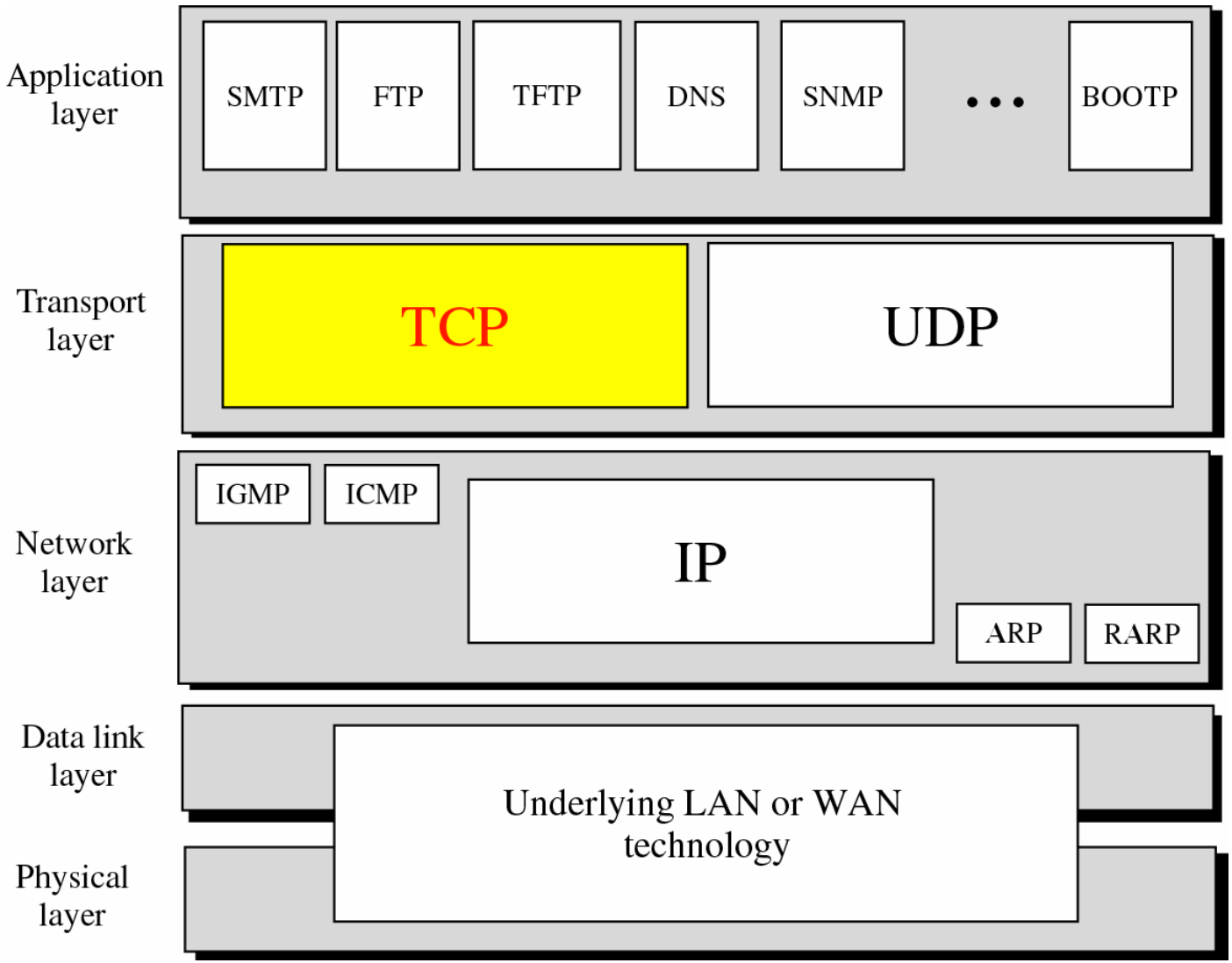
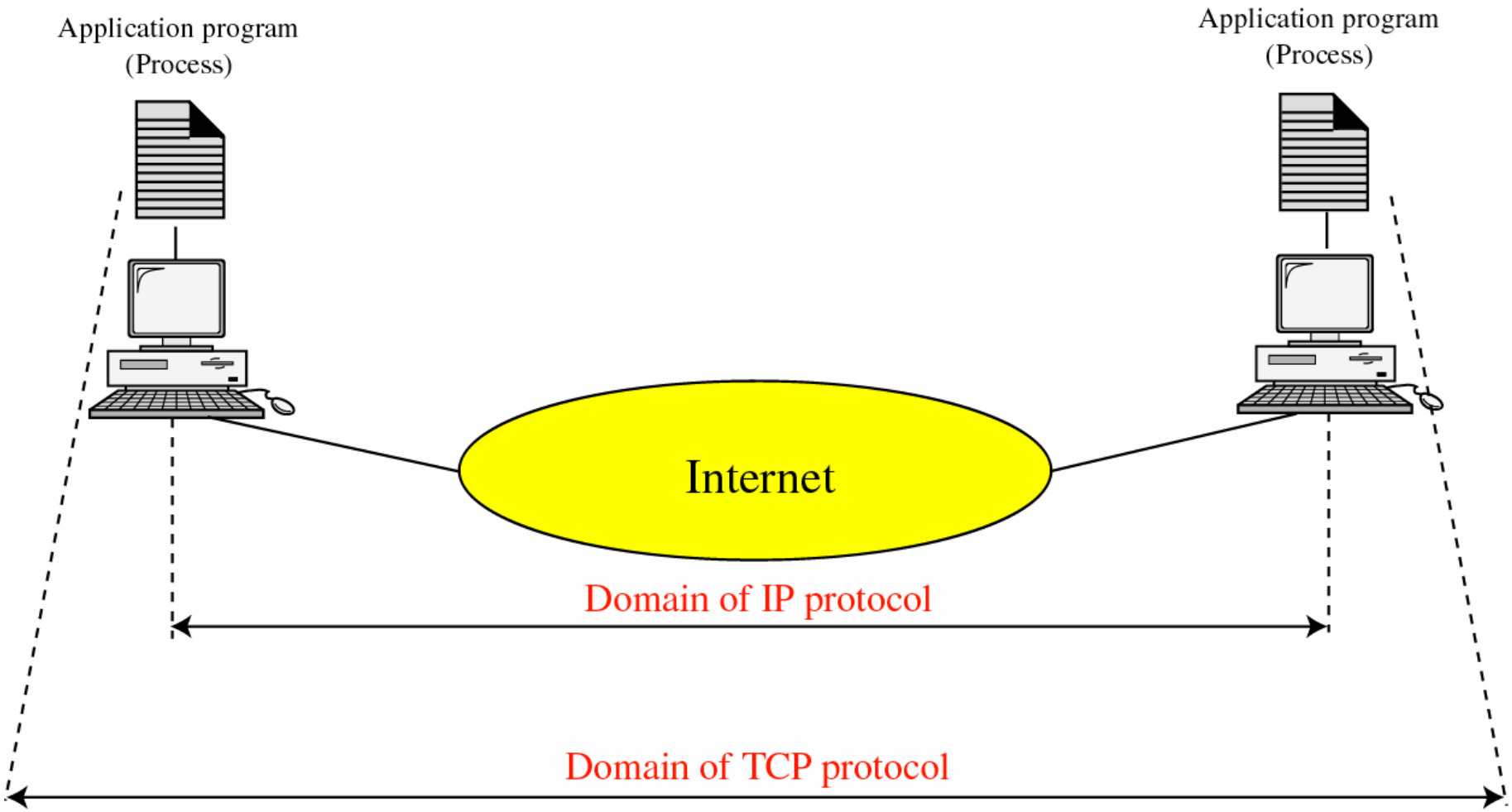
Chapter 12

Transmission Control Protocol (TCP)

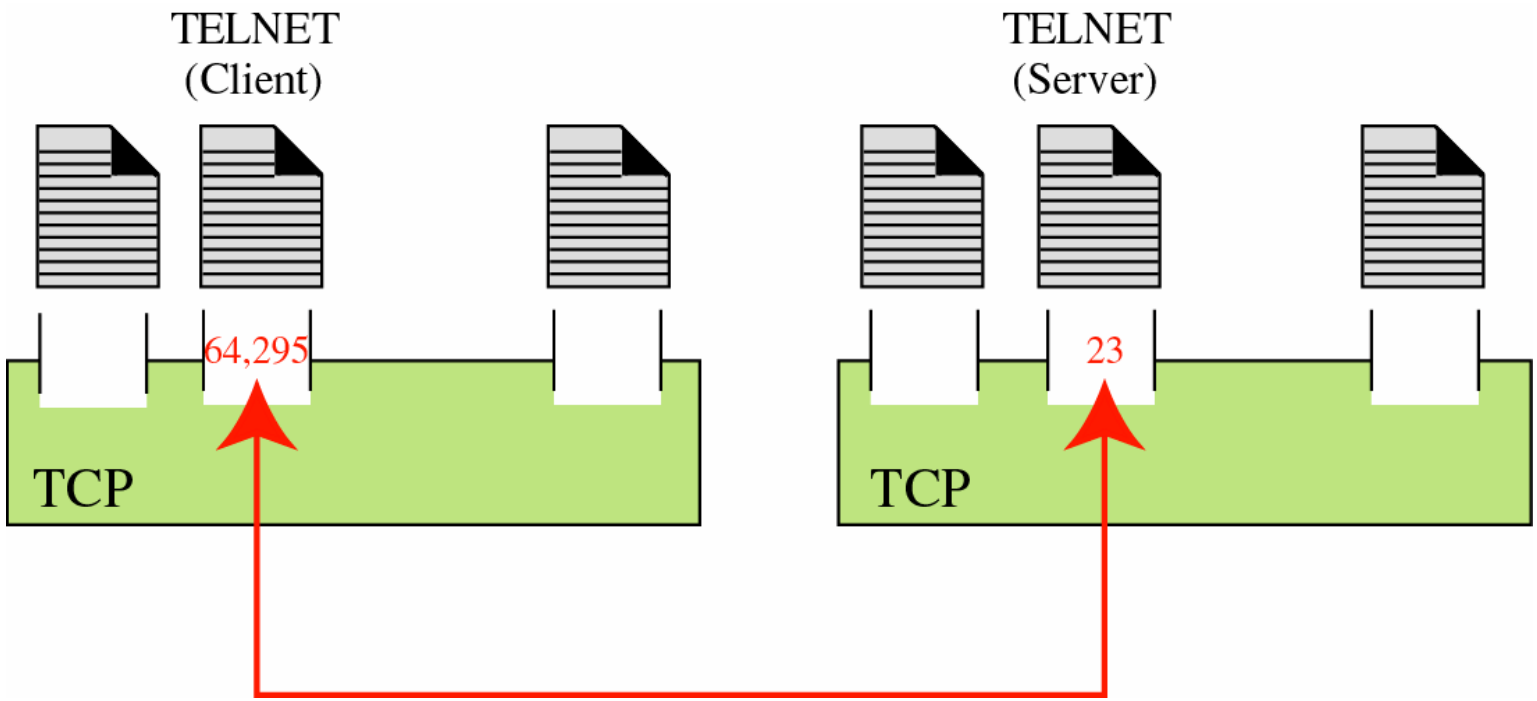
Position of TCP in TCP/IP protocol suite



TCP versus IP



Port numbers



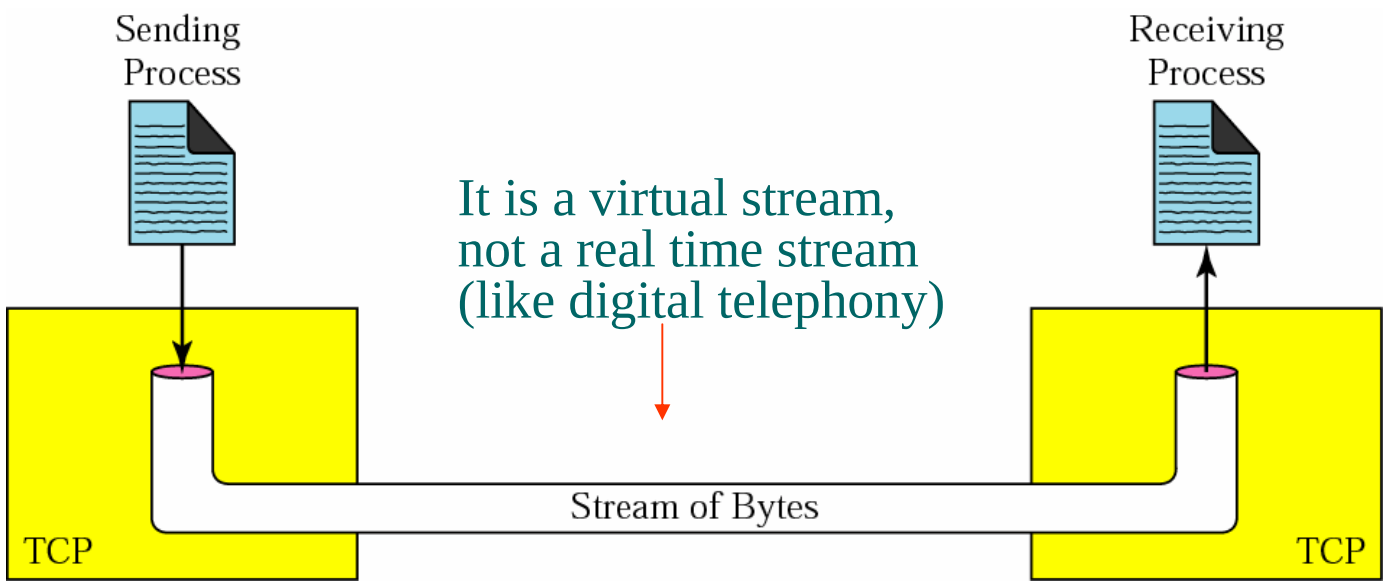
The concept of port numbers is the same as in UDP

Stream delivery

TCP, unlike UDP, is stream-oriented protocol.
This means that if a sender process sends a stream of bytes,
the receiver process will be getting exactly the same stream of bytes.
It is as if the two processes were connected with a virtual tube that
carries data across the Internet.

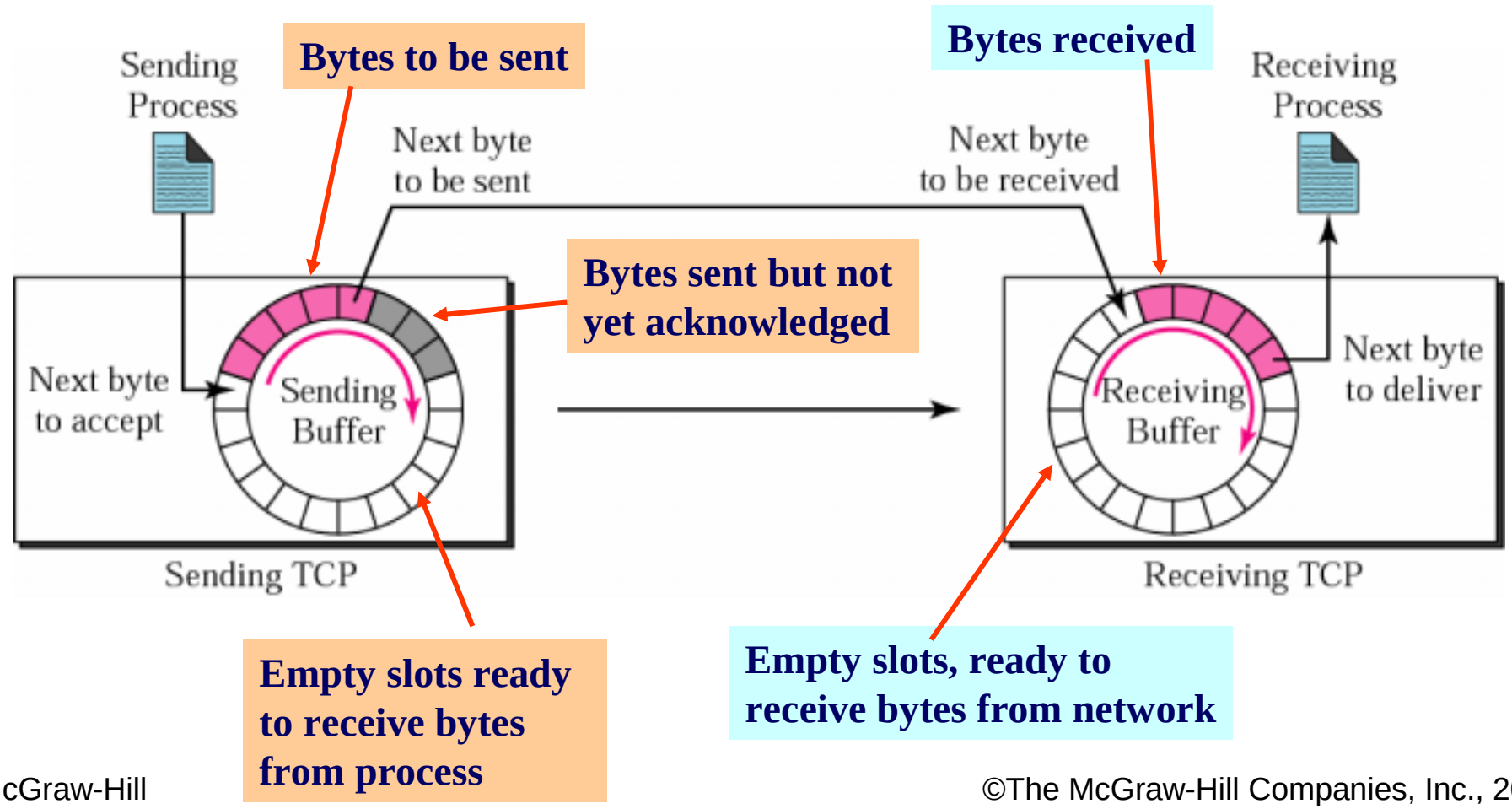
There can be two tubes carrying streams in two directions, since TCP
supports full-duplex service.

NOTICE: Digital telephony (T-1, T-3, …, ATM) supports constant bit rate (CBS)
service which allows a real-time stream as opposed to virtual tube.

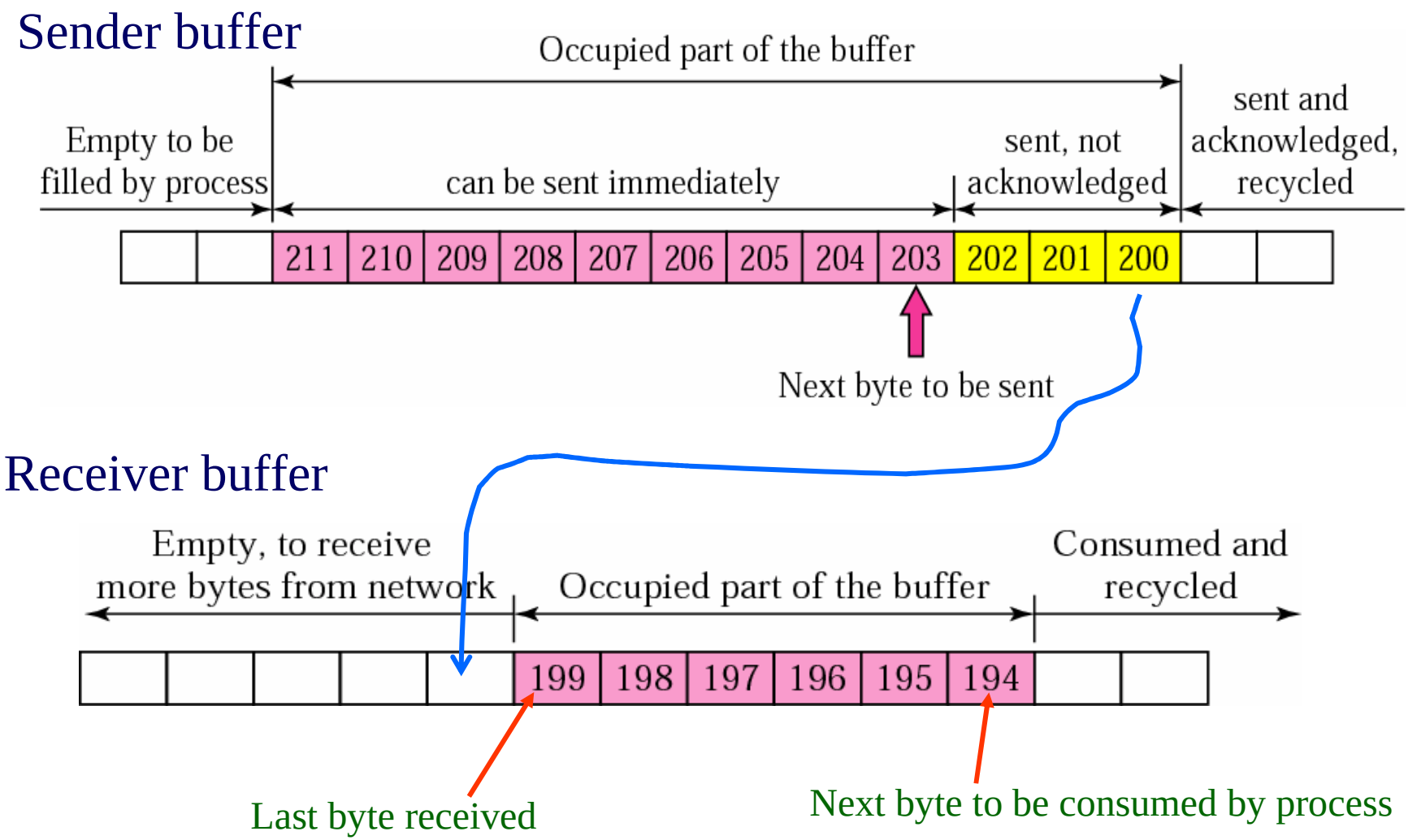


Sending and receiving buffers

Since the sending and the receiving process work at different (and unpredictable) speeds, and since the route between the sender and the receiver may cause unpredictable time varying delays, there must be two buffers to implement the "tube": sender and receiver buffer. The buffer sizes are hundreds and thousands of bytes.

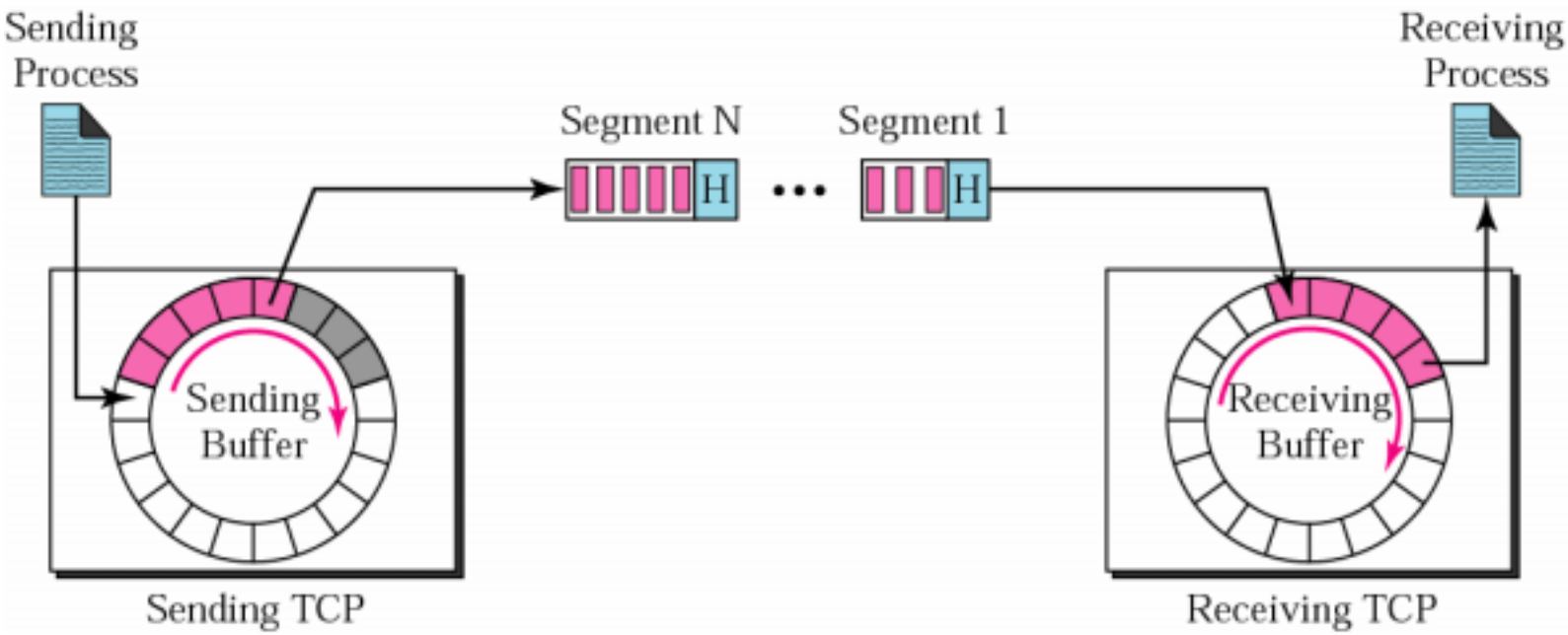


Sending and receiving buffer in flat representation



TCP segments

Since IP is a packet switching protocol the TCP stream of bytes has to be divided into groups called segments before sent to IP layer for encapsulation into IP datagrams and transmission. Segments are transparent to the sending and receiving processes, which see stream only.



In order to keep track of segments and their ordering, they have two fields: sequence number and acknowledgement number. However, these numbers do not represent the number of segments (like frame numbers in link layer error flow/error control), they rather refer to the number of bytes

The bytes of data being transferred in each connection are numbered by TCP. The numbering starts with a randomly generated number.

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

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

The sequence and acknowledge numbers are 32-bit numbers

Example 1

Imagine a TCP connection is transferring a file of 6000 bytes. The first byte is numbered 10010. What are the sequence numbers for each segment if data is sent in five segments with the first four segments carrying 1,000 bytes and the last segment carrying 2,000 bytes?

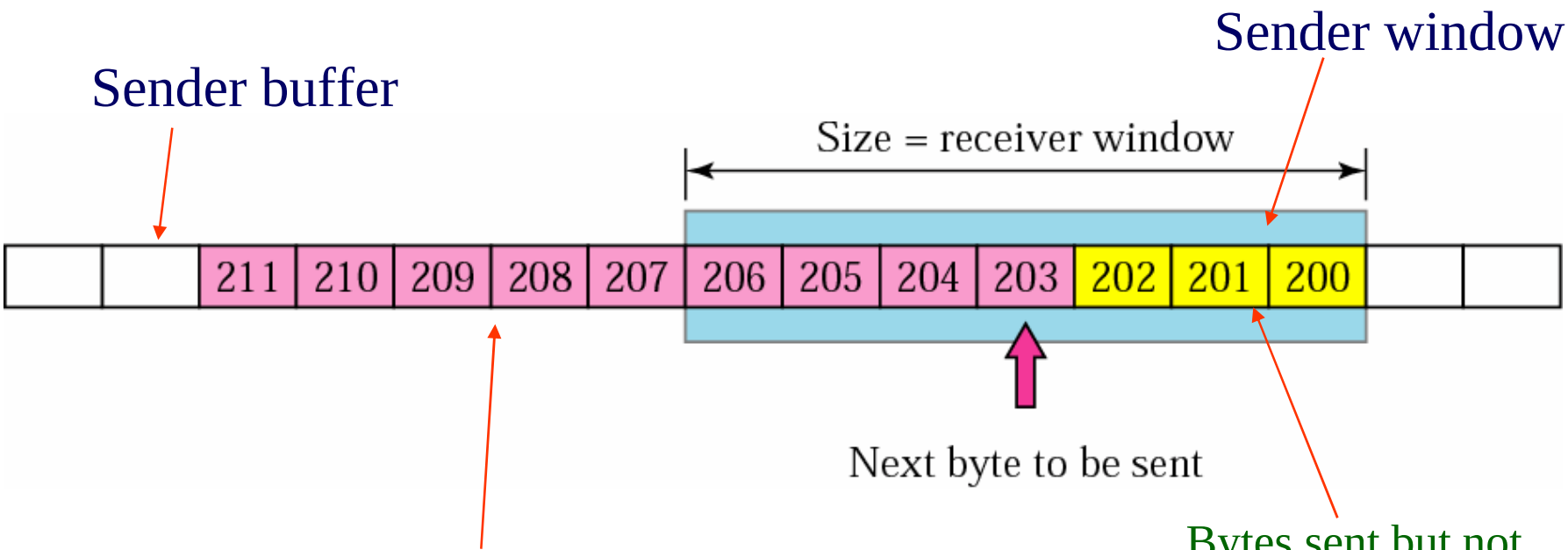
Solution

The following shows the sequence number for each segment:

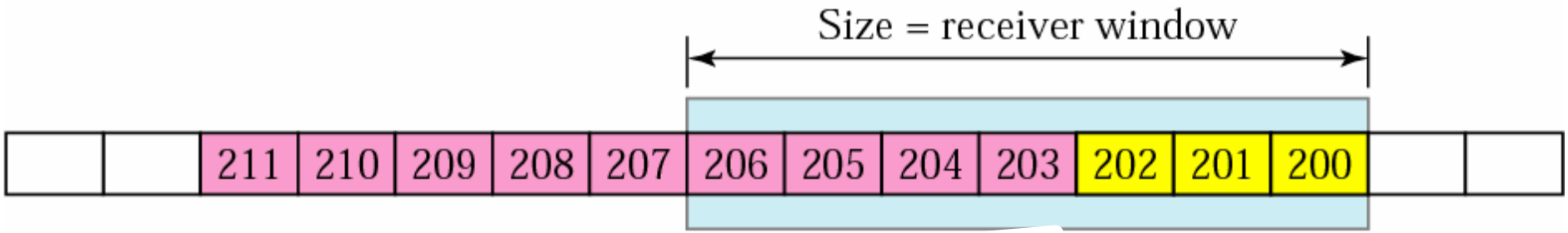
Segment 1	➔	10,010	(10,010 to 11,009)
Segment 2	➔	11,010	(11,010 to 12,009)
Segment 3	➔	12,010	(12,010 to 13,009)
Segment 4	➔	13,010	(13,010 to 14,009)
Segment 5	➔	14,010	(14,010 to 16,009)

Flow Control

Flow control is used to keep the byte stream flowing from the sender to the receiver without overflowing the buffers on both sides. In order to achieve this TCP is using the sliding window protocol.

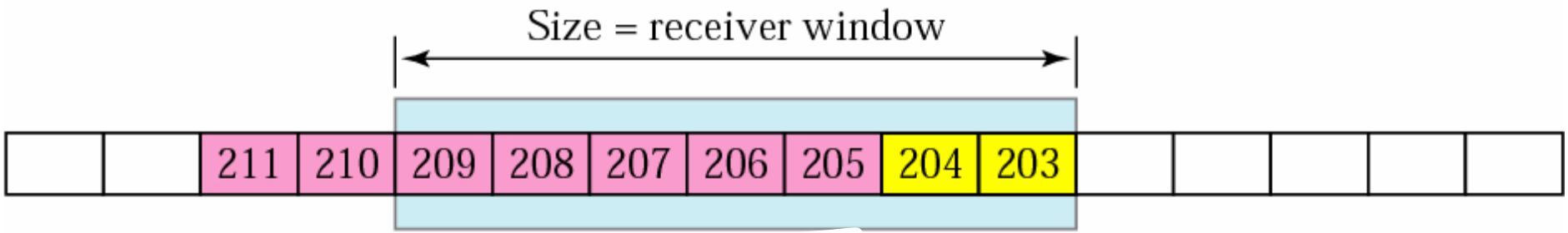


Sliding the sender window



Suppose: (1) Bytes 200, 201, 202 are acknowledged (ACK 203),
(2) Bytes 203 and 204 are sent.
(3) Receiver window size didn't change (=7)

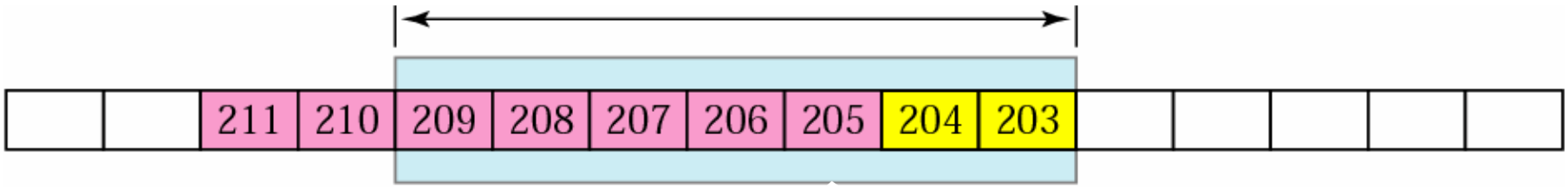
a. Before



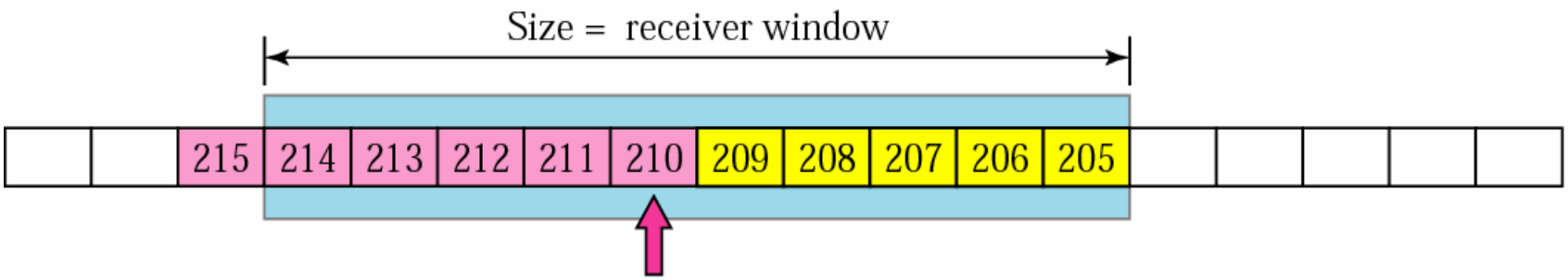
b. After

The scenario above has resulted in sliding the window

Expanding the sender window

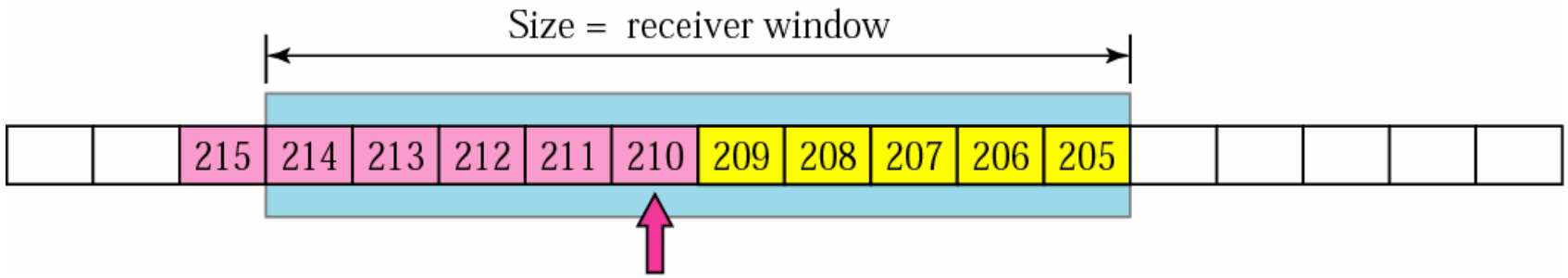


Suppose: (1) Bytes 203 and 204 are acknowledged (ACK 205),
(2) Bytes 205 - 209 are sent.
(3) Receiver window size has increased to 10
(4) Sending process has created 4 more bytes

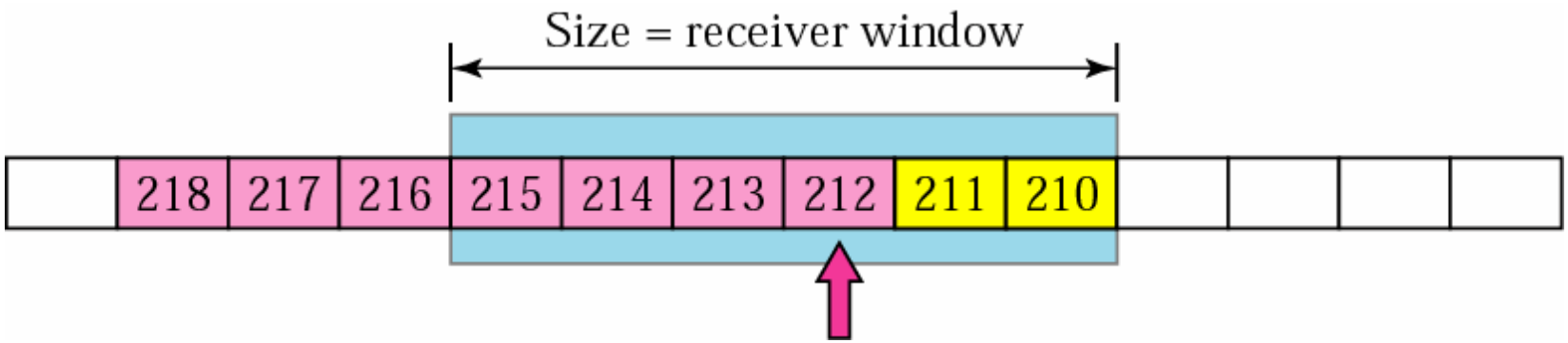


The scenario above has resulted in expanding the sender window

Shrinking the sender window



Suppose: (1) Receiver has received and acknowledged bytes 205-209 (ACK 210)
(2) Receiving process has consumed only one byte
(3) Receiver informs the sender to shrink the window to 6
(4) Sender has sent two more bytes



Some Points about TCP's Sliding Windows:

1. *The source does not have to send a full window's worth of data.*

2. *The size of the window can be increased or decreased by the destination.*

3. *The destination can send an acknowledgment at any time.*

Silly Window Syndrome

If either sender or receiver slow down significantly, results in very small segments. An extreme are one byte segments, causing IP packets of size 41 carrying only one byte of data (20 - IP header, 20-TCP header, 1-payload). This is very inefficient use of network, called silly syndrome.

Solution to that is not to send segments if the sender window has small opening, but to wait until the receiver opens the window and advertises that to the sender. How much to wait? To long wait causes interactive application (like TELNET) to suffer. Too short wait causes silly window syndrome.

Nagle's self-clocking algorithm is a simple and effective solution to that.

Silly Window Syndrome (cont.)

Nagle's self-clocking algorithm[1]

```
if (available data and window > MSS)
        Send a full segment;
else
        if (there is unacknowledged data in flight)
                Buffer the new data until ACK arrives;
        else
                Send all the new data now;
```

MSS = Maximum segment size

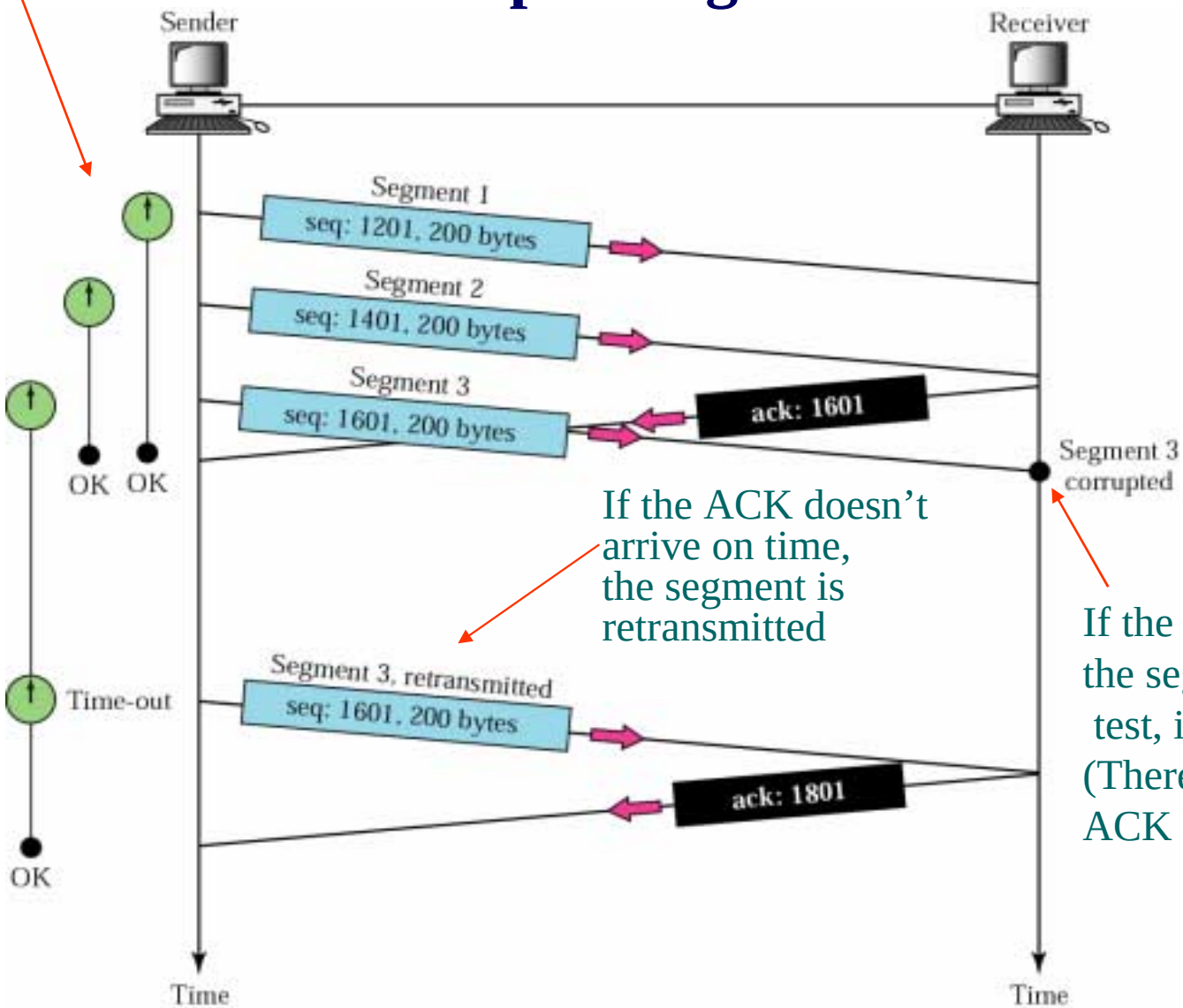
If some application cannot afford delay, the socket interface allows turning off the Nagel's algorithm (`TCP_NODELAY` option)
→ data transmitted as soon as possible.

(1) John Nagle, RFC896)

Error Control

Corrupted segment

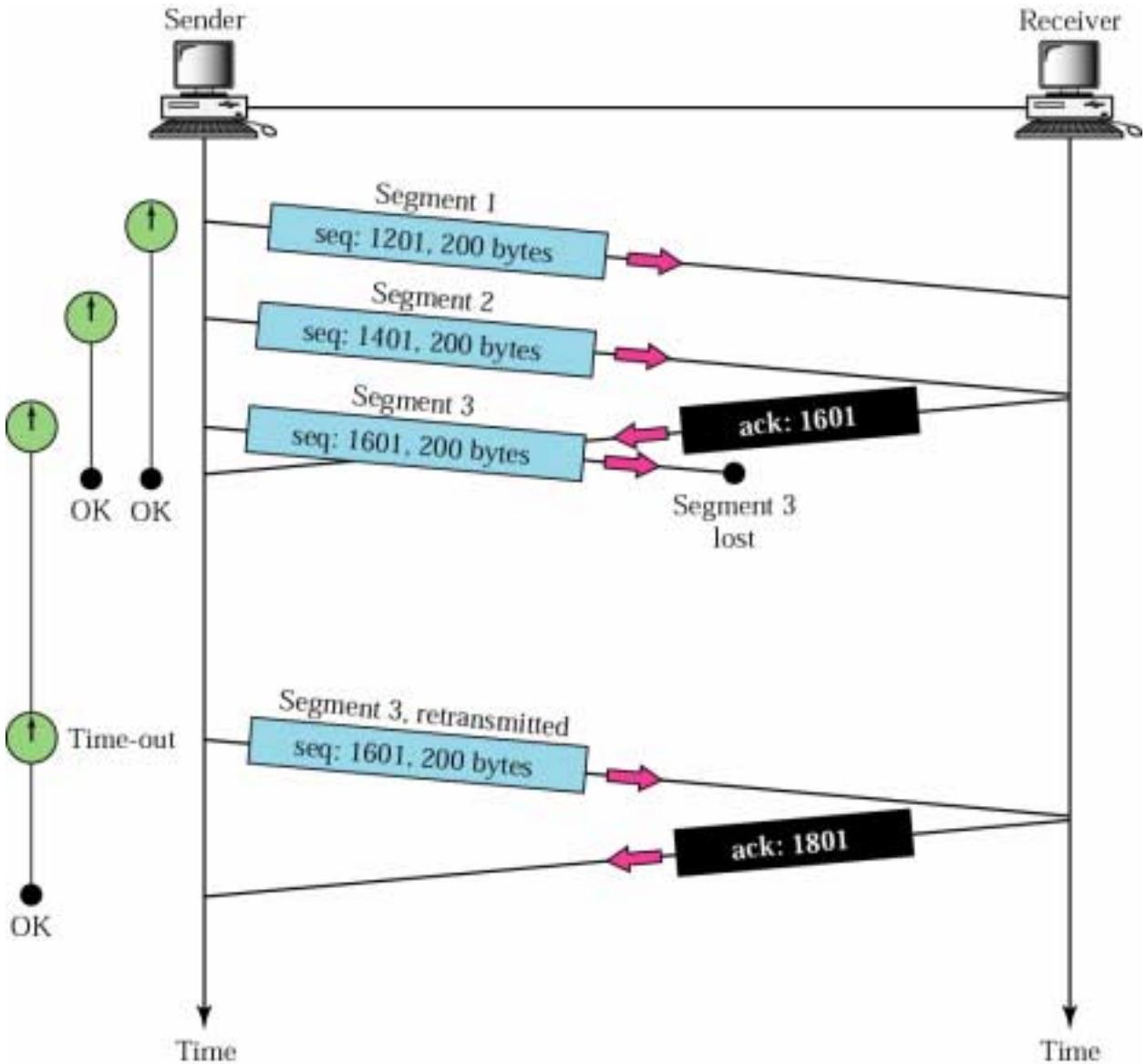
A retransmission timer is created for each segment sent



If the segment fails the segment checksum test, it is discarded. (There is no negative ACK in TCP)

Error Control (Cont.)

Lost segment (same as corrupted segment)



Error Control (Cont.)

Duplicate segment

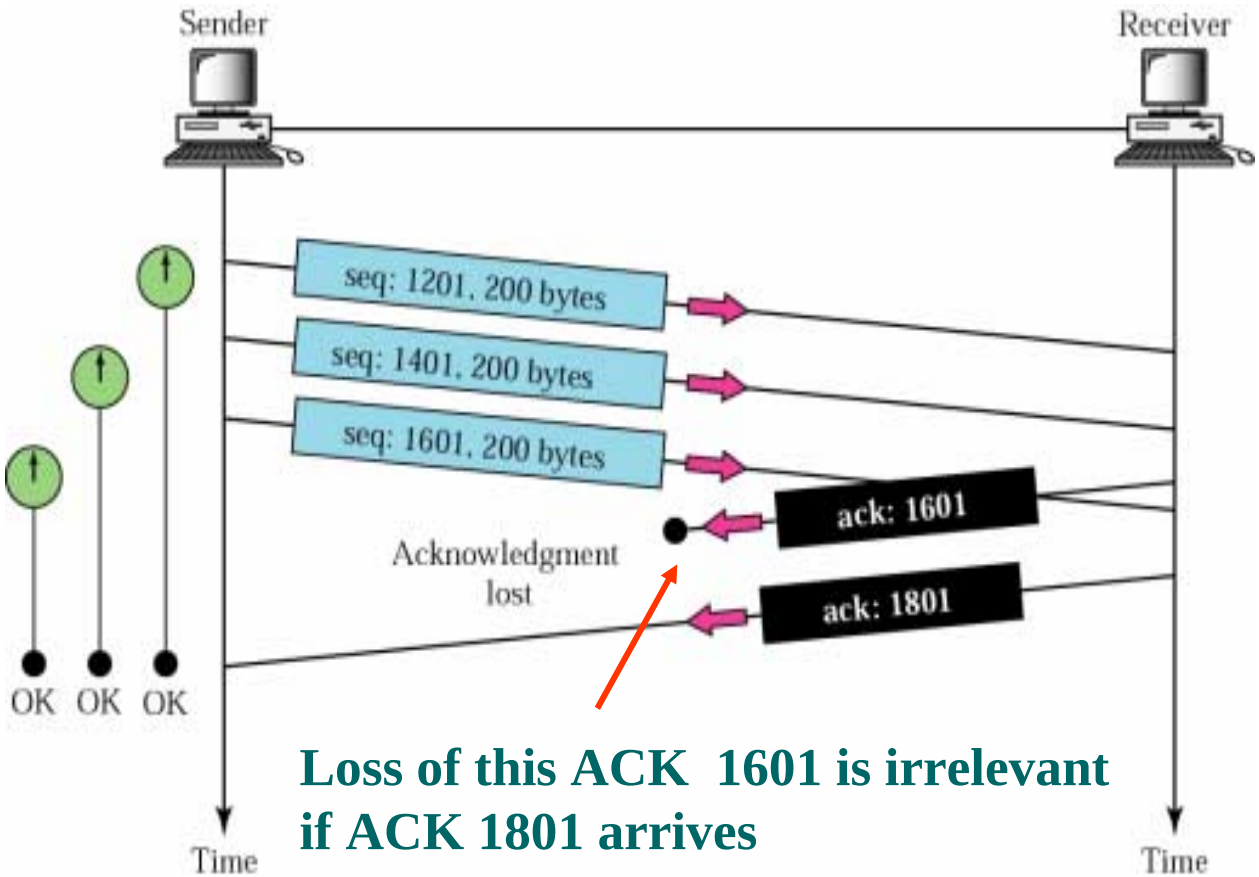
If a segment with the same sequence number as another received segment arrives it is discarded

Out-of-Order Segment

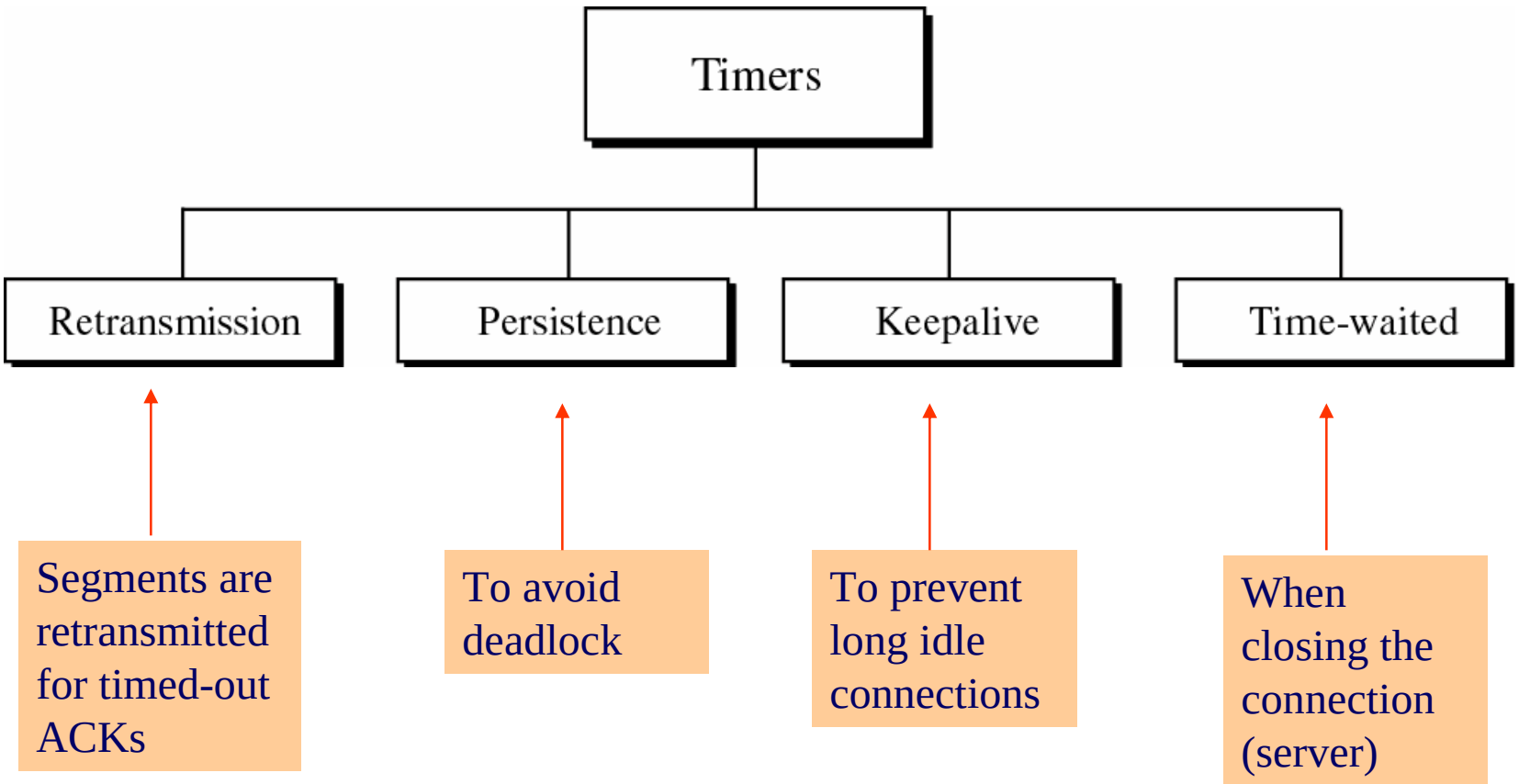
Since segments are encapsulated into IP datagrams, which are separately routed, it can happen that segments arrive to destination in a wrong order. The out of order segments are dropped and are not acknowledged.

Error Control (Cont.)

Lost acknowledgment



TCP timers



Retransmission Timer

A retransmission timer is created whenever a segment is sent. The timer is destroyed if ACK for the segment is received in time, otherwise the segment is retransmitted and the timer is restarted.

TCP cannot use the same retransmission time for all connections. Even for the same pair of processes, the retransmission time may change dynamically, depending on the current traffic situation. Calculation of retransmission time (adaptive retransmission):

$$RT = 2 * ERTT$$

Where *ERTT* is estimated round trip time predicted as:

$$ERTT := \alpha * ERTT + (1-\alpha) * RTT$$

RTT (Round trip time) is measured each time a segment is sent and the acknowledgement for that segment is received. (Measurement of RTT is achieved through time stamp option – see later). The smoothing factor α is normally between `0.8` and `0.9`.

Karn-Partridge algorithm

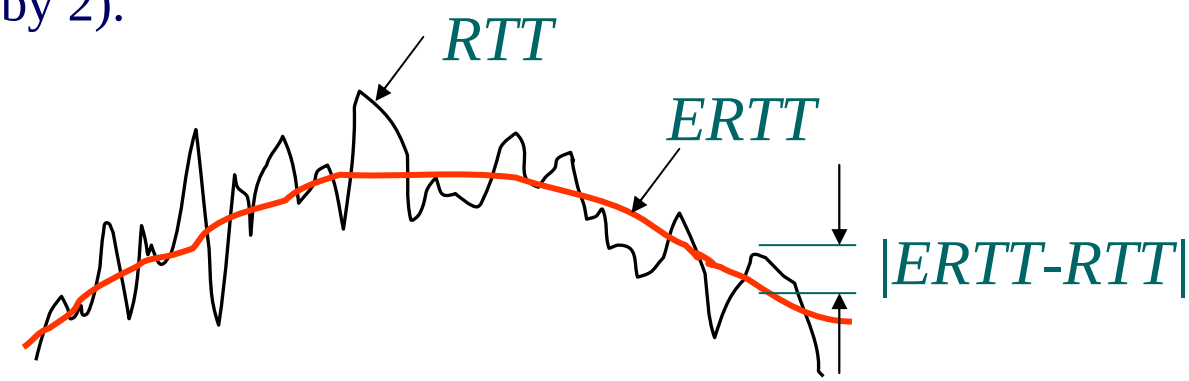
In order to measure RTT most accurately, the sampling of RTT is done only for single transmissions, i.e. the sampling is stopped if the segment is being retransmitted. In addition, at each retransmission the RT is doubled - exponential backoff policy similar to Ethernet.

Retransmission Timer (cont.)

Multiplying ERTT by two is too conservative. A more accurate approach is proposed by Jacobson/Karel's algorithm.

Jacobson/Karel's Algorithm

The previous algorithm doesn't take into account the variances of RTT samples. If variances are smaller we can trust more to ERTT (no reason to multiply ERTT by 2).



The following algorithm estimates ERTT and variances of ERTT:

$$RT = p*ERTT + q * DRTT \quad (\text{typically } p = 1, q = 4)$$

Where:

$$ERTT := \alpha * ERTT + (1-\alpha) * RTT \qquad \textit{(Average RTT)}$$

$$DRTT := \alpha * DRTT + (1-\alpha) * |ERTT-RTT| \qquad \textit{(Average deviation)}$$

Persistence Timer

TCP on both sides of connection can be deadlocked:

Receiving TCP announces zero window size

Sending TCP stops transmission until receiving the ACK and the announcement of nonzero window size.

The receiver finally sends the ACK. If that ACK gets lost (ACKs are not ACKed and retransmitted!), the receiver TCP would think that the segment is done and waits for the next segments.

Both TCPs are now waiting for each other.

Therefore, when sending TCP receives an ACK with zero window size, it starts persistence timer. After the timer goes off, sending TCP sends a probe (a special one byte segment) which alerts the receiving TCP to resend the ACK. Time out of persistence timer is set to RT. If the ACK doesn't come in time, the sending TCP sends another probe, and the RT is doubled. This is repeated until RT reaches the threshold value (usually 60 sec). Until that the probe is sent every 60 sec until the window reopens.

KeepaliveTimer

Used to prevent that an idle TCP connection stays open forever. Therefore the keepalive timer is put at the server side. Whenever server TCP hears from a client it resets the timer. The timeout is typically 2 hours. After that, server sends a probe (each 75 sec). If there is no answer after 10 probes, it assumes that the client is down and terminates the connection.

Time-Waited Timer

Used during connection termination. When TCP closes the connection, it must first wait enough time to receive all segments and acknowledgements that are possibly in flight. Once received these are discarded and TCP can finally close. The value of this timer is typically twice the expected lifetime of a segment.

Typical value of the maximum segment lifetime (MSL) is 16 seconds

Congestion Control

The flow control between source and destination uses advertised window size (AW) as the basic tool. This is however not enough – there is congestion caused by the network (the receiver may have non-full buffer and large AW, but the router(s) in between might be in trouble). If routers are congested, the IP packets are dropped, which causes retransmission of TCP segments, which in turn worsens the congestion. Therefore, for the purpose of congestion control, TCP maintains congestion window (CW). The effective window size is now:

$$W = min\{CW, AW\} - \underbrace{(Last\_Byte\_Sent - Last\_Byte\_ACKed)}_{\text{Bytes in flight}}$$

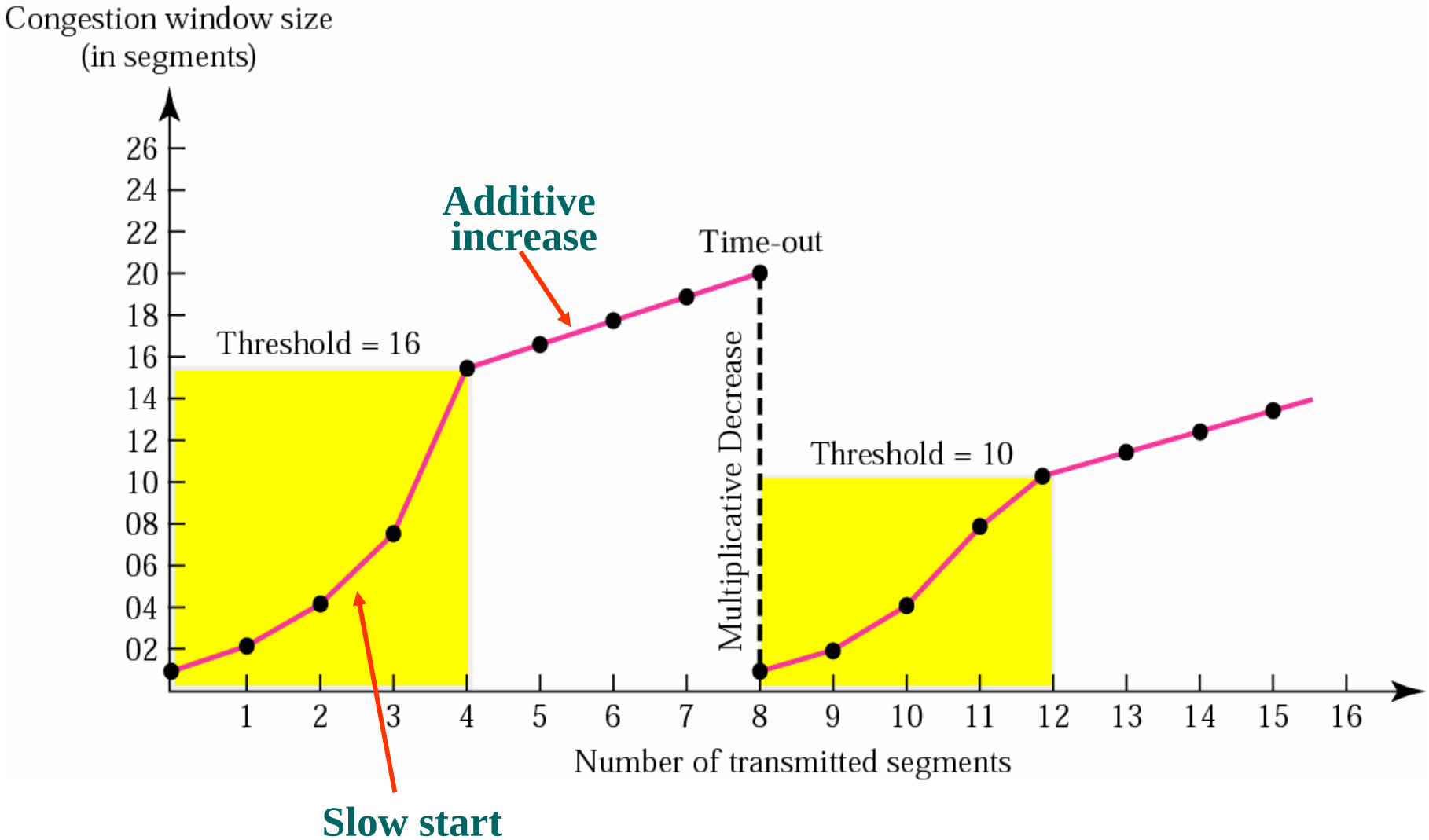
The CW size is determined by mechanism called additive increase/multiplicative decrease (AIMD):

Congestion Control (cont.)

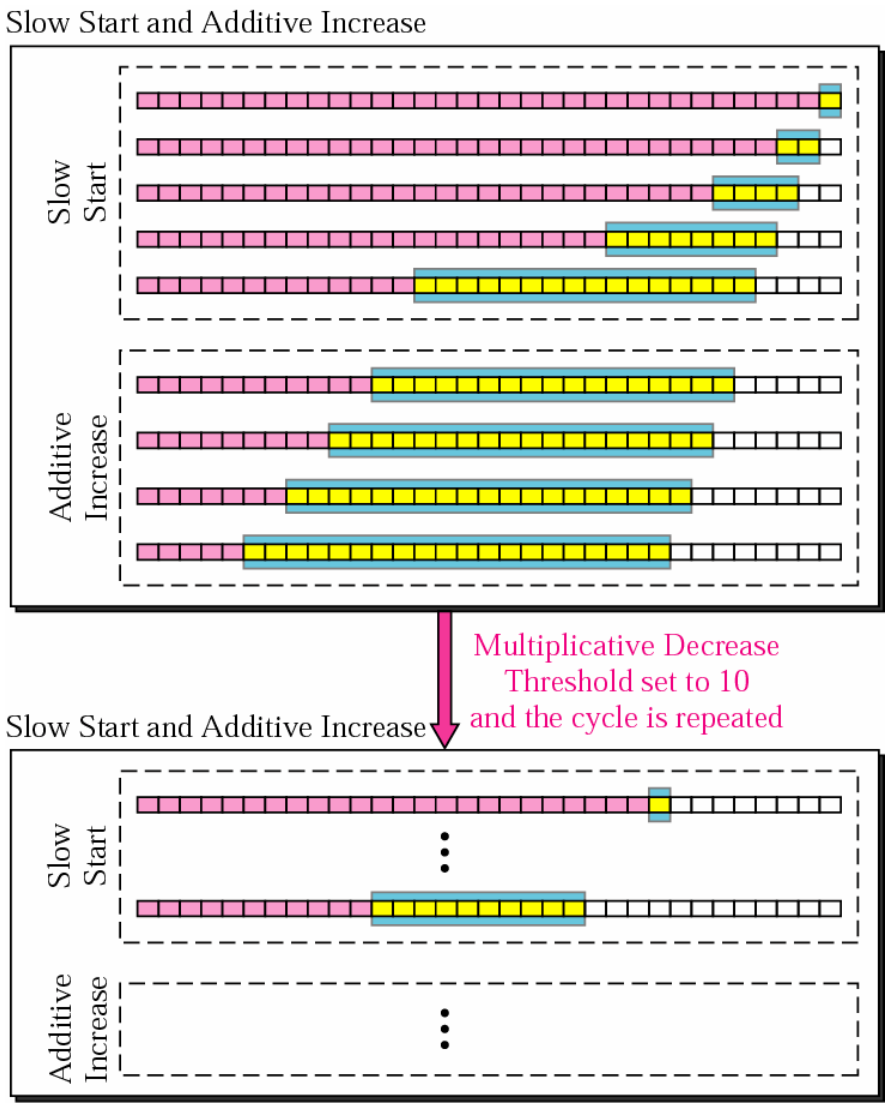
Additive increase/multiplicative decrease:

```
CW = 1*MSS;      // Initialize congestion window size
CWT = n*MSS;     // Initialize CW threshold (typically n = 16)
while (1) {      // Loop forever
    Wait for ACK or time-out;
    if (ACK received && CW <= CWT)
        CW = CW + 1*MSS;                  // "Slow start"
    elseif (ACK received && CW > CWT)
        CW = CW + (MSS/CWT)*MSS;          // Additive increase (by a fraction of MSS)
        CW = min(CW, WMAX);               // Limit increase
    elseif (time-out)                     // This indicates congestion in network
        CW = 1*MSS;
        CWT = ½ CWT;                      // Multiplicative decrease
}
```

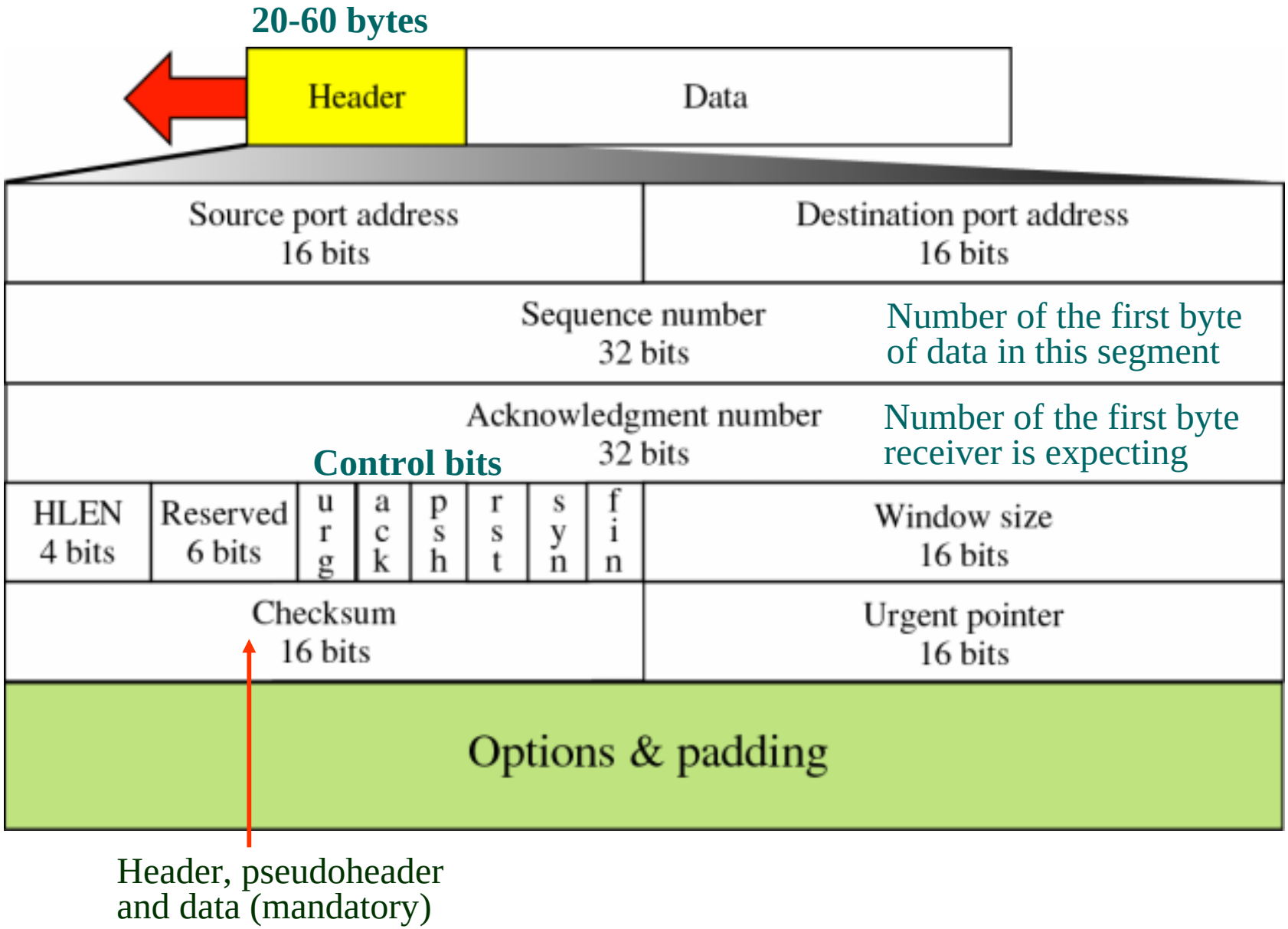
Additive Increase/Multiplicative decrease



Congestion avoidance strategies



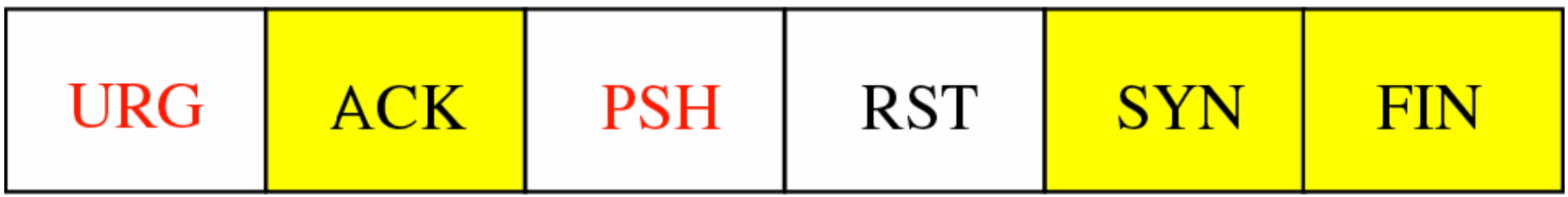
TCP segment format



Control field

Actually request to start the connection establishment

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

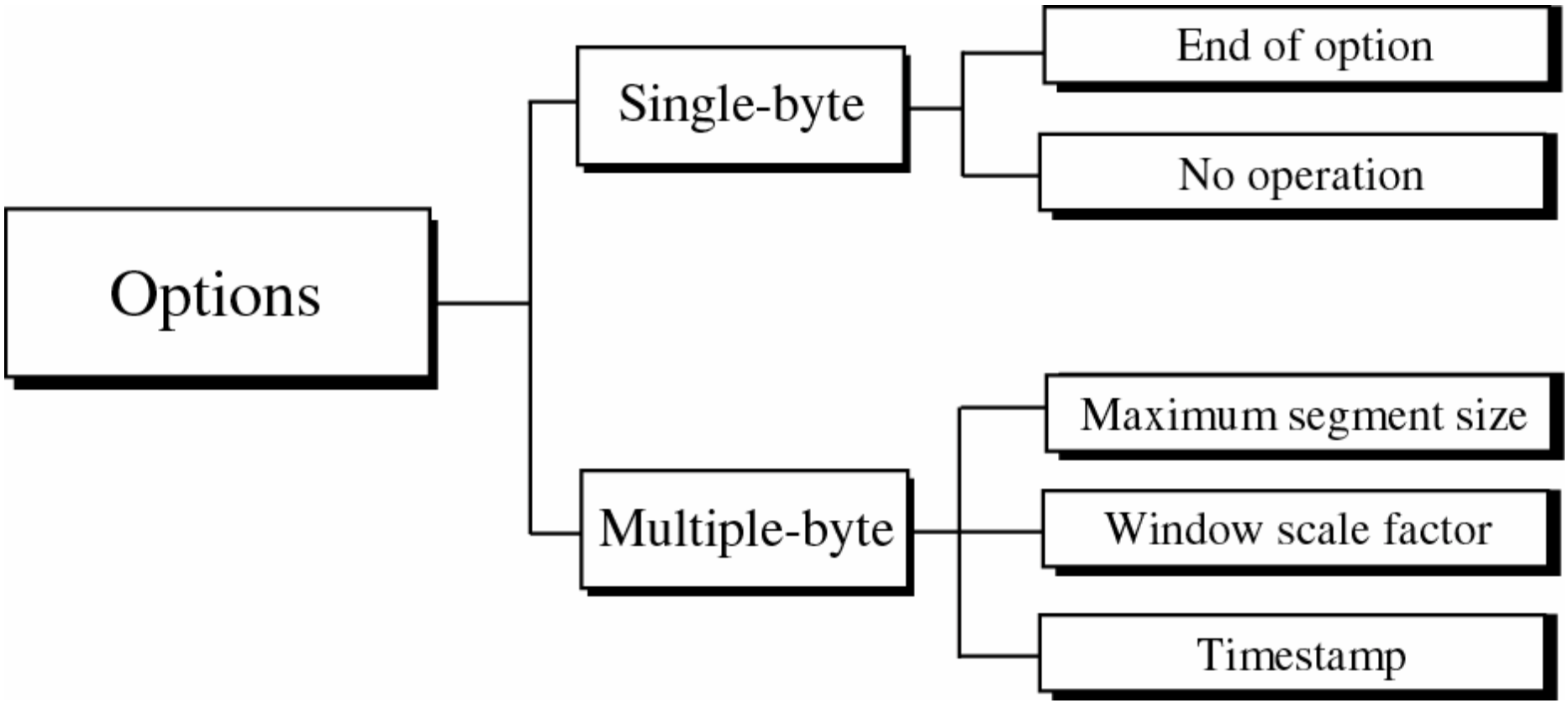


Push and urgent operations

In some interactive applications we don't want to wait that TCP fills the segment with data, instead we want an immediate response (for example an immediate echo after a single keystroke in TELNET).
For this purpose the application is requesting from the sending TCP a no buffer request (push). This request is also sent to the destination (PSH flag) asking the TCP to deliver data immediately to the receiving application.

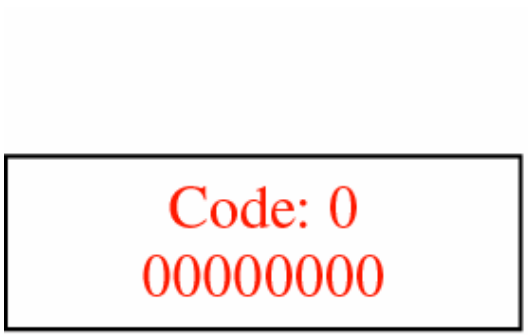
Sometimes there are urgent data that we want to deliver out of order, in the middle of the data stream transfer (example: TCP wants to abort a long data stream that already has been sent, i.e. wants to send Control-C ahead of data). Such urgent data are placed at the beginning of an urgent segment (with URG flag). The urgent pointer points to the end of the urgent data in the segment. The receiving TCP extracts the urgent data from the segment and delivers them to the application immediately and out of order.

Options

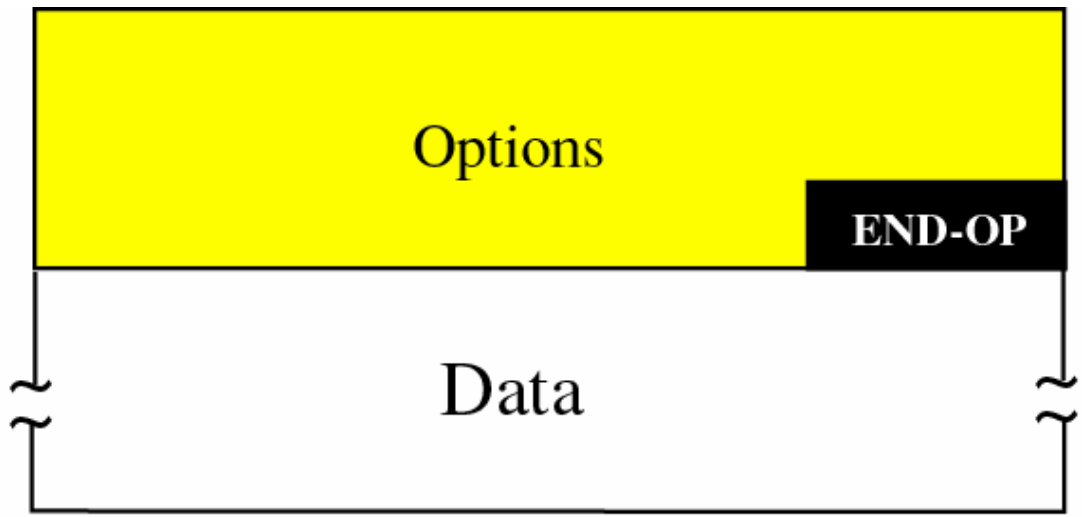


Similar to IP datagram options

End of option option

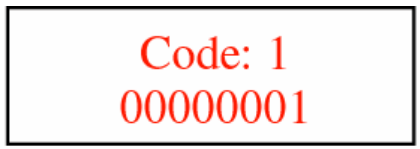


a. End of option

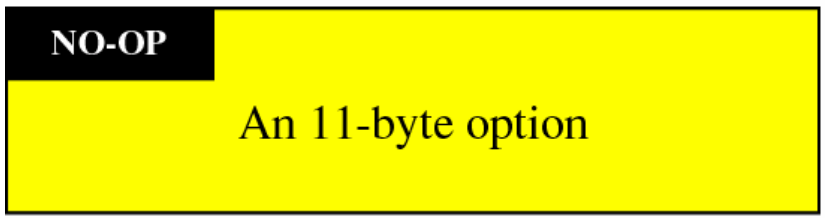


b. Used for padding

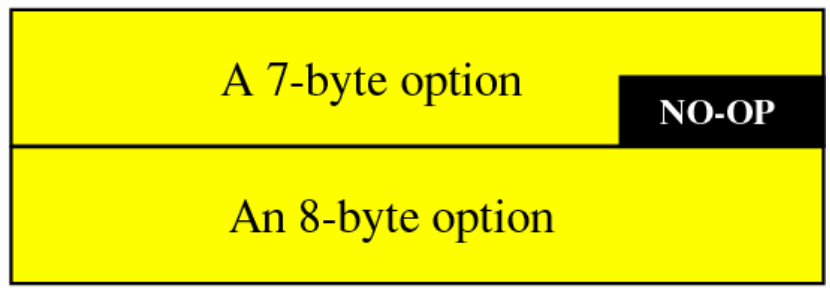
No operation option



a. No operation option



b. Used to align beginning of an option



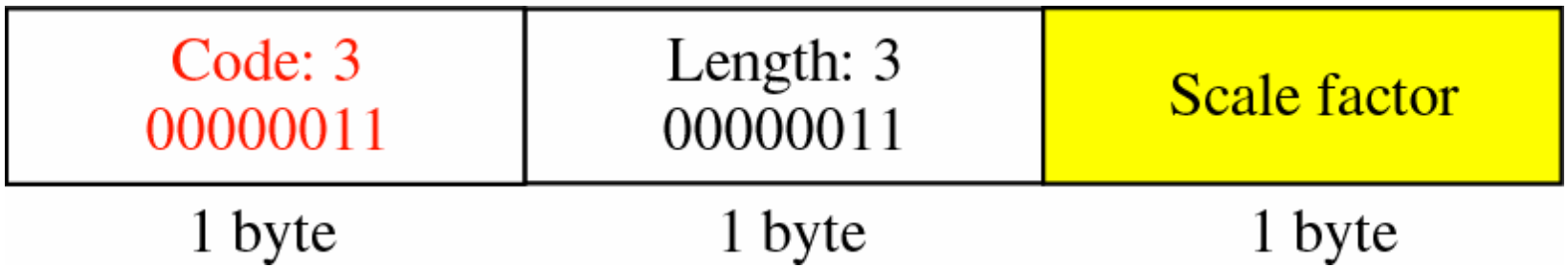
c. Used to align the next option

Maximum segment size option

Actually, maximum data size that can be received by the destination. It is determined by destination at the connection establishment time. The default is 536.

Window scale factor option

(Used only at connection establishment time)



From the header

Scaling factor

New Window Size = WS*2^{SF}

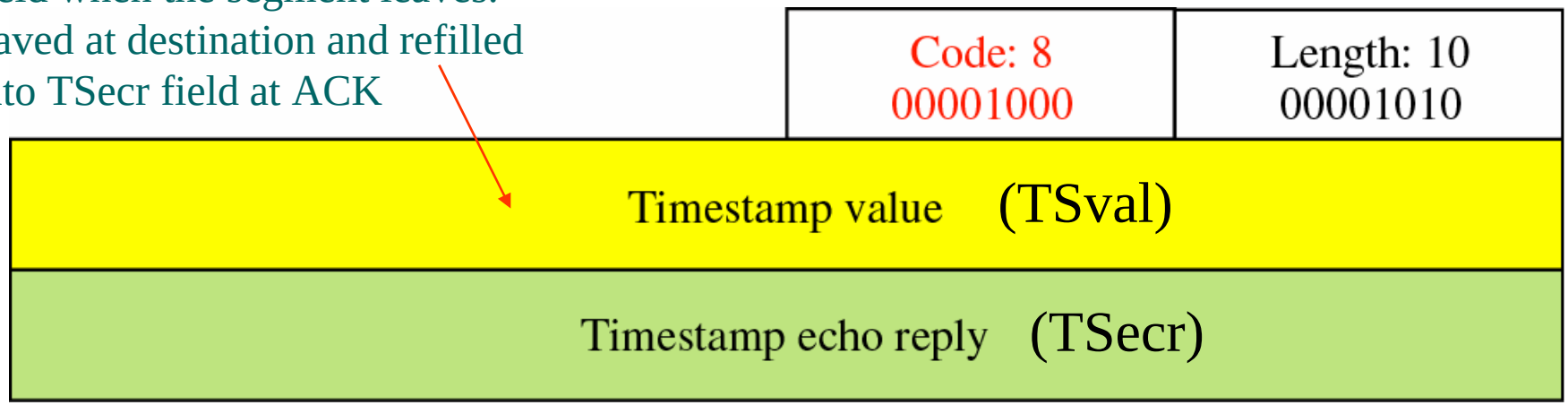
Maximum allowed by TCP: SF max = 16, $2^{16}*2^{16}=2^{32}$

When is needed such huge window size?
Communication at very long distances over high bandwidth links.
(For example, propagation of a packet at distance 6000 miles takes approx 40 ms and 80 ms to get ACK. During that time could be sent 12 Mbytes over OC-24

Timestamp option

Used to measure RTT

Filled by the source into TSval field when the segment leaves. Saved at destination and refilled into TSecr field at ACK



Source
(clock at transmission = 20)

Destination
(clock at transmission = 200)

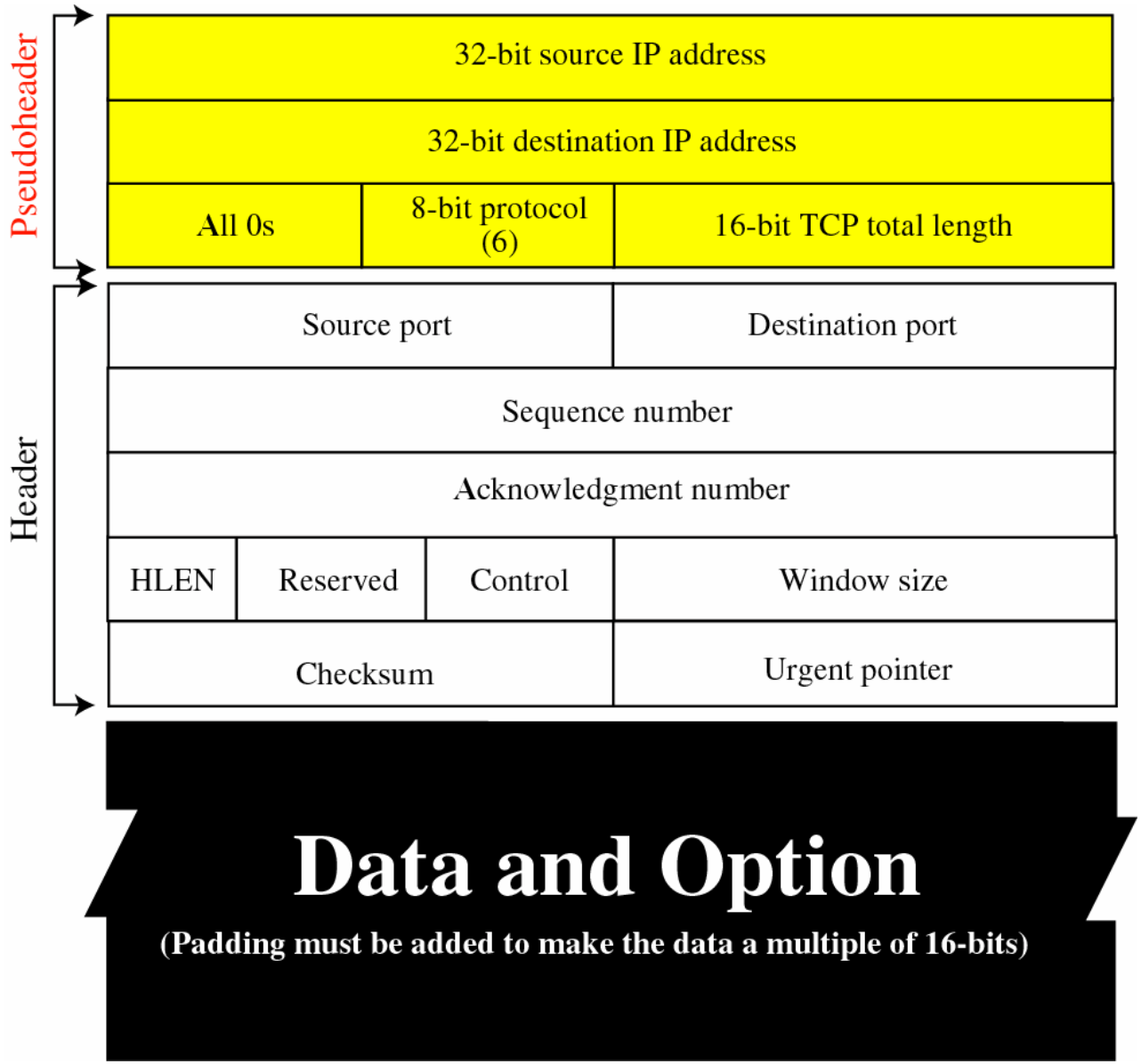
Source
(clock at receiving= 390)

RTT = 390-20=270

Clocks of source and destination don't have to be synchronized since the time measurement is done at the same side.

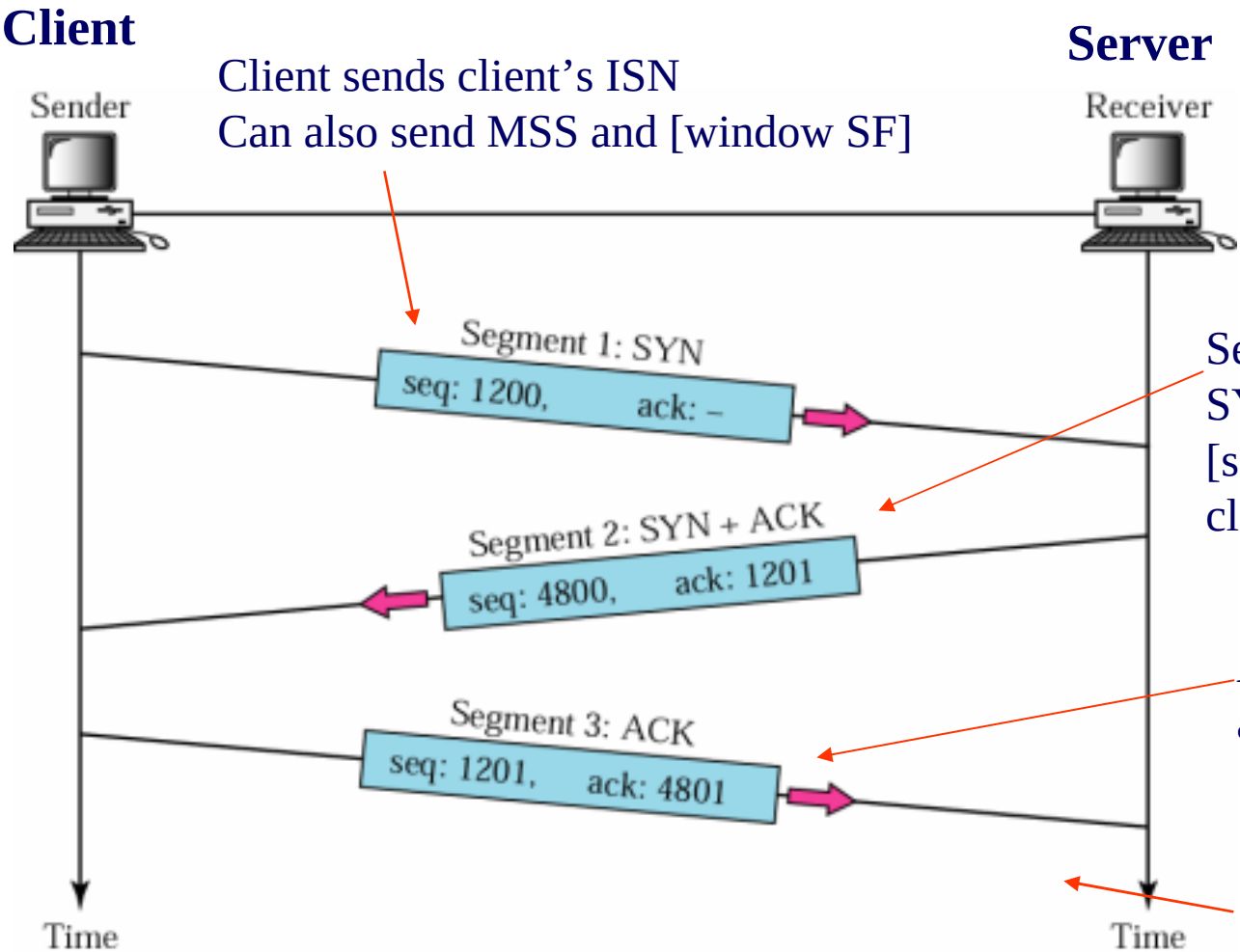
Tricky: TCP is using timestamp segment and its ACK to store the outgoing transmission time (20)

Pseudoheader added to the TCP datagram



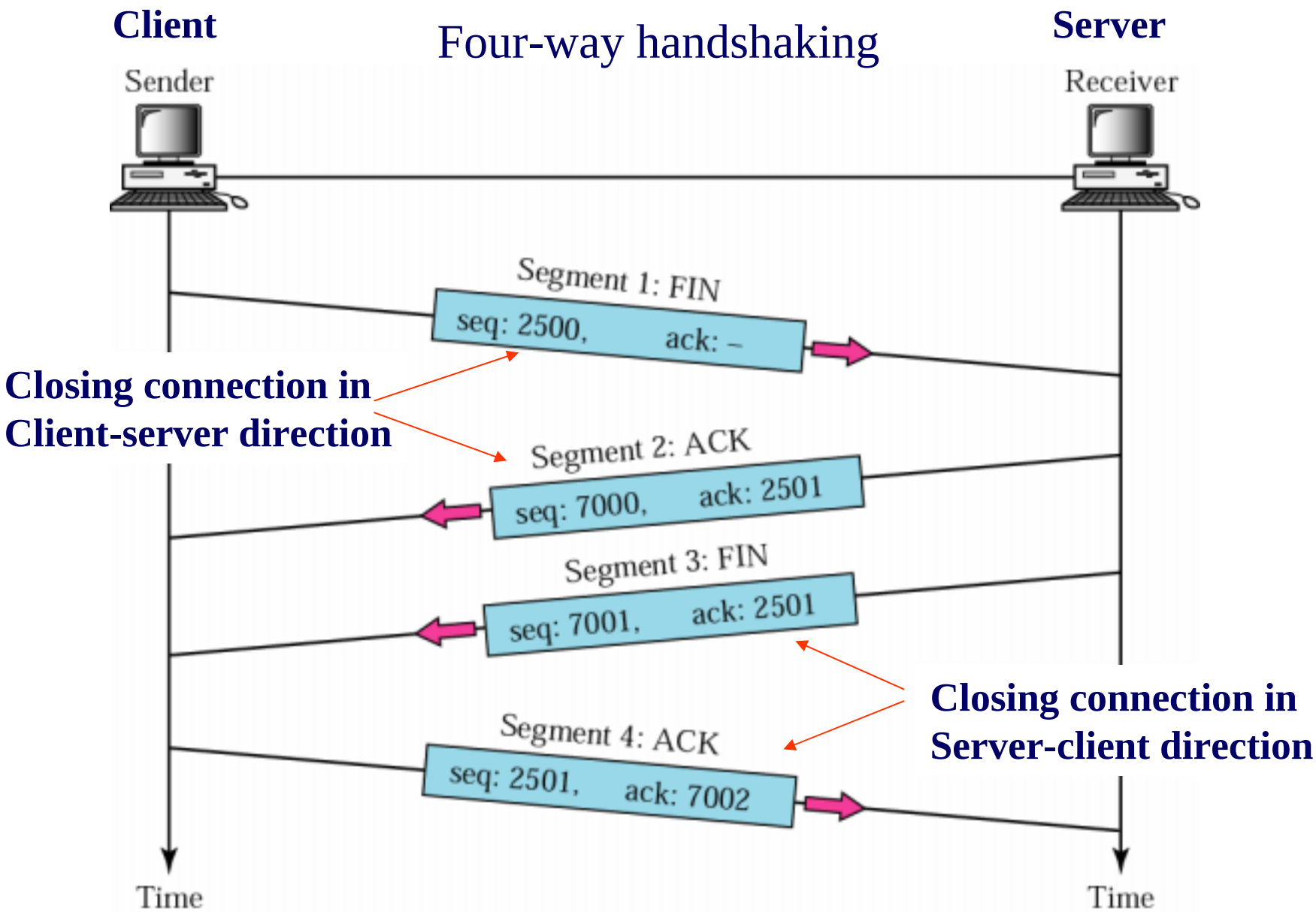
Connection Establishment

Three-way handshaking



ISN = Initial sequence number

Connection Termination

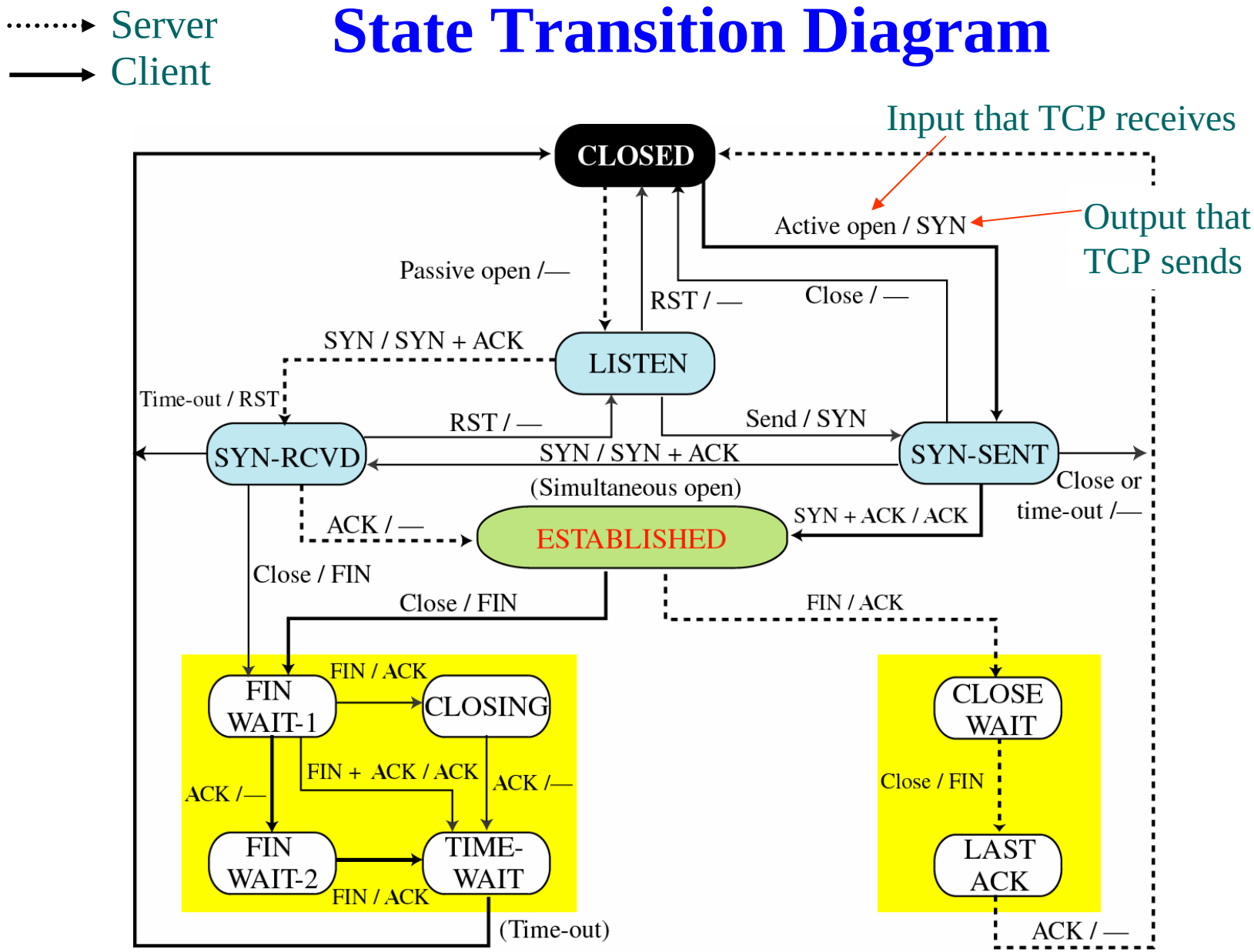


Implementation of TCP

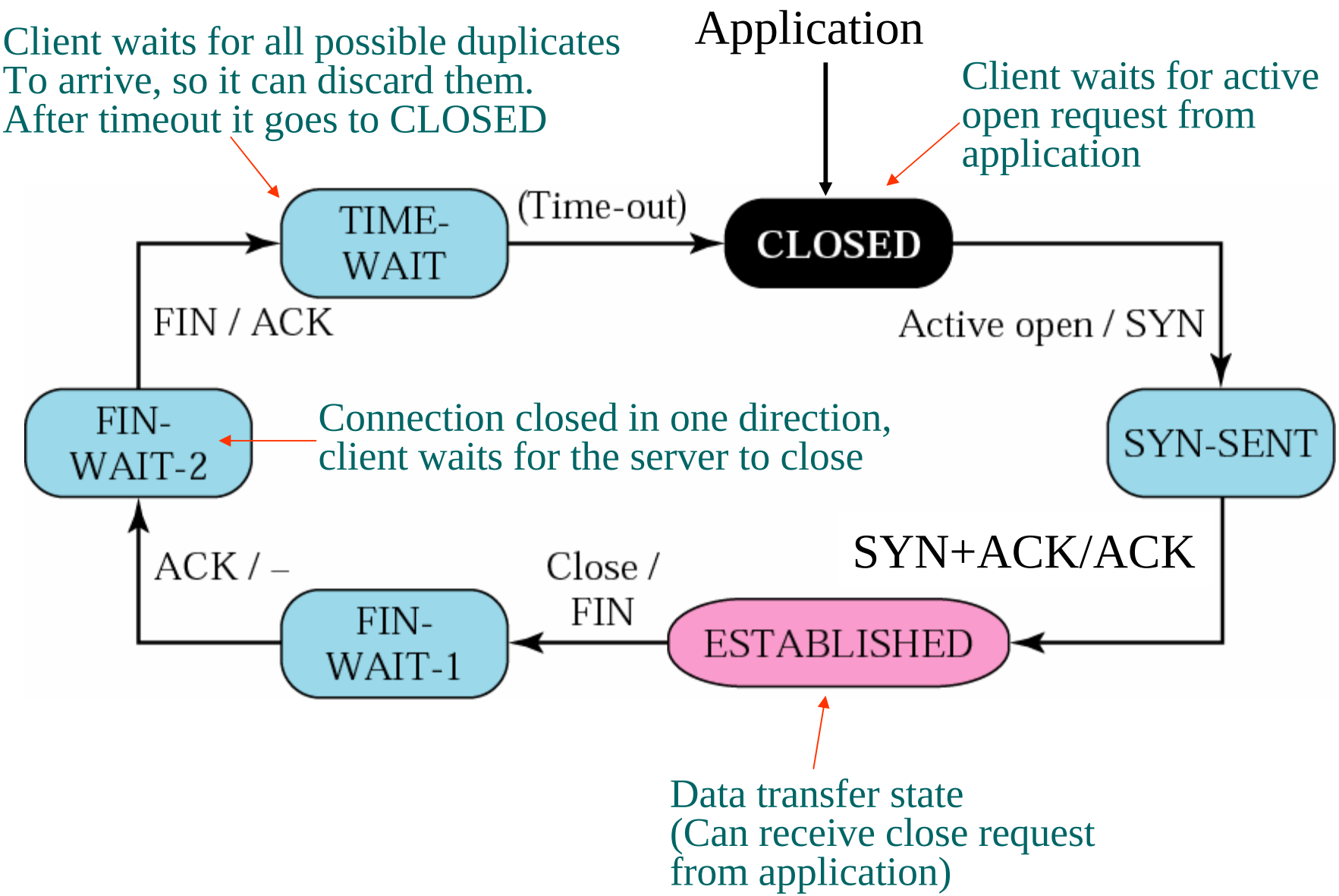
TCP connection management is implemented as a finite state machine (FSM) with the following states:

State	*Description*
CLOSED	There is no connection
LISTEN	The server is waiting for call from a client
SYN-SENT	A connection request is sent, waiting for ACK
SYN-RCVD	A connection request is received
ESTABLISHED	Connection is established
FIN-WAIT-1	The application has requested closing of connection
FIN-WAIT-2	The other side has accepted the closing of connection
CLOSING	Both sides have decided to close simultaneously
TIME-WAIT	Waiting for retransmitted segments to die
CLOSE-WAIT	The server is waiting for application to close
LAST-ACK	The server is waiting for the last acknowledgement

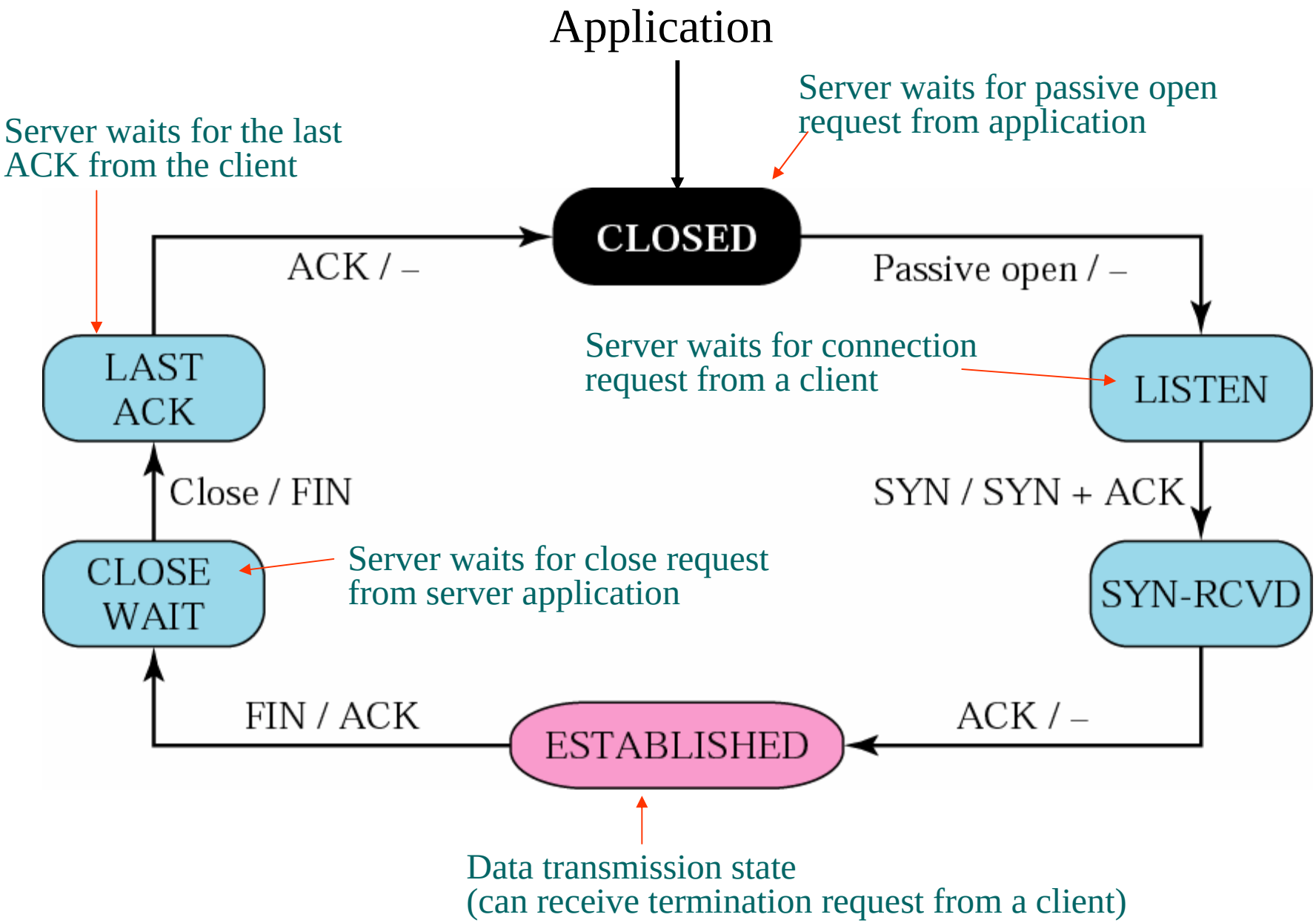
Connection Management State Transition Diagram



Client Diagram



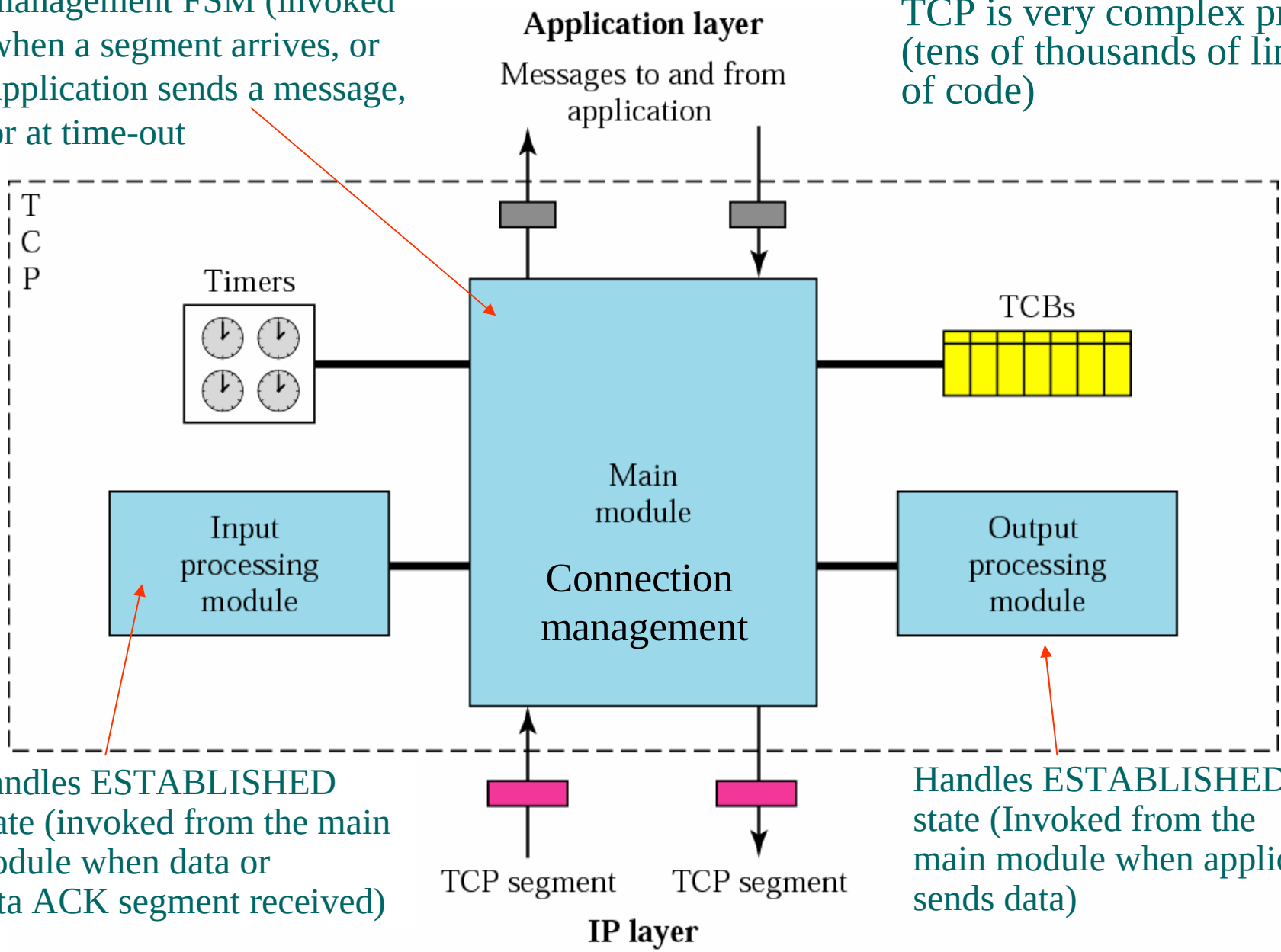
Server Diagram



TCP package

Implements the connection management FSM (invoked when a segment arrives, or application sends a message, or at time-out

TCP is very complex program (tens of thousands of lines of code)



Handles ESTABLISHED State (invoked from the main module when data or data ACK segment received)

Handles ESTABLISHED state (Invoked from the main module when application sends data)

Excerpt from the main module

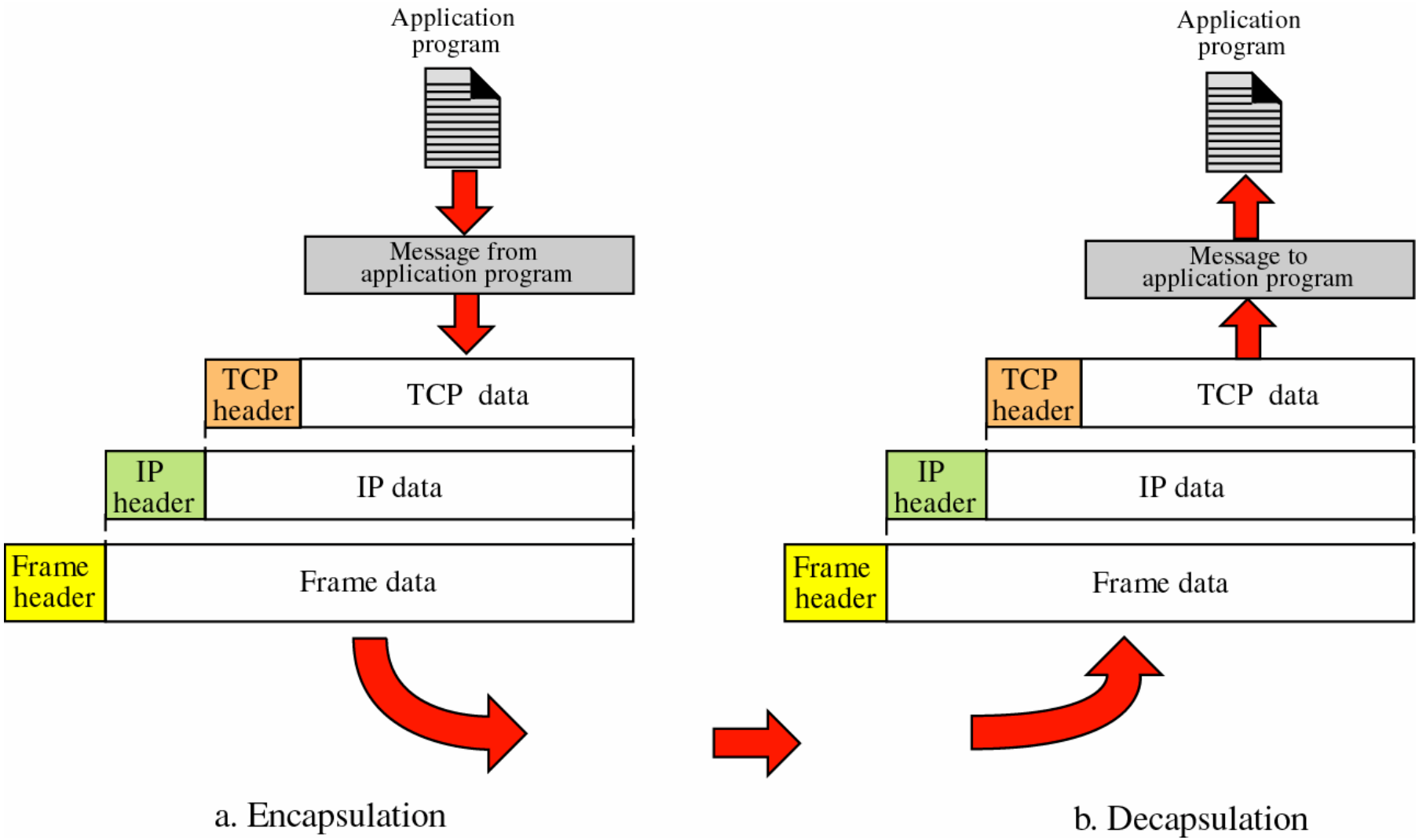
```
Receive a TCP segment, a message from an application, or time-out event;
if (corresponding entry in TCB not found)
        Create a new entry in TCB with state = CLOSED;
Get the pointer to TCB entry (ptcb);
Switch (ptcb->state){
. . . . . .
case ESTABLISHED:
        if (FIN segment received)
                send  an ACK segment; change state to CLOSE-WAIT;
        if ("close" message from application received)
                send FIN segment; change state to FIN-WAIT-1;
        if (RST or SYN segment received)
                issue an error message;
        if (data or ACK segment received)
                call the Input Module;
        if ("send" message from application received)
                call the Output Module;
        return;
. . . . . .
}
```

Transmission Control Block (TCB)

TCP maintains necessary data in TCB for each open connection.
A TCB entry has following fields:

TCP State (CLOSED, LISTEN, SYN-SENT,…)
Process ID
Local IP address and port number
Remote IP address and port number
Local interface
Local window (contains all information associated with local sliding window)
Remote window (contains all information associated with remote sliding window)
Sending sequence number
Receiving sequence number
Sending ACK number
Parameters (RTT, DRTT, ERTT, CW, CWT, WMAX, MSS)
Time-outs (retransmission, persistence, keepalive…)
Buffer size (local buffer)
Pointers to send and receive buffers

Encapsulation and decapsulation



Multiplexing and demultiplexing