Transmission Control Protocol (TCP)

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

Figure 1 *TCP/IP protocol suite*

Figure 1 shows the relationship of TCP to the other protocols in the TCP/IP protocol suite. TCP lies between the application layer and the network layer, and serves as the intermediary between the application programs and the network operations.

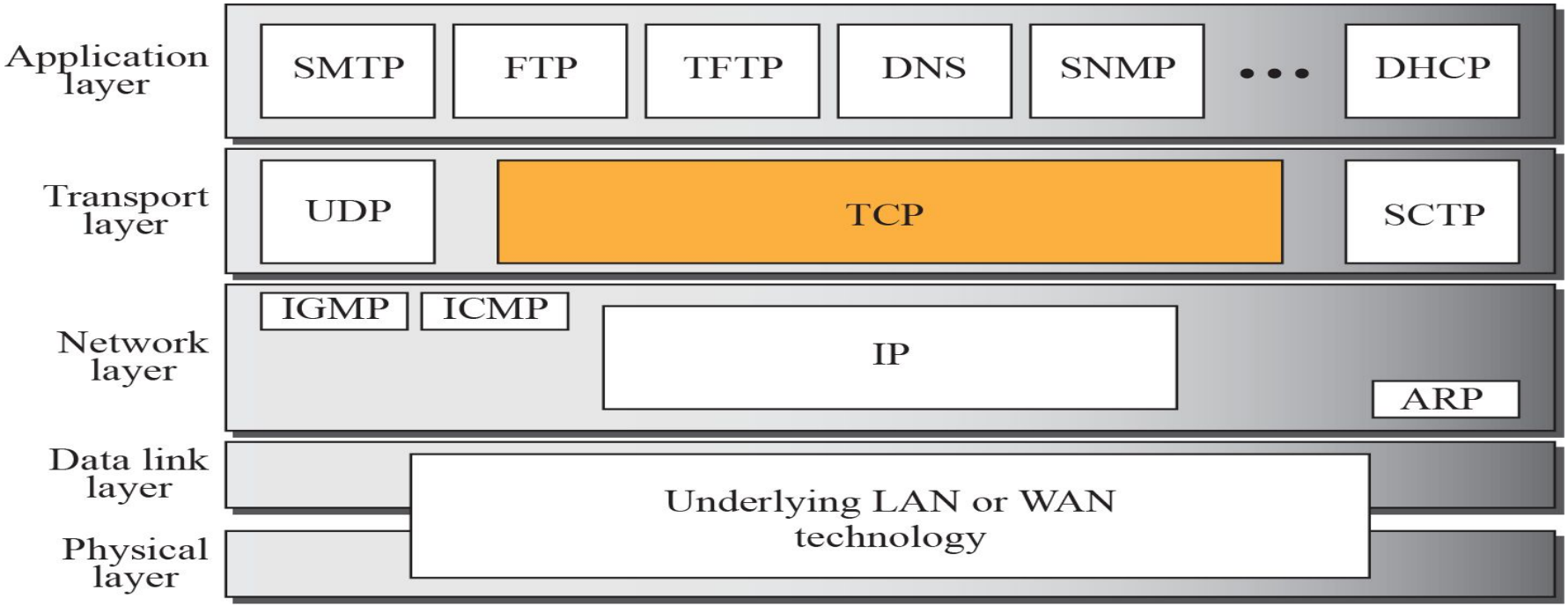


Table 15.1 *Well-known Ports used by TCP*

Port	*Protocol*	*Description*
7	Echo	Echoes a received datagram back to the sender
9	Discard	Discards any datagram that is received
11	Users	Active users
13	Daytime	Returns the date and the time
17	Quote	Returns a quote of the day
19	Chargen	Returns a string of characters
20 and 21	FTP	File Transfer Protocol (Data and Control)
23	TELNET	Terminal Network
25	SMTP	Simple Mail Transfer Protocol
53	DNS	Domain Name Server
67	BOOTP	Bootstrap Protocol
79	Finger	Finger
80	HTTP	Hypertext Transfer Protocol

Figure 2 *Stream delivery*

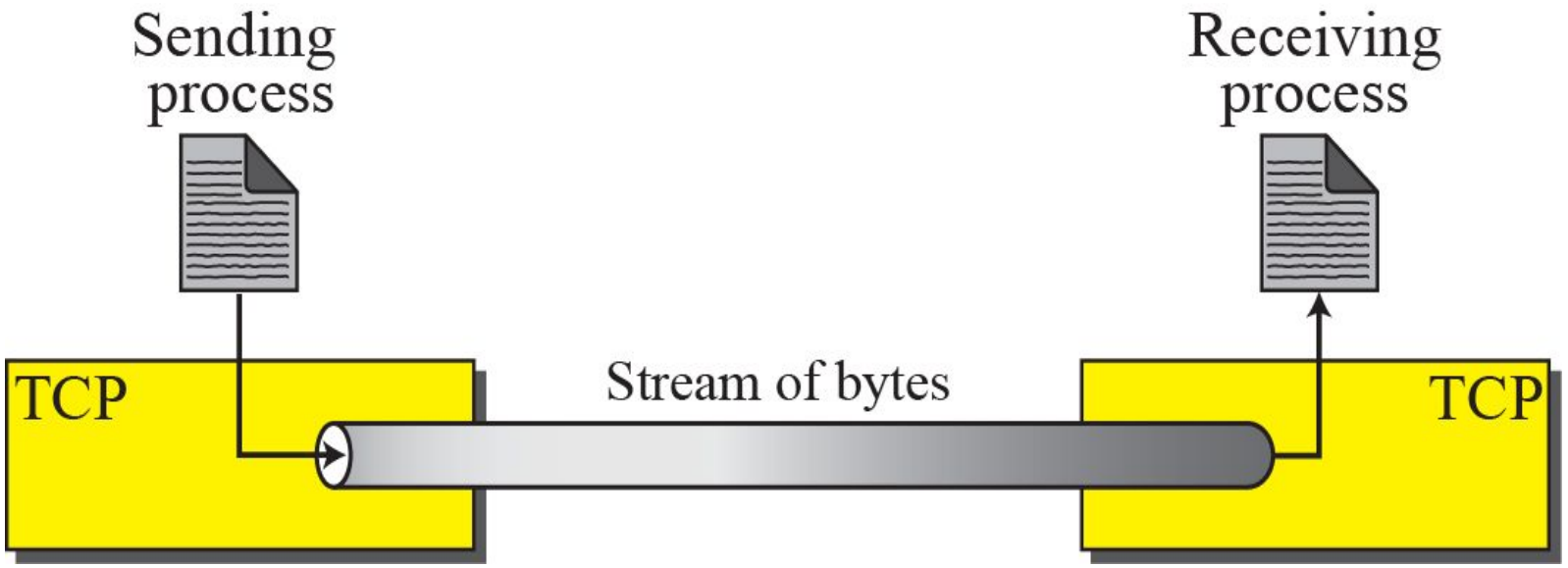


Figure 3 *Sending and receiving buffers*

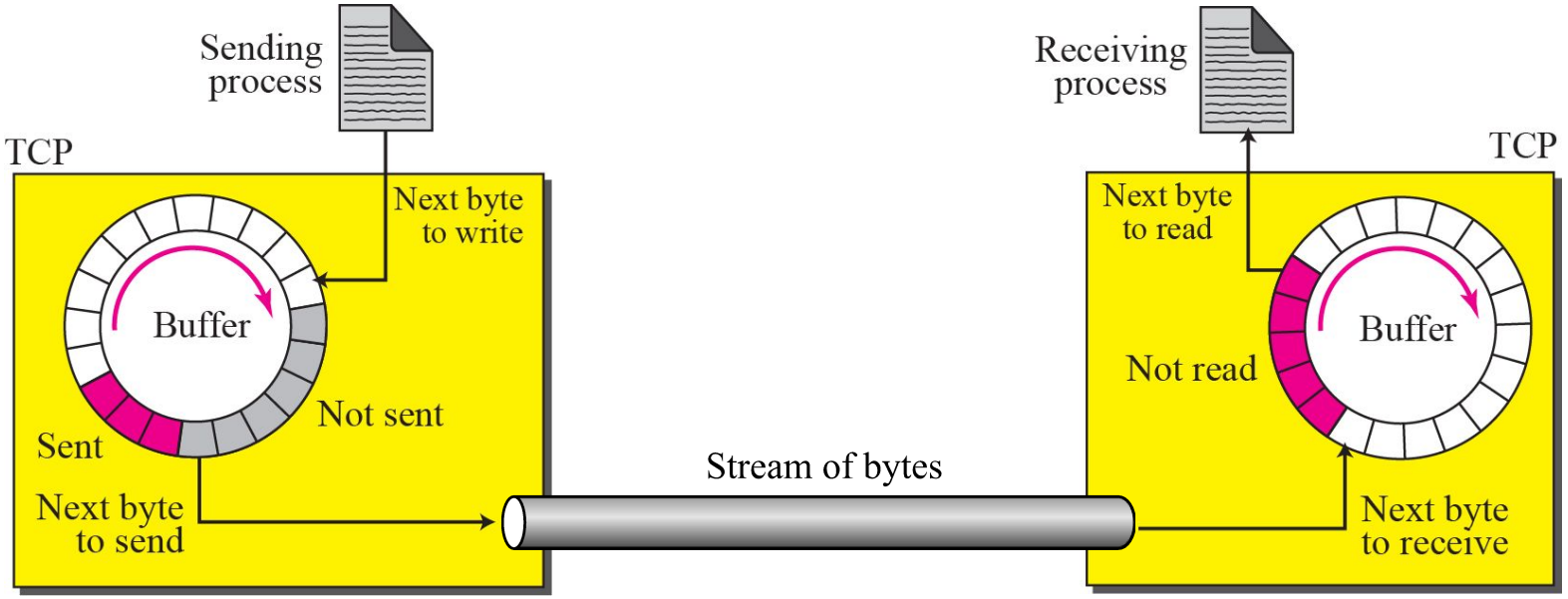
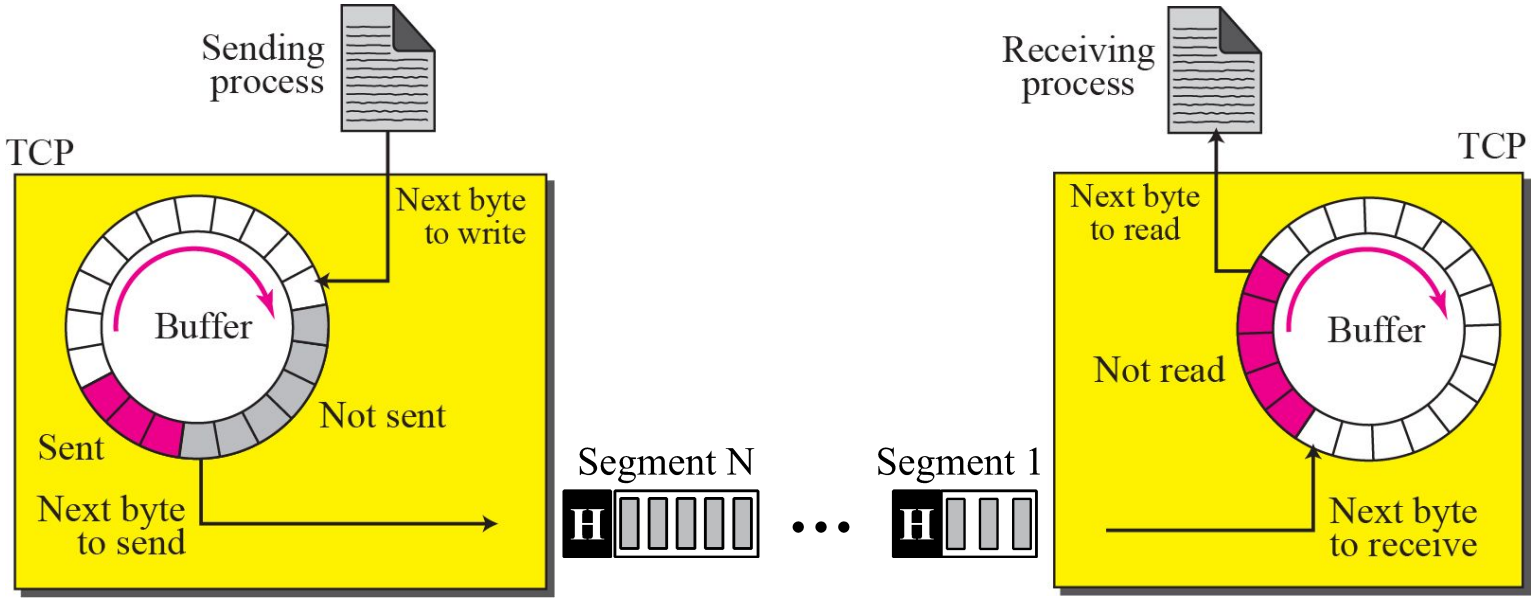


Figure 4 *TCP segments*



TCP FEATURES

TCP has several features that are given as follows.

- ✔ **Numbering System**
- ✔ **Flow Control**
- ✔ **Error Control**
- ✔ **Congestion Control**

Note

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

The numbering starts with an arbitrarily generated number.

Example 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.

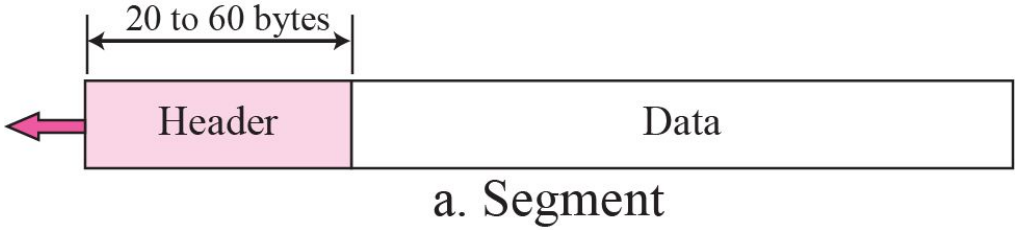
The acknowledgment number is cumulative.

SEGMENT

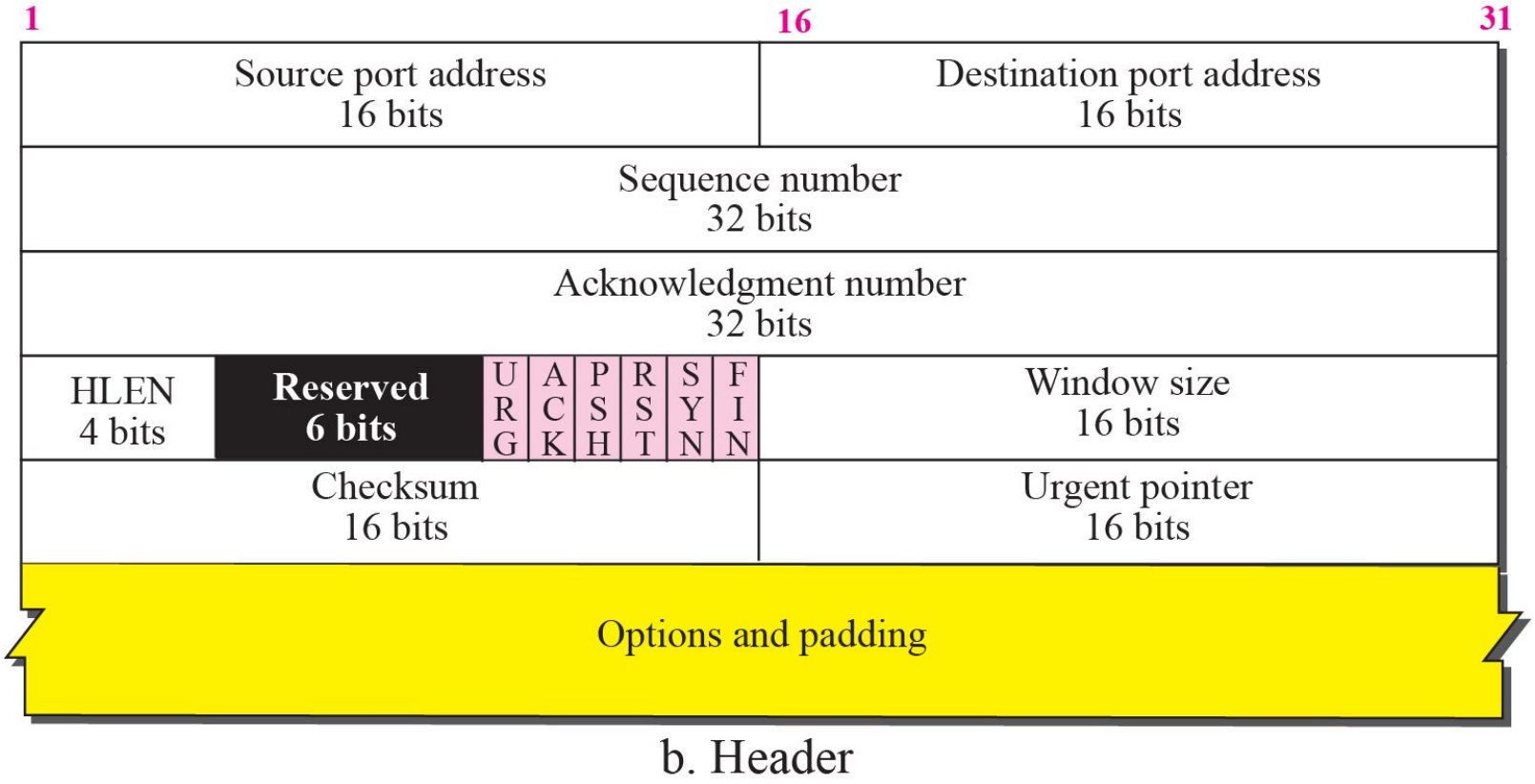
Before discussing TCP in more detail, let us discuss the TCP packets themselves. A packet in TCP is called a segment.

- ✔ **Format**
- ✔ **Encapsulation**

Figure 5 *TCP segment format*



a. Segment



b. Header

Figure 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

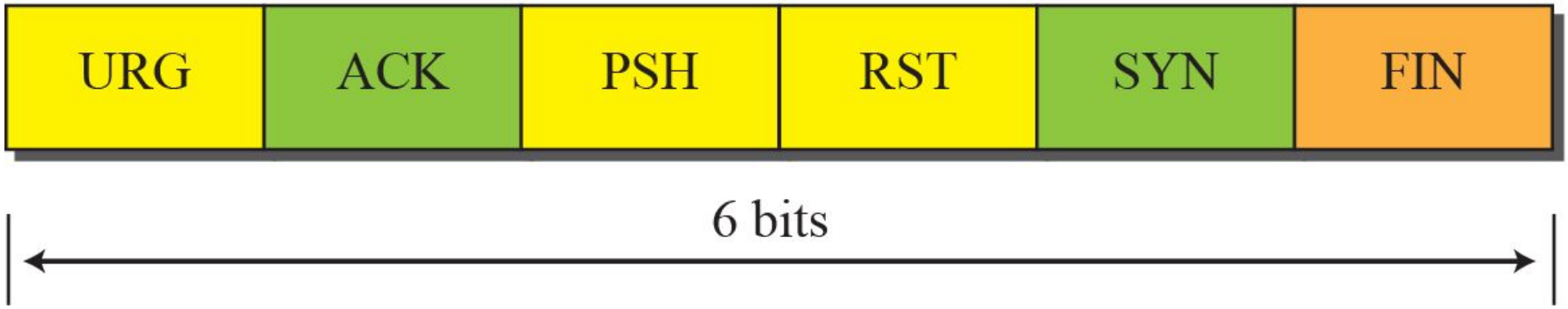
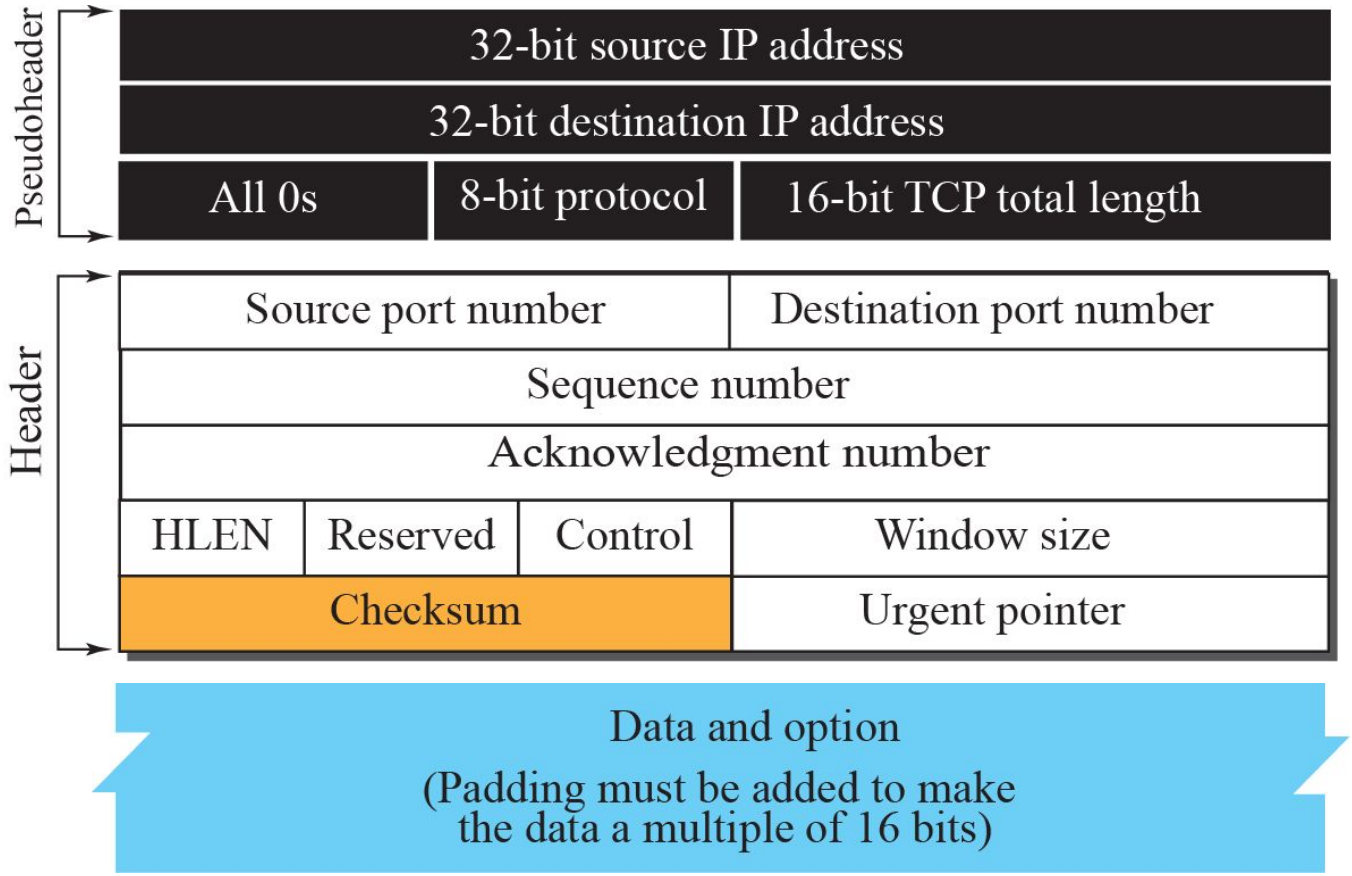


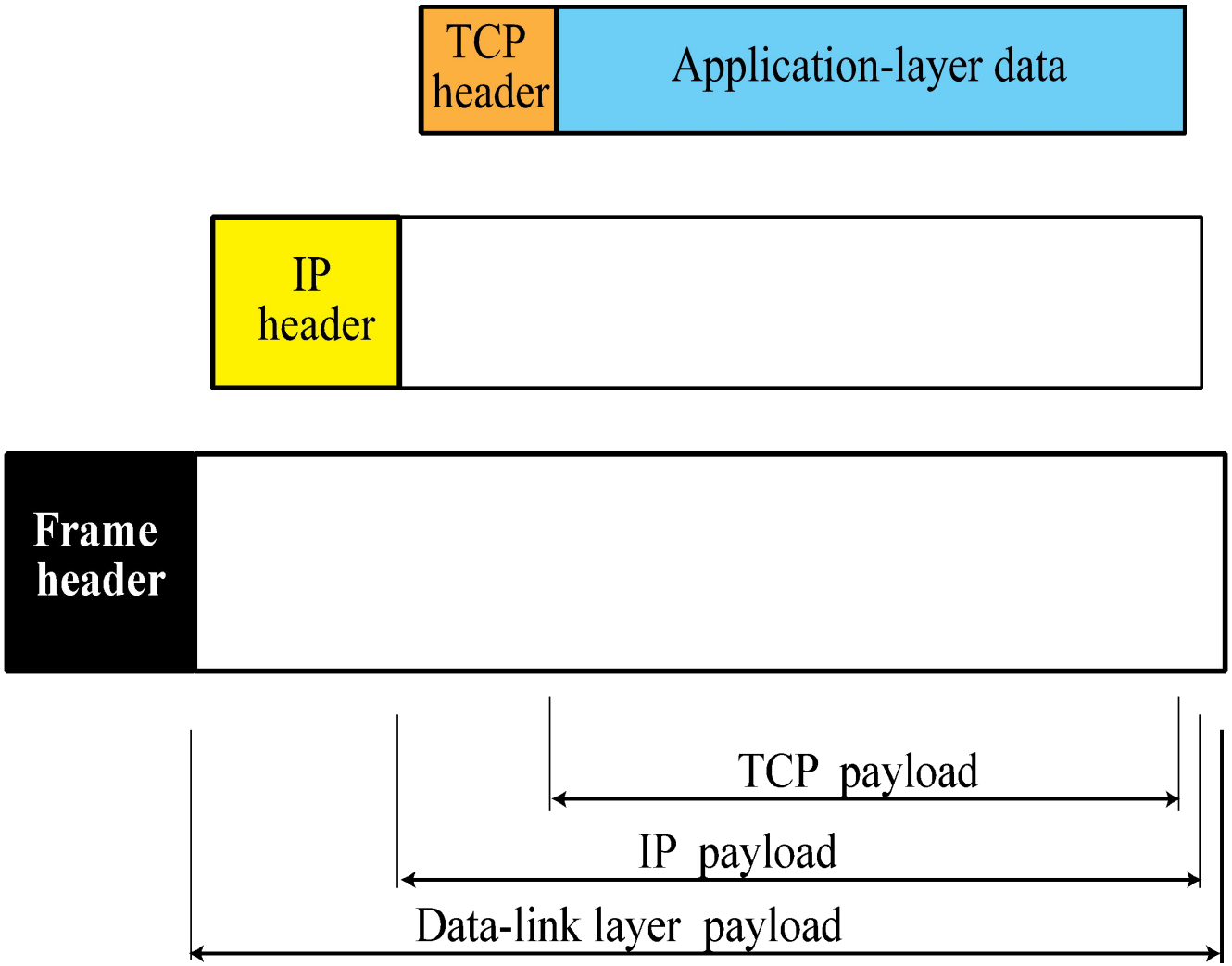
Figure 7 *Pseudoheader added to the TCP segment*



Note

The use of the checksum in TCP is mandatory.

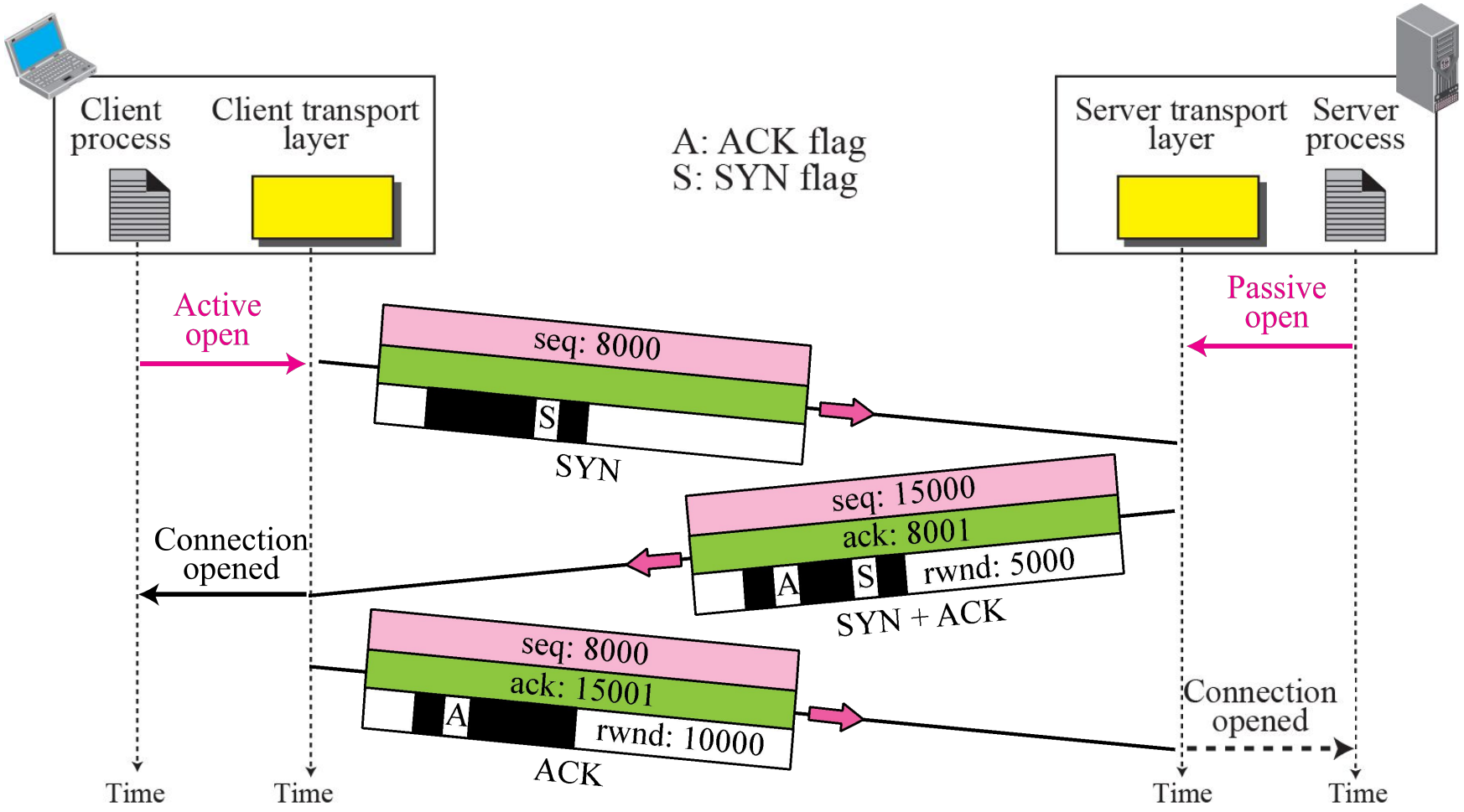
Figure 8 *Encapsulation*



A TCP CONNECTION

- **TCP is connection-oriented. It establishes a virtual path between the source and destination.**
- **All of the segments belonging to a message are sent over this virtual path.**
- **You may wonder how TCP, which uses the services of IP, a connectionless protocol, can be connection-oriented.**
- **TCP uses the services of IP to deliver individual segments to the receiver, but it controls the connection itself.**
- **If a segment is lost or corrupted, it is retransmitted.**

Figure 9 *Connection establishment using three-way handshake*



Note

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

Note

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

Note

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

Figure 10 *Data Transfer*

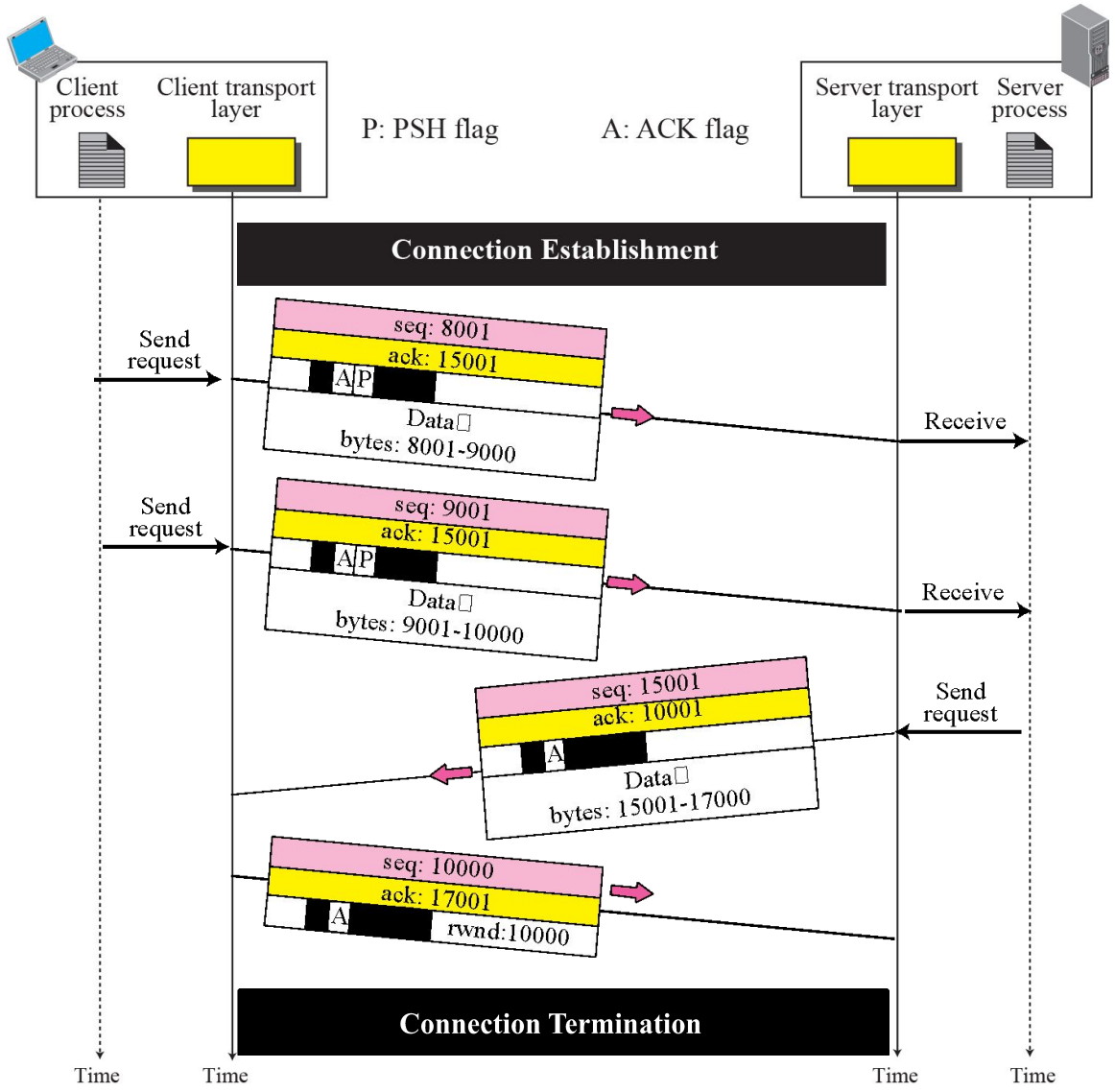
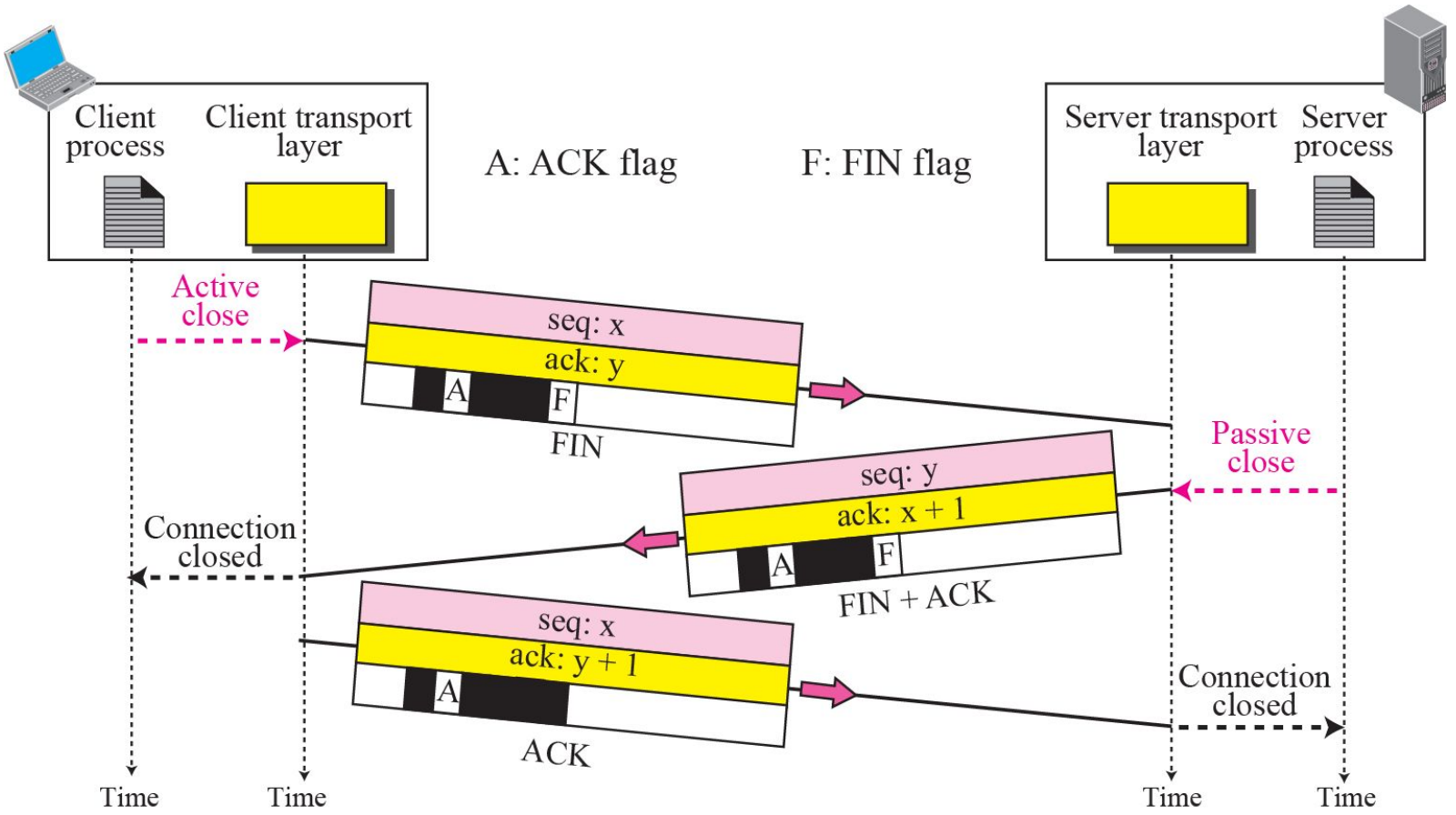


Figure 11 *Connection termination using three-way handshake*



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 12 *Half-Close*

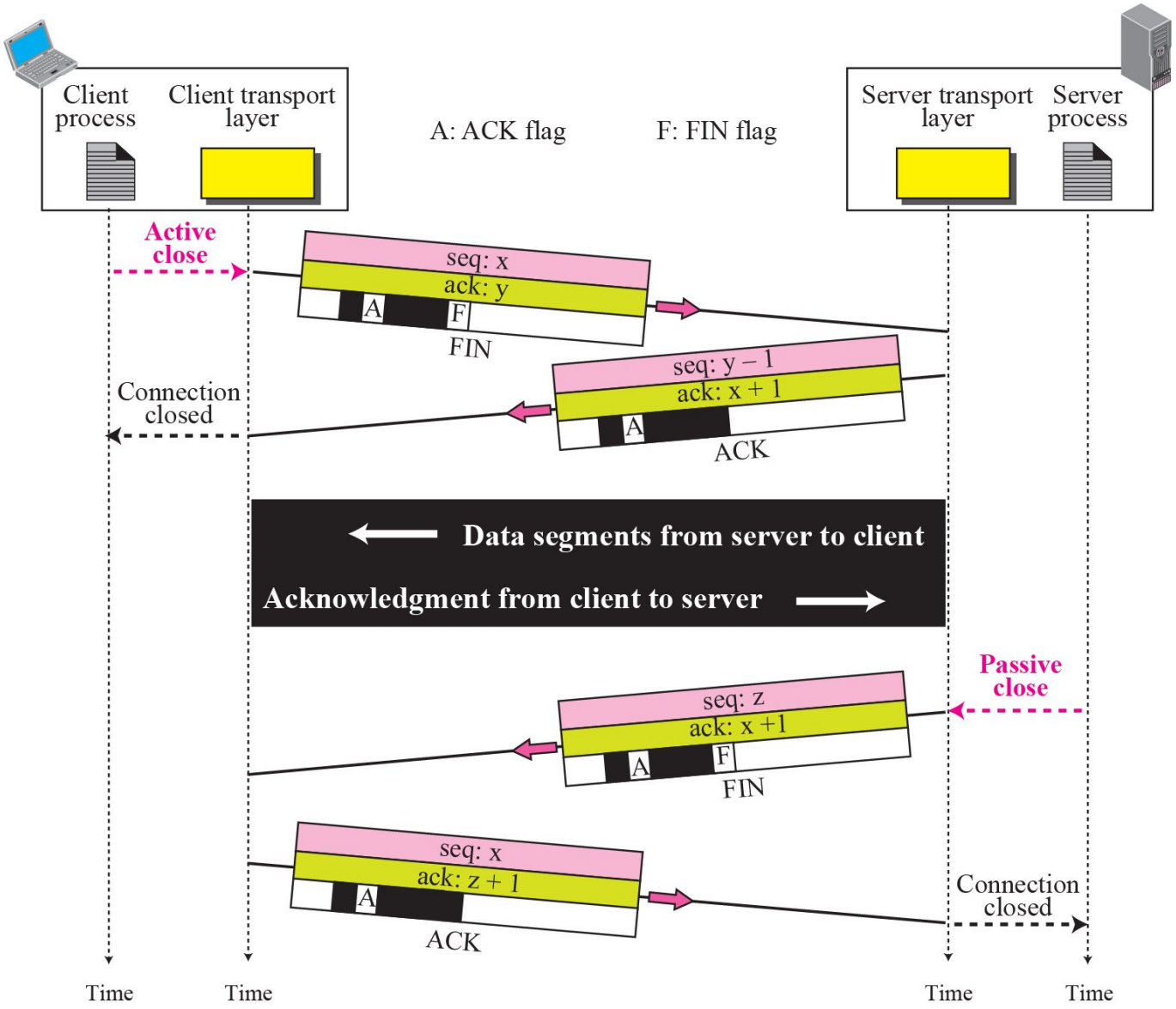


Fig 15 *Time-line diagram for connection establishment and half-close termination*

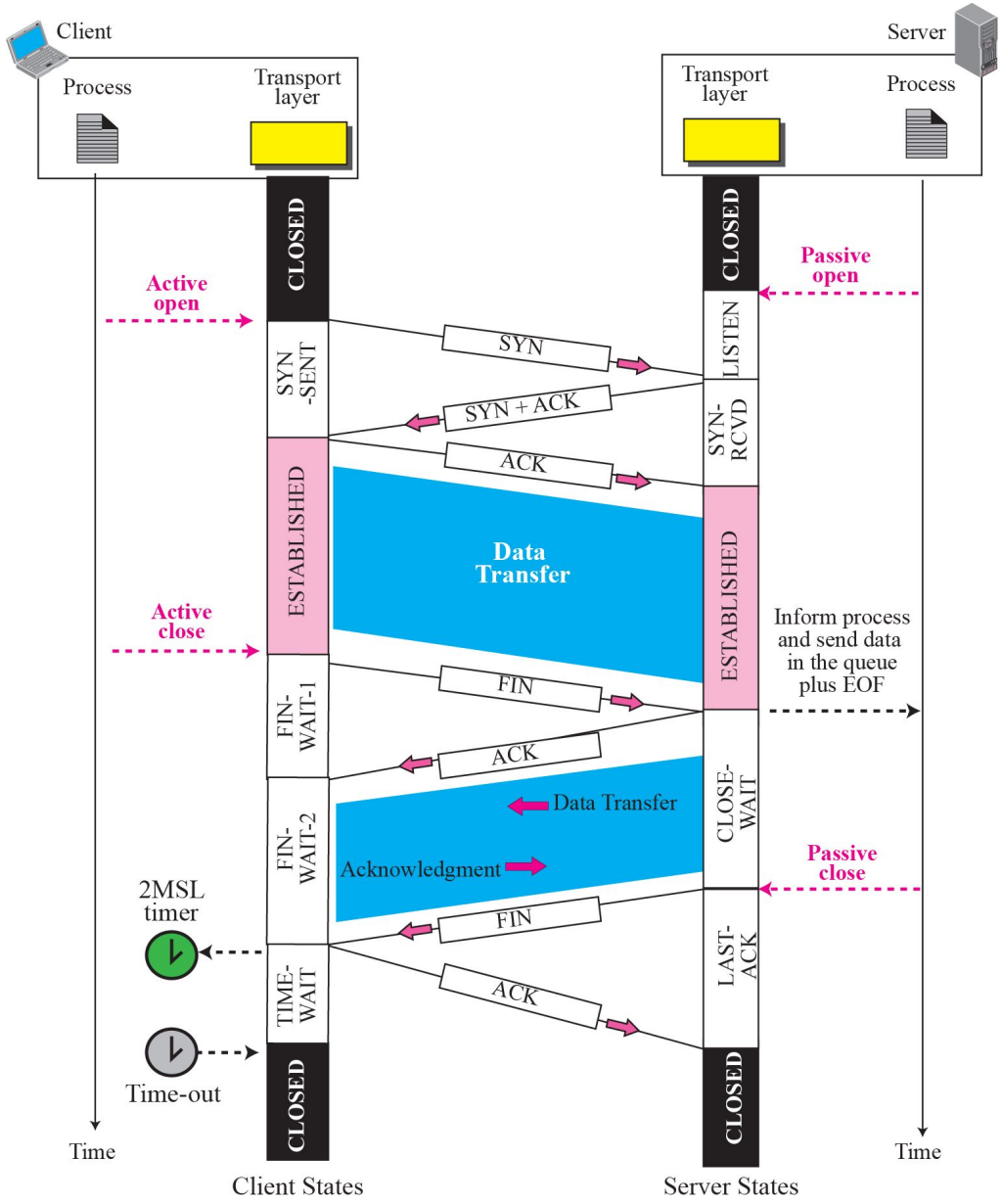


Figure 17 *Time line for a common scenario*

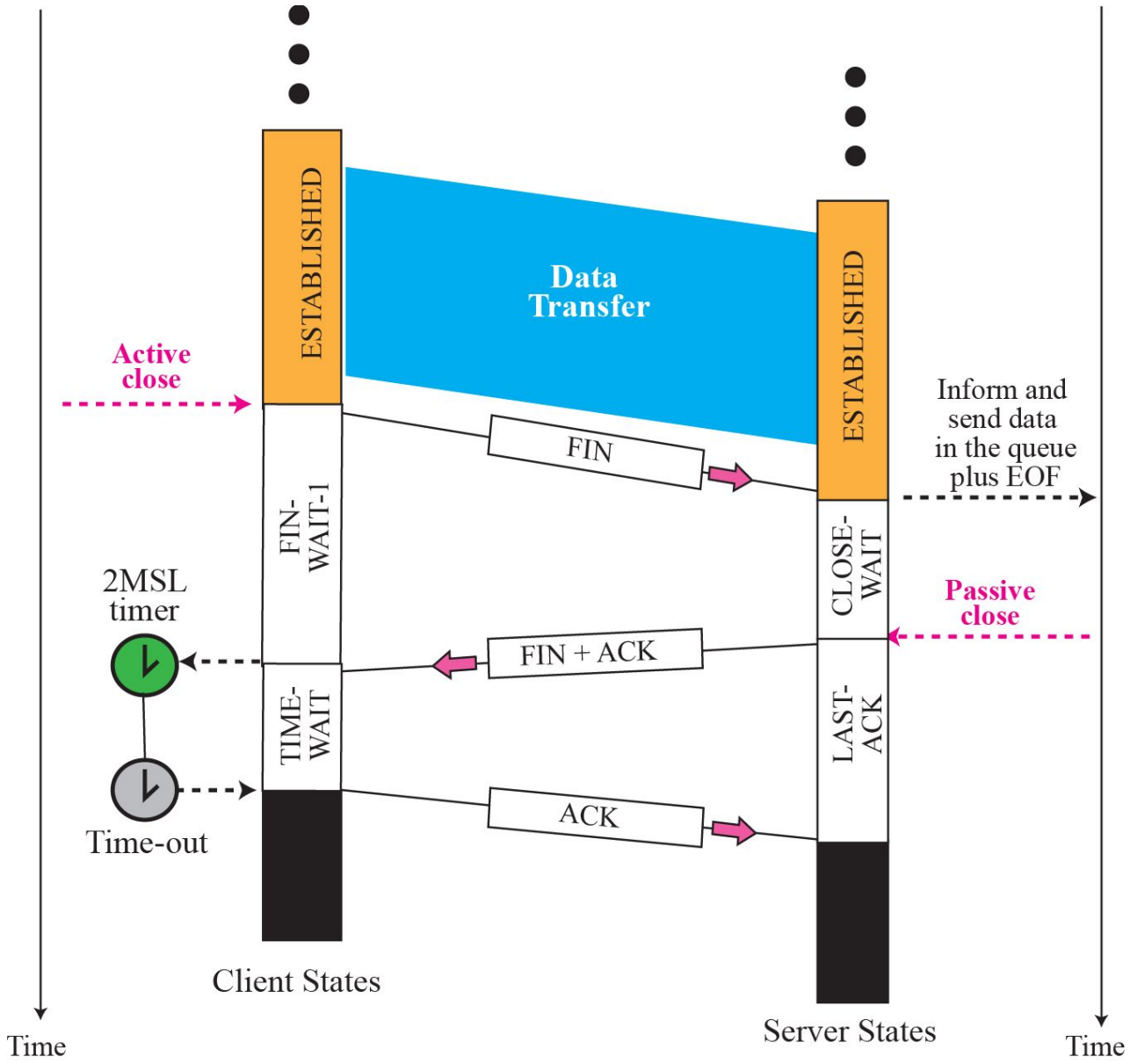


Figure 18 *Simultaneous open*

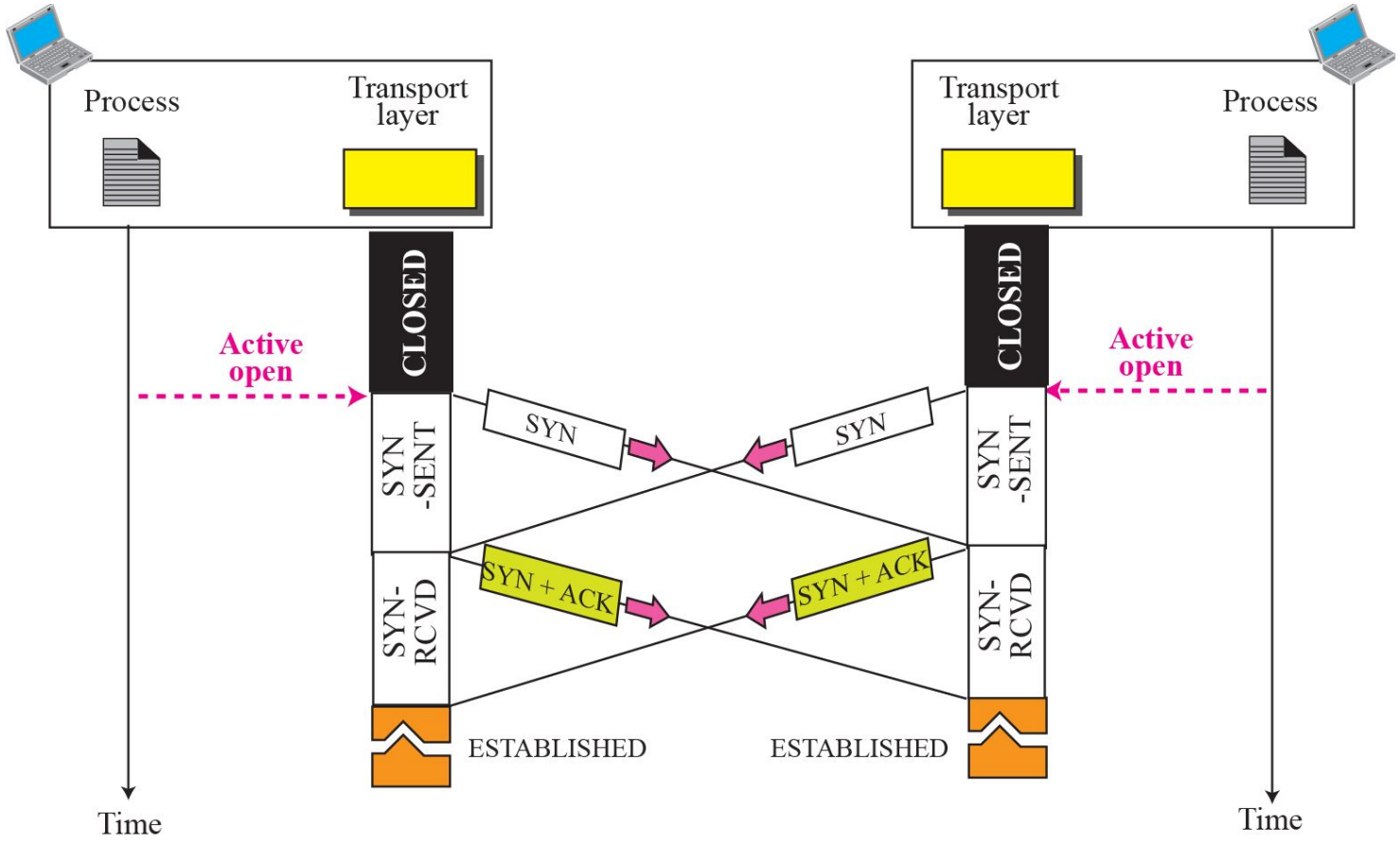


Figure 19 *Simultaneous close*

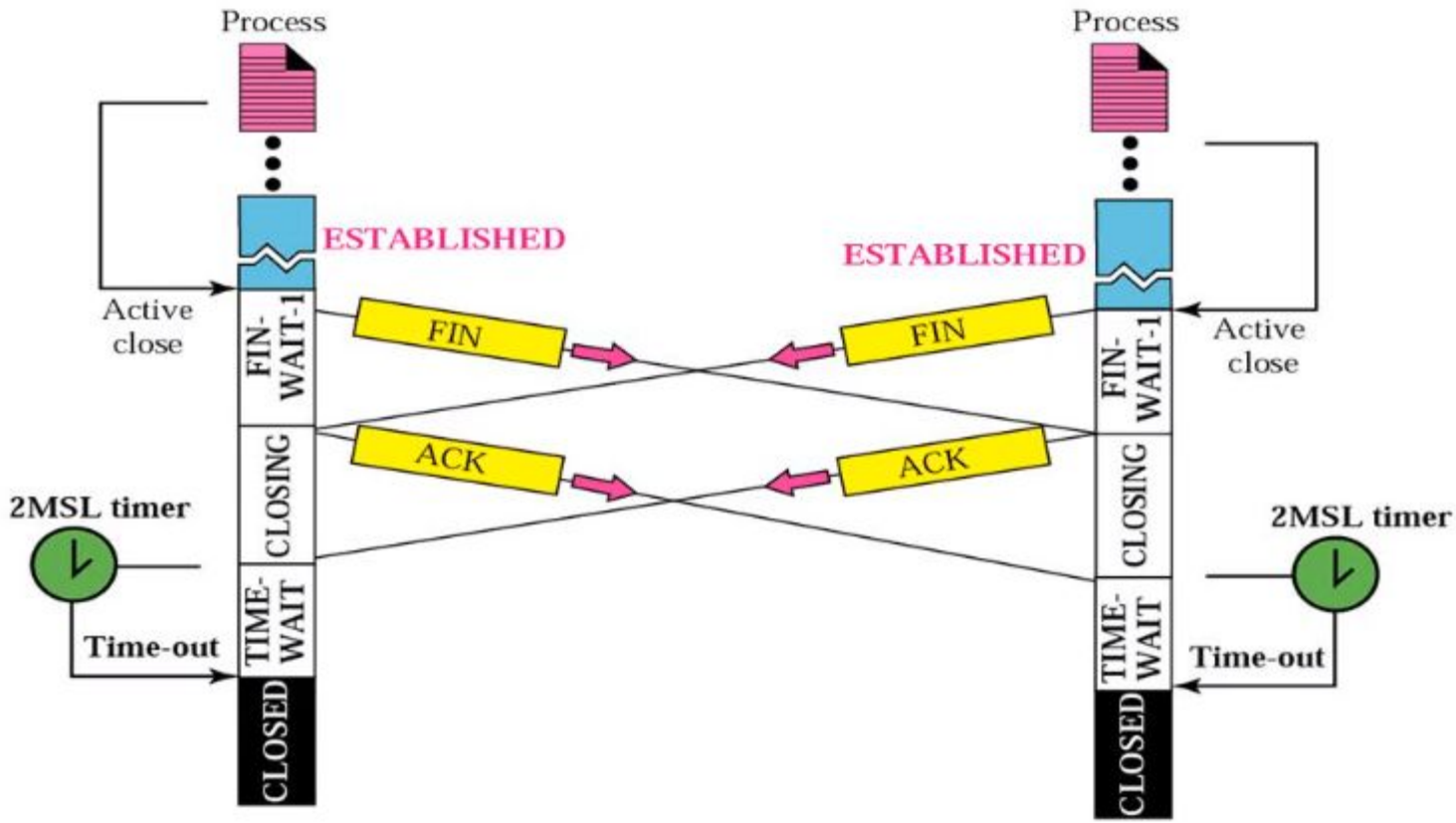


Figure 20 *Denying a connection*

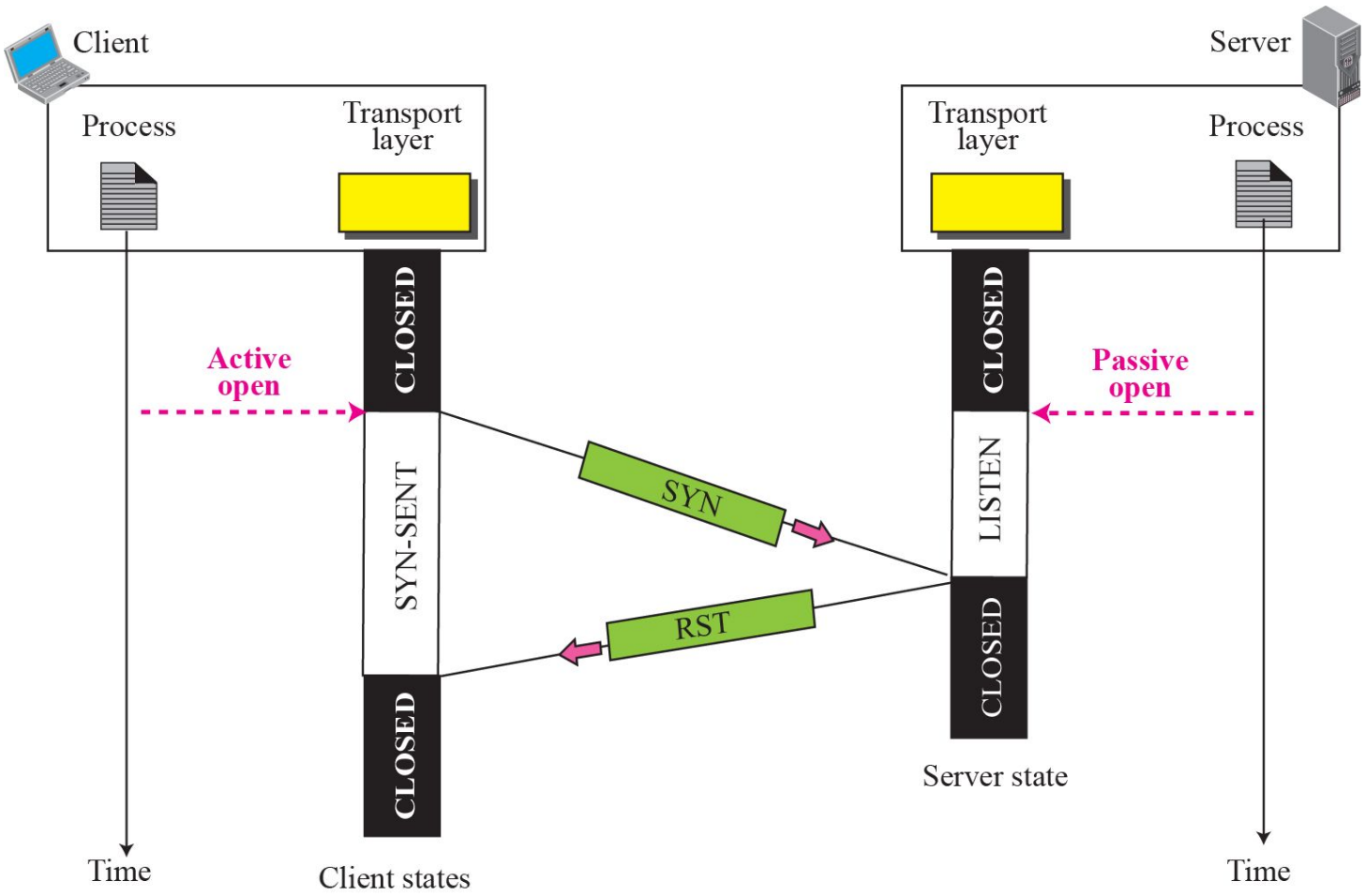
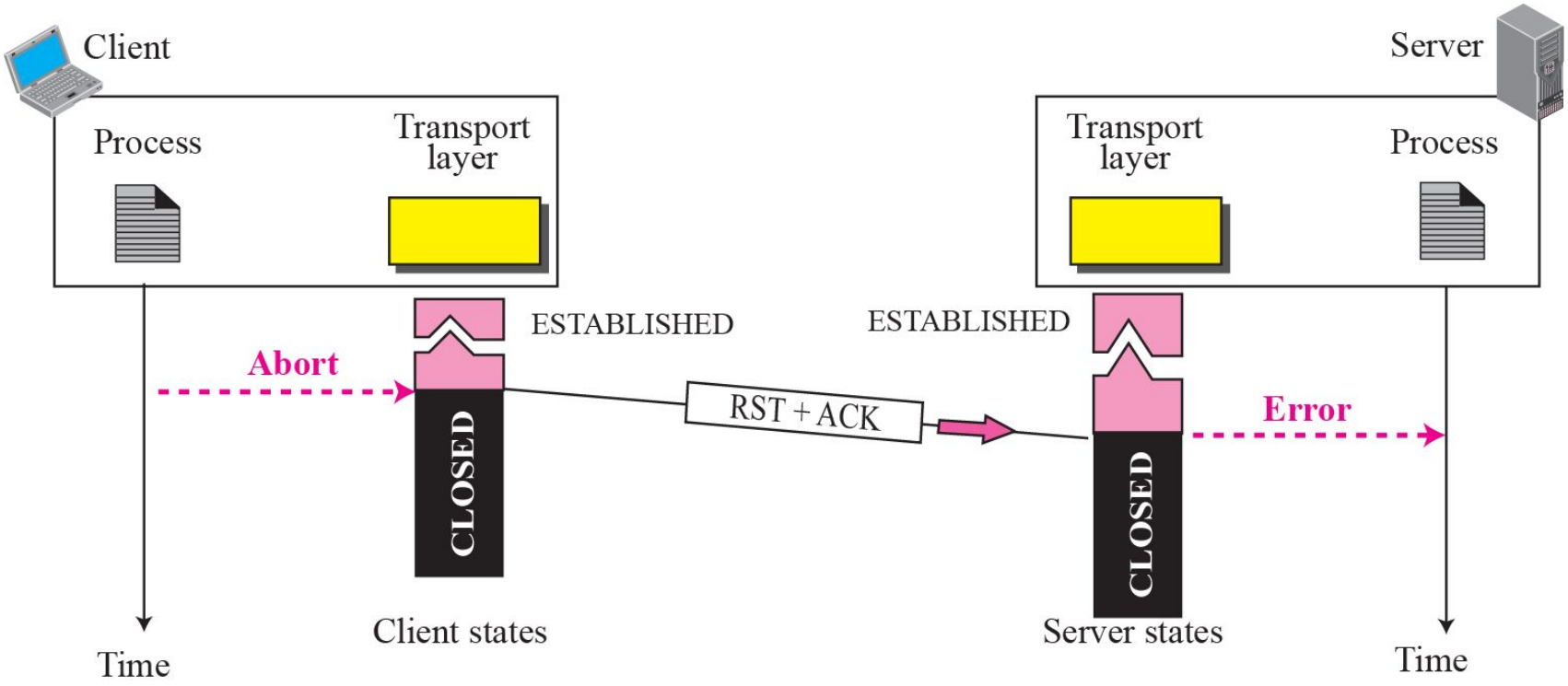


Figure 21 *Aborting a connection*

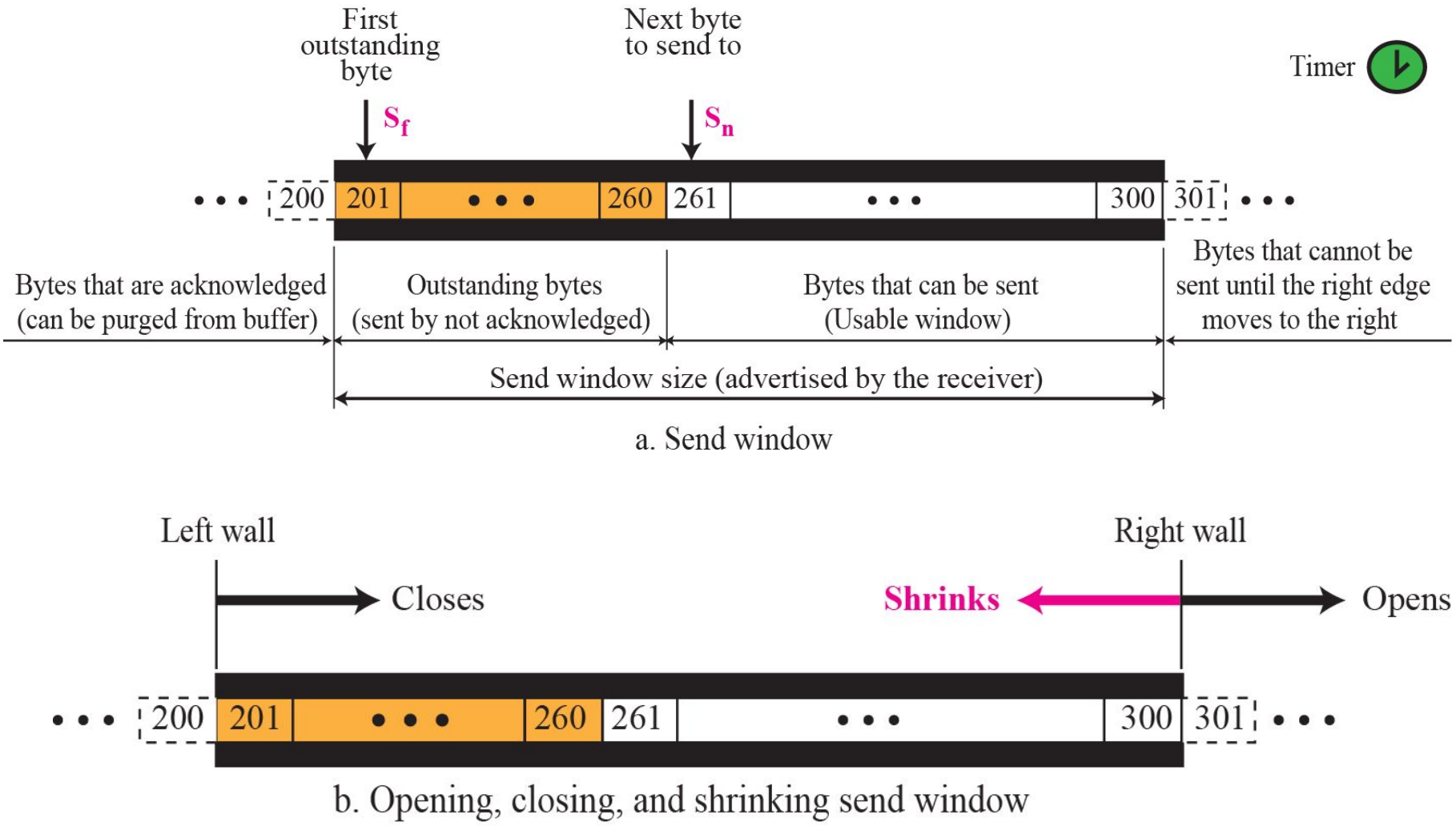


WINDOWS IN TCP

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 assumption that communication is only unidirectional; the bidirectional communication can be inferred using two unidirectional communications with piggybacking.

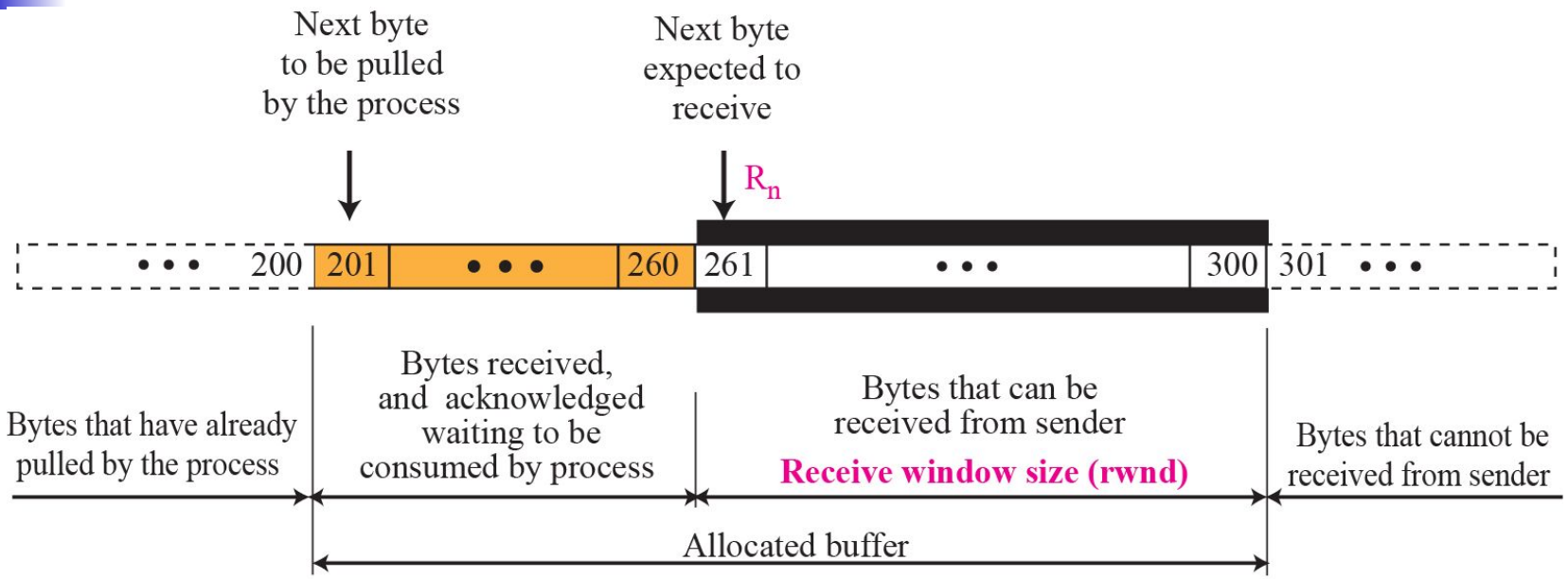
Figure 22 *Send window in TCP*



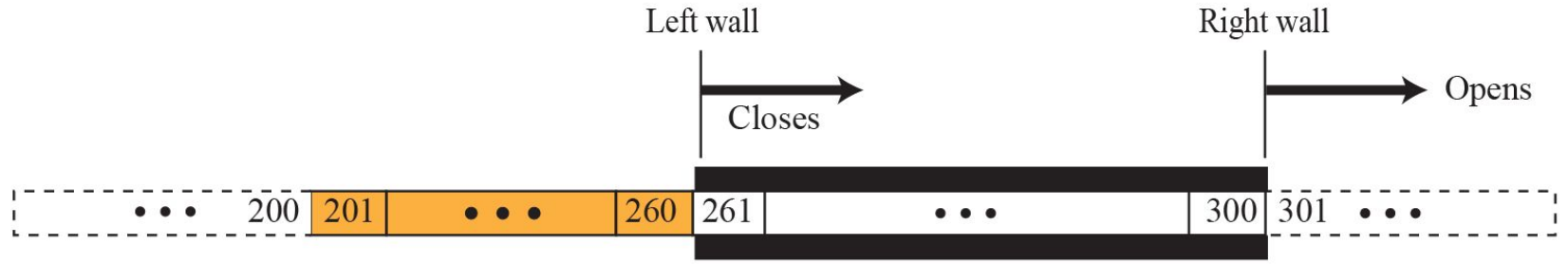
Differences

- One difference is the nature of entities related to the window. The window in SR numbers pockets, but the window in the TCP numbers bytes. Although actual transmission in TCP occurs segment by segment, the variables that control the window are expressed in bytes.
- The second difference is that, in some implementations, TCP can store data received from the process and send them later, but we assume that the sending TCP is capable of sending segments of data as soon as it receives them from its process.
- Another difference is the number of timers. The theoretical Selective Repeat protocol may use several timers for each packet sent, but the TCP protocol uses only one timer.

Figure 23 *Receive window in TCP*



a. Receive window and allocated buffer



b. Opening and closing of receive window

Differences

- The first difference is that TCP allows the receiving process to pull data at its own pace. This means that part of the allocated buffer at the receiver may be occupied by bytes that have been received and acknowledged, but are waiting to be pulled by the receiving process. The receive window size is then always smaller or equal to the buffer size, as shown in the above figure. The receiver window size determines the number of bytes that the receive window can accept from the sender before being overwhelmed (flow control). In other words, the receive window size, normally called *rwnd*, can be determined as:

  rwnd = buffer size - **number of waiting bytes to be pulled**

- The second difference is the way acknowledgments are used in the TCP protocol. Remember that an acknowledgement in SR is selective, defining the uncorrupted packets that have been received. The major acknowledgment mechanism in TCP is a cumulative acknowledgment announcing the next expected byte to receive (in this way TCP looks like GBN discussed in Chapter 13). The new versions of TCP, however, uses both cumulative and selective acknowledgements.

FLOW CONTROL

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

- **Opening and Closing Windows**
- **Shrinking of Windows**
- **Silly Window Syndrome**

Figure 24 *Data flow and flow control feedbacks in TCP*

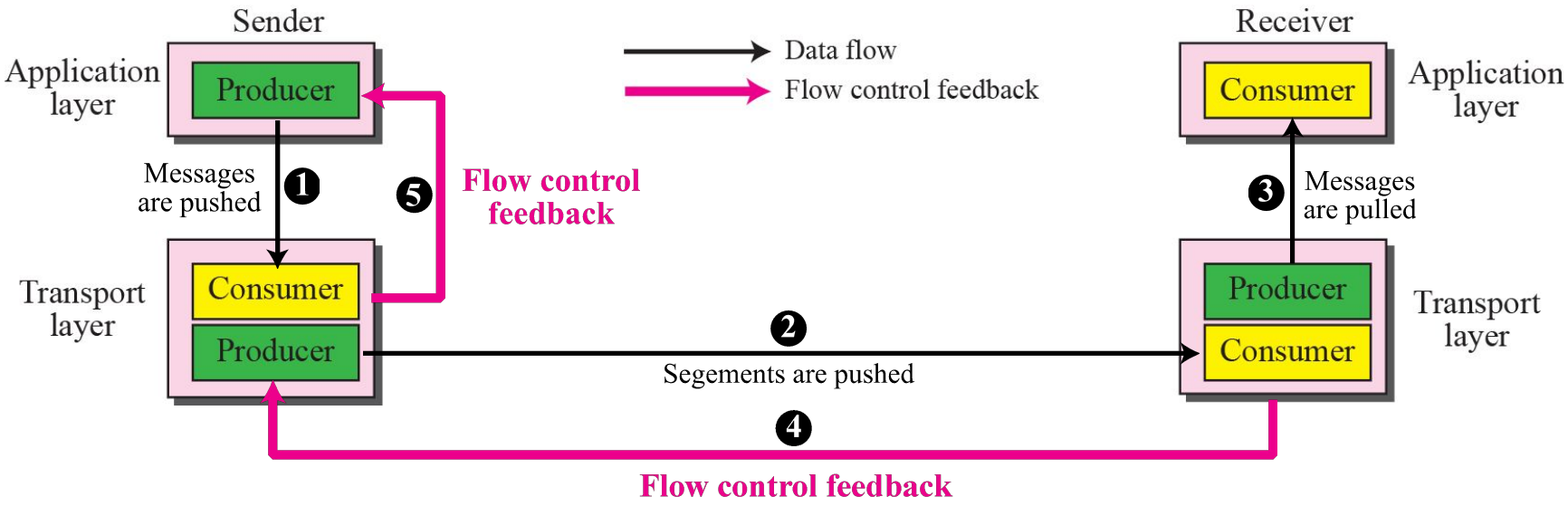
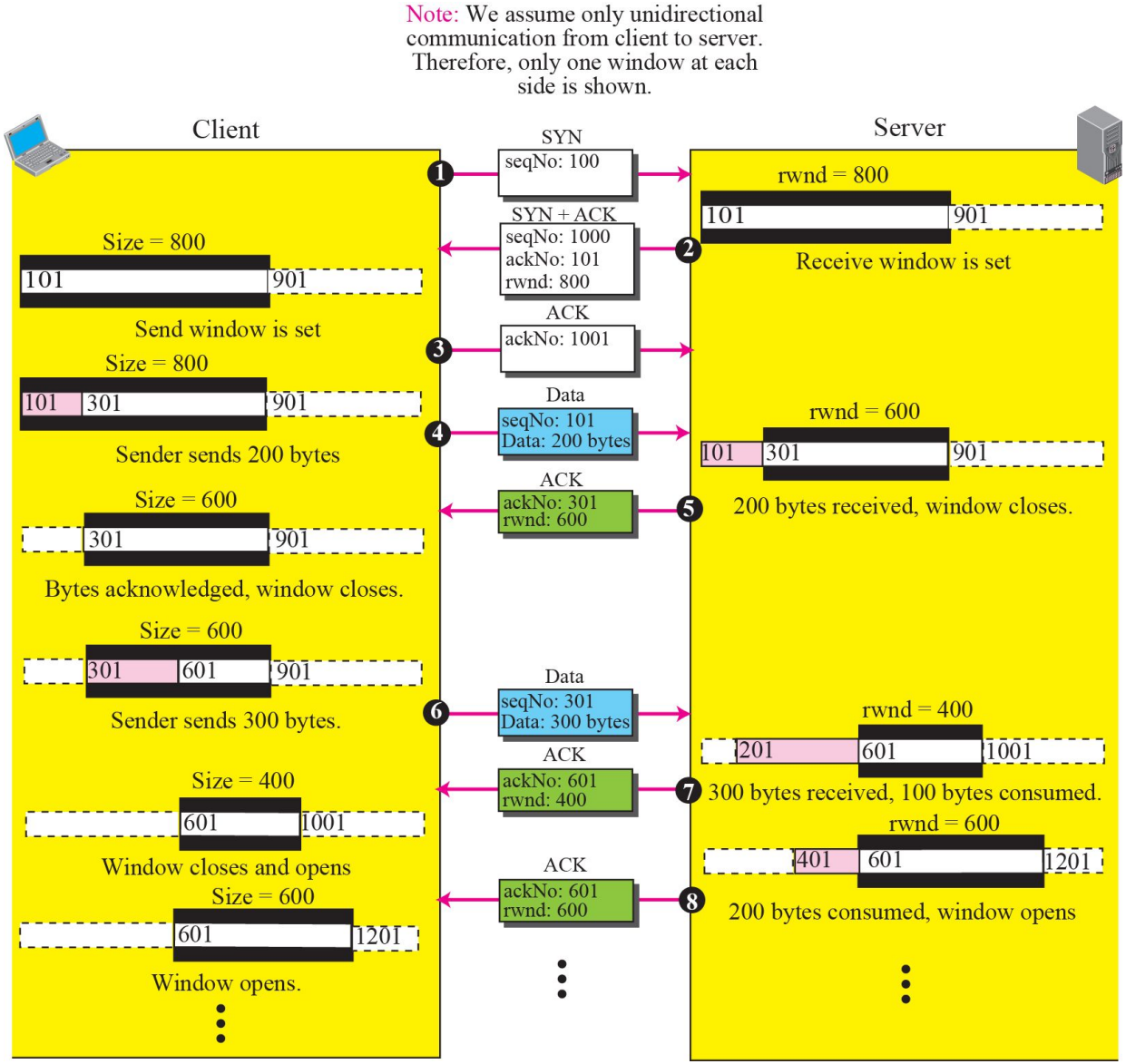


Figure 25 *An example of flow control: Opening and Closing Windows*

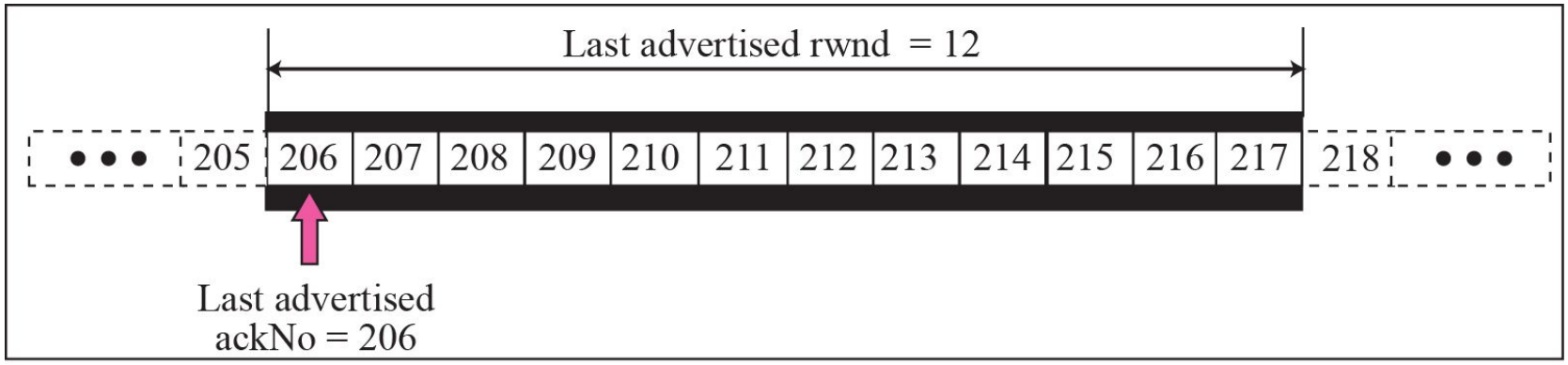


Example 15.2

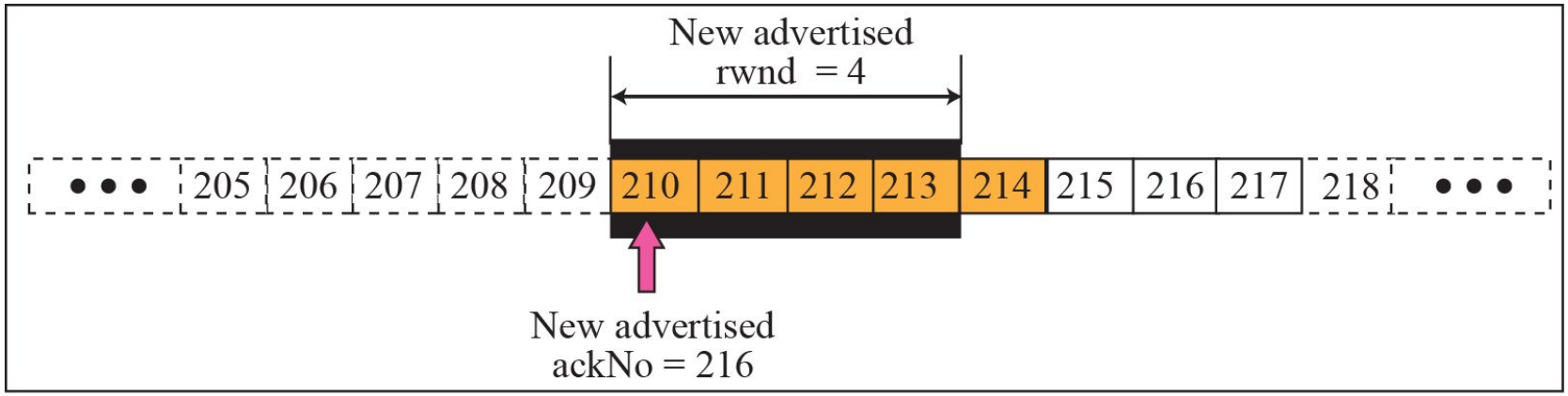
- Next figure shows the reason for the mandate in window shrinking.
  - **new ackNo + new rwnd ≥ last ackNo +last rwnd**

- 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 26 *Example 2*



a. The window after the last advertisement



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

Silly Window Syndrome

- A serious problem can arise in the sliding window operation when either the sending application program creates data slowly or the receiving application program consumes data slowly, or both.
- Any of these situations results in the sending of data in very small segments, which reduces the efficiency of the operation.
- For example, if TCP sends segments containing only 1 byte of data, it means that a 41-byte datagram (20 bytes of TCP header and 20 bytes of IP header) transfers only 1 byte of user data.
- Here the overhead is 41/1, which indicates that we are using the capacity of the network very inefficiently.
- The inefficiency is even worse after accounting for the data link layer and physical layer overhead. This problem is called the **silly window syndrome.**
- For each site, we first describe how the problem is created and then give a proposed solution.

- The solution is to prevent the sending TCP from sending the data byte by byte.
- The sending TCP must be forced to wait and collect data to send in a larger block.
- How long should the sending TCP wait? If it waits too long, it may delay the process.
- if it does not wait long enough, it may end up sending small segments. Nagle found an elegant solution

Nagle's Algorithm

- The sending TCP sends the first piece of data it receives from the sending application program even if it is only 1 byte.
- After sending the first segment, the sending TCP accumulates data in the output buffer and waits until either the receiving TCP sends an acknowledgment or until enough data has accumulated to fill a maximum-size segment. At this time, the sending TCP can send the segment.
- Step 2 is repeated for the rest of the transmission. Segment 3 is sent immediately if an acknowledgment is received for segment 2, or if enough data have accumulated to fill a maximum-size segment.

Clark's solution

- **Clark's solution** is to send an acknowledgment as soon as the data arrive, but to announce a window size of zero until either there is enough space to accommodate a segment of maximum size or until at least half of the receive buffer is empty.

Congestion control

- An important issue in the Internet is **congestion.**
- Congestion in a network may occur if the **load** on the network—the number of packets sent to the network—is greater than the *capacity* of the network—the number of packets a network can handle.
- **Congestion control** refers to the mechanisms and techniques to control the congestion and keep the load below the capacity

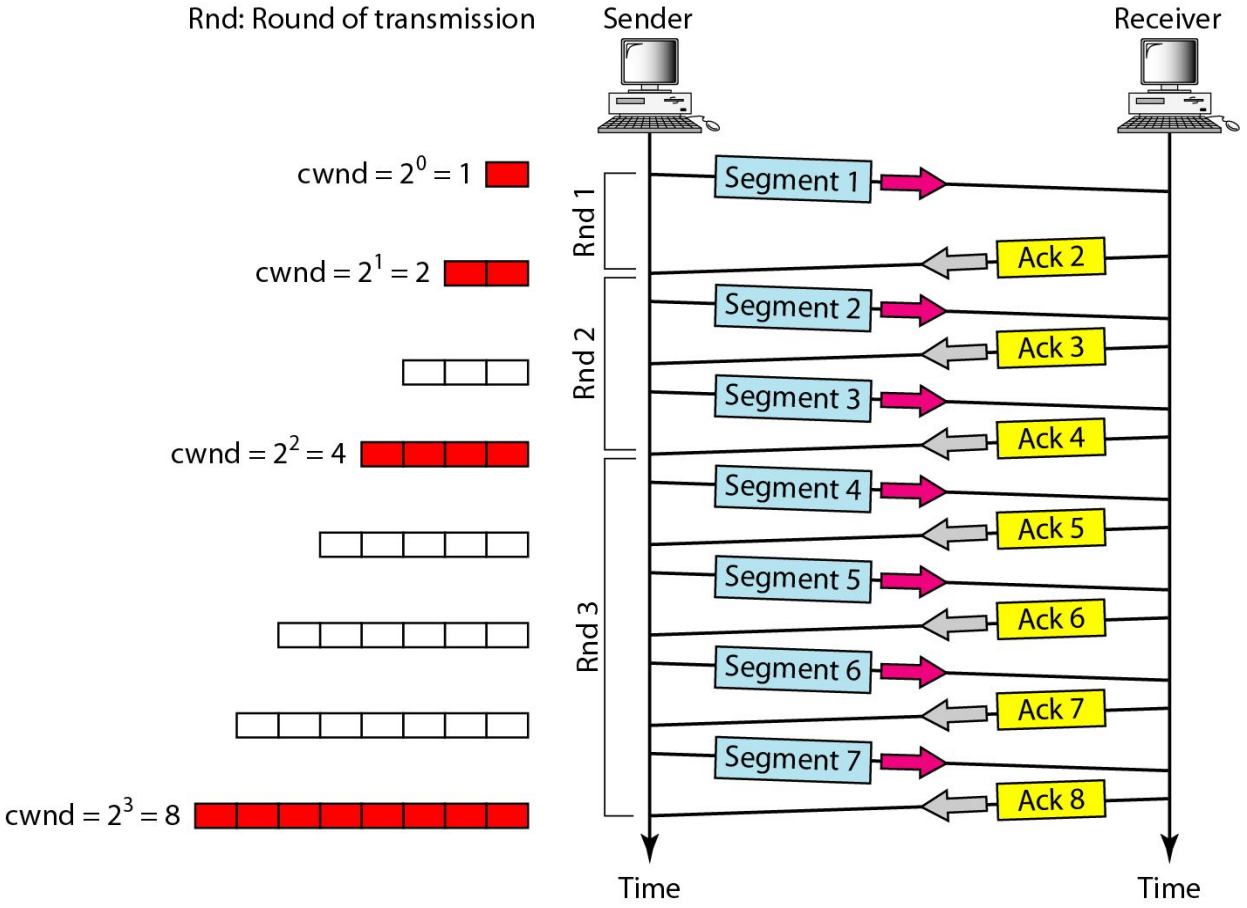
Congestion Window

Actual window size =minimum (rwnd, cwnd)

Congestion Policy

- TCP's general policy for handling congestion is based on three phases: slow start, congestion avoidance, and congestion detection.
- In the slow start phase, the sender starts with a slow rate of transmission, but increases the rate rapidly to reach a threshold.
- When the threshold is reached, the rate of increase is reduced.
- Finally if ever congestion is detected, the sender goes back to the slow start or congestion avoidance phase.

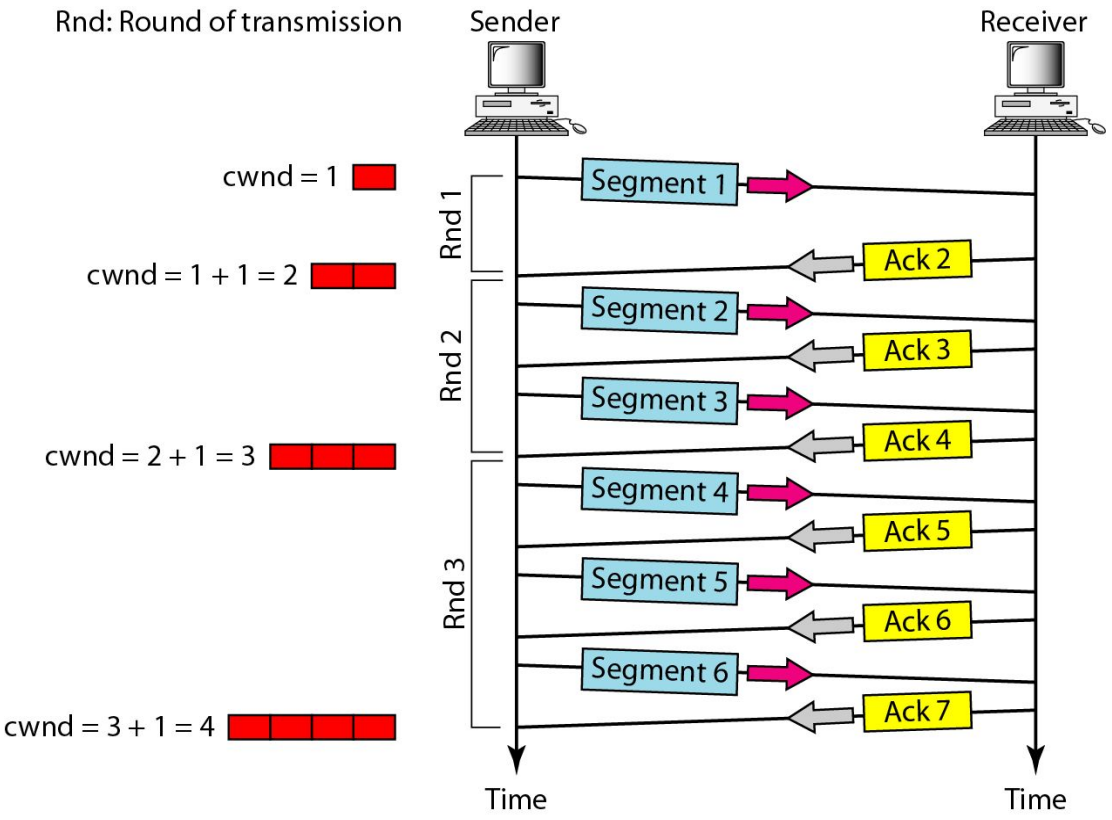
Figure 24.8 *Slow start, exponential increase*



Note

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

Figure 24.9 *Congestion avoidance, additive increase*



Note

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

Congestion Detection: Multiplicative Decrease

An implementation reacts to congestion detection in one of the following ways:

- If detection is by time-out, a new slow start phase starts.
- If detection is by three ACKs, a new congestion avoidance phase starts.

Figure 24.10 *TCP congestion policy summary*

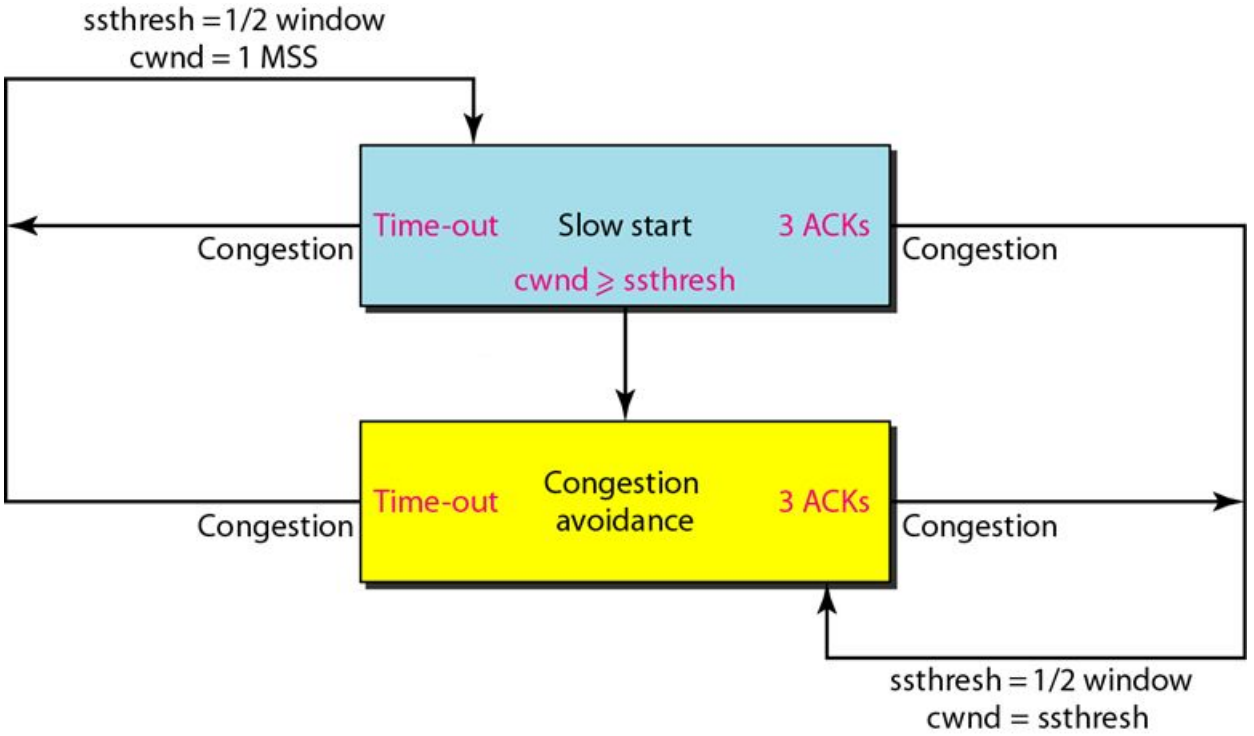
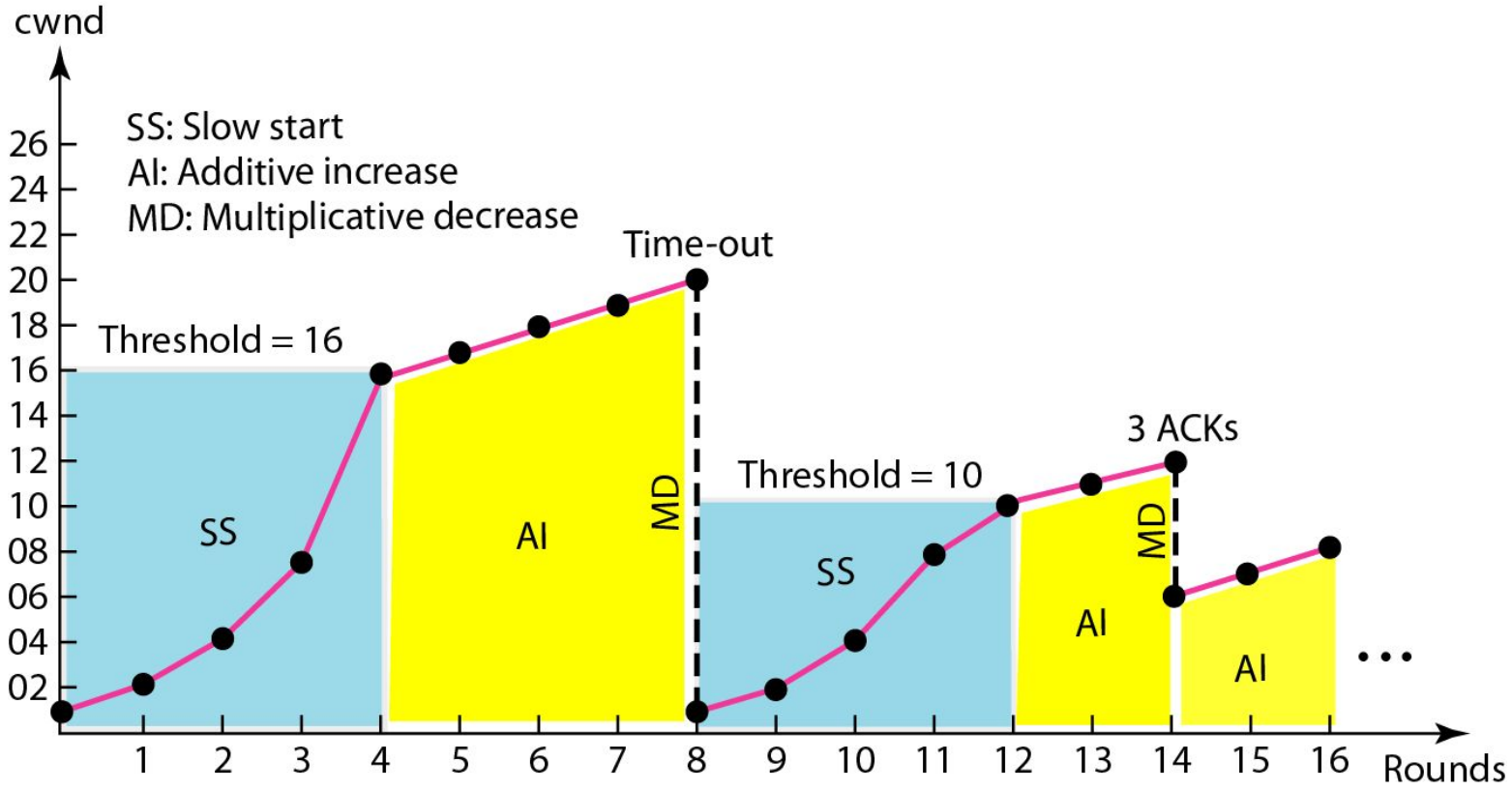
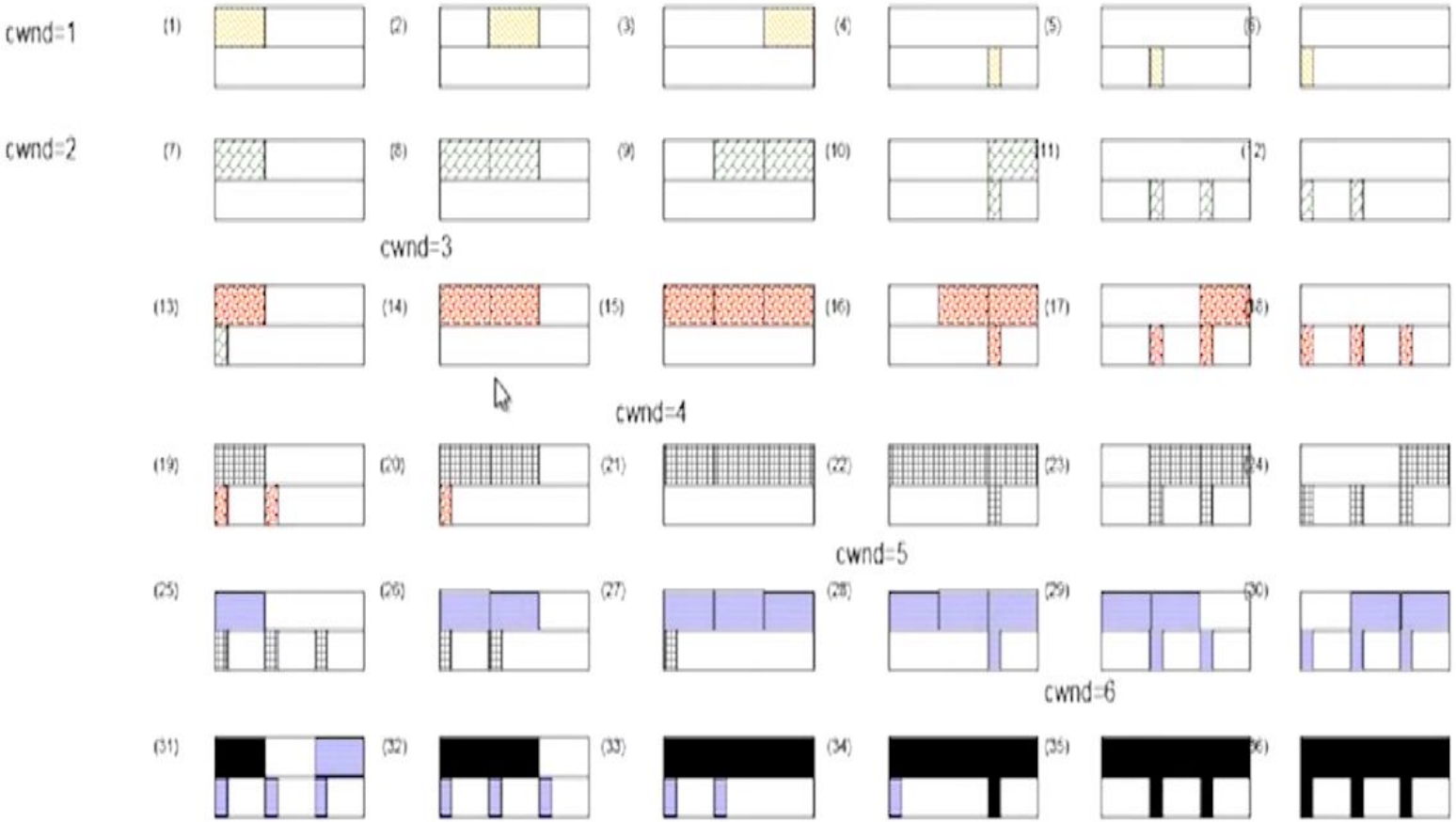


Figure 24.11 *Congestion example*

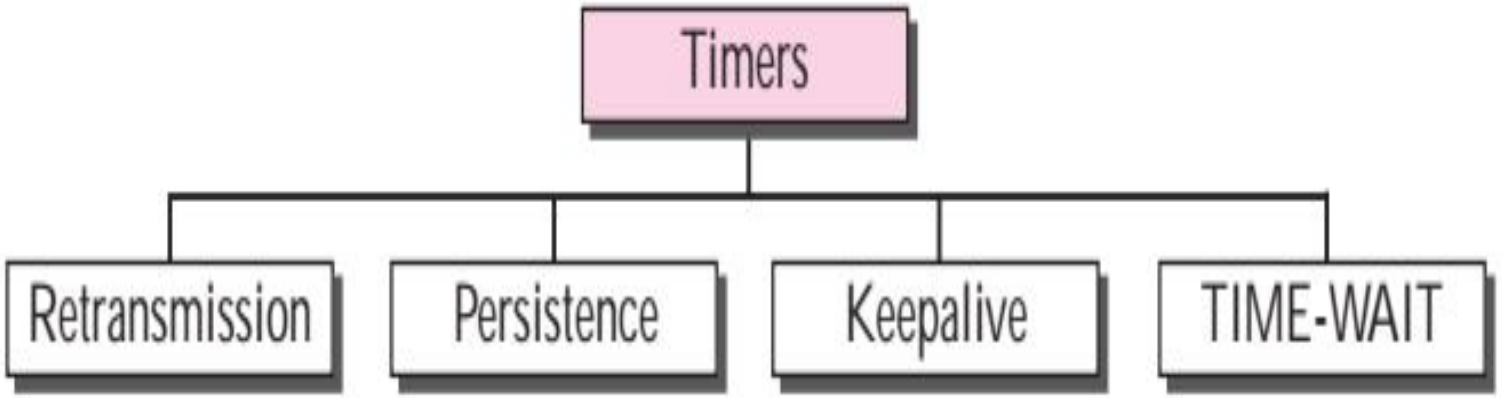


Performance of Bulk Data Transfers

Steps of filling the pipe using Congestion Avoidance



TCP TIMERS



Retransmission Timer

To retransmit lost segments, TCP employs one retransmission timer (for the whole connection period) that handles the retransmission time-out (RTO), the waiting time for an acknowledgment of a segment. We can define the following rules for the retransmission timer:

1. When TCP sends the segment in front of the sending queue, it starts the timer.
2. When the timer expires, TCP resends the first segment in front of the queue, and restarts the timer.
3. When a segment (or segments) are cumulatively acknowledged, the segment (or segments) are purged from the queue.
4. If the queue is empty, TCP stops the timer; otherwise, TCP restarts the timer

Round-Trip Time (RTT)

- To calculate the retransmission time-out (RTO), we first need to calculate the **roundtrip time (RTT).** However, calculating RTT in TCP is an involved process that we explain step by step with some examples.
- Calculate **Measured RTT**

  In TCP, there can be only one RTT measurement in progress at any time.
- **Smoothed RTT**
  The measured RTT, RTT_M, is likely to change for each round trip. The fluctuation is so high in today's Internet that a single measurement alone cannot be used for retransmission time-out purposes. Most implementations use a smoothed RTT, called RTT_S, which is a weighted average of RTT_M and the previous RTT_S

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

The value of α is implementation-dependent, but it is normally set to 1/8. In other words, the new RTT_S is calculated as 7/8 of the old RTT_S and 1/8 of the current RTT_M.

RTT Deviation Most implementations do not just use RTT_S; they also calculate the RTT deviation, called RTT_D, based on the RTT_S and RTT_M using the following formulas:

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

The value of β is also implementation-dependent, but is usually is set to 1/4.

Retransmission Time-out (RTO)

The value of RTO is based on the smoothed round-trip time and its deviation. Most implementations use the following formula to calculate the RTO:

Original	$\rightarrow$	**Initial value**
After any measurement	$\rightarrow$	$\mathbf{RTO = RTT_S + 4 \times RTT}$

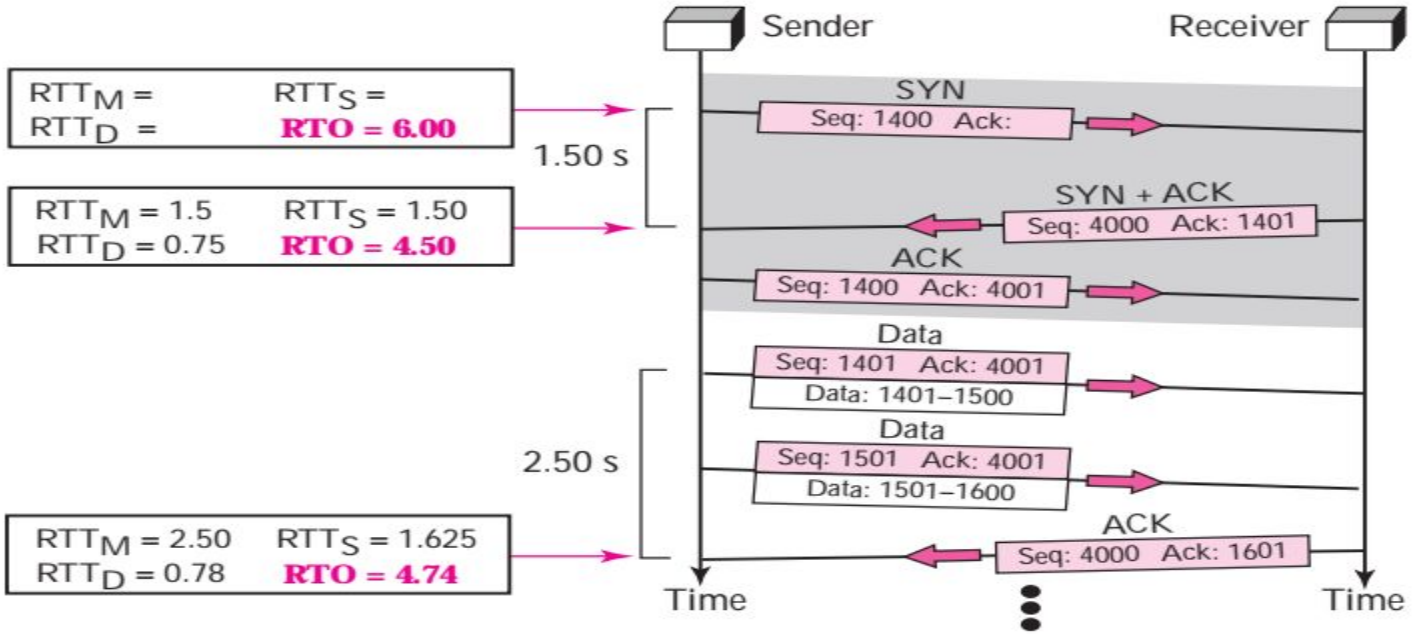
In other words, take the running smoothed average value of RTT_S, and add four times the running smoothed average value of RTT_D (normally a small value).

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 variables at this moment:

RTO = 6



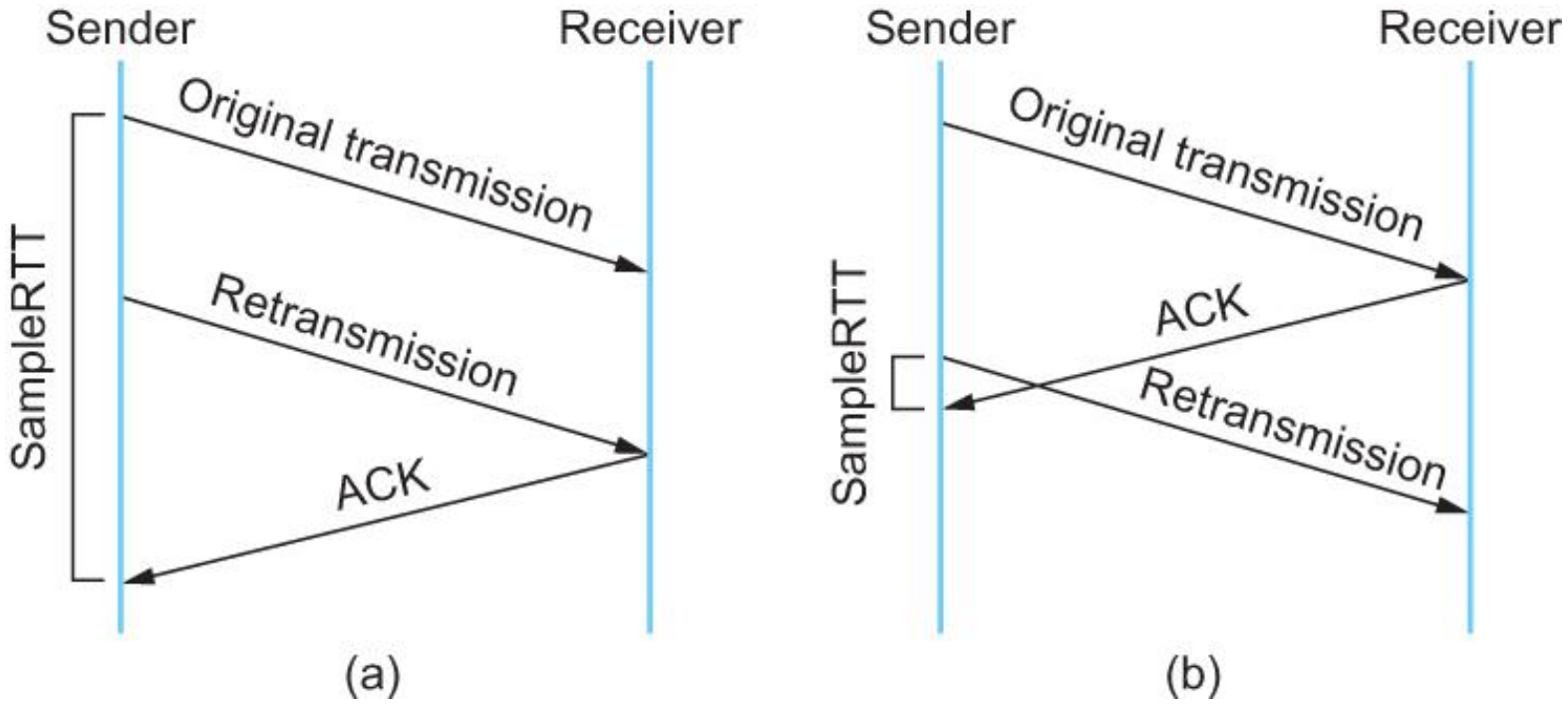
2. When the SYN+ACK segment arrives, RTT_M is measured and is equal to 1.5 seconds. The following shows the values of these variables:

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

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. 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 (accumulative), its arrival finalizes the value of RTT_M for the first segment. The values of these variables are now as shown below.

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

Karn/Partridge Algorithm



Associating the ACK with (a) original transmission versus (b) retransmission

Karn's algorithm

- Do not consider the round-trip time of a retransmitted segment in the calculation of RTTs. Do not update the value of RTTs until you send a segment and receive an acknowledgment without the need for retransmission.

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

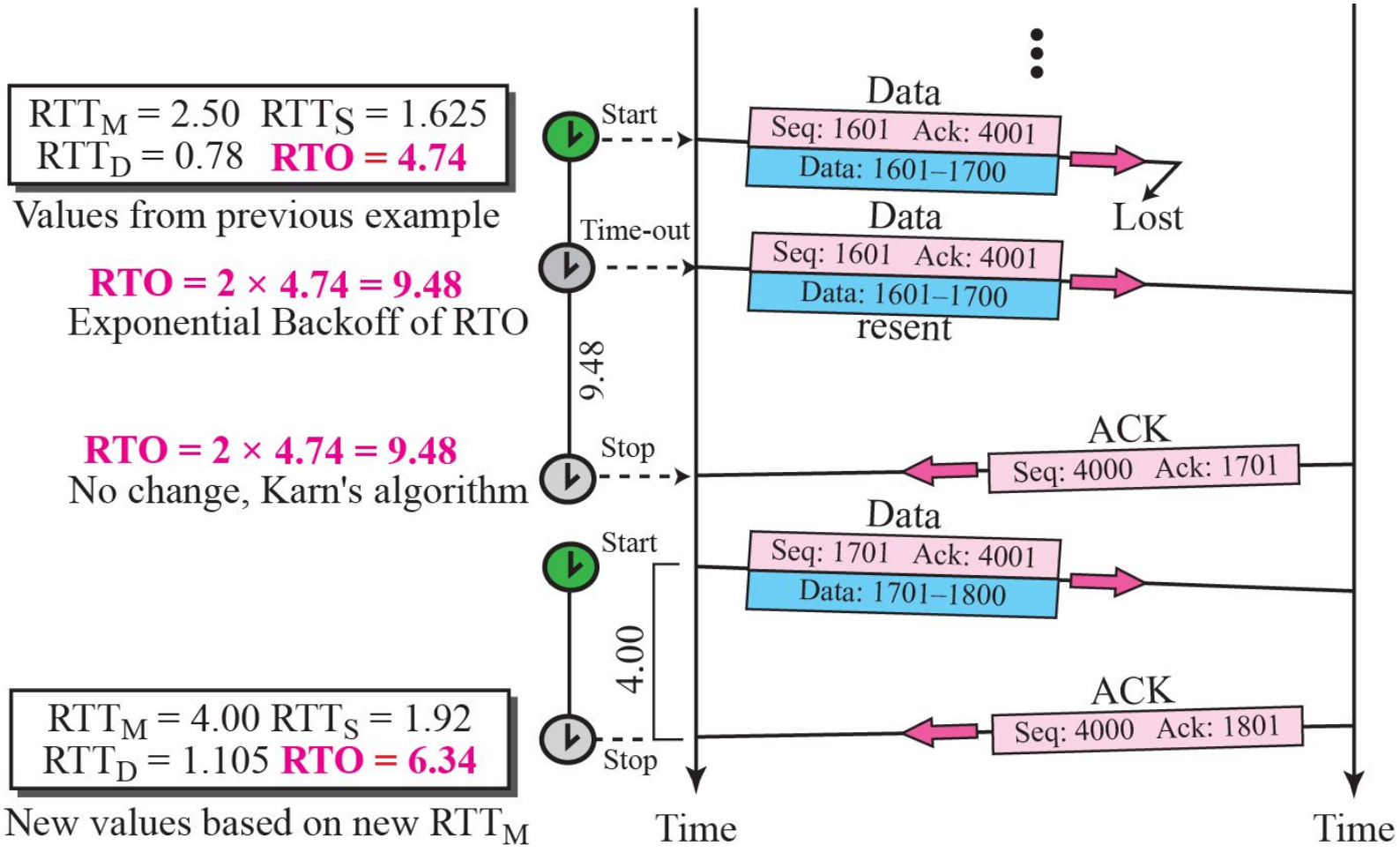
Example 4

Figure 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 40 *Example 4*



OPTIONS

- The TCP header can have up to 40 bytes of optional information.
- Options convey additional information to the destination or align other options.
- We can define two categories of options: 1-byte options and multiple-byte options.
- The first category contains two types of options:
  end of option list and no operation.
- The second category, in most implementations, contains five types of options:
  maximum segment size, window scale factor, timestamp, SACK-permitted, and SACK.

Figure 41 *Options*

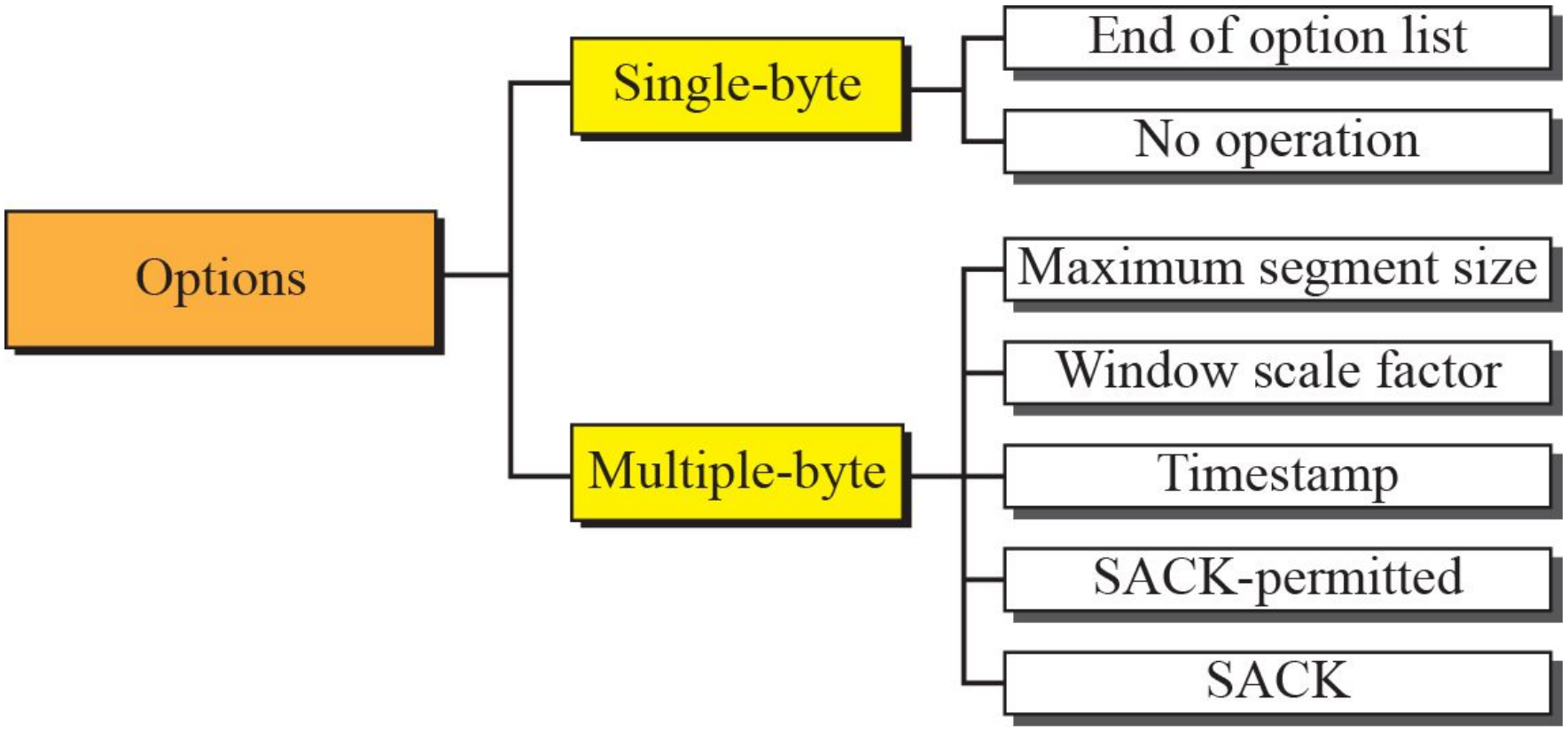
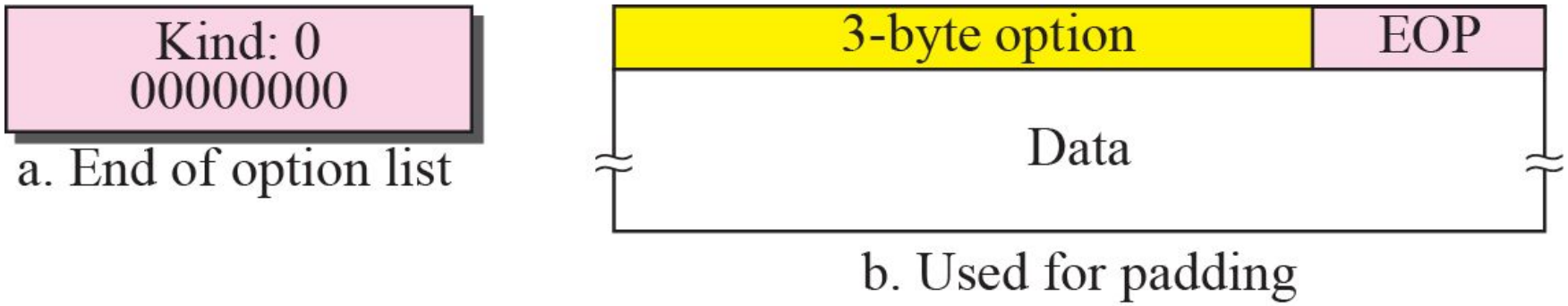


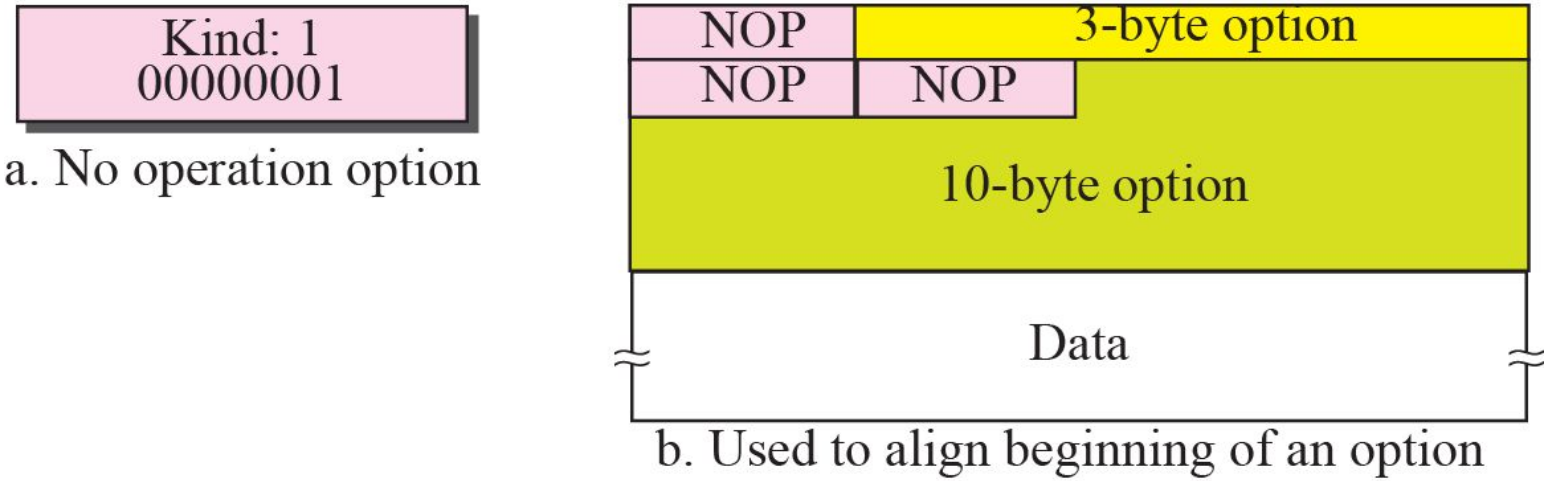
Figure 42 *End-of-option option*



Note

EOP can be used only once.

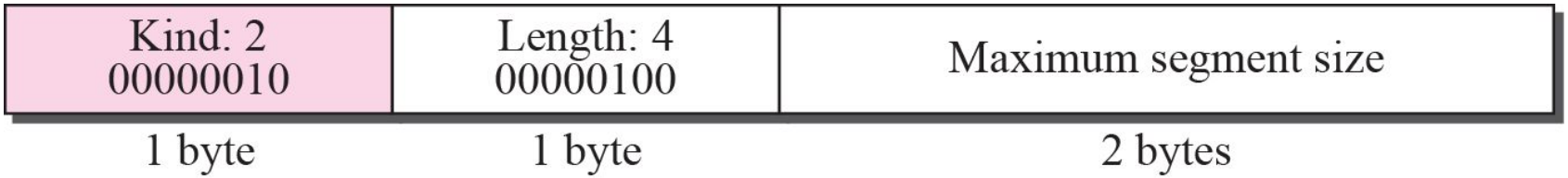
Figure 43 *No-operation option*



Note

NOP can be used more than once.

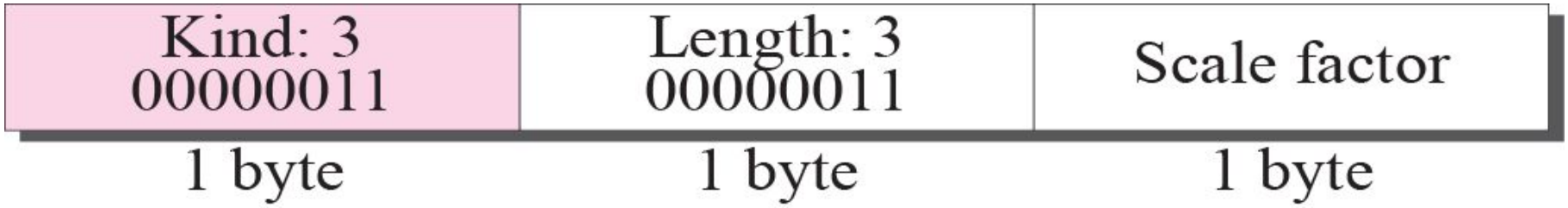
Figure 44 *Maximum-segment-size option*



Note

The value of MSS is determined during connection establishment and does not change during the connection.

Figure 45 *Window-scale-factor option*



$$\text{New window size} = \text{window size defined in the header} \times 2^{\text{window scale factor}}$$

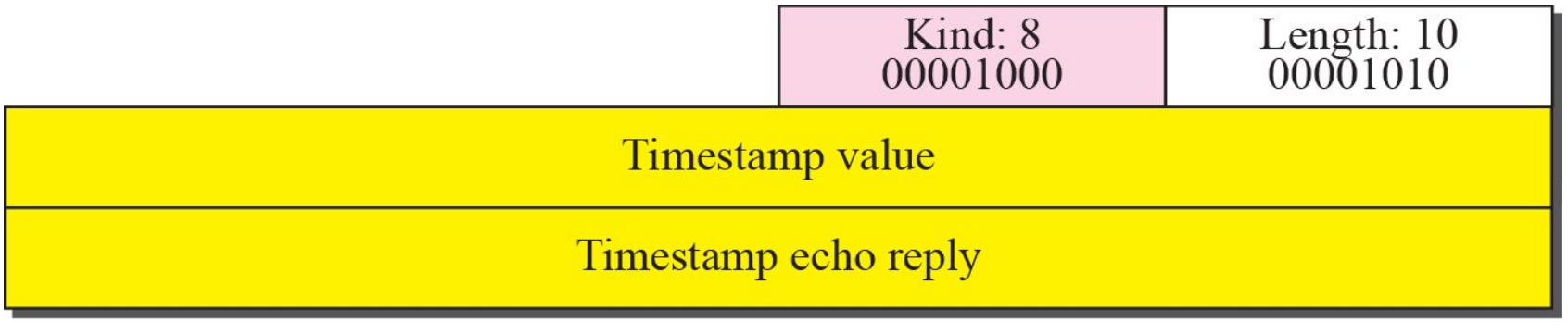
- Although the scale factor could be as large as 255, the largest value allowed by TCP/IP is 14, which means that the maximum window size is 2^16 × 2^14 = 2^30

- which is less than the maximum value for the sequence number.

Note

The value of the window scale factor can be determined only during connection establishment; it does not change during the connection.

Figure 46 *Timestamp option*



- **One application of the timestamp option is the calculation of round-trip time (RTT).**

- **The timestamp option can also be used for protection against wrapped sequence numbers (PAWS).**

Example 5

Below Fig shows an example that calculates the round-trip time for one end. Everything must be flipped if we want to calculate the RTT for the other end.

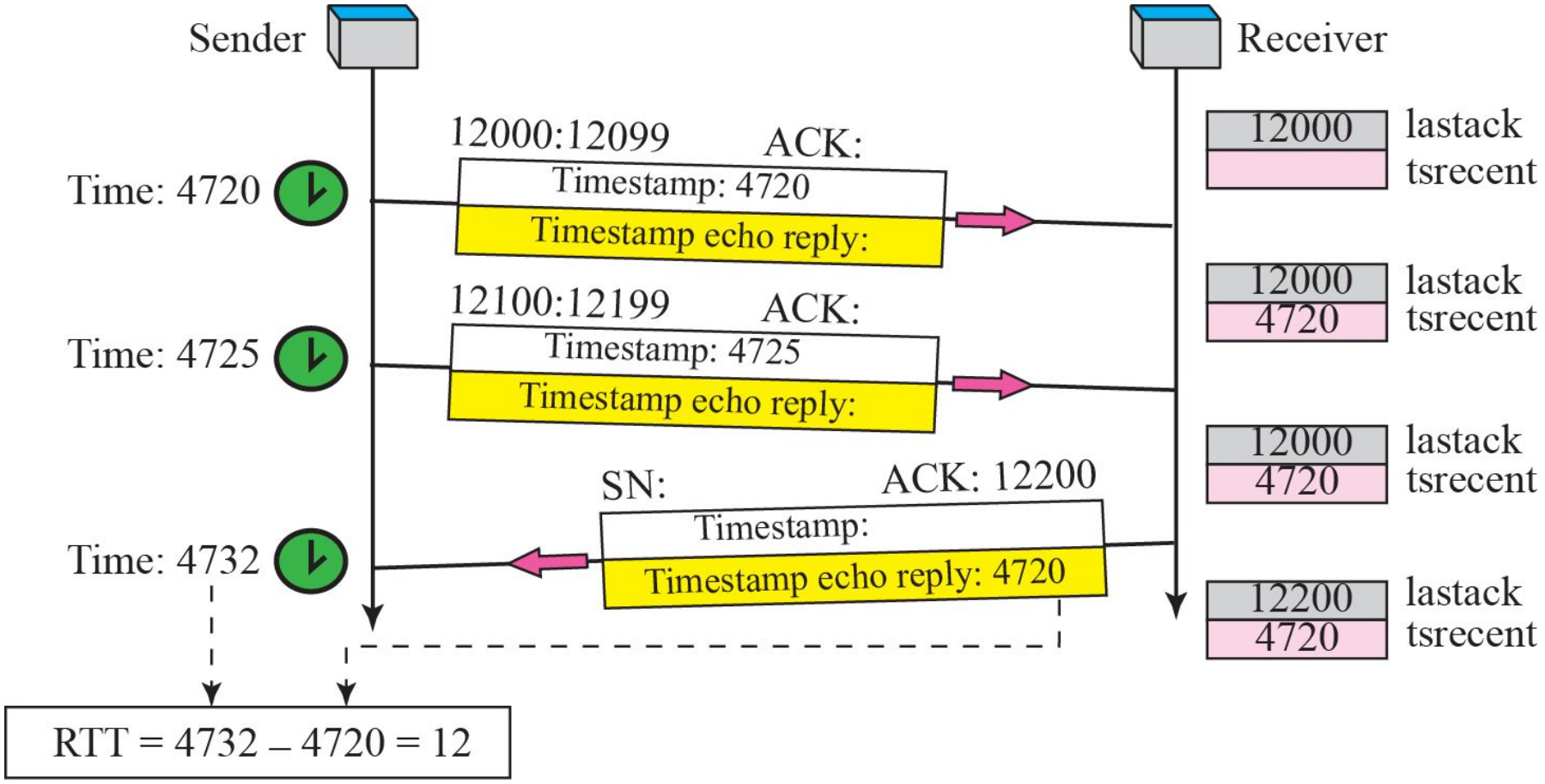
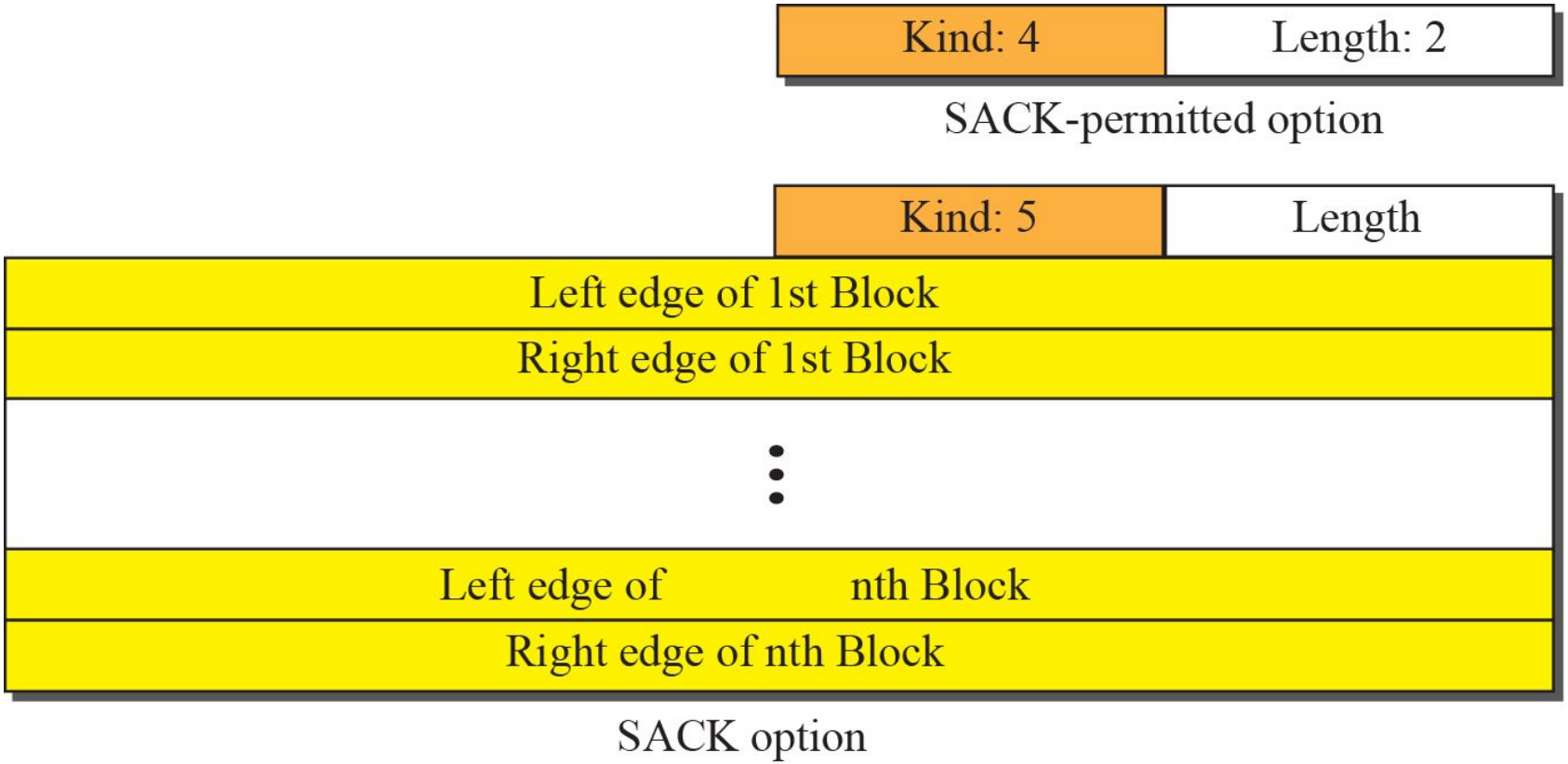
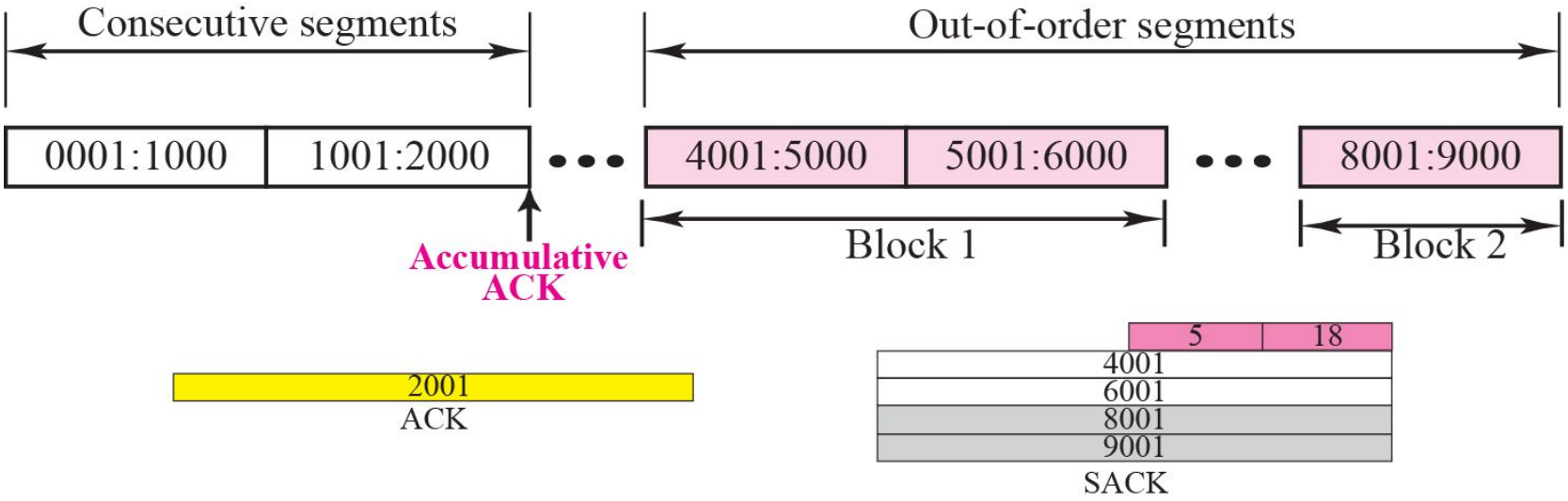


Figure 48 *SACK*



Example 6

Let us see how the SACK option is used to list out-of-order blocks. In the below Figure an end has received five segments of data.

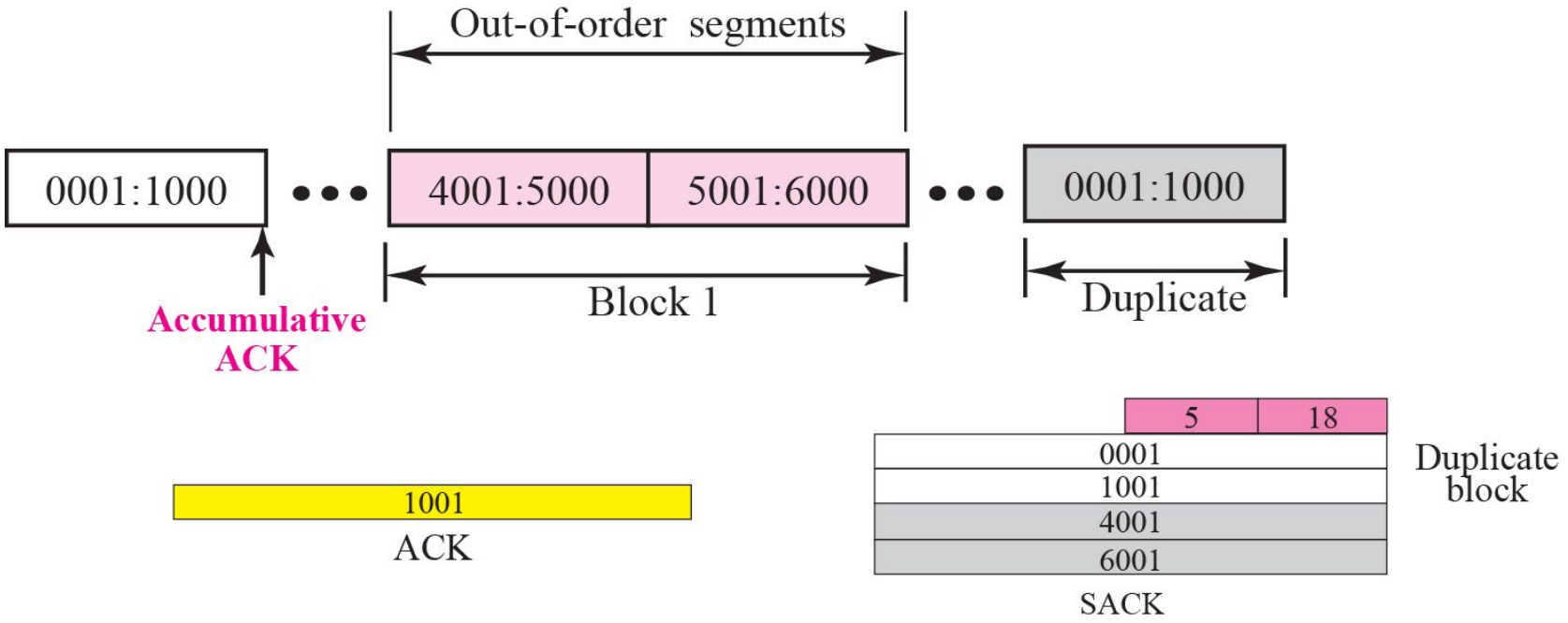


Example 7

Figure 50 shows how a duplicate segment can be detected with a combination of ACK and SACK. In this case, we have some out-of-order segments (in one block) and one duplicate segment. To show both out-of-order and duplicate data, SACK uses the first block, in this case, to show the duplicate data and other blocks to show out-of-order data.

Note that only the first block can be used for duplicate data. The natural question is how the sender, when it receives these ACK and SACK values, knows that the first block is for duplicate data (compare this example with the previous example). The answer is that the bytes in the first block are already acknowledged in the ACK field; therefore, this block must be a duplicate.

Figure 50 *Example 7*



Example 8

Figure 51 shows what happens if one of the segments in the out-of-order section is also duplicated. In this example, one of the segments (4001:5000) is duplicated.

The SACK option announces this duplicate data first and then the out-of-order block. This time, however, the duplicated block is not yet acknowledged by ACK, but because it is part of the out-of-order block (4001:5000 is part of 4001:6000), it is understood by the sender that it defines the duplicate data.

Figure 51 *Example 8*