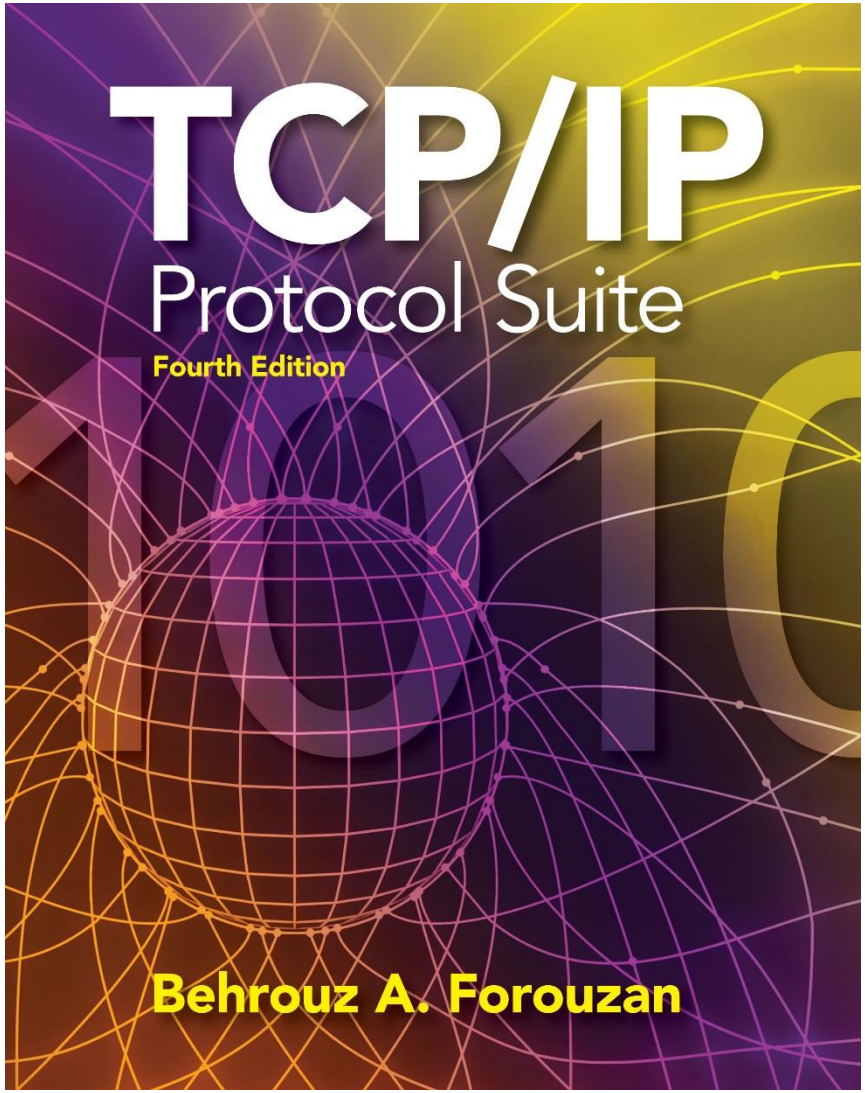
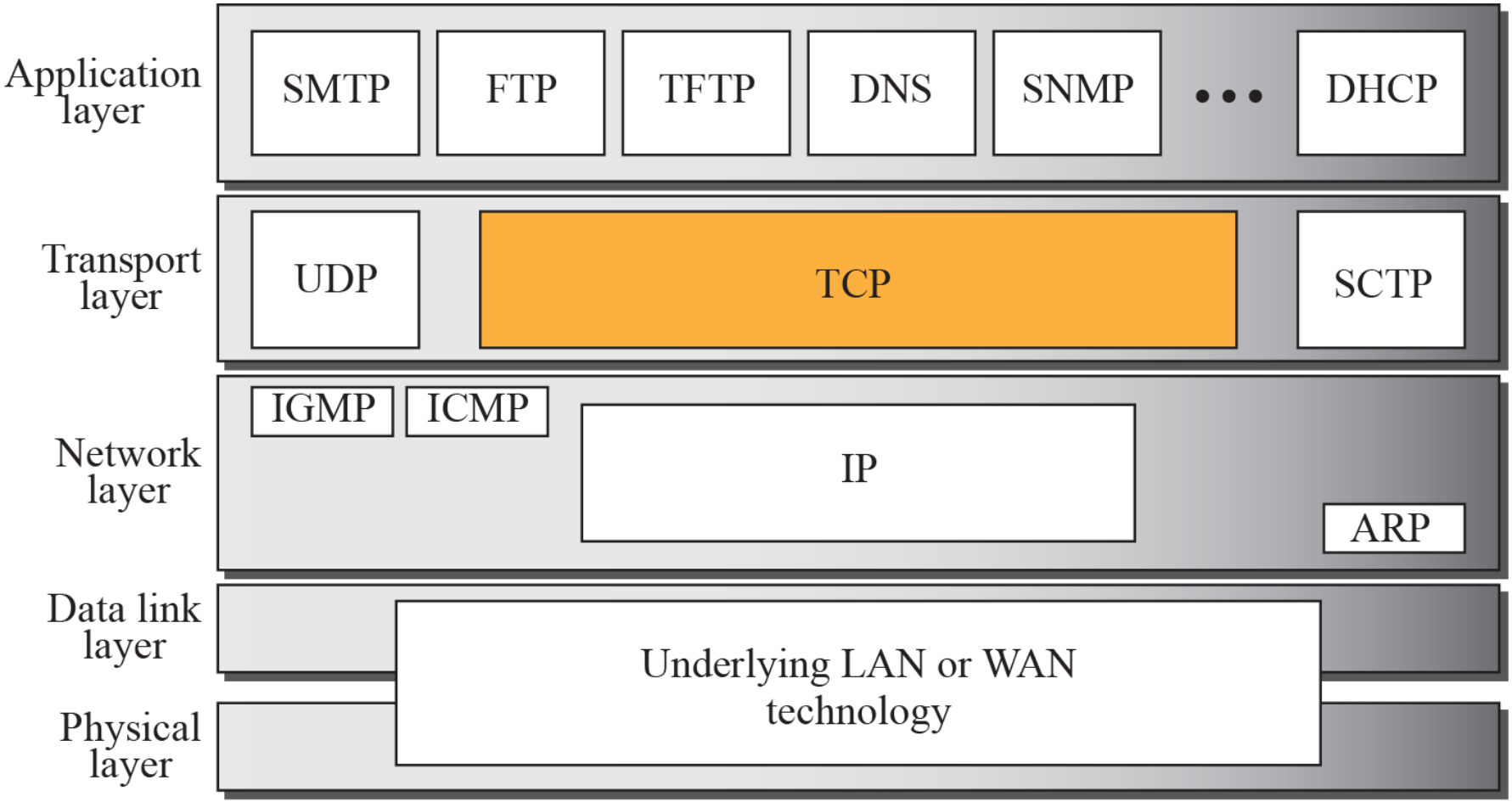


Figure 15.1 *TCP/IP protocol suite*



Note

The bytes of data being transferred in each connection are numbered by TCP.

The numbering starts with an arbitrarily generated number.

Example 15.1

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

Solution

The following shows the sequence number for each segment:

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

Note

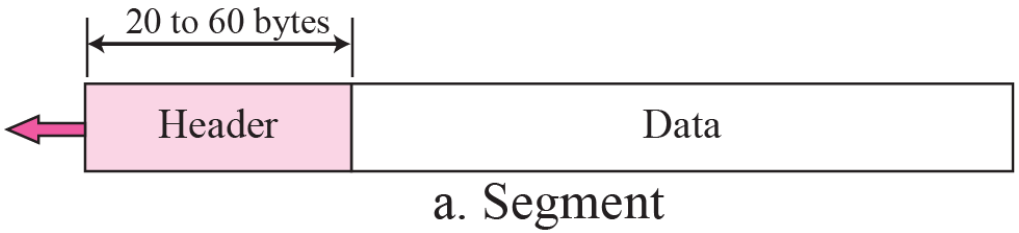
The value in the sequence number field of a segment defines the number assigned to the first data byte contained in that segment.

Note

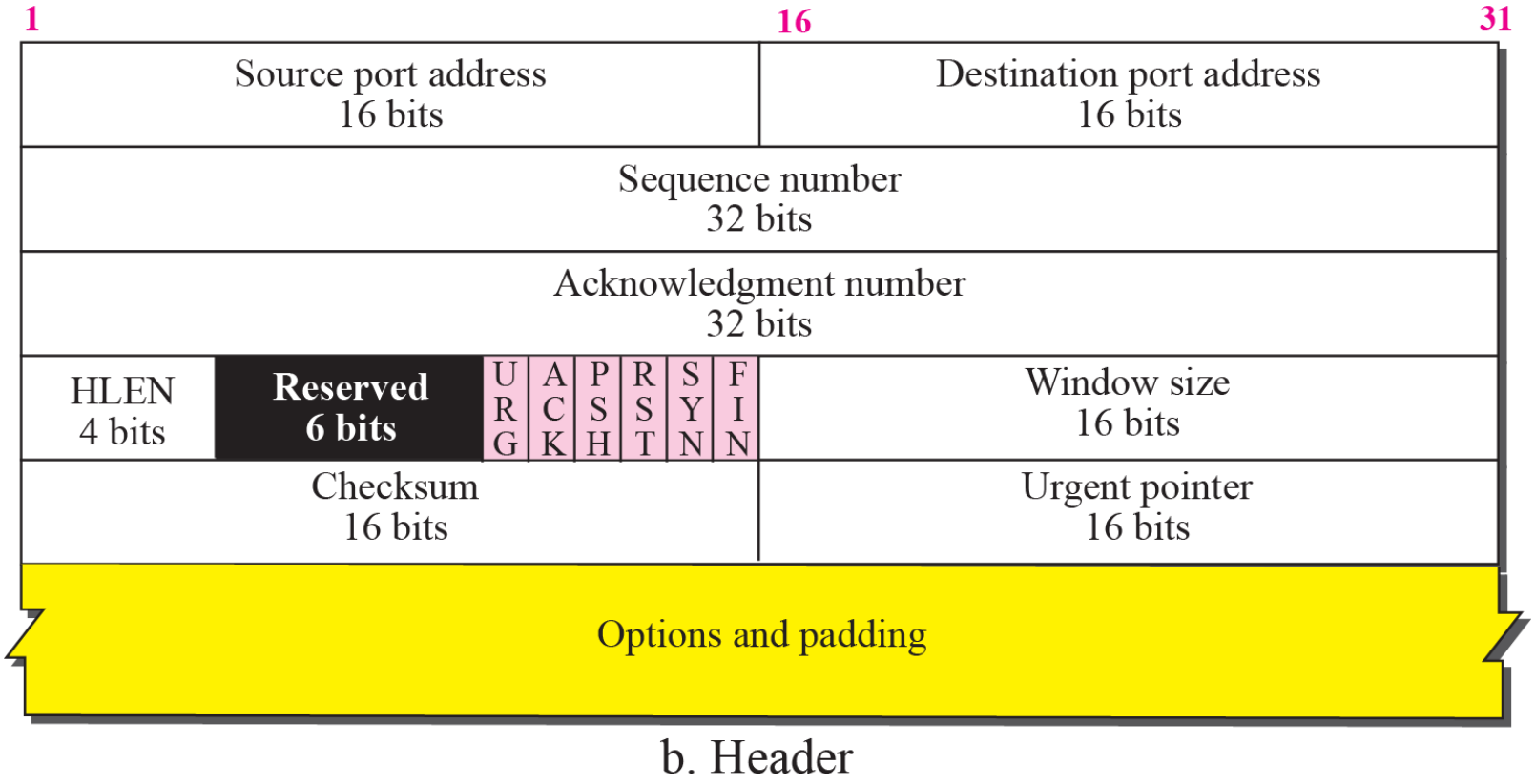
The value of the acknowledgment field in a segment defines the number of the next byte a party expects to receive.

The acknowledgment number is cumulative.

Figure 15.5 *TCP segment format*



a. Segment



b. Header

Figure 15.6 *Control field*

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

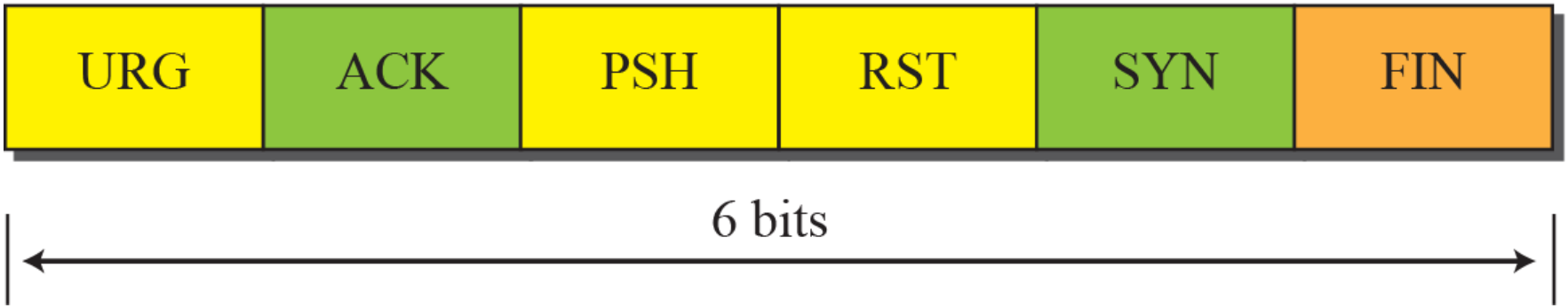


Figure 15.7 *Pseudoheader added to the TCP segment*

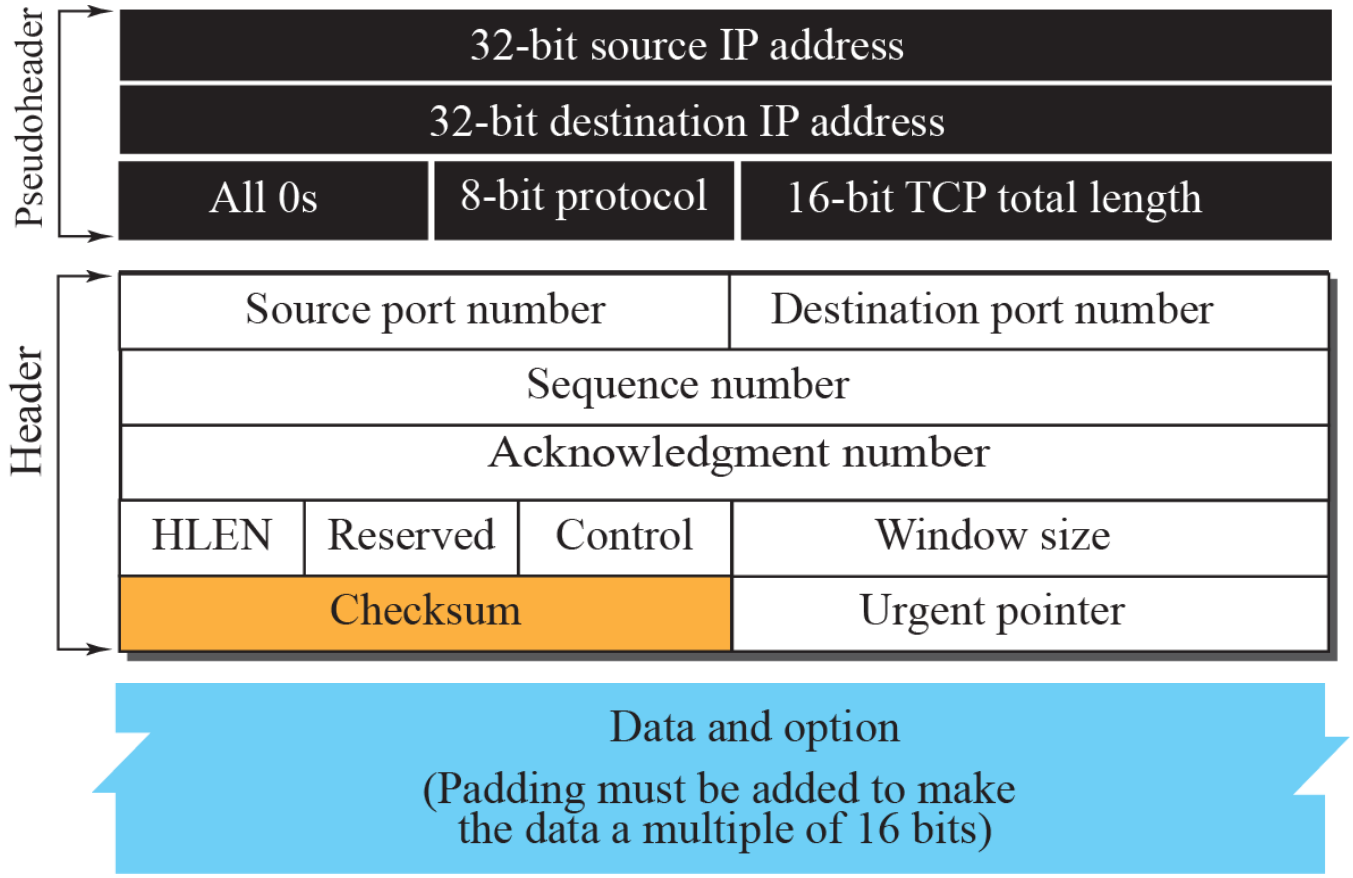


Figure 15.8 *Encapsulation*

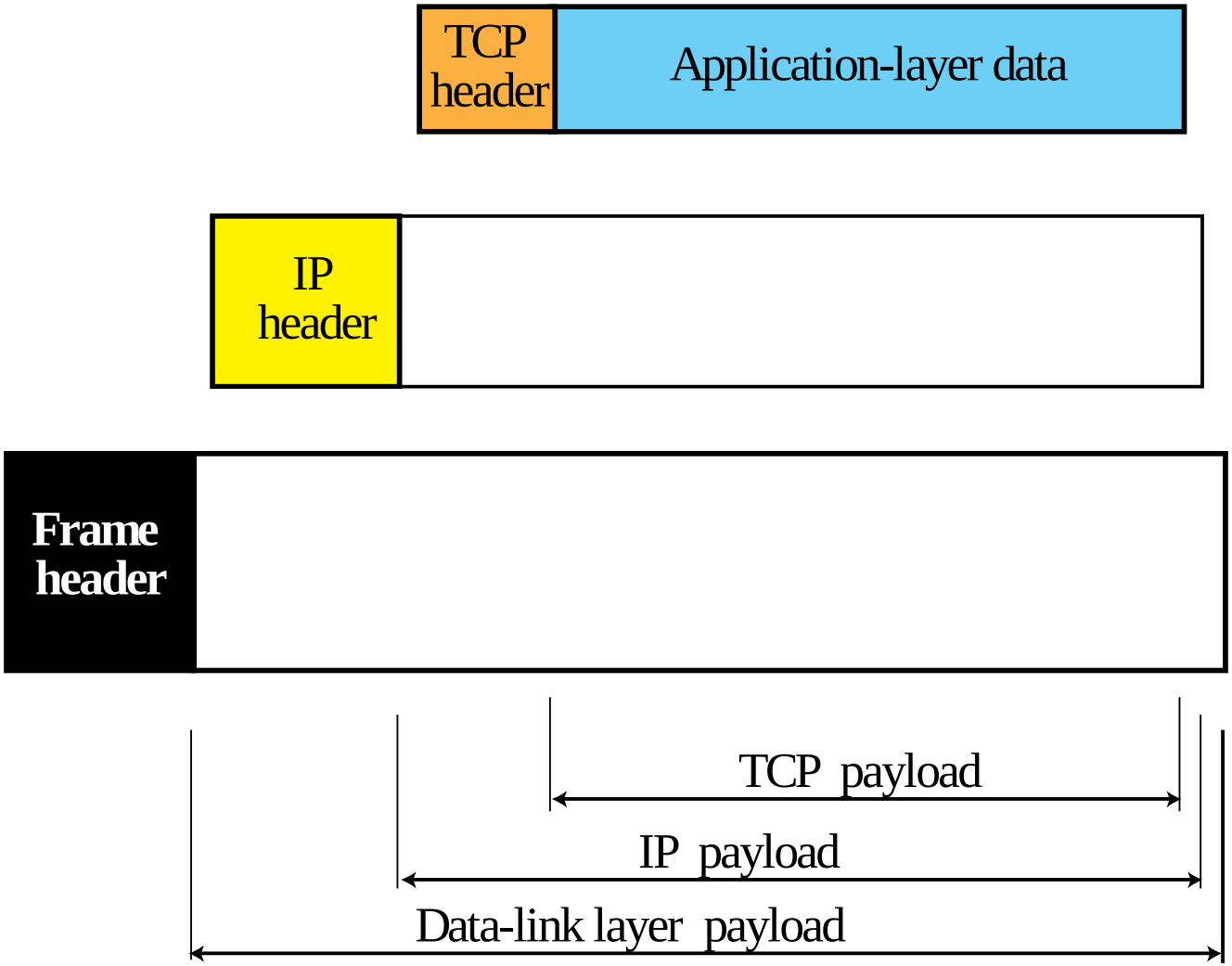


Figure 15.9 *Connection establishment using three-way handshake*

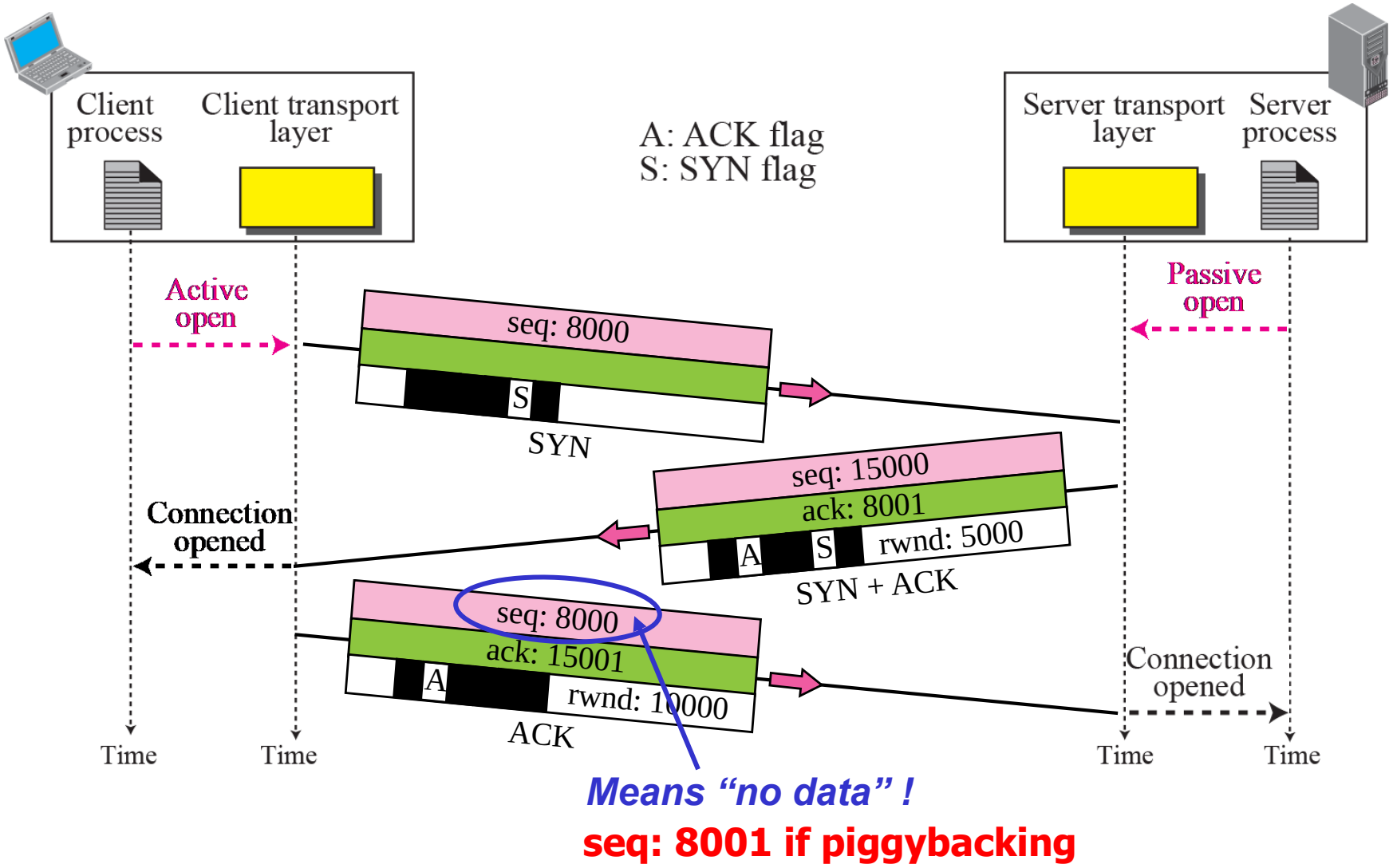
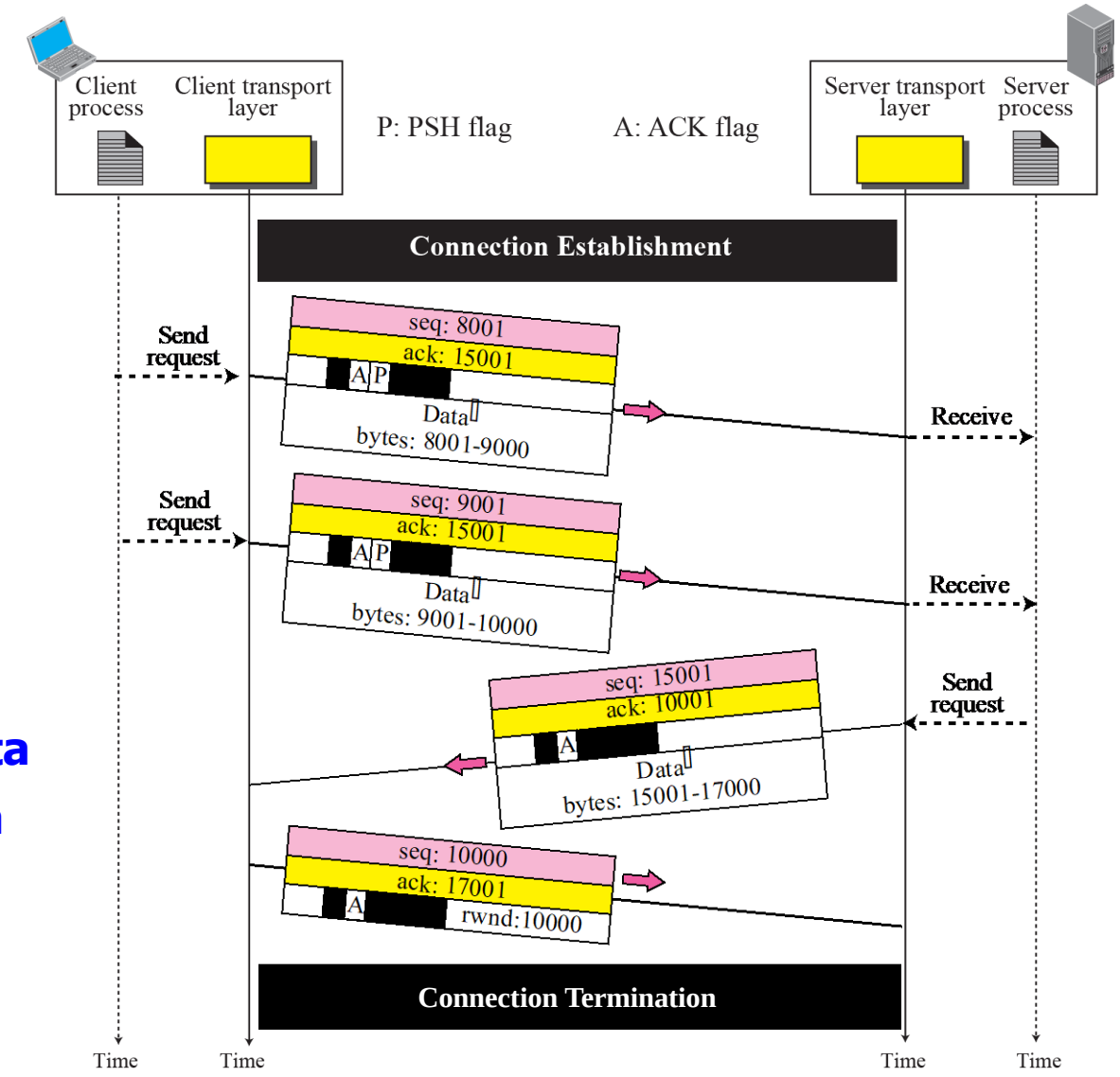


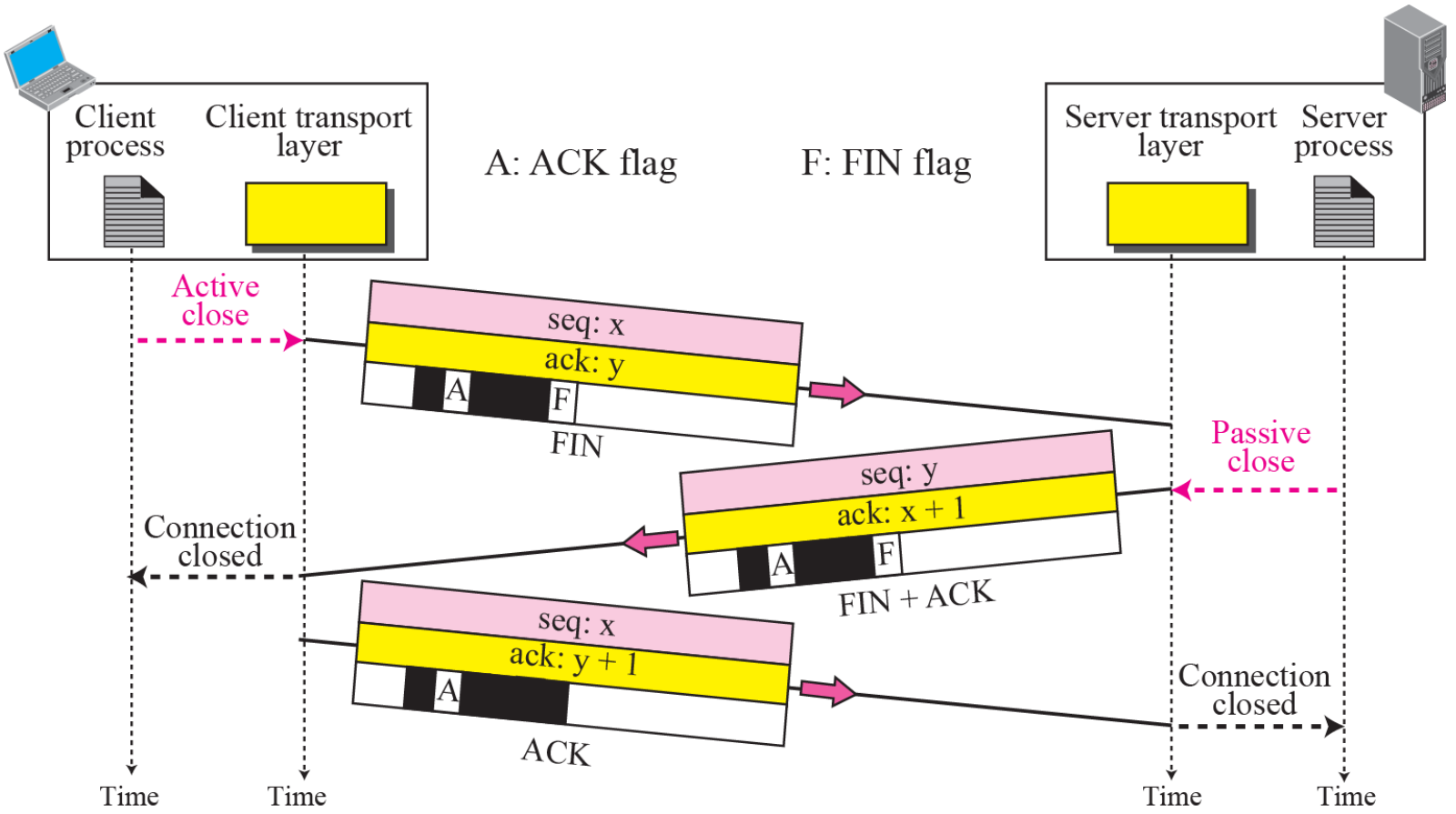
Figure 15.10 *Data Transfer*



Pushing data

Urgent data

Figure 15.11 *Connection termination using three-way handshake*



Note

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

Note

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

Figure 15.12 *Half-Close*

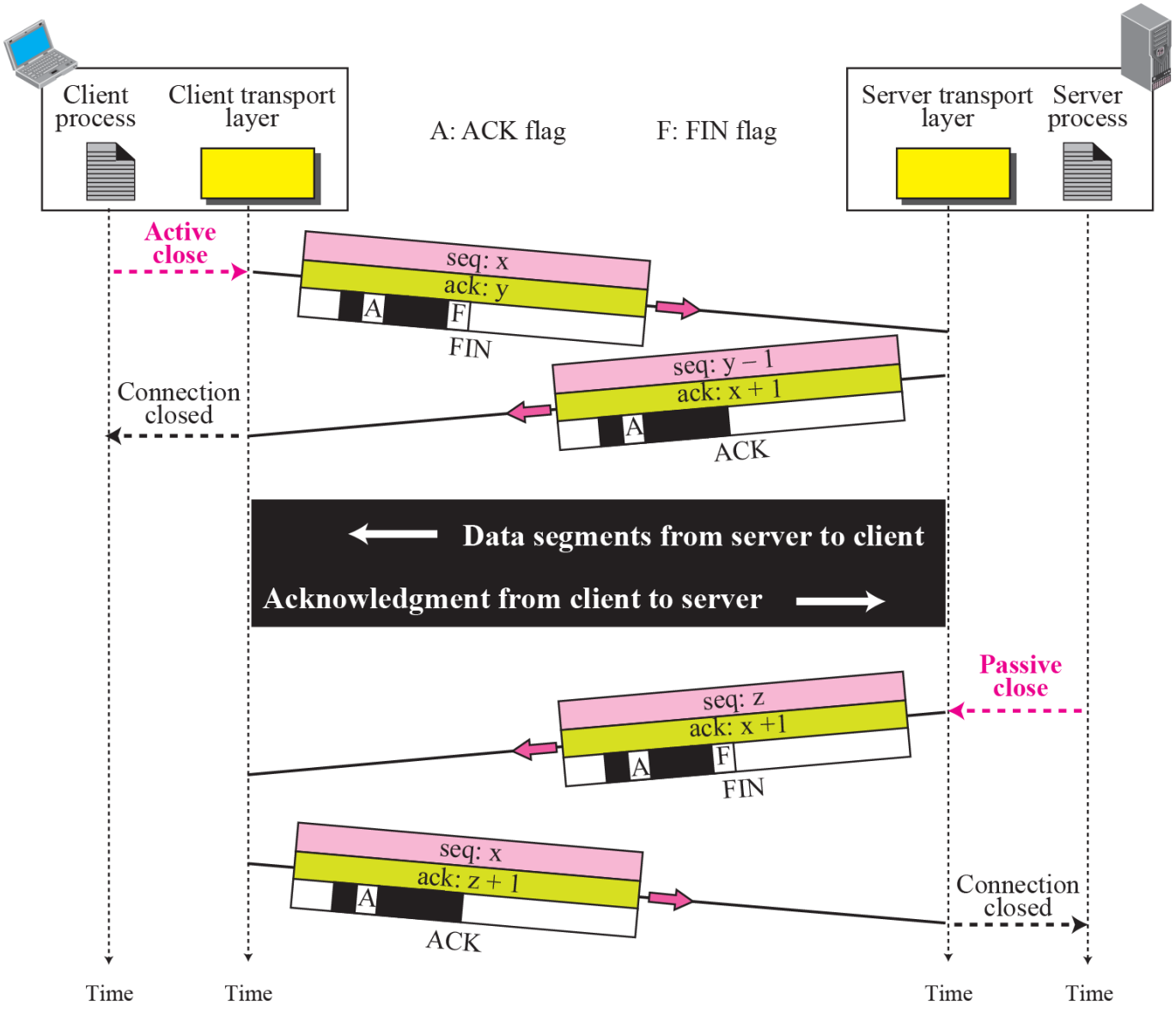
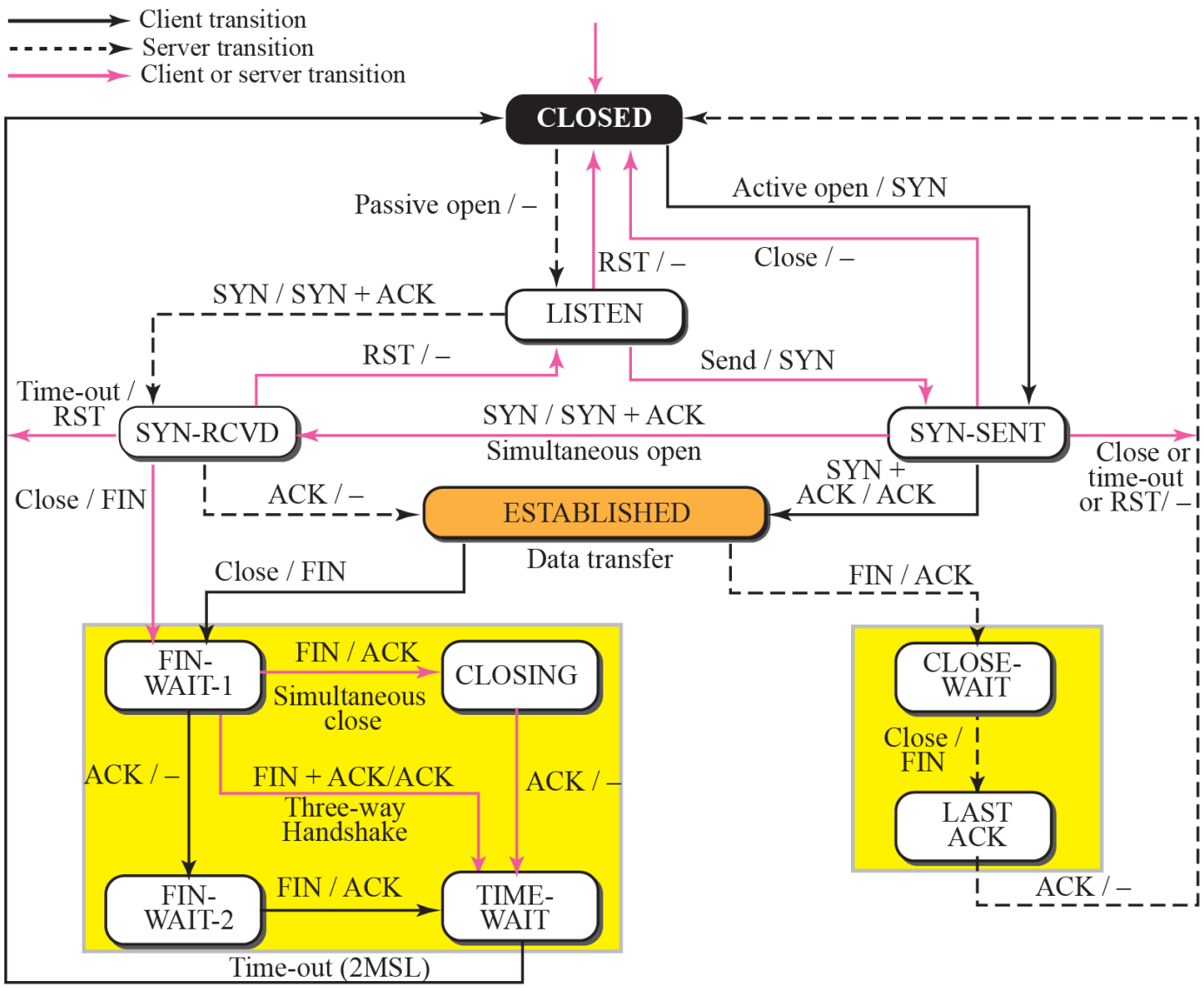


Figure 15.13 *State transition diagram*



Note

The state marked as ESTBLISHED in the FSM is in fact two different sets of states that the client and server undergo to transfer data.

Table 15.2 *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 15.14 *Transition diagram for connection and half-close termination*

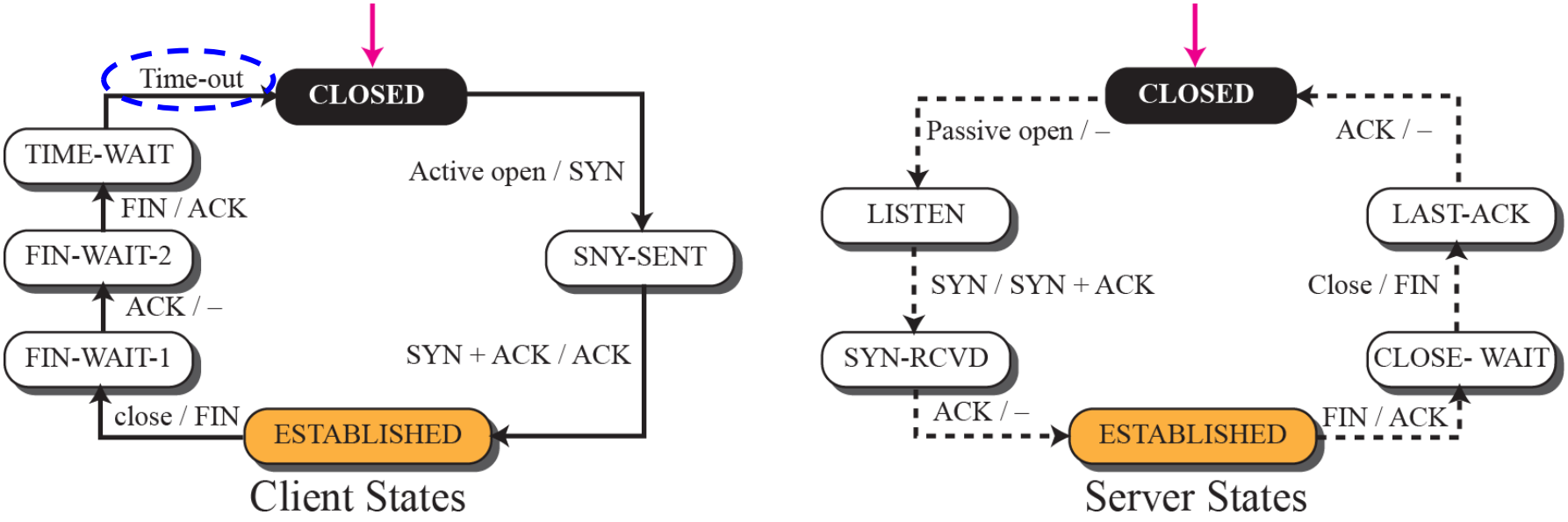


Figure 15.15 *Time-line diagram for Figure 15.14*

1. **Enough time for an ACK to be lost and a new FIN to arrive. If during the TIME-WAIT state, a new FIN arrives, the client sends a new ACK and restarts the 2MSL timer**
2. **To prevent a duplicate segment from one connection appearing in the next one, TCP requires that incarnation cannot take place unless 2MSL amount of time has elapsed.**

   Another solution: the ISN of the incarnation is greater than the last seq. # used in the previous connection.

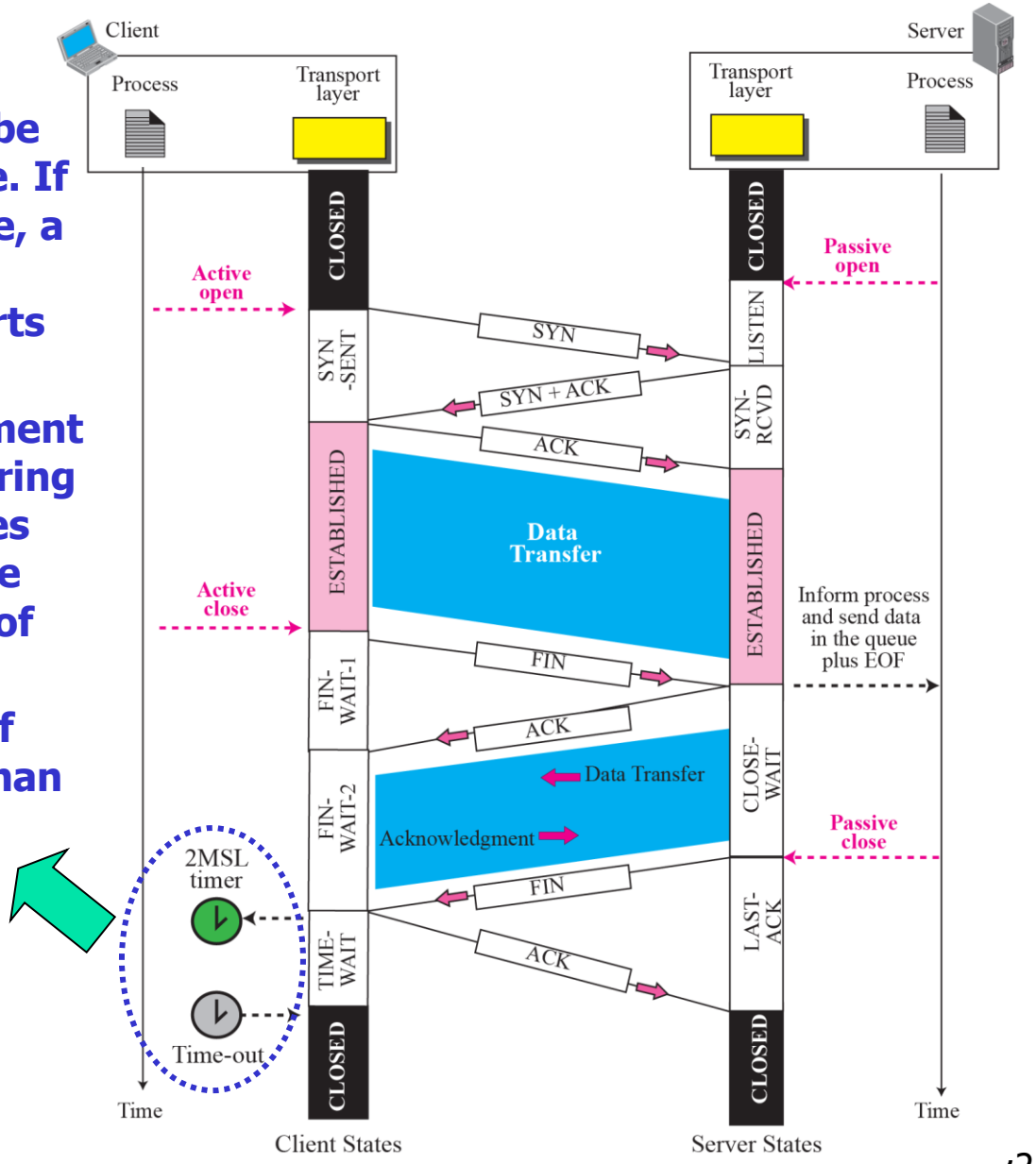


Figure 15.16 *Transition diagram for a common scenario*

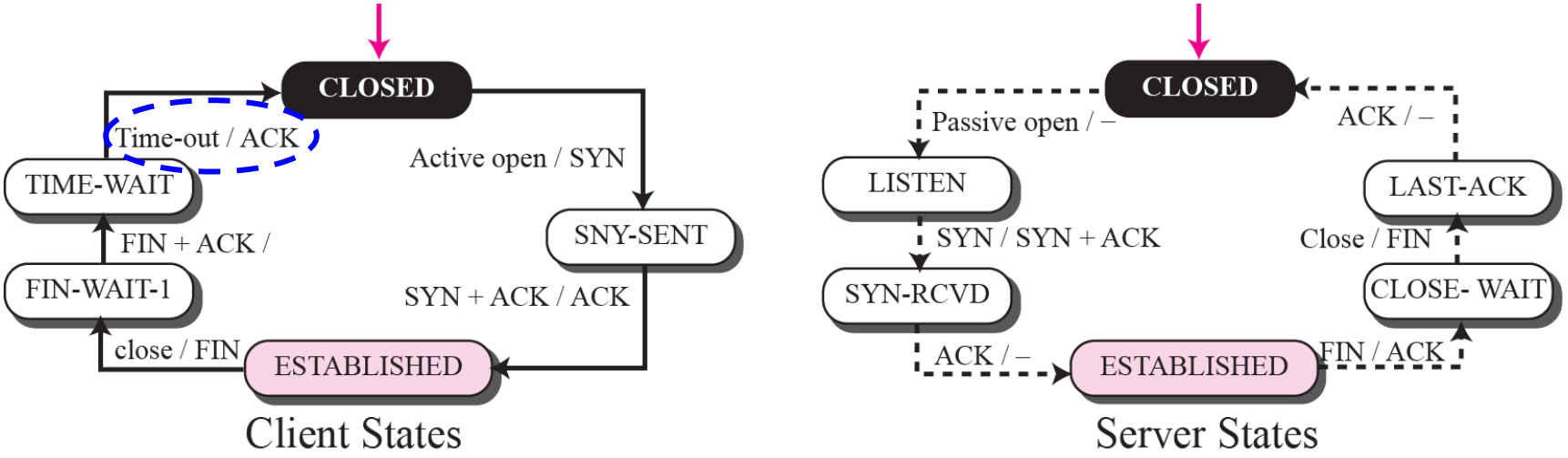


Figure 15.17 *Time line for a common scenario*

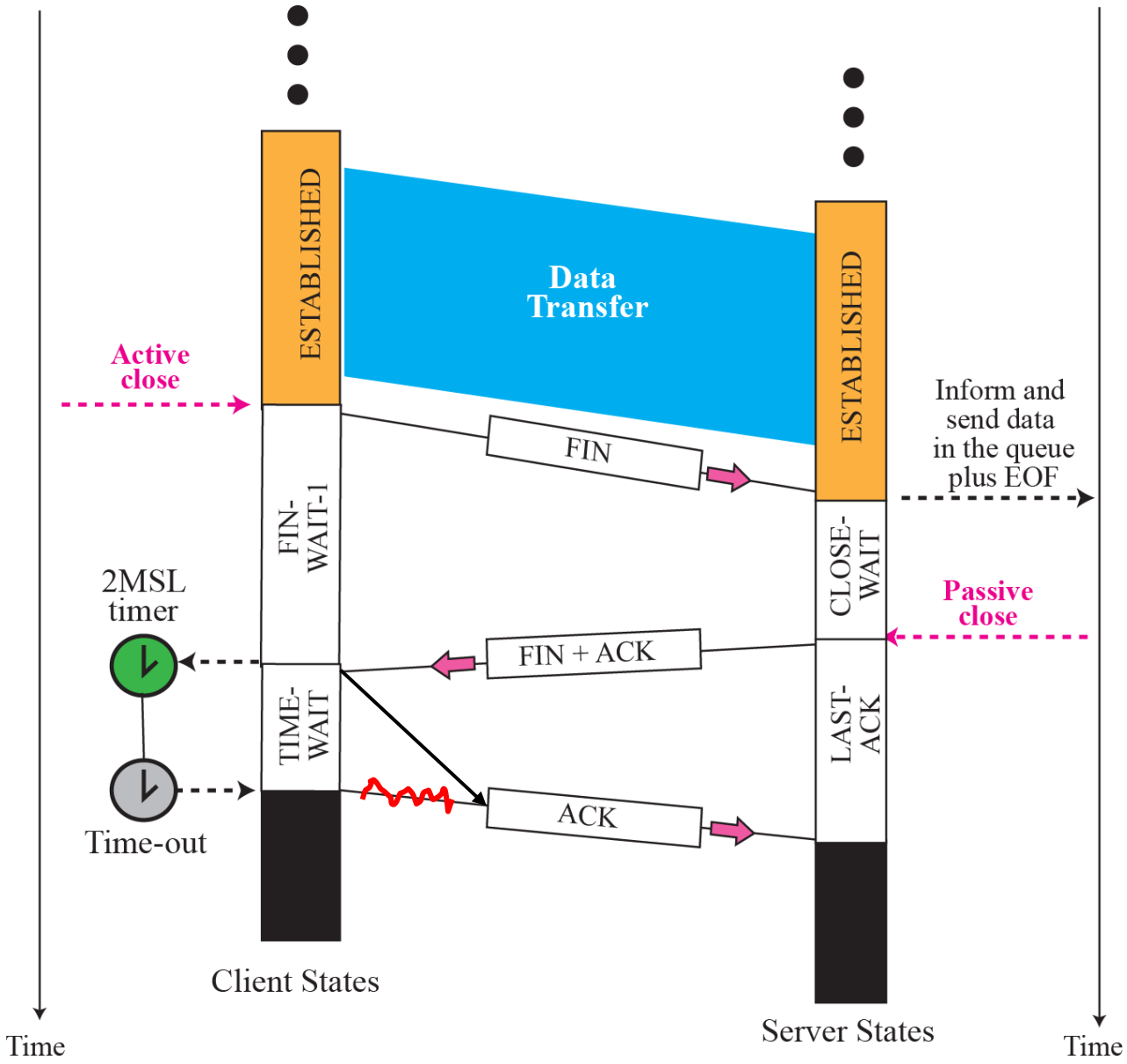


Figure 15.18 *Simultaneous open*

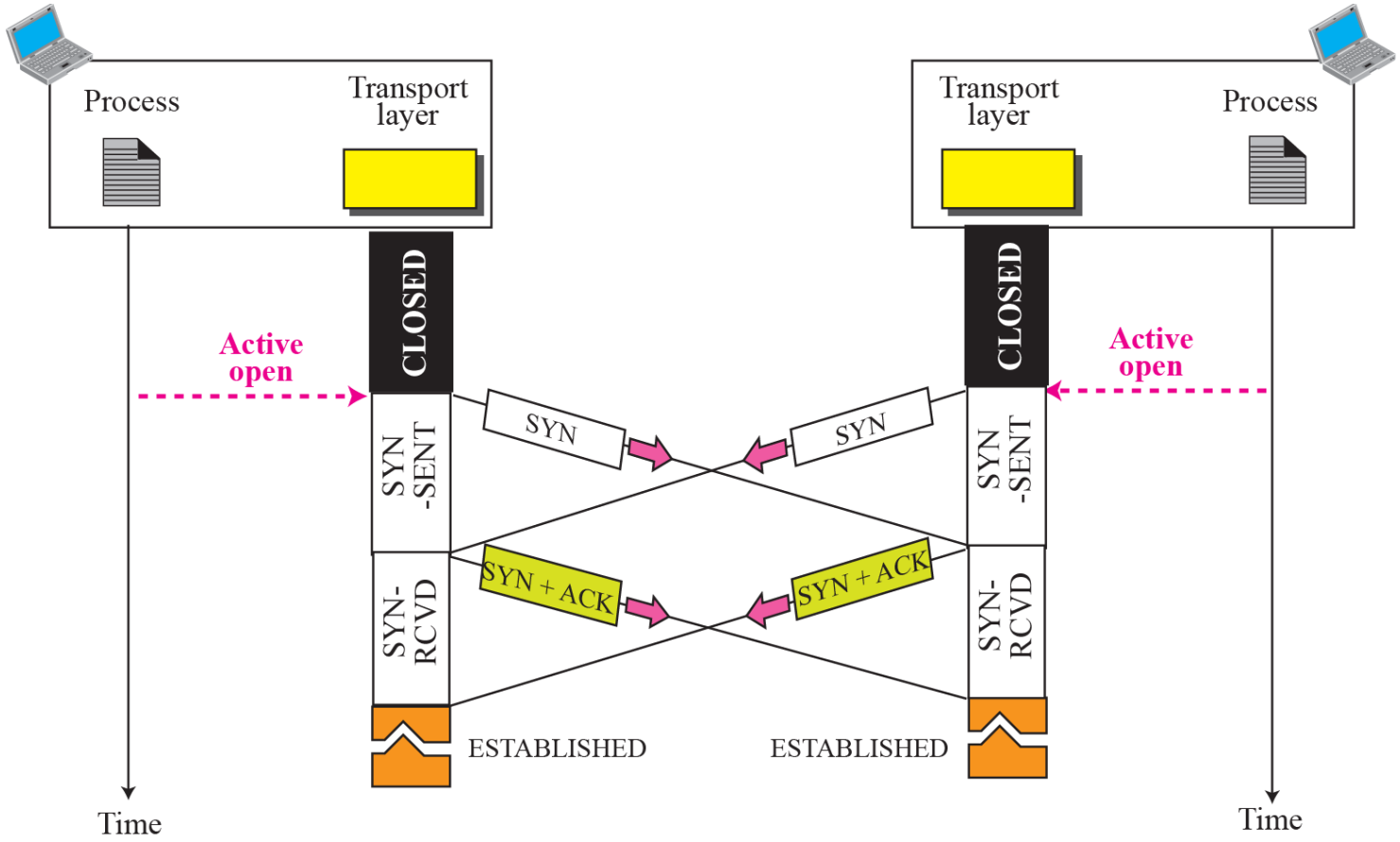


Figure 15.19 *Simultaneous close*

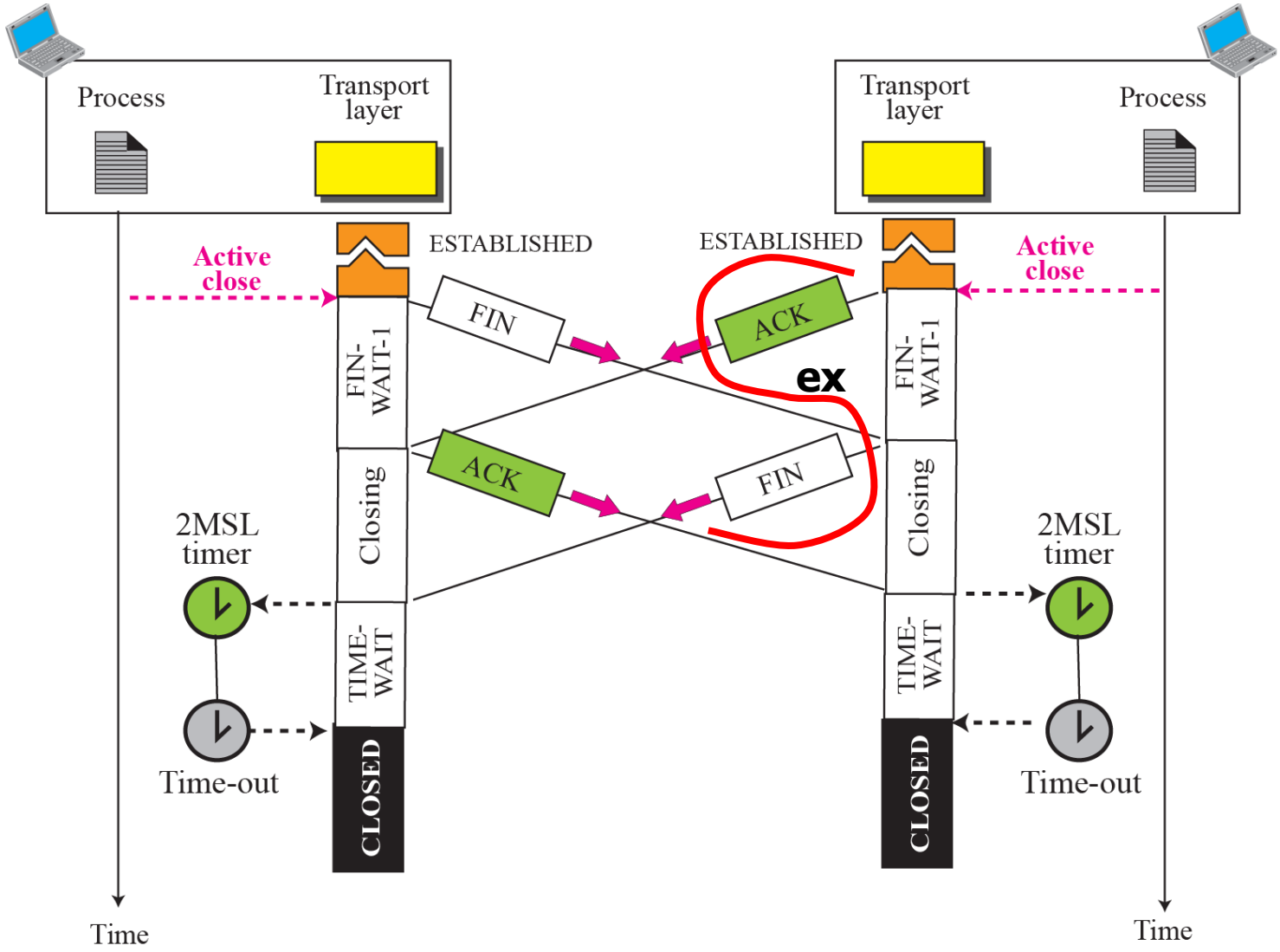


Figure 15.20 *Denying a connection*

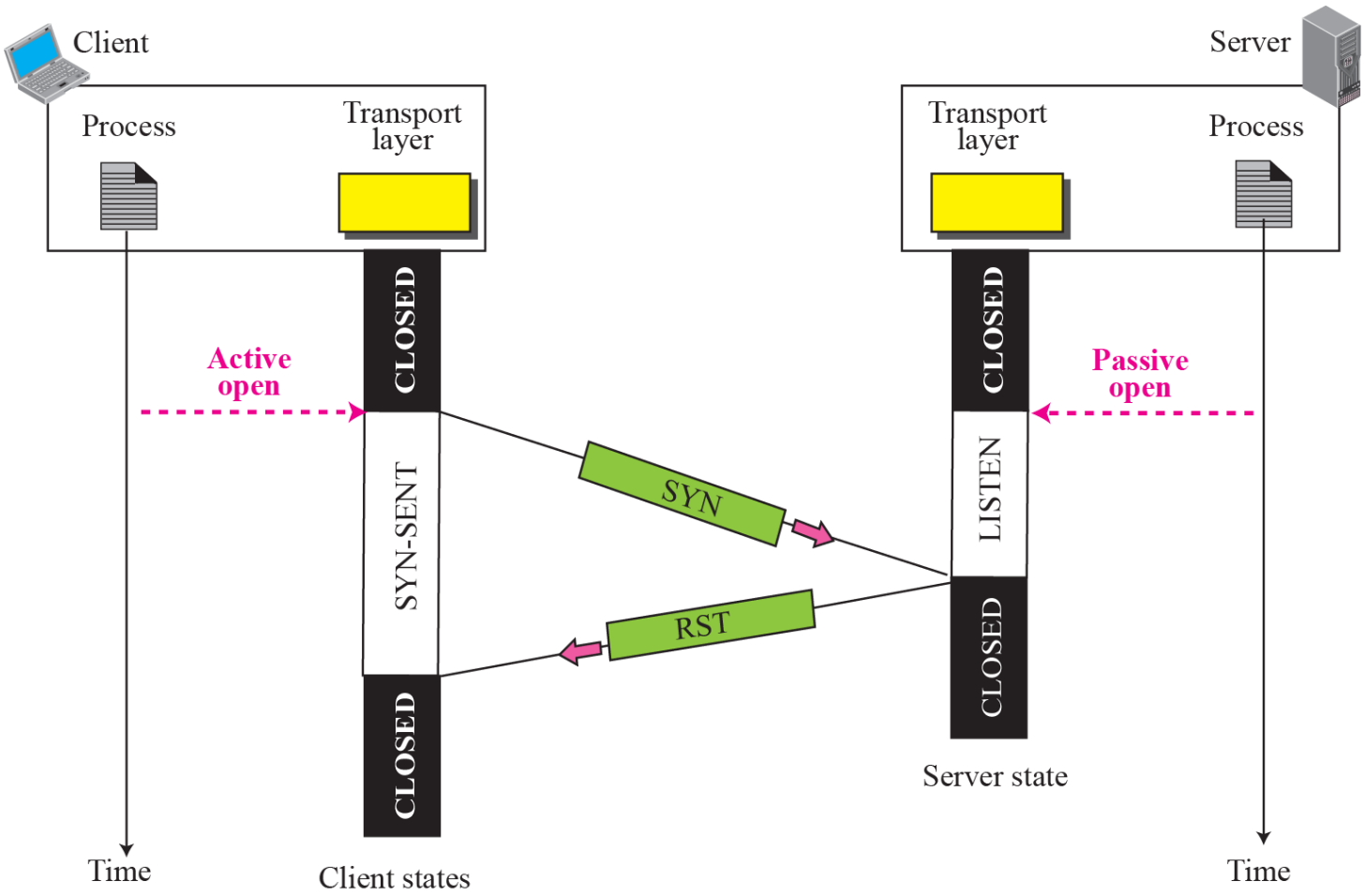


Figure 15.21 *Aborting a connection*

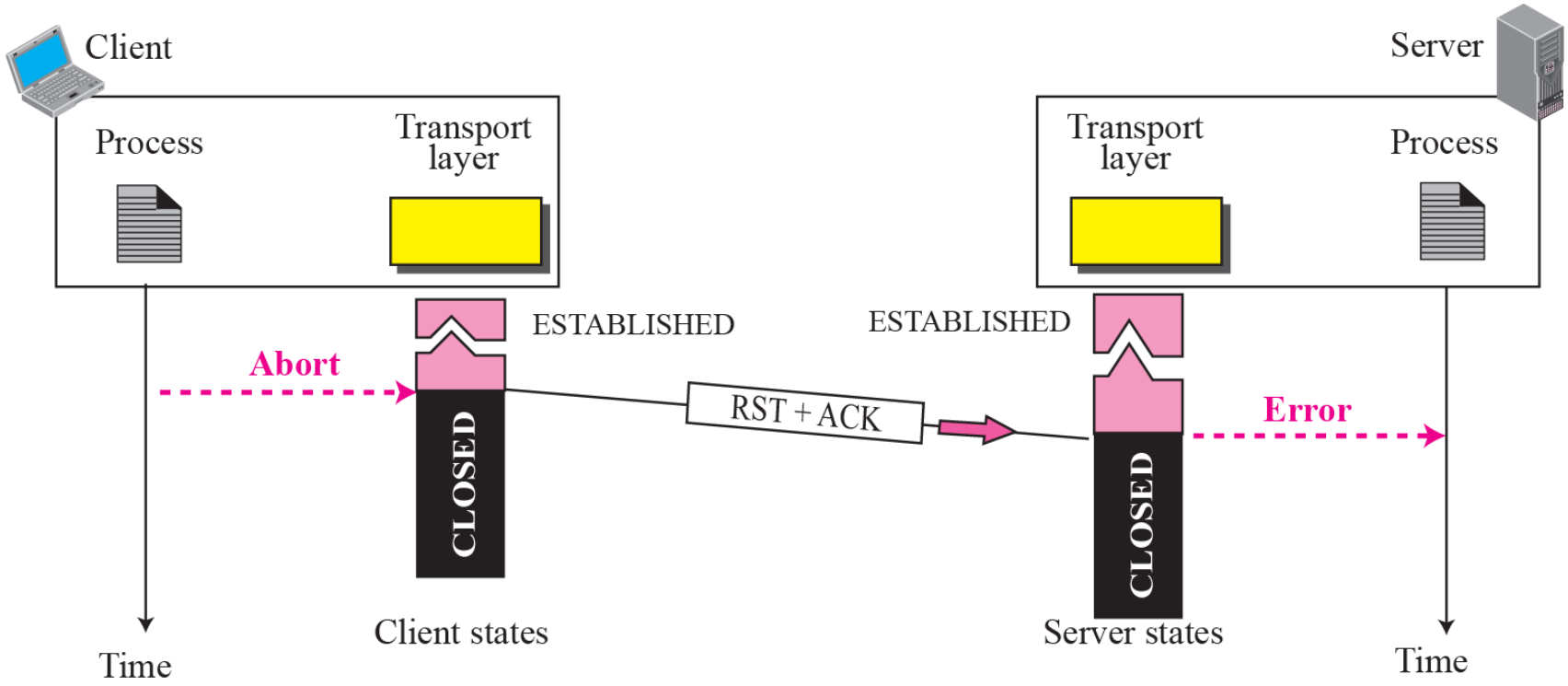


Figure 15.22 *Send window in TCP*

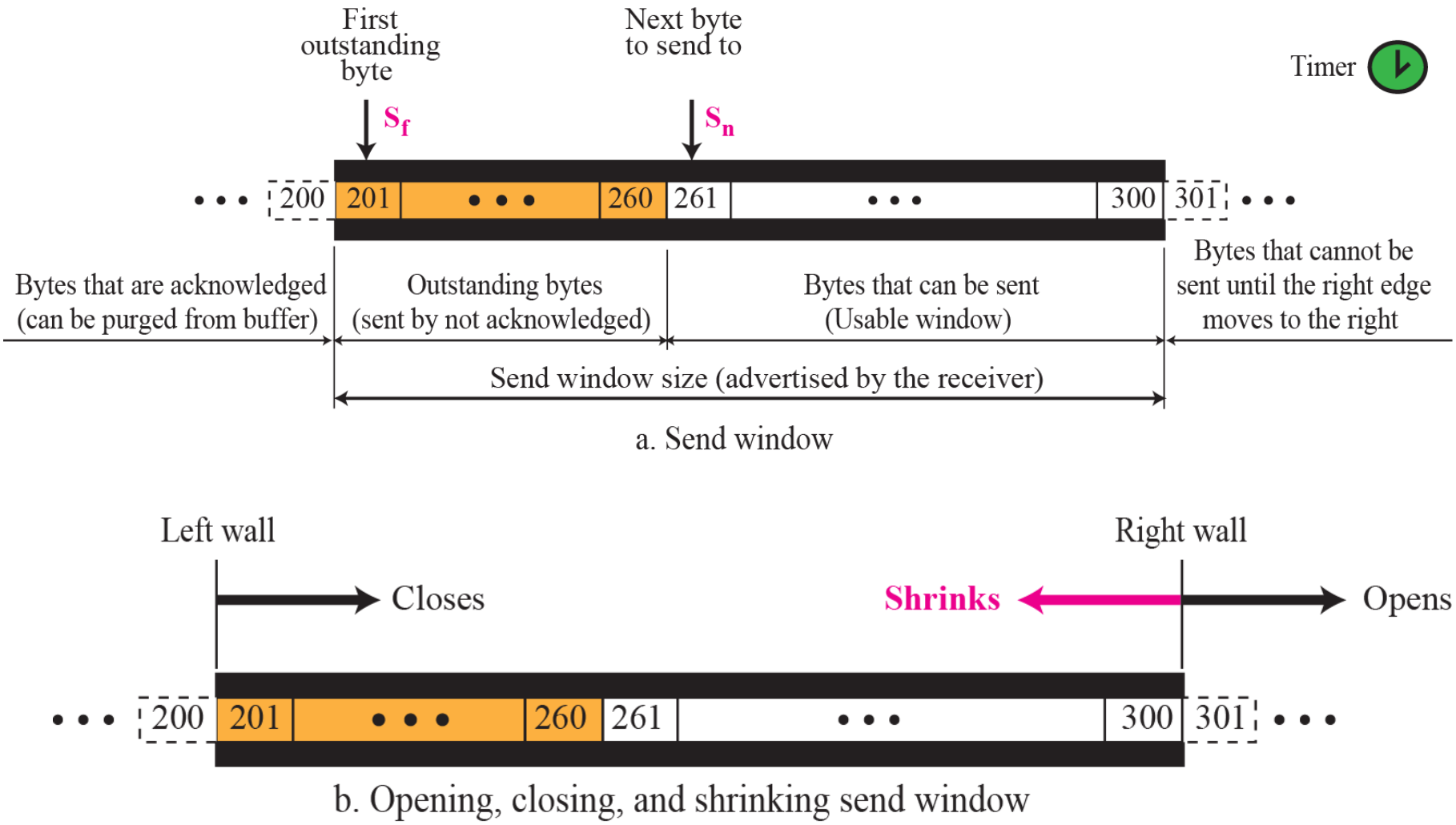
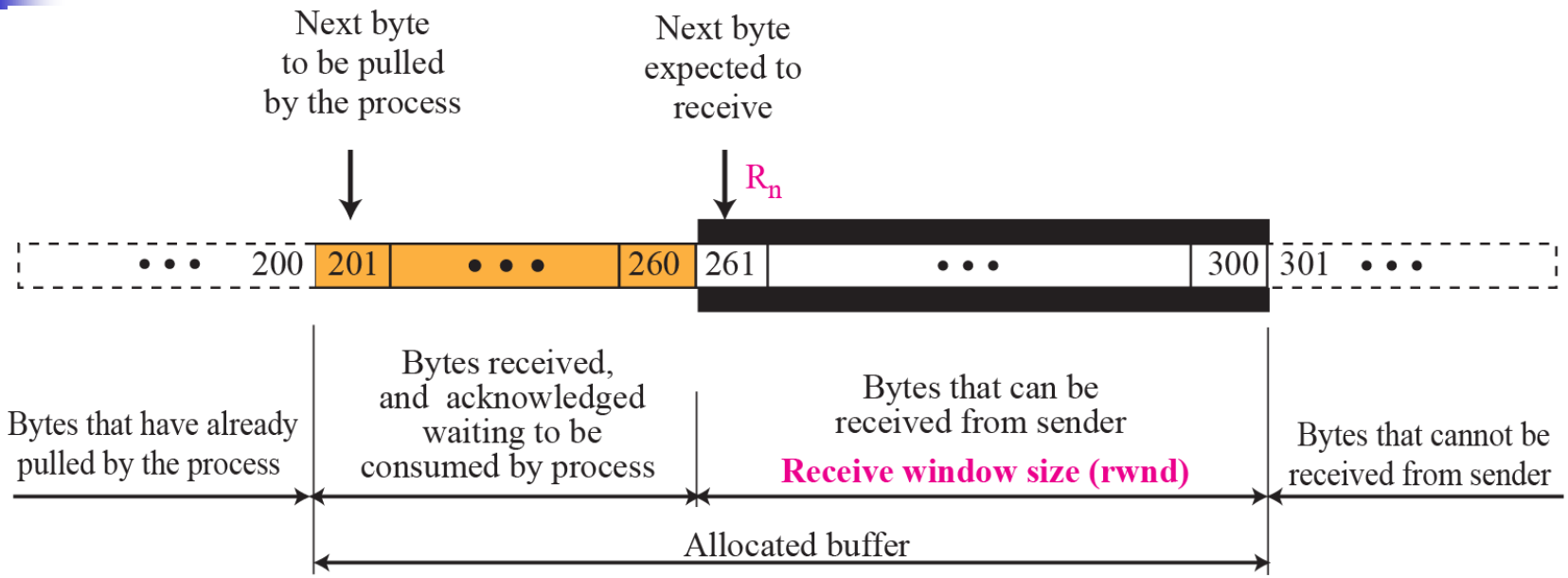
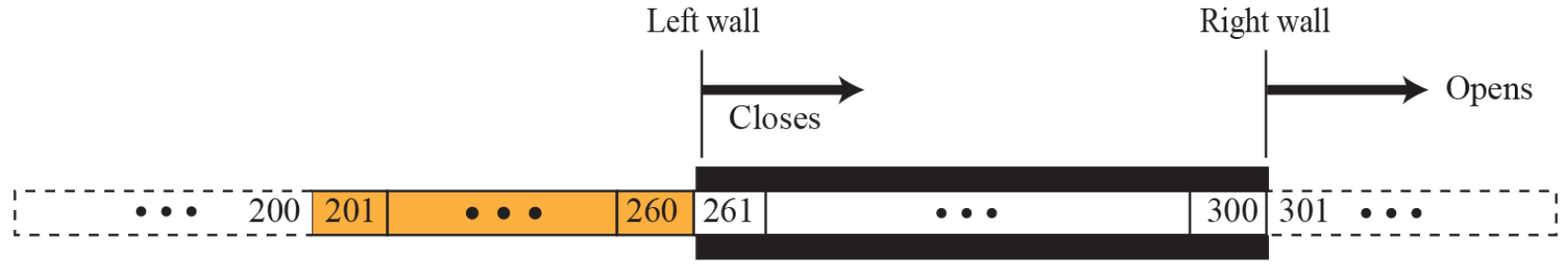


Figure 15.23 *Receive window in TCP*



a. Receive window and allocated buffer



b. Opening and closing of receive window

Figure 15.24 *TCP/IP protocol suite*

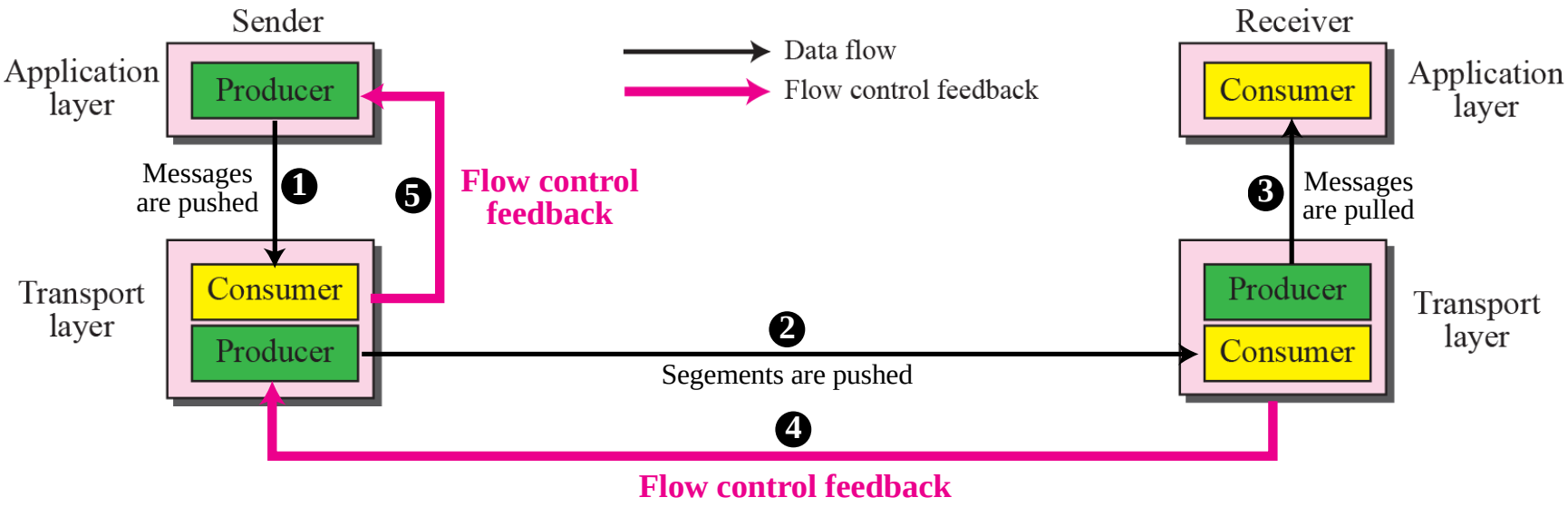
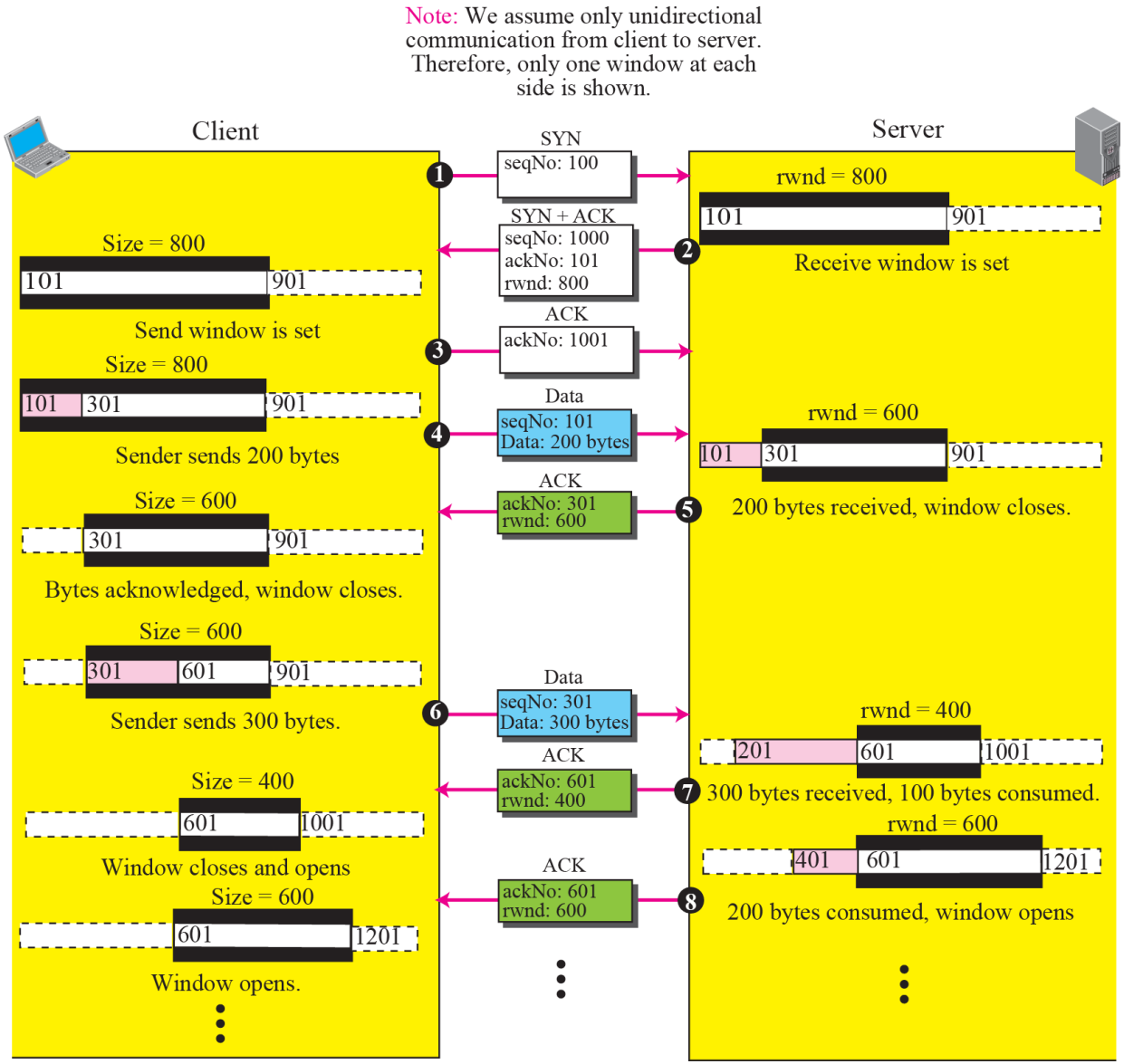


Figure 15.25 *An example of flow control*

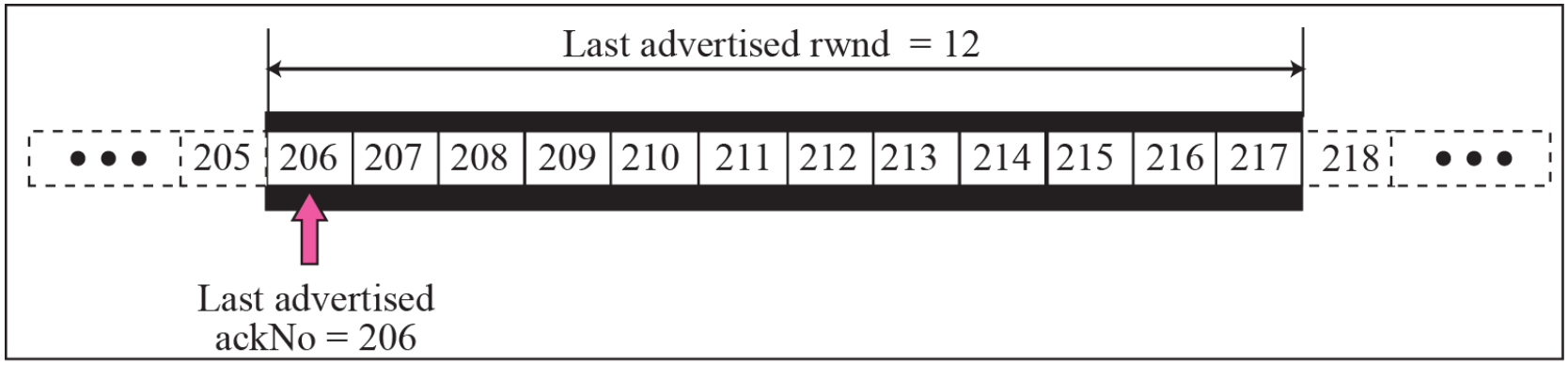


Example 15.2

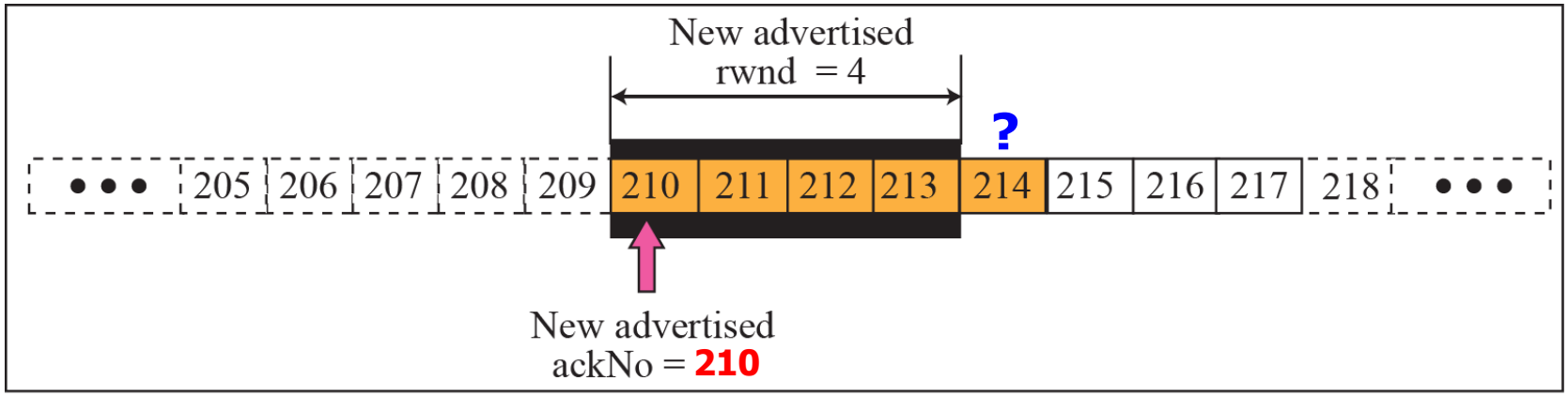
Figure 15.26 shows the reason for the mandate in window shrinking. Part a of the figure shows values of last acknowledgment and *rwnd*. Part b shows the situation in which the sender has sent bytes 206 to 214. Bytes 206 to 209 are acknowledged and purged. The new advertisement, however, defines the new value of *rwnd* as 4, in which 210 + 4 < 206 + 12. When the send window shrinks, it creates a problem: byte 214 which has been already sent is outside the window. The relation discussed before forces the receiver to maintain the right-hand wall of the window to be as shown in part a because the receiver does not know which of the bytes 210 to 217 has already been sent. One way to prevent this situation is to let the receiver postpone its feedback until enough buffer locations are available in its window. In other words, the receiver should wait until more bytes are consumed by its process.

Figure 15.26 *Example 15.2*

Prevent the shrinking of the send window:
new ackNo + new rwnd >= last ackNo + last rwnd



a. The window after the last advertisement



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

Silly Window Syndrome (1)

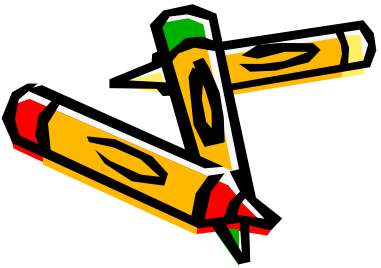
- **Sending data in very small segments**

1. Syndrome created by the Sender
   - Sending application program creates data slowly (e.g. 1 byte at a time)
   - Wait and collect data to send in a larger block
   - How long should the sending TCP wait?
   - Solution: Nagle's algorithm
   - Nagle's algorithm takes into account (1) the speed of the application program that creates the data, and (2) the speed of the network that transports the data

Silly Window Syndrome (2)

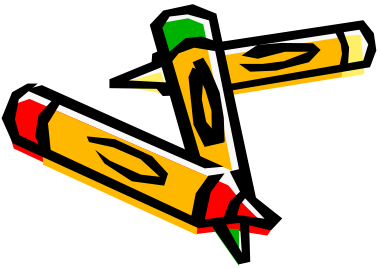
2. Syndrome created by the Receiver

- Receiving application program consumes data slowly (e.g. 1 byte at a time)
- The receiving TCP announces a window size of 1 byte. The sending TCP sends only 1 byte...
- Solution 1: Clark's solution
- Sending an ACK but announcing a window size of zero until there is enough space to accommodate a segment of max. size or until half of the buffer is empty



Silly Window Syndrome (3)

- Solution 2: Delayed Acknowledgement
- The receiver waits until there is decent amount of space in its incoming buffer before acknowledging the arrived segments
- The delayed acknowledgement prevents the sending TCP from sliding its window. It also reduces traffic.
- Disadvantage: it may force the sender to retransmit the unacknowledged segments
- To balance: should not be delayed by more than 500ms

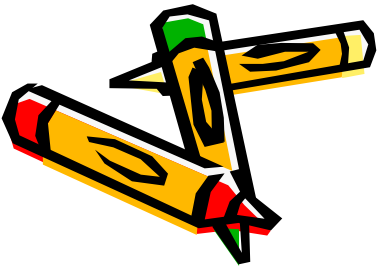


Note

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

Acknowledgement Type

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

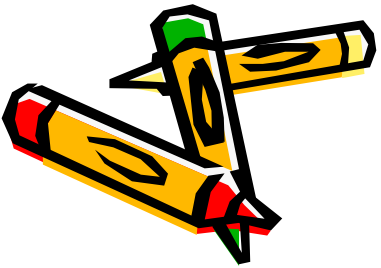


Note

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

Rules for Generating ACK (1)

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



Rules for Generating ACK (2)

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

Figure 15.29 *Normal operation*

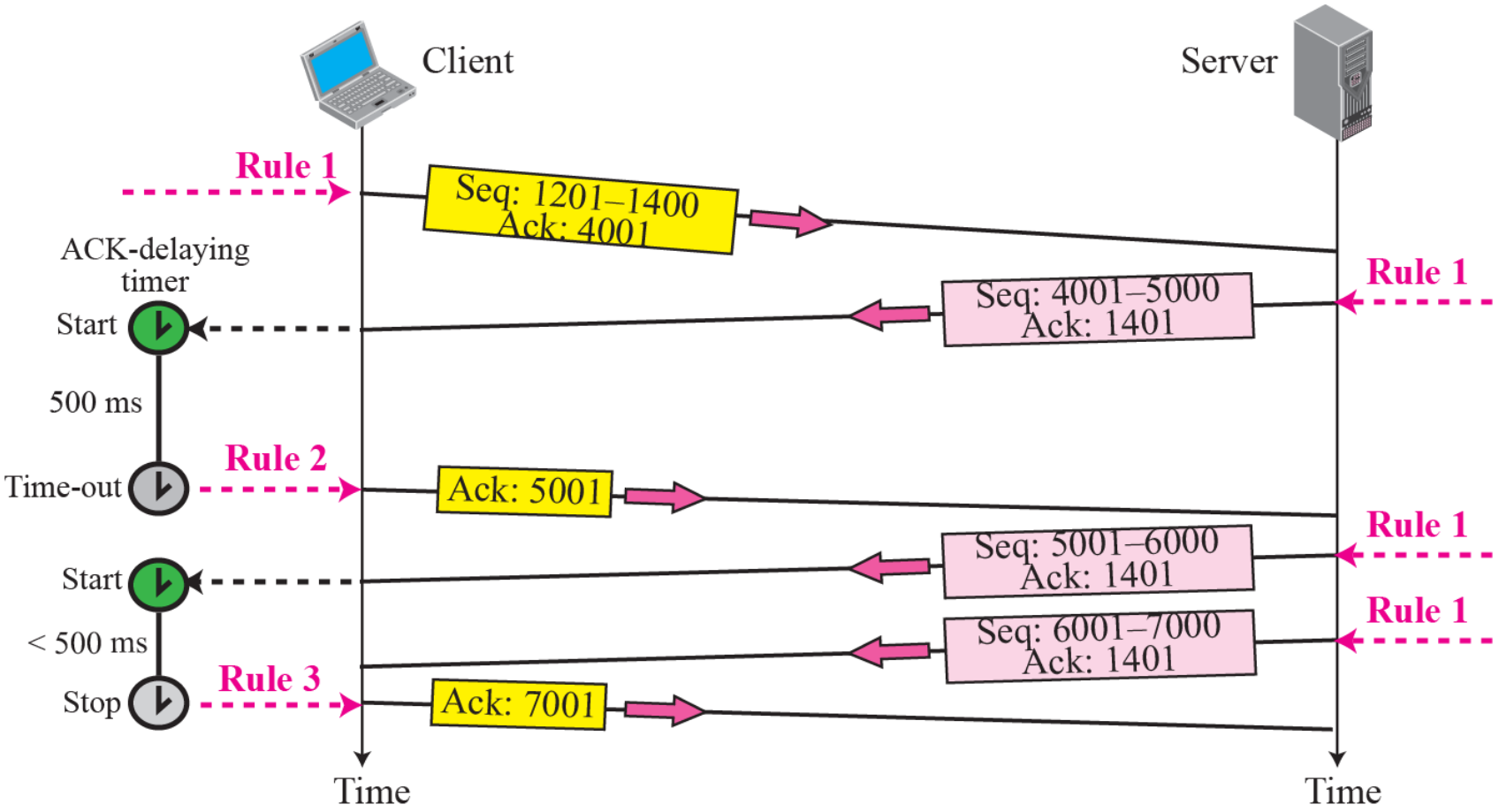
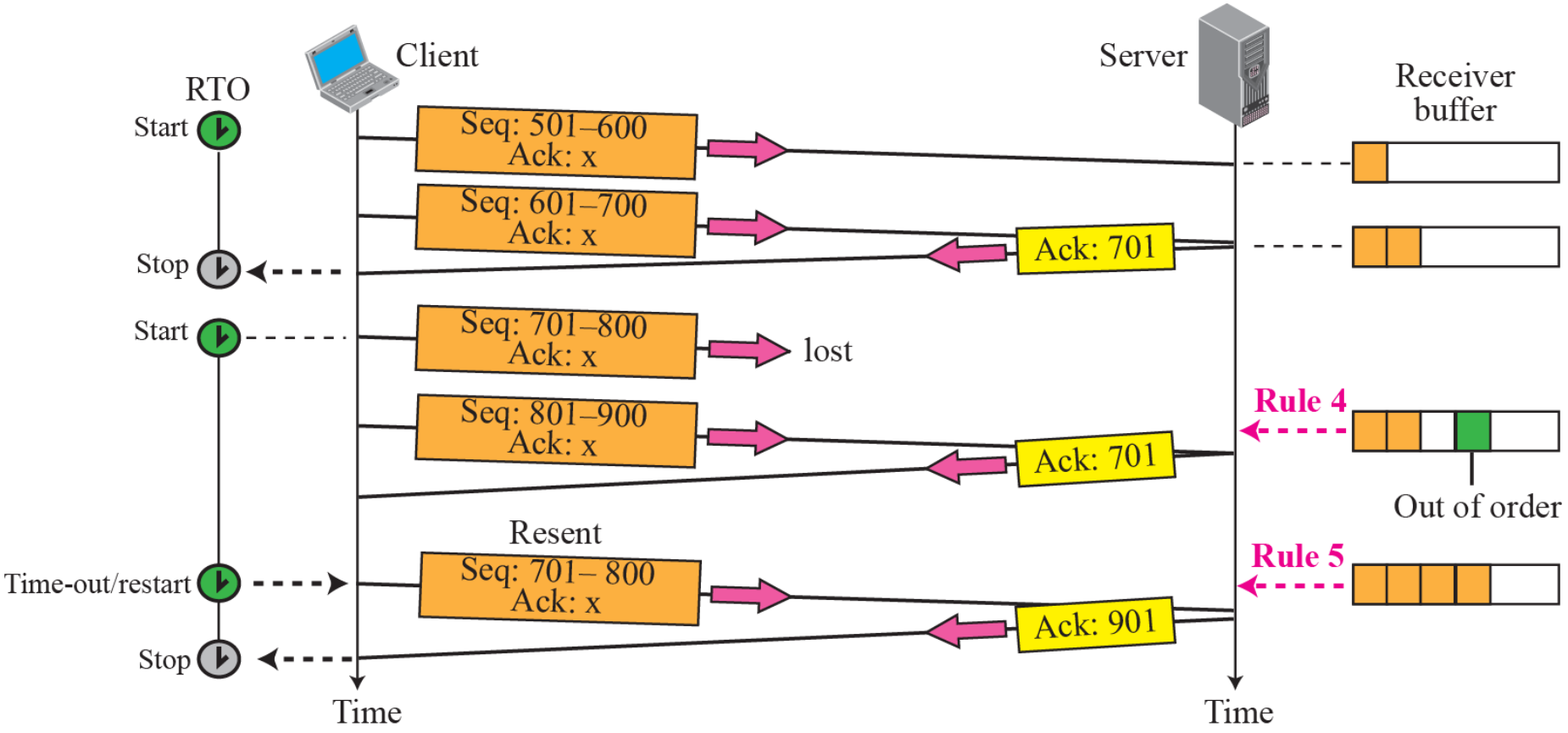


Figure 15.30 *Lost segment*



Note

The receiver TCP delivers only ordered data to the process.

Figure 15.31 *Fast retransmission*

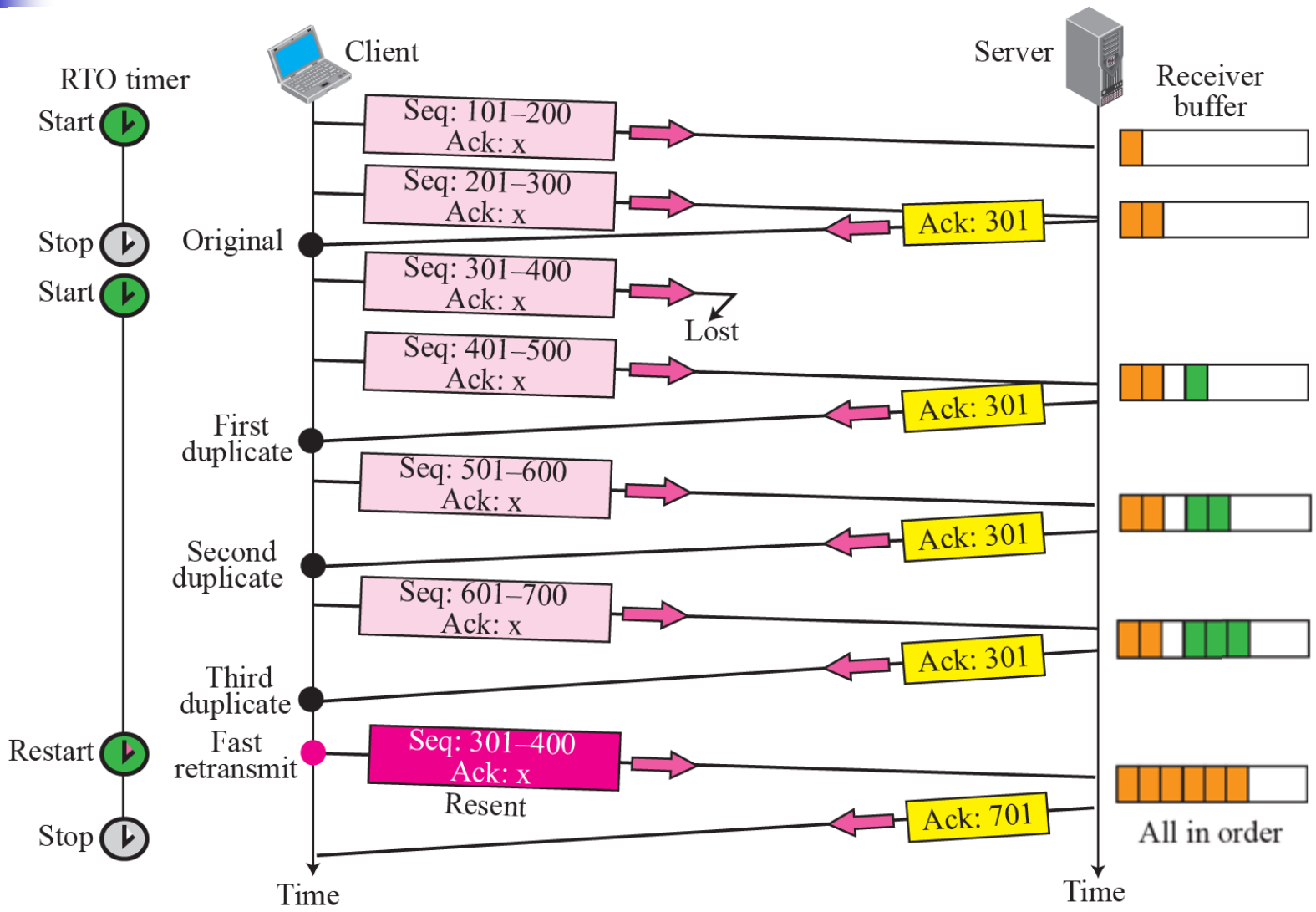


Figure 15.32 *Lost acknowledgment*

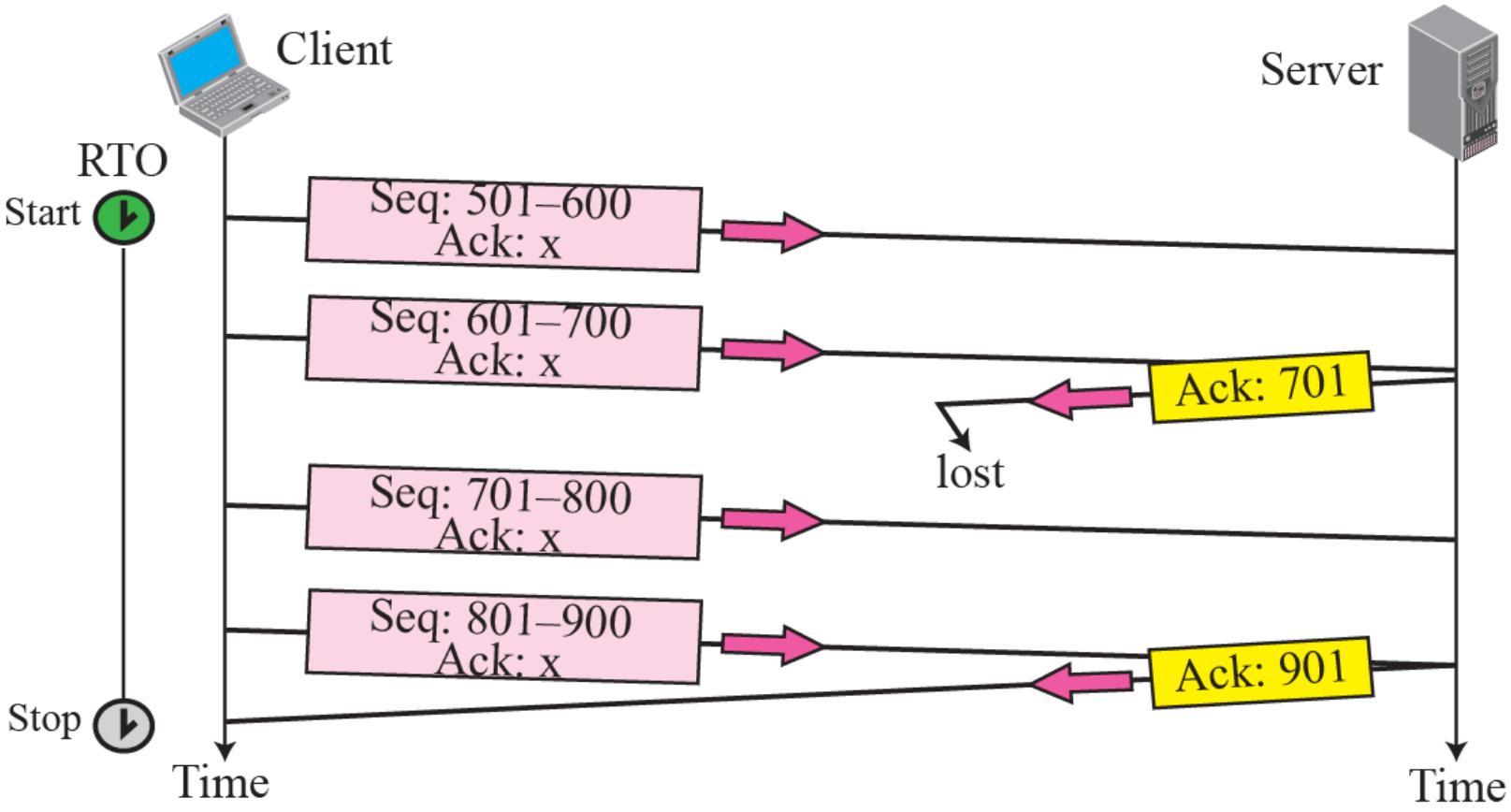
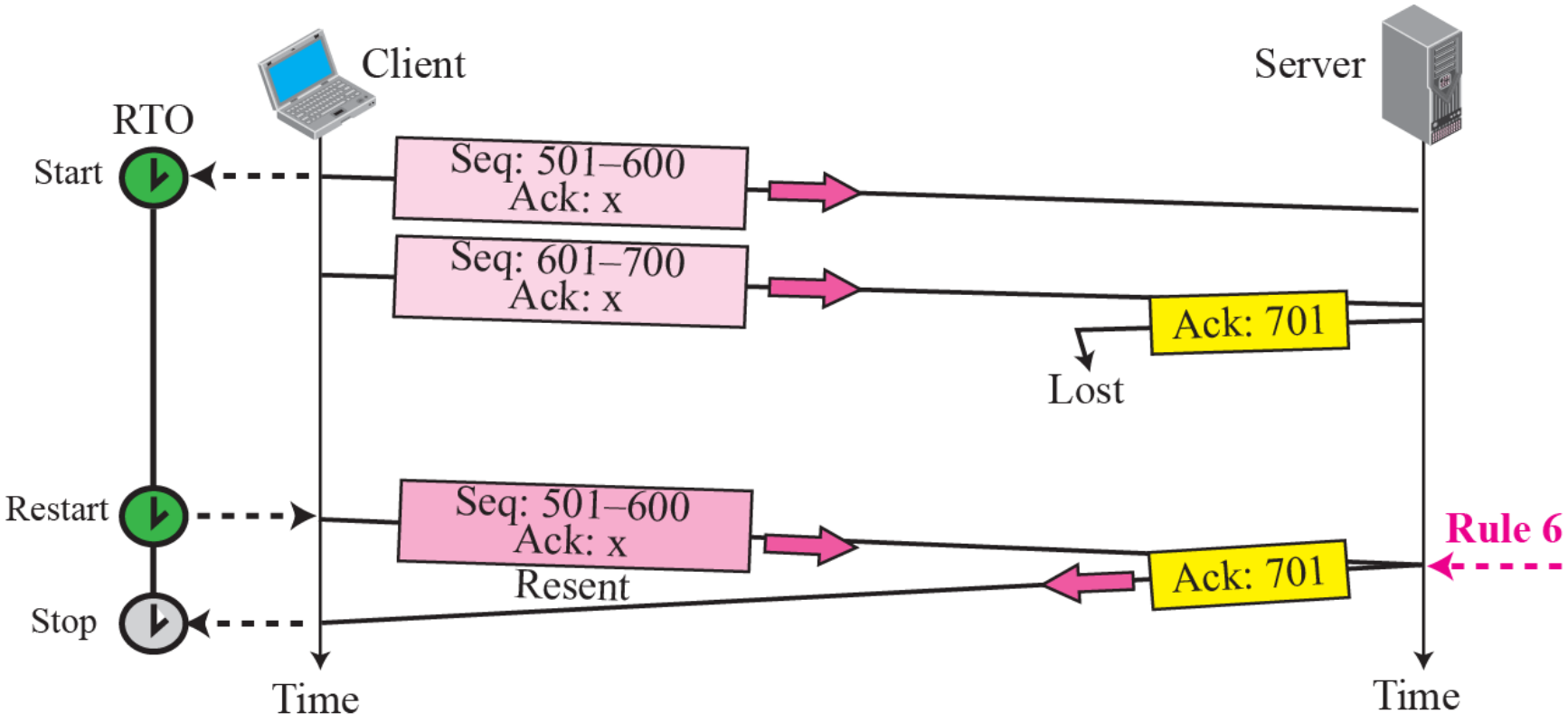


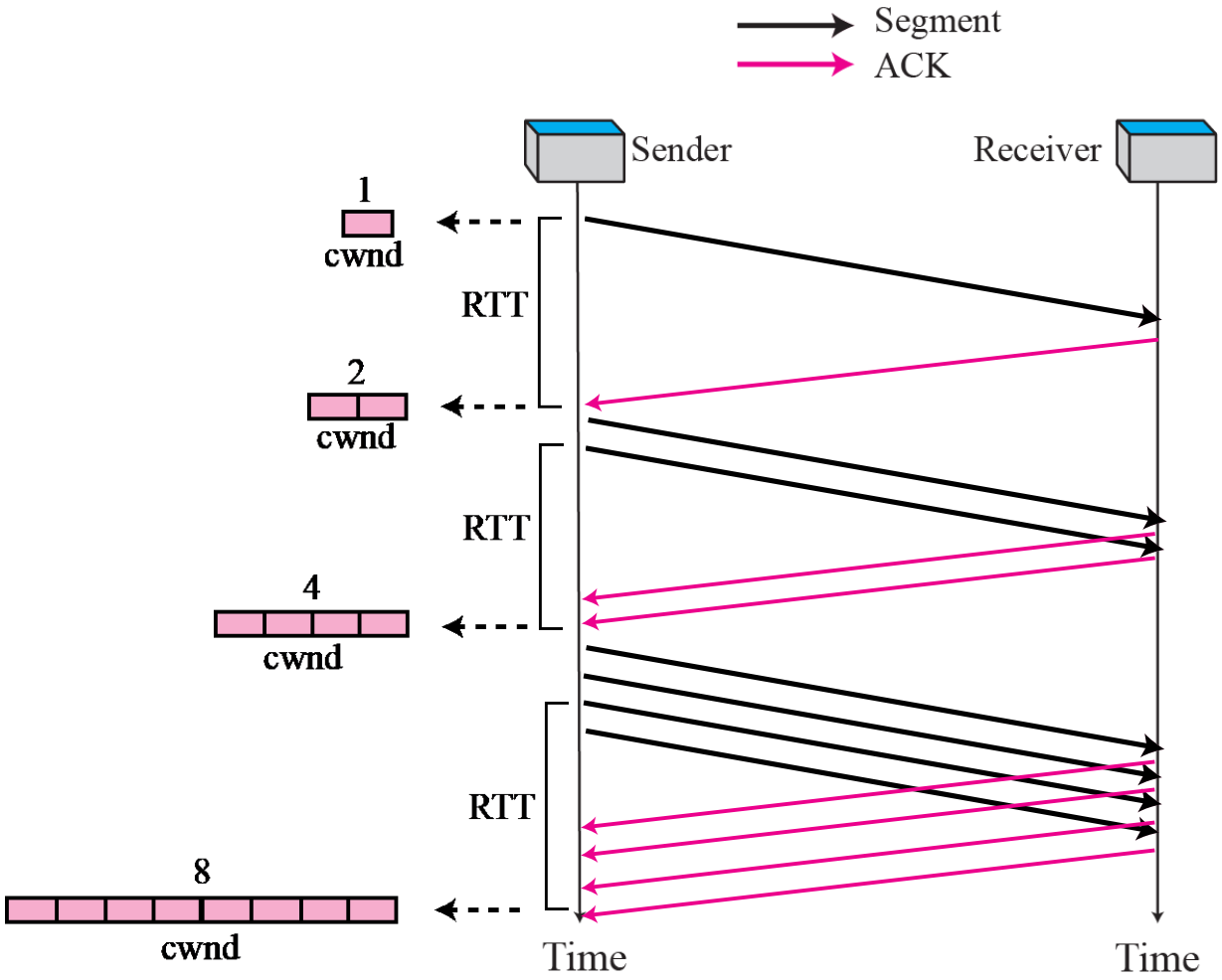
Figure 15.33 *Lost acknowledgment corrected by resending a segment*



Note

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

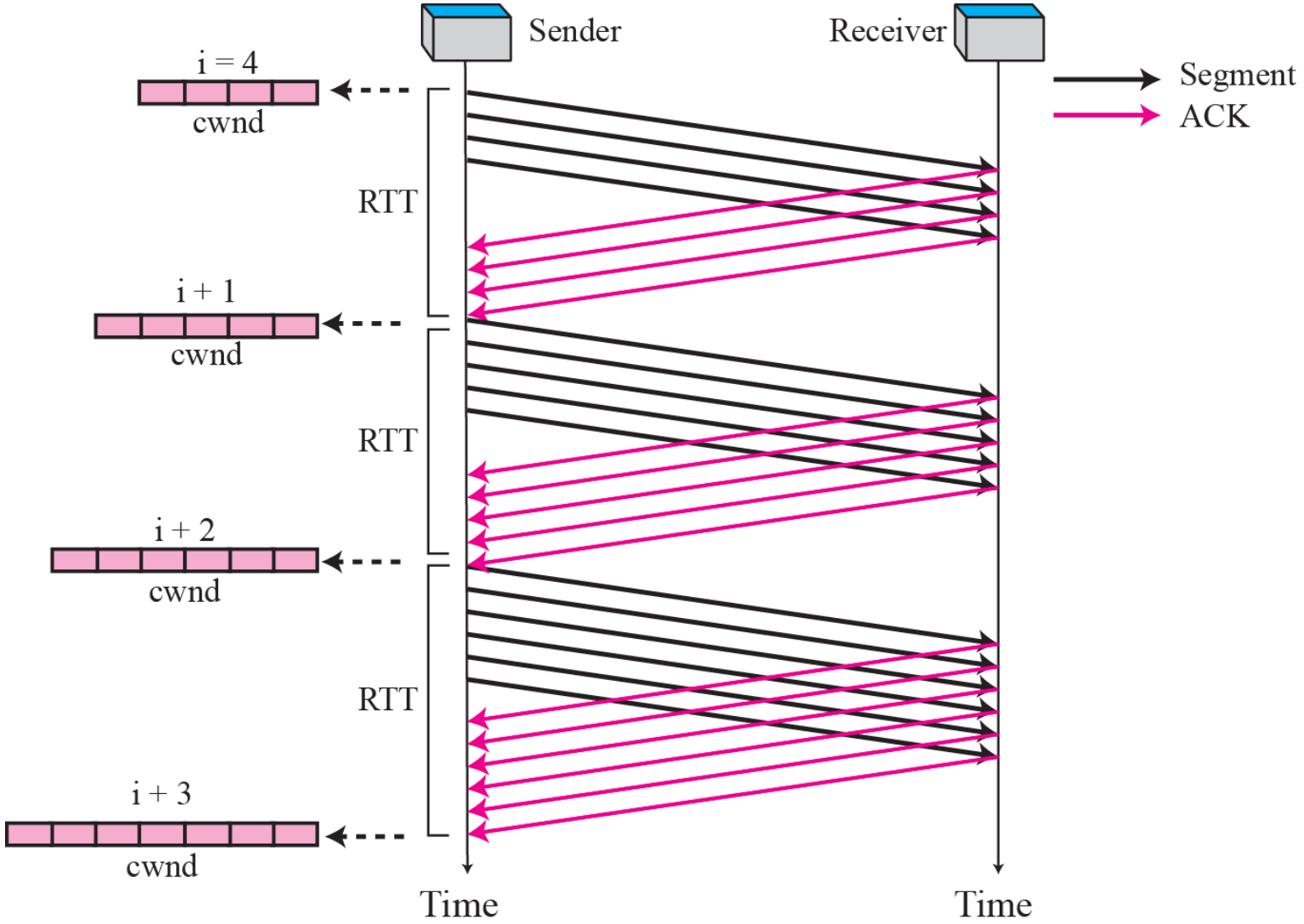
Figure 15.34 *Slow start, exponential increase*



Note

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

Figure 15.35 *Congestion avoidance, additive increase*



Note

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

Figure 15.36 *TCP Congestion policy summary*

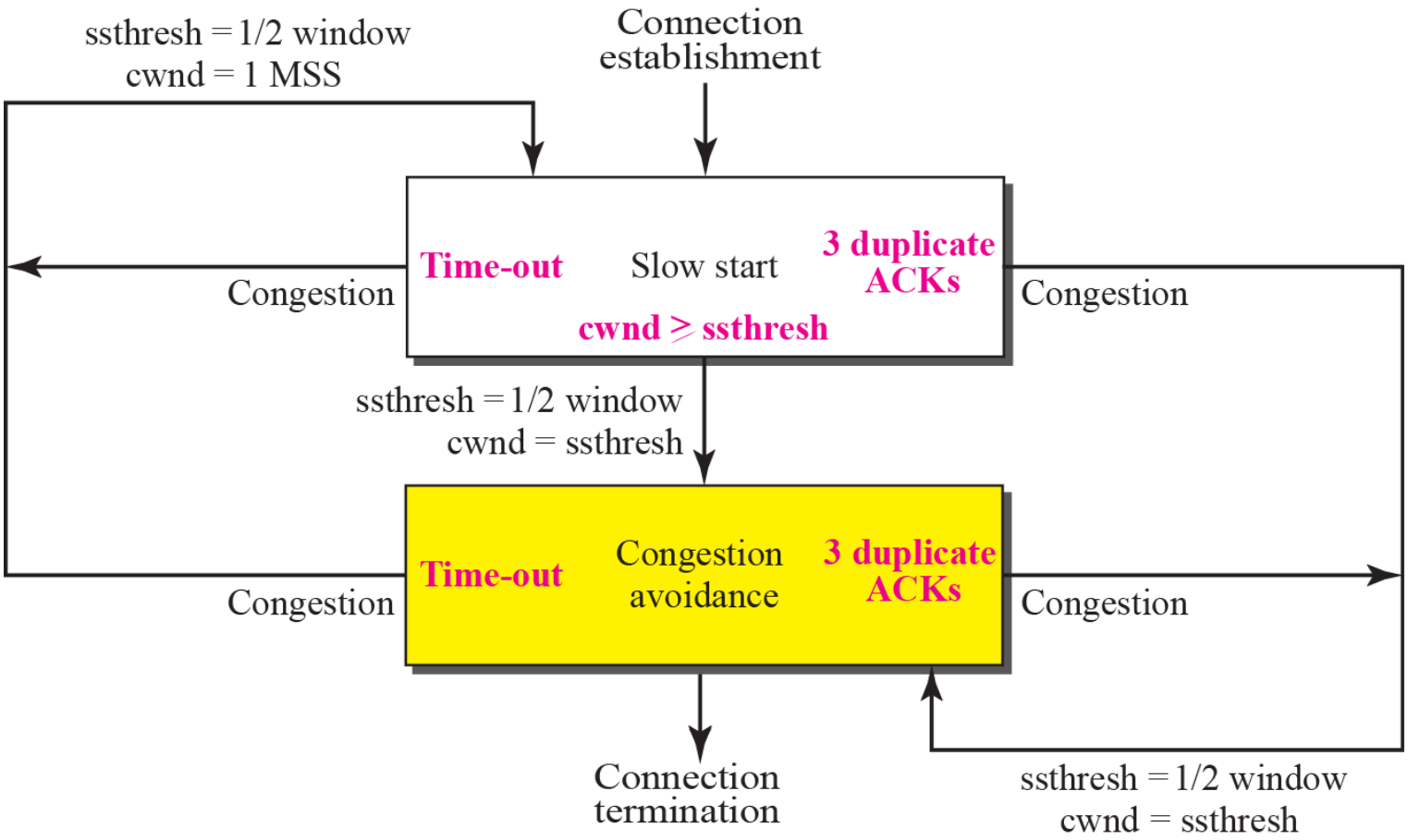


Figure 15.37 *Congestion example*

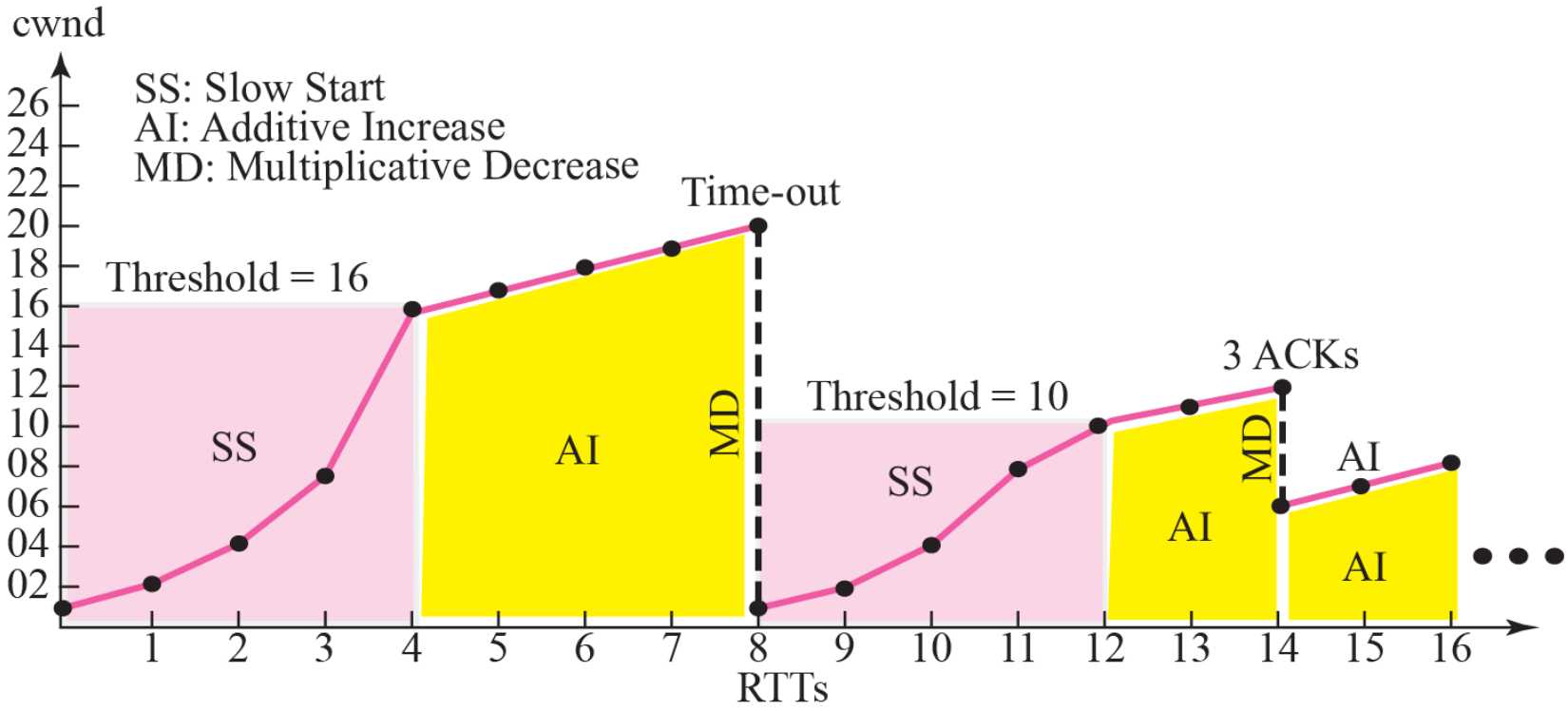
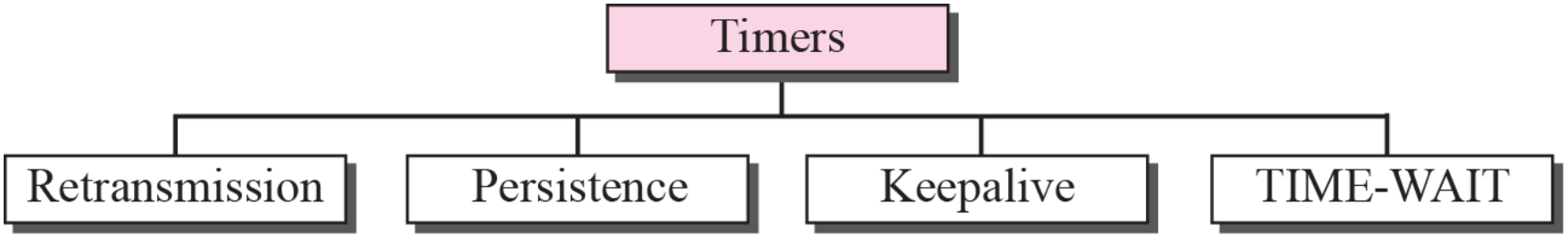


Figure 15.38 *TCP timers*



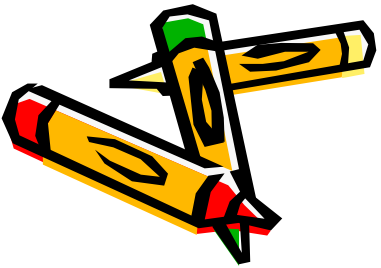
Note

In TCP, there can be only one RTT measurement in progress at any time.

Since the segments and their ACKs do not have a 1-1 relationship

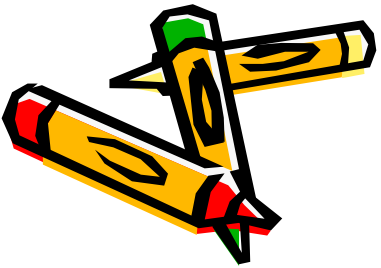
Calculation of RTO (1)

- Smoothed RTT: RTT_S
  - Original → No value
  - After 1st measurement → $RTT_S = RTT_M$
  - 2nd … → $RTT_S = (1-\alpha)*RTT_S + \alpha*RTT_M$
- RTT Deviation : RTT_D
  - Original → No value
  - After 1st measurement → $RTT_D = 0.5*RTT_M$
  - 2nd … → $RTT_D = (1-\beta)*RTT_D + \beta*|RTT_S - RTT_M|$



Calculation of RTO (2)

- **Retransmission Timeout (RTO)**
  - Original → Initial value
  - After any measurement
    → $RTO = RTT_S + 4RTT_D$
- **Example 10 (page 322)**
  - $\alpha = 1/8$
  - $\beta = 1/4$



Example 15.3

Let us give a hypothetical example. Figure 15.39 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 RTT_M, RTT_S, or RTT_D. The value of RTO is set to 6.00 seconds. The following shows the value of these variable at this moment:

$$RTO = 6$$

2. When the SYN+ACK segment arrives, RTT_M is measured and is equal to 1.5 seconds.

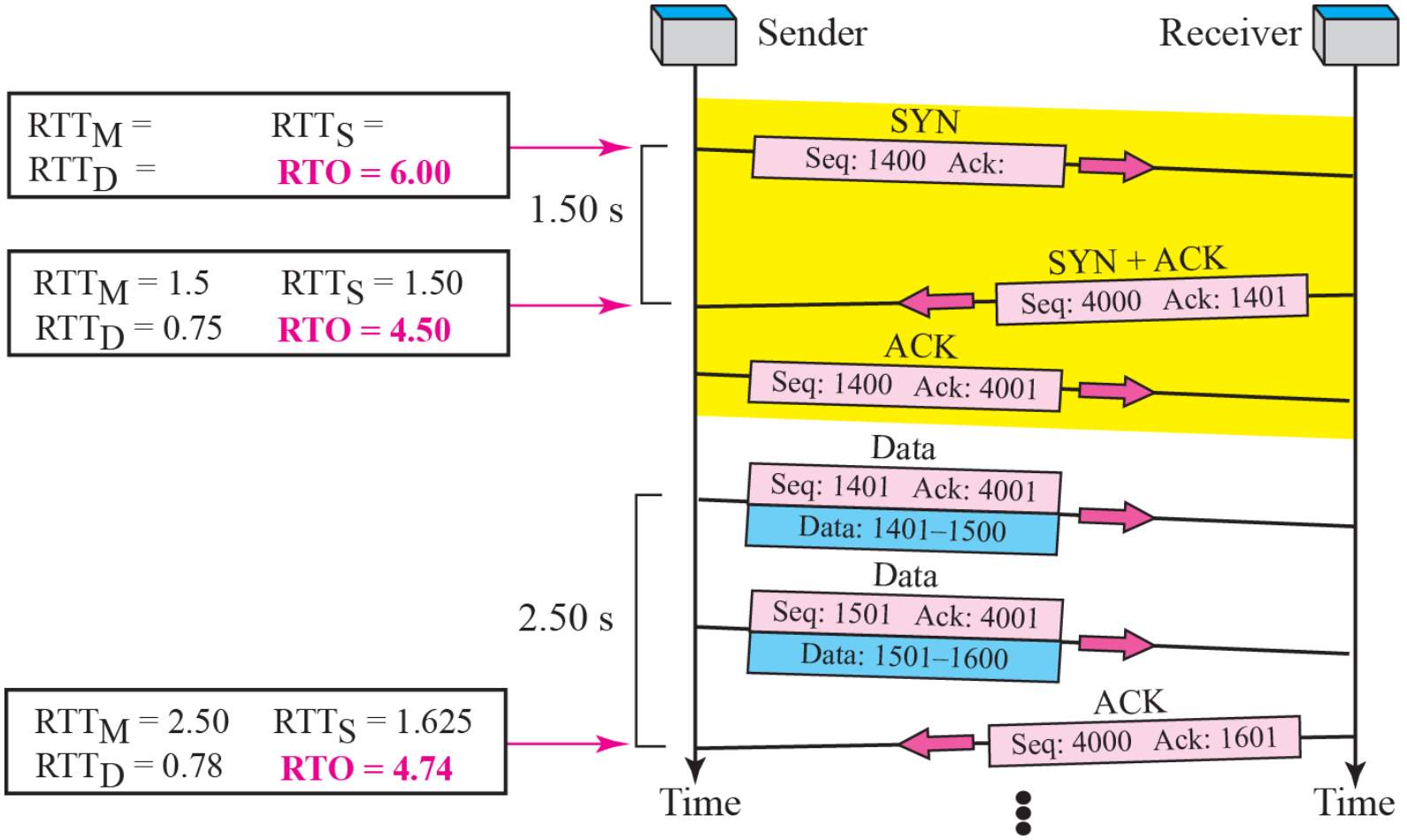
$$\begin{aligned} RTT_M &= 1.5 \\ RTT_S &= 1.5 \\ RTT_D &= (1.5) / 2 = 0.75 \\ RTO &= 1.5 + 4 \times 0.75 = 4.5 \end{aligned}$$

Example 15.3 *Continued*

3. When the first data segment is sent, a new RTT measurement starts. No RTT measurement starts for the second data segment because a measurement is already in progress. The arrival of the last ACK segment is used to calculate the next value of RTT_M. Although the last ACK segment acknowledges both data segments (cumulative), its arrival finalizes the value of RTT_M for the first segment. The values of these variables are now as shown below.

$$RTT_M = 2.5$$
$$RTT_S = 7/8 \times 1.5 + (1/8) \times 2.5 = 1.625$$
$$RTT_D = 3/4\ (7.5) + (1/4) \times |1.625 - 2.5| = 0.78$$
$$RTO = 1.625 + 4 \times 0.78 = 4.74$$

Figure 15.39 *Example 15.3*



Note

TCP does not consider the RTT of a retransmitted segment in its calculation of a new RTO.

Example 15.4

Figure 15.40 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 15.40 *Example 15.4*