UNIT - 4

Transport Layer

- INTRODUCTION
- TRANSPORT-LAYER PROTOCOLS
- INTERNET TRANSPORT-LAYER PROTOCOLS
  - ✓ USER DATAGRAM PROTOCOL (UDP)
  - ✓ **TRANSMISSION CONTROL PROTOCOL (TCP)**

TRANSMISSION CONTROL PROTOCOL (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.

TCP Services

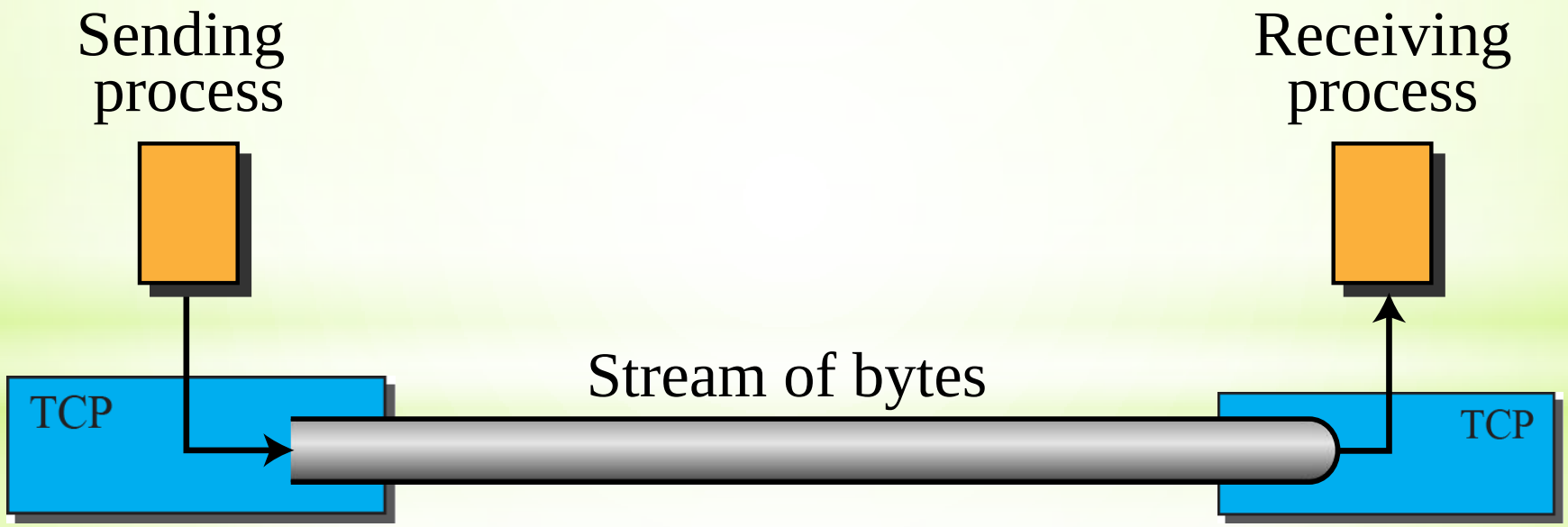
- Process-to-Process Communication
- Stream Delivery Service
  - Sending and Receiving Buffers
  - Segments
- Full-Duplex Communication
- Multiplexing and Demultiplexing
- Connection-Oriented Service
- Reliable Service

TCP Services

- Process-to-Process Communication
  - TCP provides process-to-process communication using port numbers.
  - Senders port number & Destination Port number are placed in header of TCP Segment
- Stream Delivery Service
  - TCP groups a number of bytes together into a packet called a segment
  - TCP adds a header to each segment (for control purposes) and delivers the segment to the network layer for transmission. The segments are encapsulated in an IP datagram and transmitted. This entire operation is transparent to the receiving process.
  - The segments are not necessarily all the same size.

Stream delivery

- TCP, unlike UDP, is a stream-oriented protocol.
- TCP, allows the sending process to deliver data as a stream of bytes and allows the receiving process to obtain data as a stream of bytes.

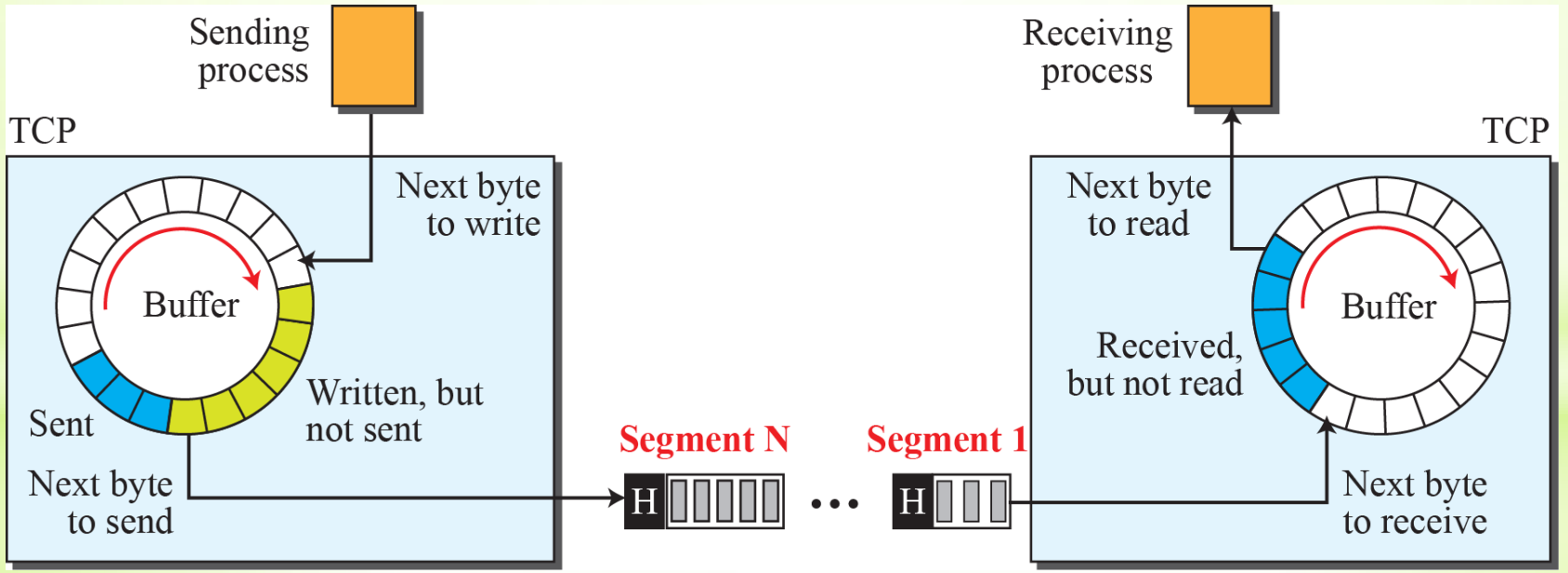


Sending and receiving buffers

- There are two buffers, the sending buffer and the receiving buffer, one for each direction
- The TCP sender keeps these bytes in the buffer until it receives an acknowledgment.

TCP segments

- TCP groups a number of bytes together into a packet called a segment
- TCP adds a header to each segment (for control purposes) and delivers the segment to the network layer for transmission. The segments are encapsulated in an IP datagram and transmitted. This entire operation is transparent to the receiving process.
- The segments are not necessarily all the same size.



Full-Duplex Communication

TCP offers full-duplex service, where data can flow in both directions at the same time. Each TCP endpoint then has its own sending and receiving buffer, and segments move in both directions.

Multiplexing and Demultiplexing

TCP performs multiplexing at the sender and demultiplexing at the receiver. However, since TCP is a connection-oriented protocol, a connection needs to be established for each pair of processes.

Connection-Oriented Service

TCP is a connection-oriented protocol. With Three Phases:

1. The two TCP's establish a logical connection between them.
2. Data are exchanged in both directions.
3. The connection is terminated.

Reliable Service

TCP is a reliable transport protocol. It uses an acknowledgment mechanism to check the safe and sound arrival of data.

TCP Features

To provide the services mentioned in the previous section, TCP has several features:

- Numbering System
  - Byte Number
    - The bytes of data being transferred in each connection are numbered by TCP.
    - The numbering starts with an arbitrarily generated number.
    - Numbering is independent in each direction.
    - TCP chooses an arbitrary number between 0 and $2^{32} - 1$ for the number of the first byte

❖ Sequence Number

- After the bytes have been numbered, TCP assigns a sequence number to each segment that is being sent. The sequence number, in each direction, is defined as follows:
  1. The sequence number of the first segment is the ISN (initial sequence number), which is a random number.
  2. The sequence number of any other segment is the sequence number of the previous segment plus the number of bytes (real or imaginary) carried by the previous segment.

❖ Acknowledgment Number

- TCP is full duplex; when a connection is established, both parties can send and receive data at the same time
- 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.

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

Segment

A packet in TCP is called a segment.

- Format

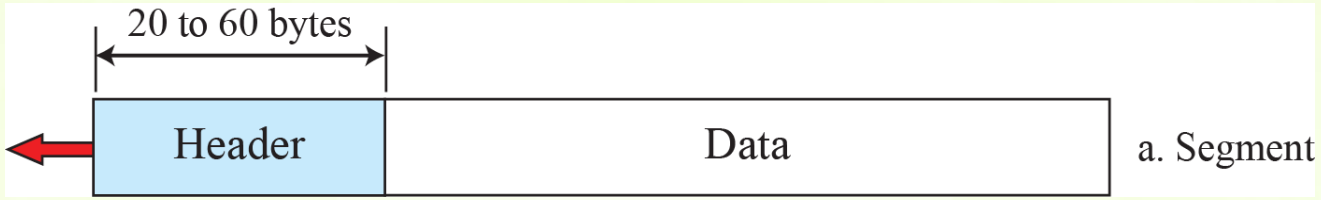
The segment consists of a header of 20 to 60 bytes, followed by data from the application program.

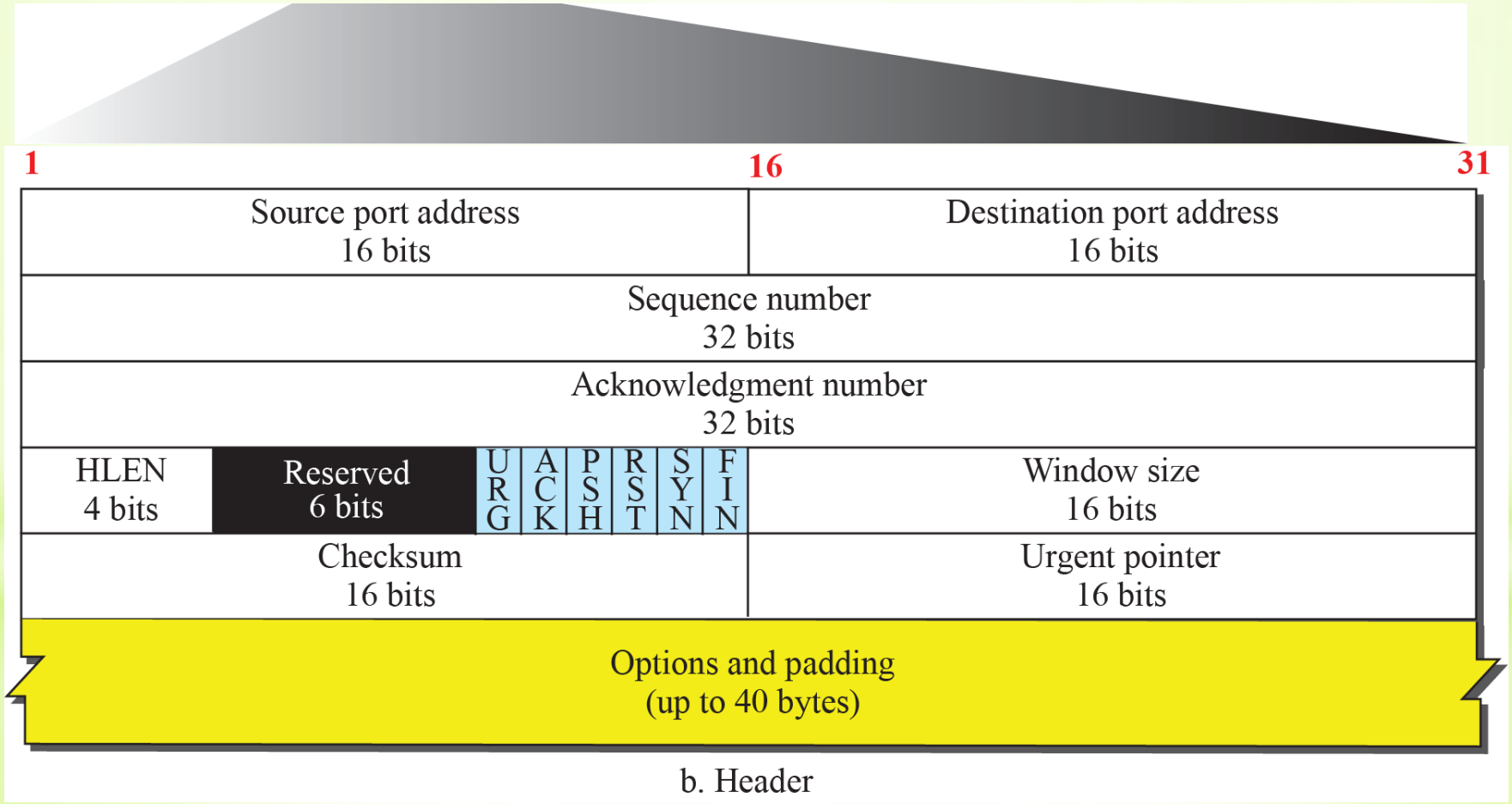
The header is 20 bytes if there are no options and up to 60 bytes if it contains options.

- Encapsulation

A TCP segment encapsulates the data received from the application layer. The TCP segment is encapsulated in an IP datagram, which in turn is encapsulated in a frame at the data-link layer

TCP segment format

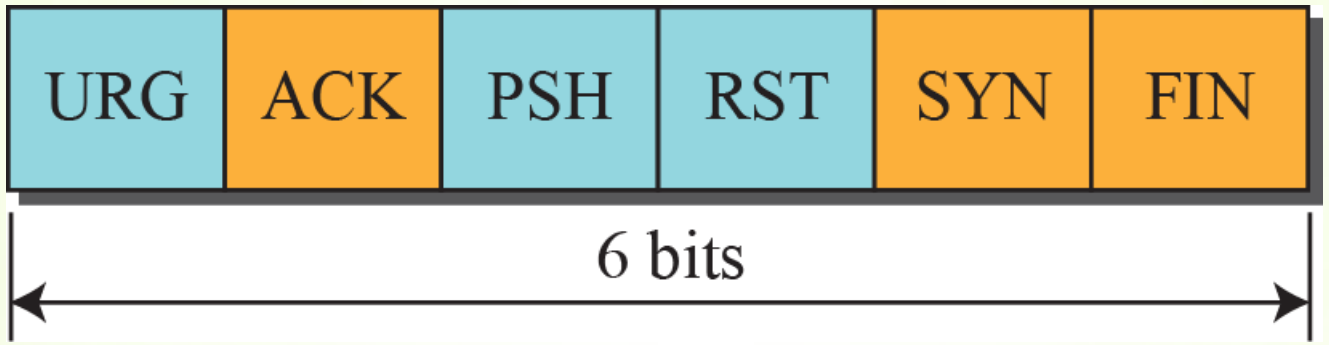




b. Header

- **Source port address (16 bits):** defines the port number of the application program in the host that is sending the segment
- **Destination port address (16 bits):** defines the port number of the application program in the host that is receiving the segment
- **Sequence number(32 bits):** defines the number assigned to the first byte of data contained in this segment
- **Acknowledgment number(32 bits):** defines the byte number that the receiver of the segment is expecting to receive from the other party
- **Header length(4 bits):** indicates the number of 4-byte words in the TCP header.

Control Field(6 bits): defines 6 different control bits or flags. One or more of these bits can be set at a time. These bits enable flow control, connection establishment and termination, connection abortion, and the mode of data transfer in TCP.



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

Window size(16 bits): defines the window size of the sending TCP in bytes. The maximum size of the window is 65,535 bytes. This value is normally referred to as the receiving window (rwnd) and is determined by the receiver.

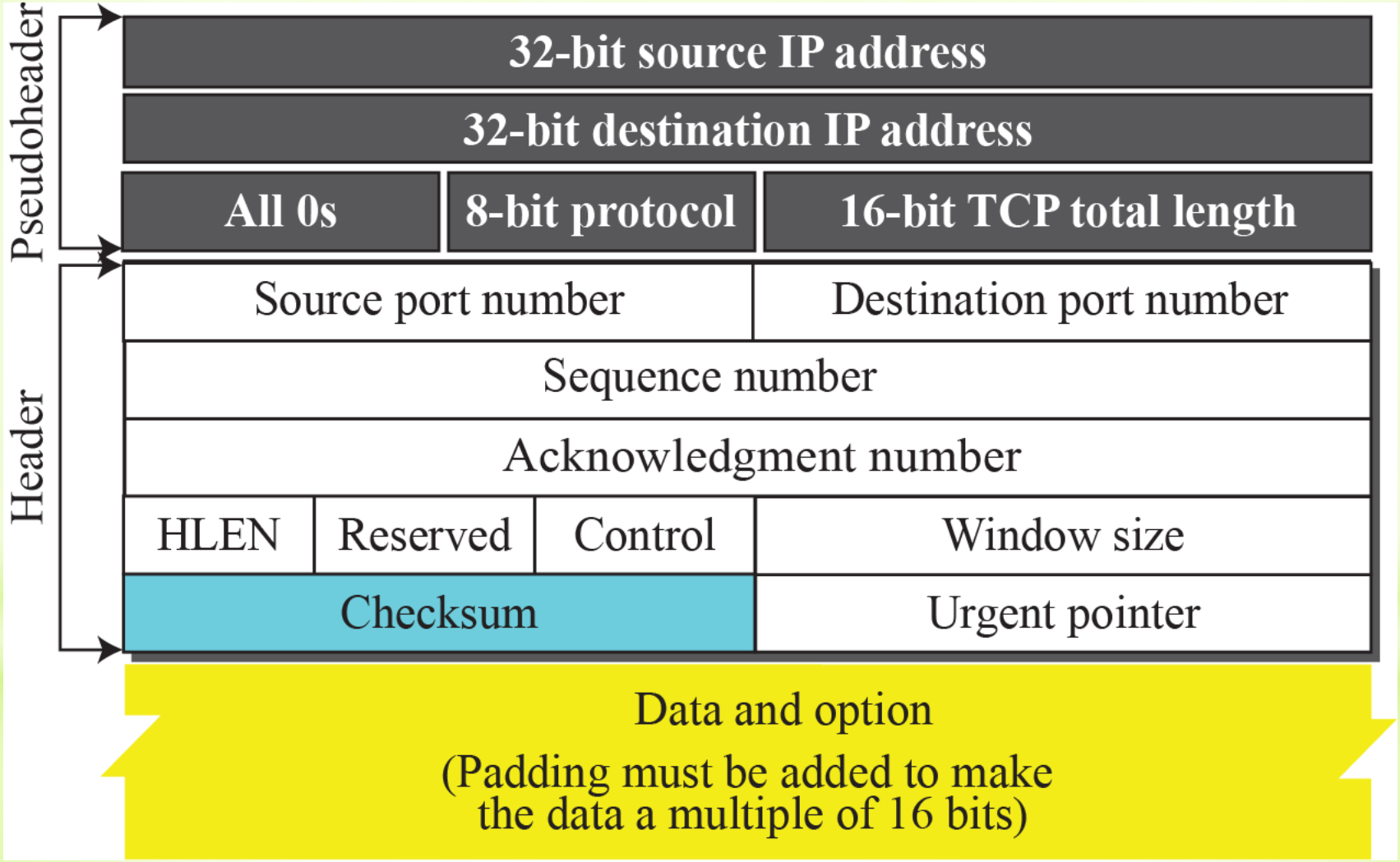
Checksum(16 bits): The calculation of the checksum for TCP follows the same procedure as the one described for UDP. However, the use of the checksum in the UDP datagram is optional, whereas the use of the checksum for TCP is mandatory.

Urgent pointer(16 bits): which is valid only if the urgent flag is set, is used when the segment contains urgent data. It defines a value that must be added to the sequence number to obtain the number of the last urgent byte in the data section of the segment.

Options: There can be up to 40 bytes of optional information in the TCP header.

Pseudoheader added to the TCP datagram

TCP pseudoheader, the value for the protocol field is 6.



A TCP Connection

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

(continued)

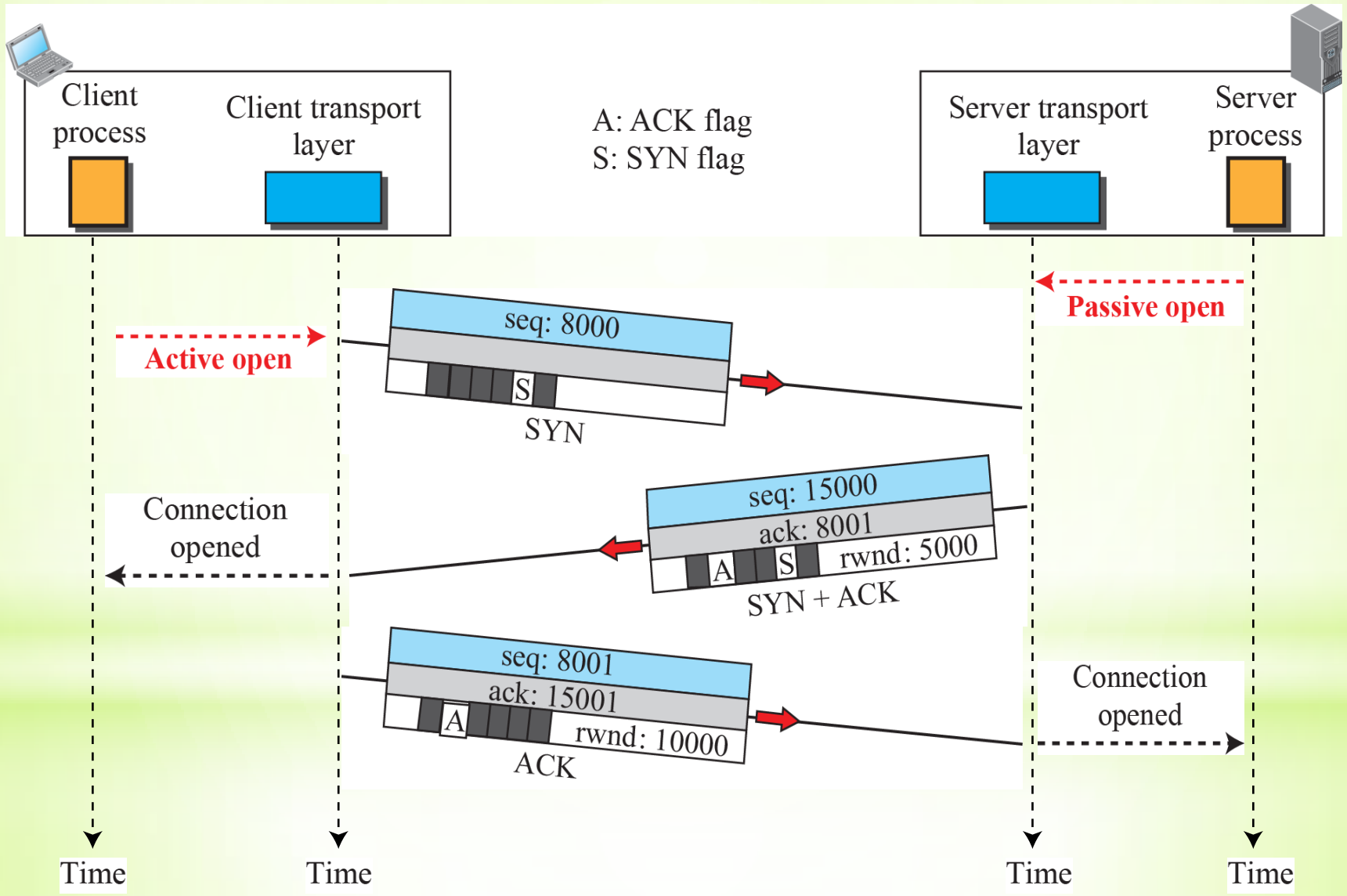
- Connection Establishment
  - Three-Way Handshaking
  - SYN Flooding Attack
- Data Transfer
  - Pushing Data
  - Urgent Data
- Connection Termination
  - Three-Way Handshaking
  - Half-Close
- Connection Reset

Connection Establishment

- TCP transmits data in full-duplex mode.
- The connection establishment in TCP is called ***three-way handshaking***.
- The process starts with the server. The server program tells its TCP that it is ready to accept a connection. This request is called a *passive open*.
- The client program issues a request for an *active open*.
- The connection establishment is done by exchanging 3 messages between the two parties:
    - SYN segment
    - SYN + ACK segment
    - ACK segment

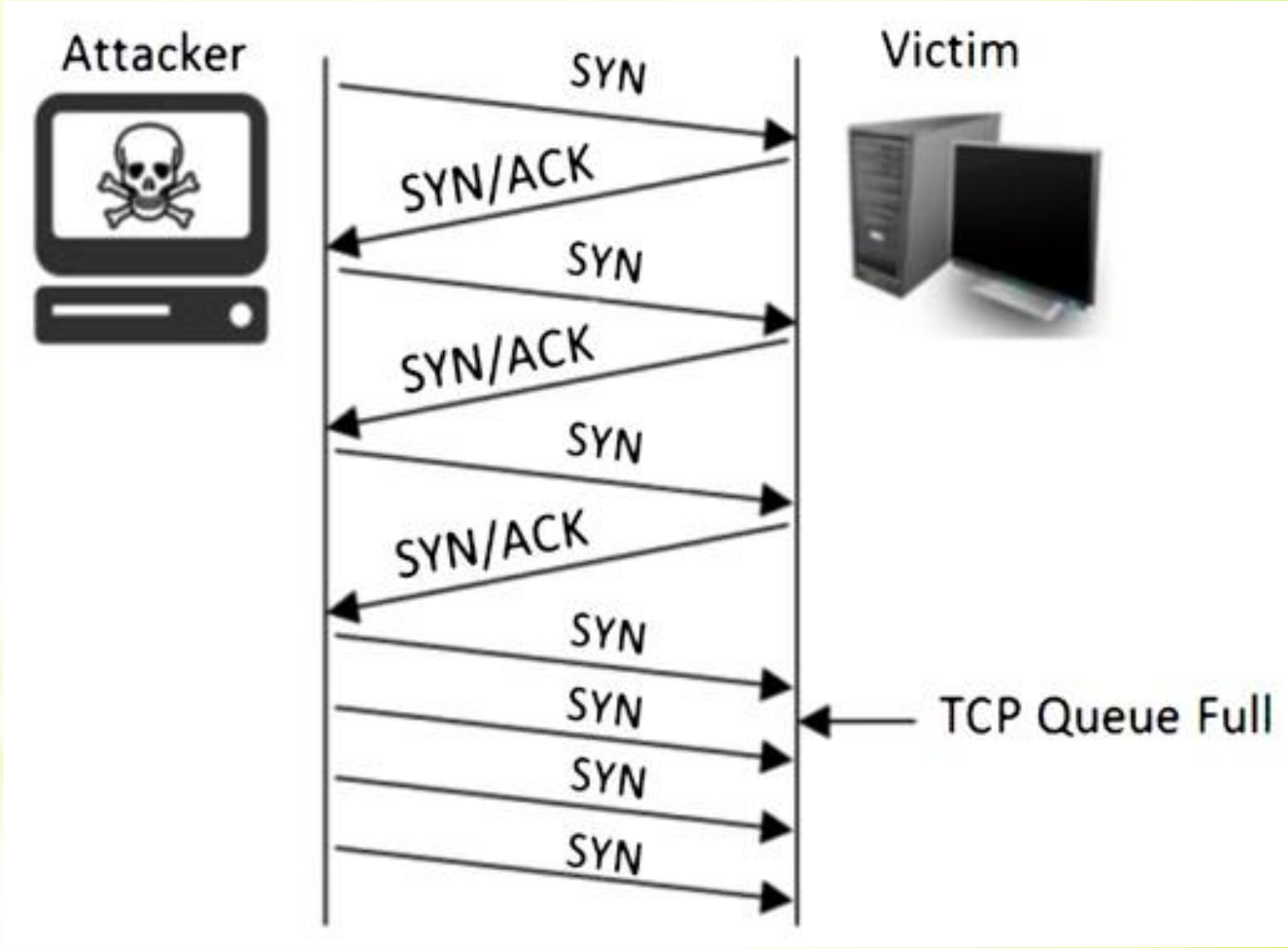
- A SYN segment cannot carry data, but it consumes one sequence number.
- A SYN + ACK segment cannot carry data, but it does consume one sequence number.
- An ACK segment, if carrying no data, consumes no sequence number.

Connection establishment using three-way handshaking

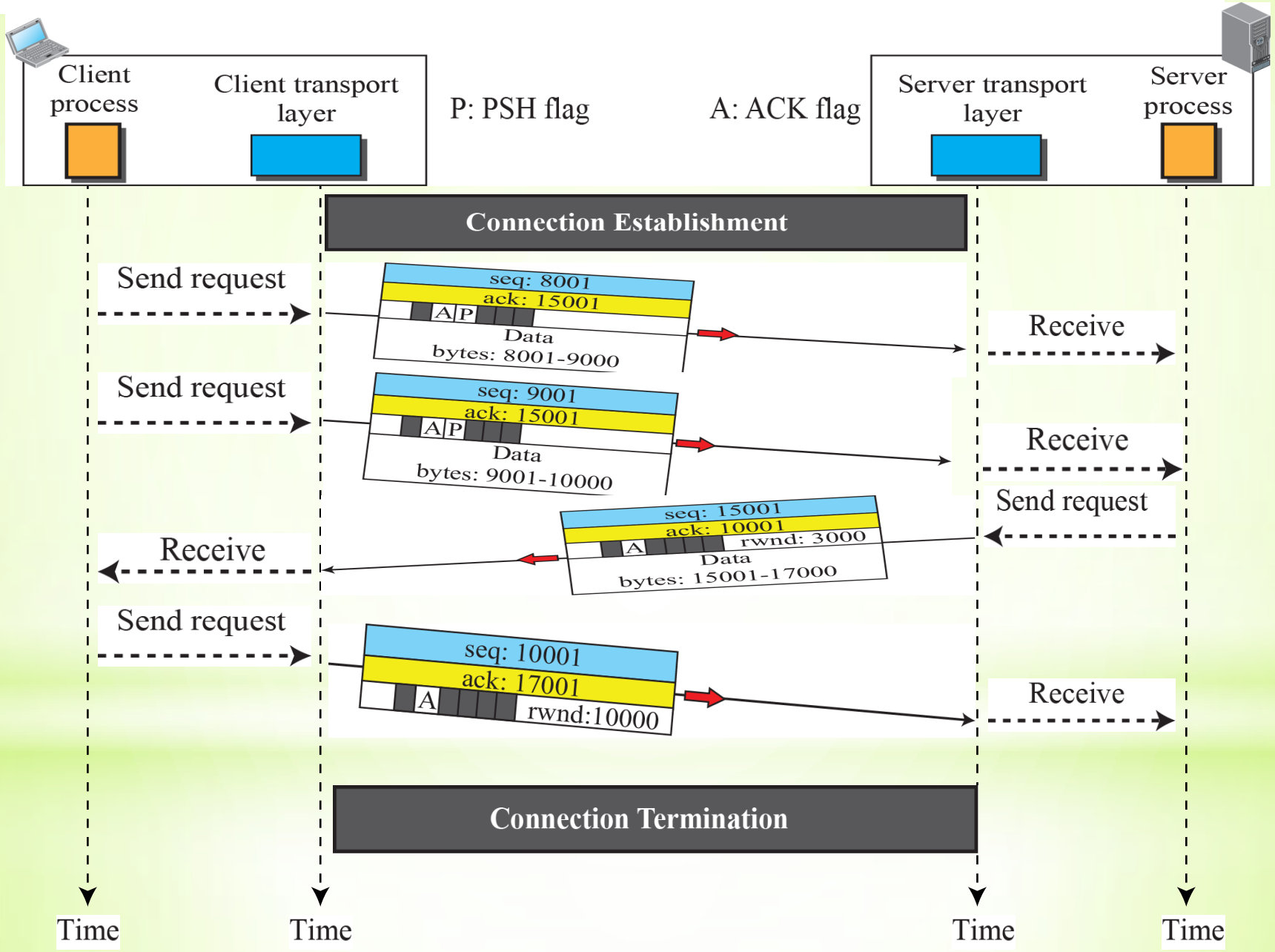


Denial of Service attack / SYN flooding attack

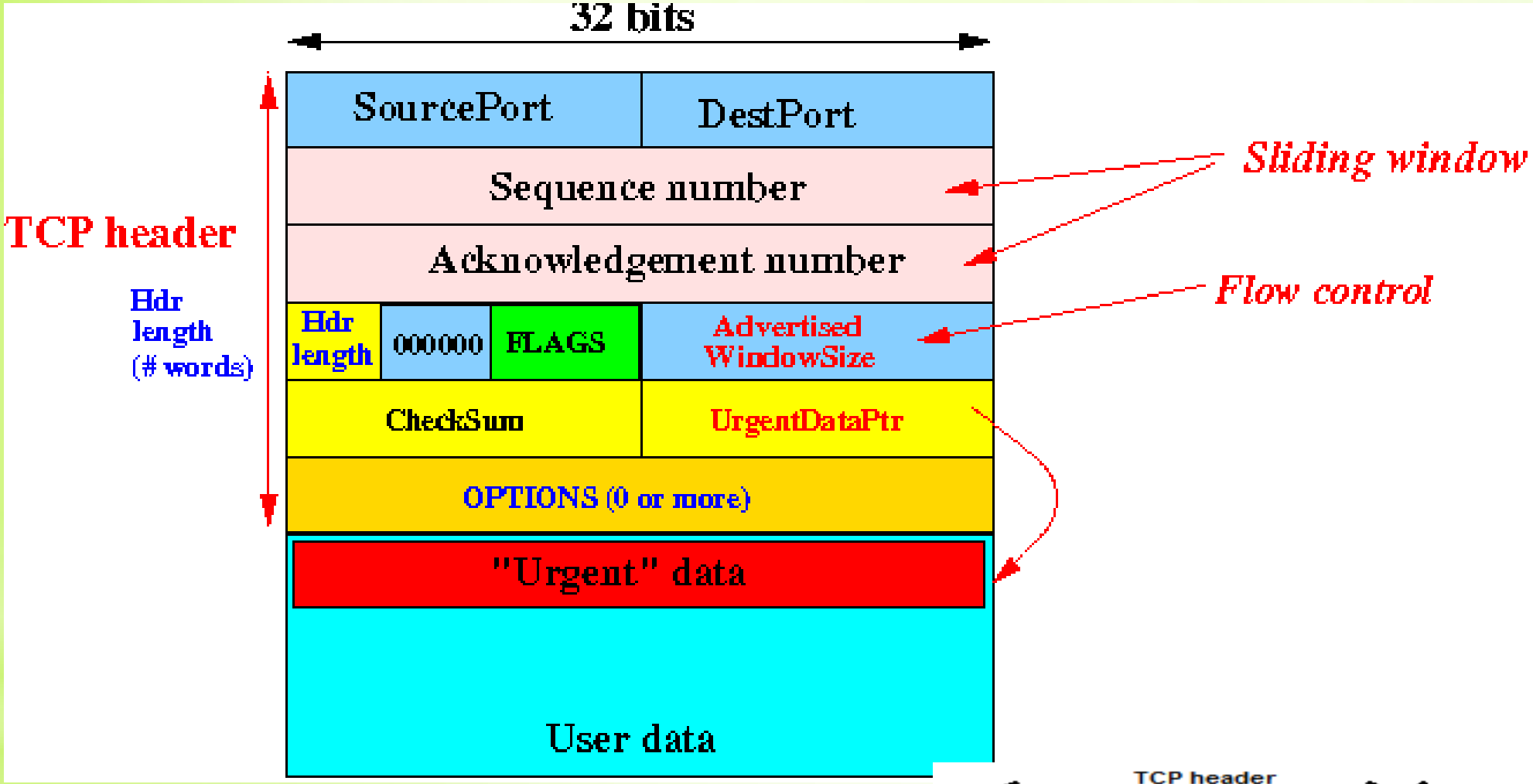
- The connection establishment procedure in TCP is susceptible to a serious security problem called ***SYN flooding attack***.
- This SYN flooding attack belongs to a group of security attacks known as a ***denial of service attack***
- The server, assuming that the clients are issuing an active open, allocates the necessary resources, such as creating transfer control block (TCB) tables and setting timers.
- One recent strategy is to postpone resource allocation until the server can verify that the connection request is coming from a valid IP address, by using what is called a ***cookie***.

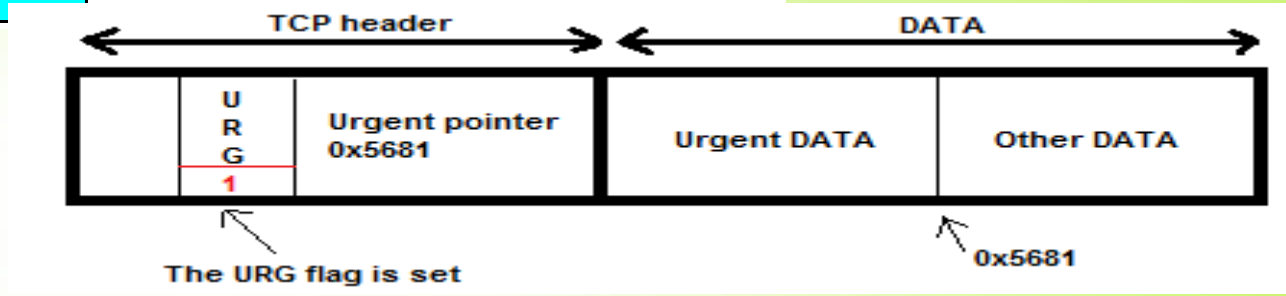


Data transfer



Urgent Pointer





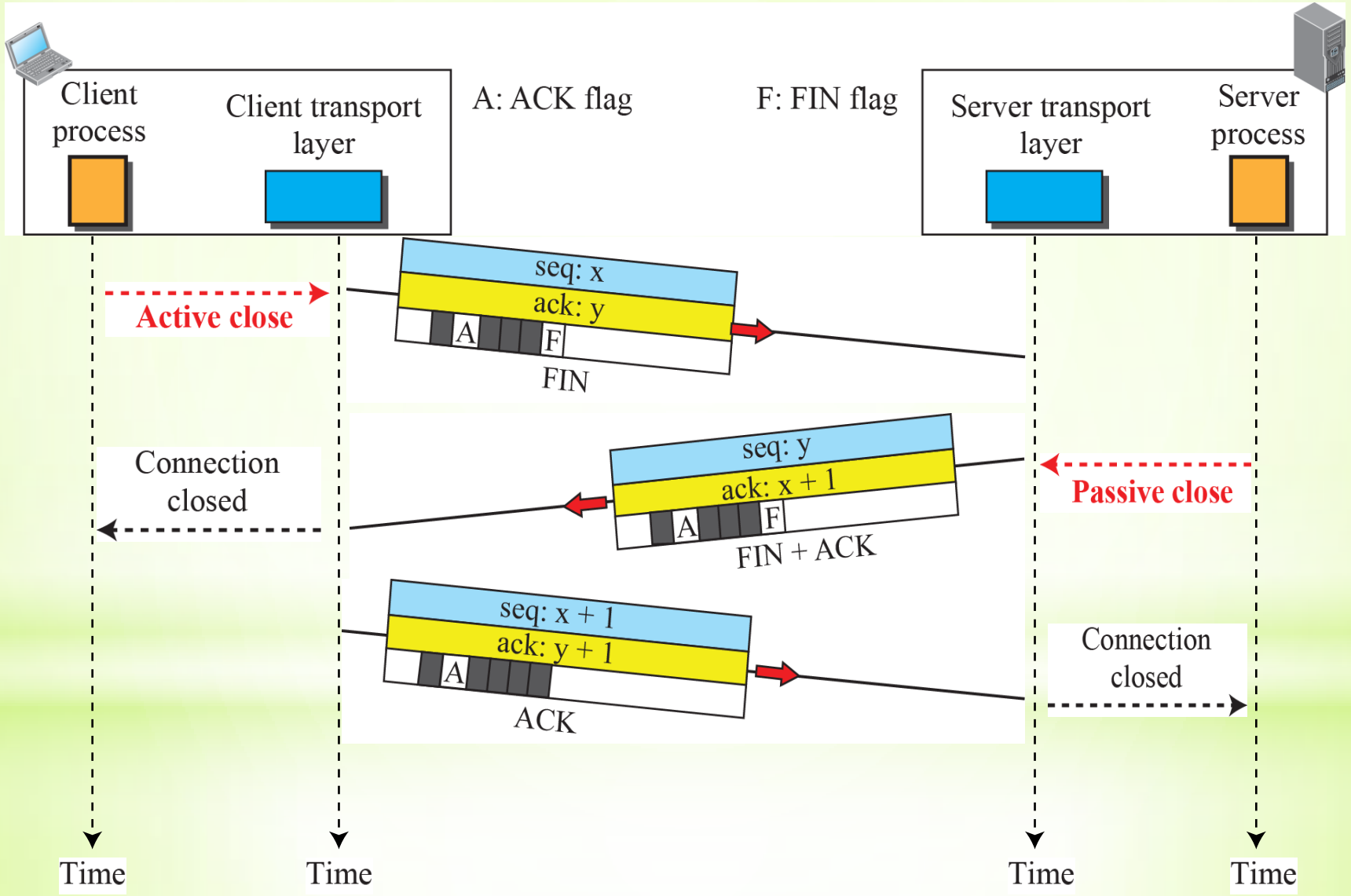
Connection Termination

- Most implementations today allow two options for connection termination: three-way handshaking and four-way handshaking with a half-close option.

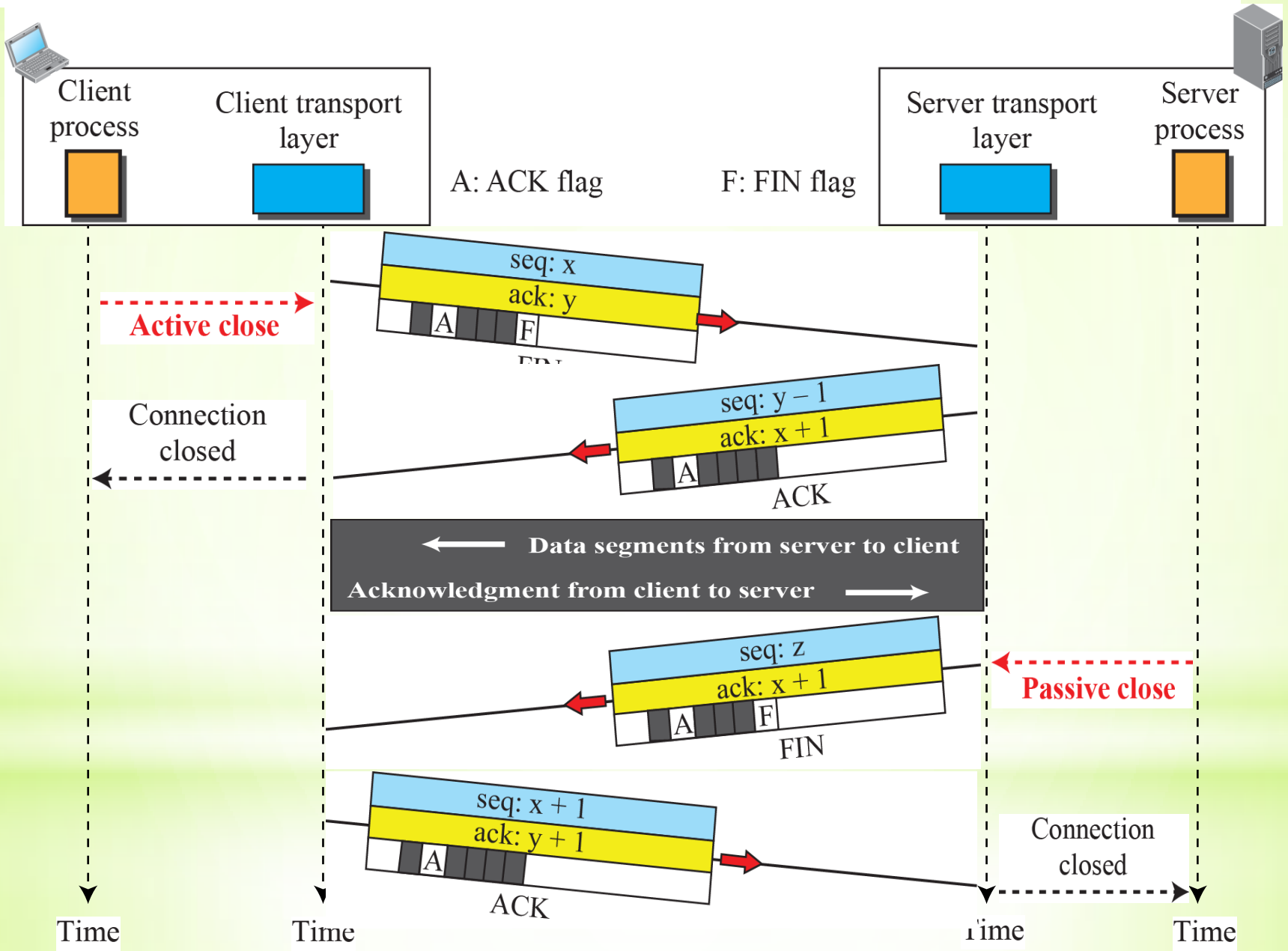
- *Three-way handshaking* for connection termination includes exchange of three messages between the two parties:
    - FIN segment
    - FIN + ACK segment
    - ACK segment

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

Connection termination using three-way handshaking



Half-close


Client process
Client transport layer
A: ACK flag
F: FIN flag
Server transport layer
Server process
seq: x
ack: y
A
F
Active close
seq: y – 1
ack: x + 1
A
ACK
Connection closed
Data segments from server to client
Acknowledgment from client to server
seq: z
ack: x + 1
A
F
FIN
Passive close
seq: x + 1
ack: y + 1
A
ACK
Connection closed
Time
Time
Time
Time

State Transmission Diagram

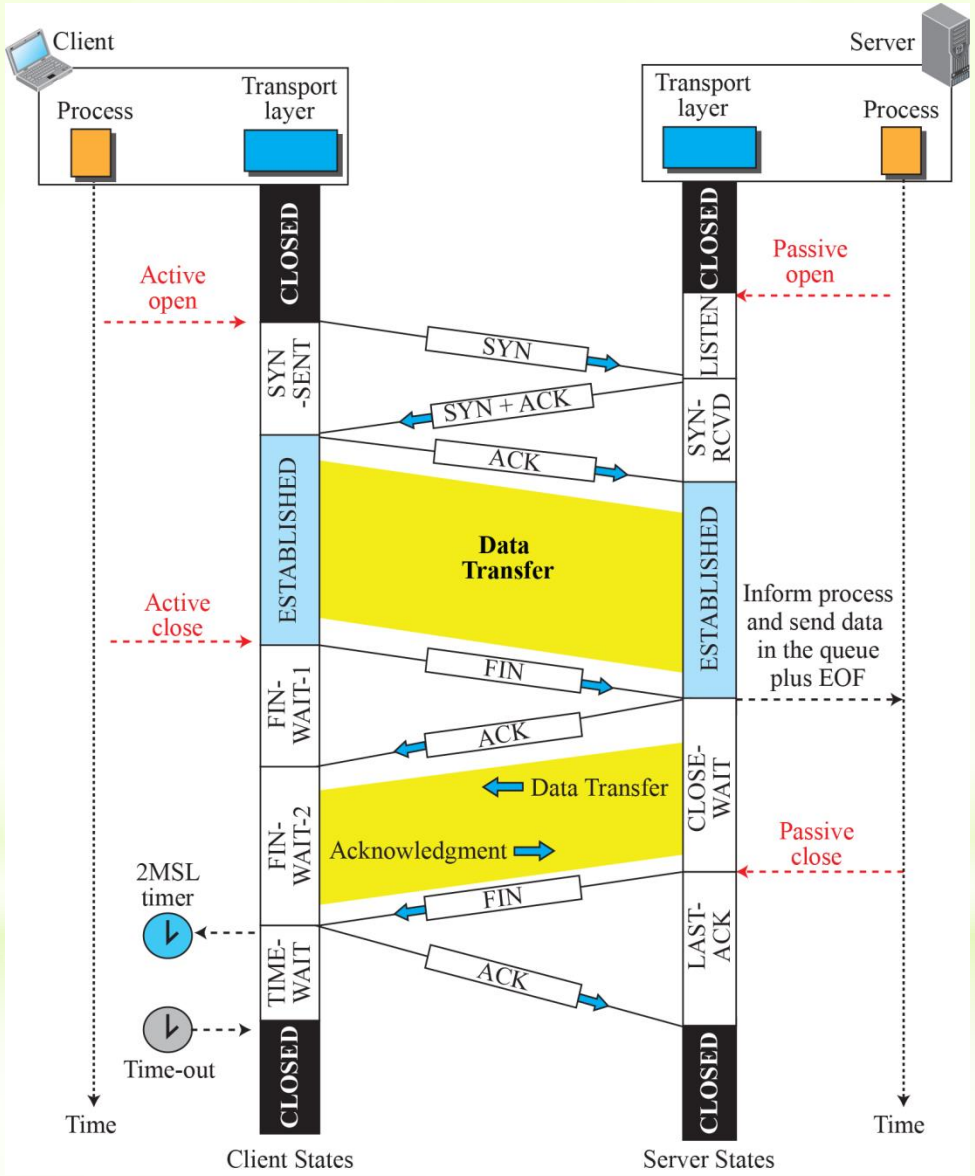
To keep track of all the different events happening during connection establishment, connection termination, and data transfer, TCP is specified as the finite state machine (FSM).

- Scenarios
  - A Half-Close Scenario

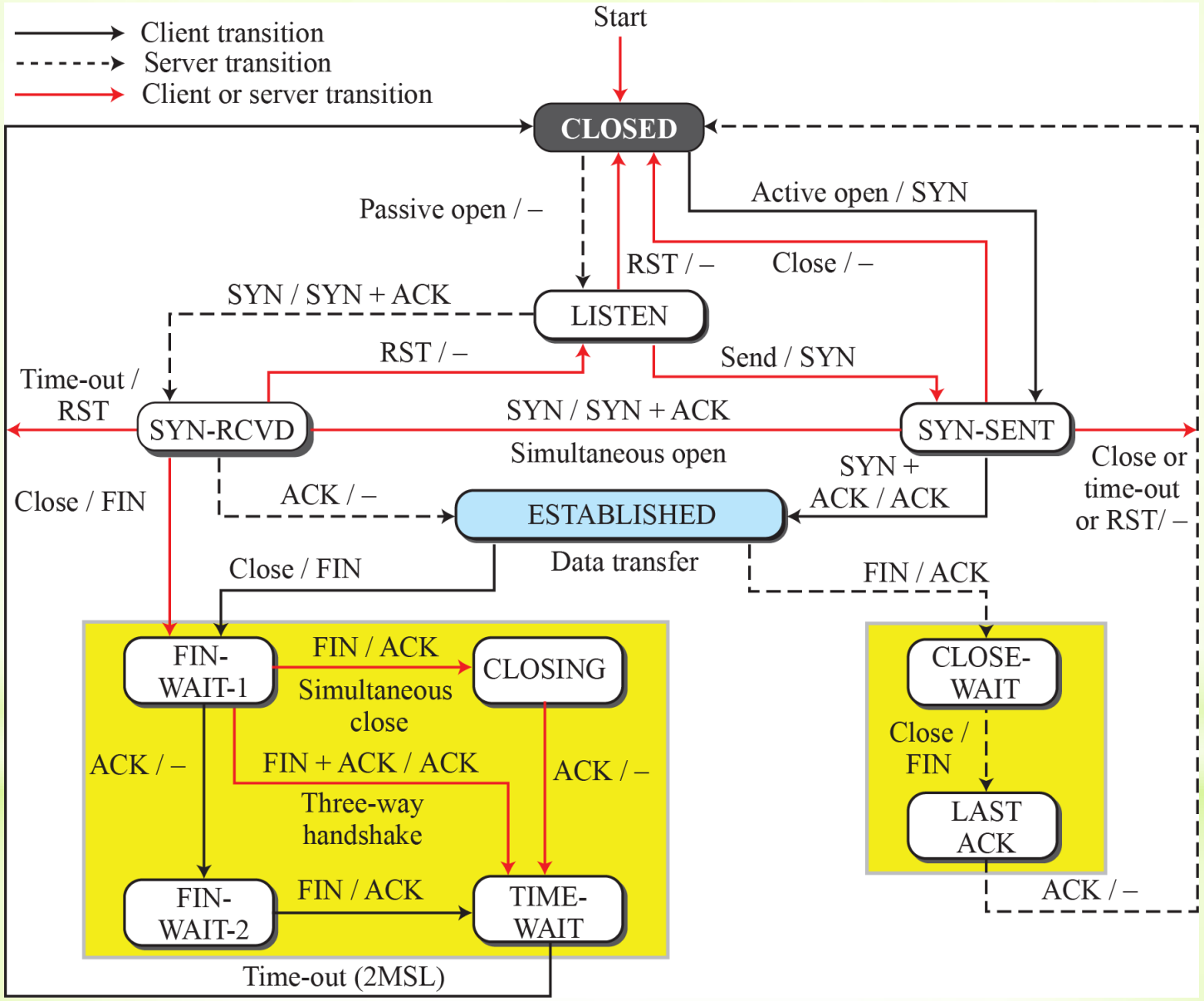
States for TCP

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

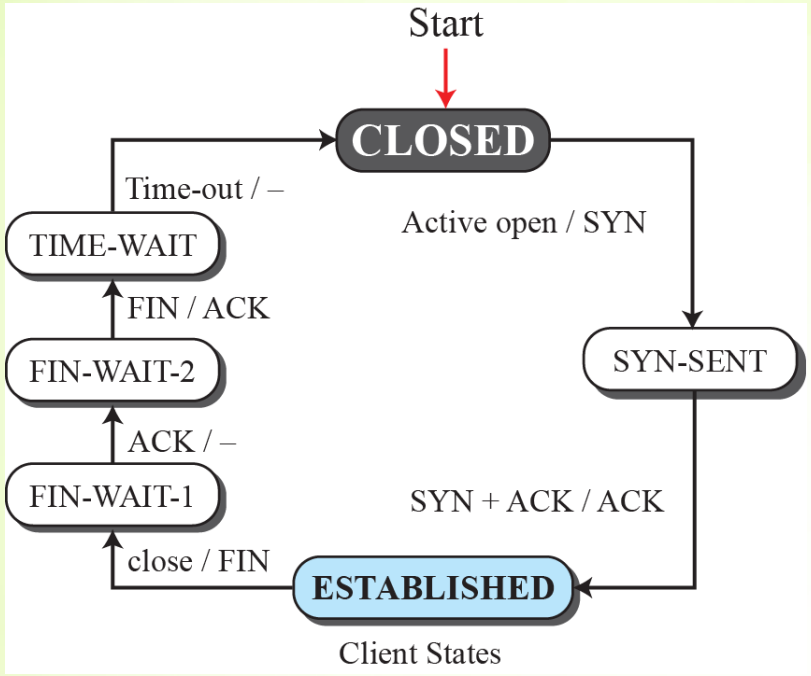
Time-line diagram for a common scenario

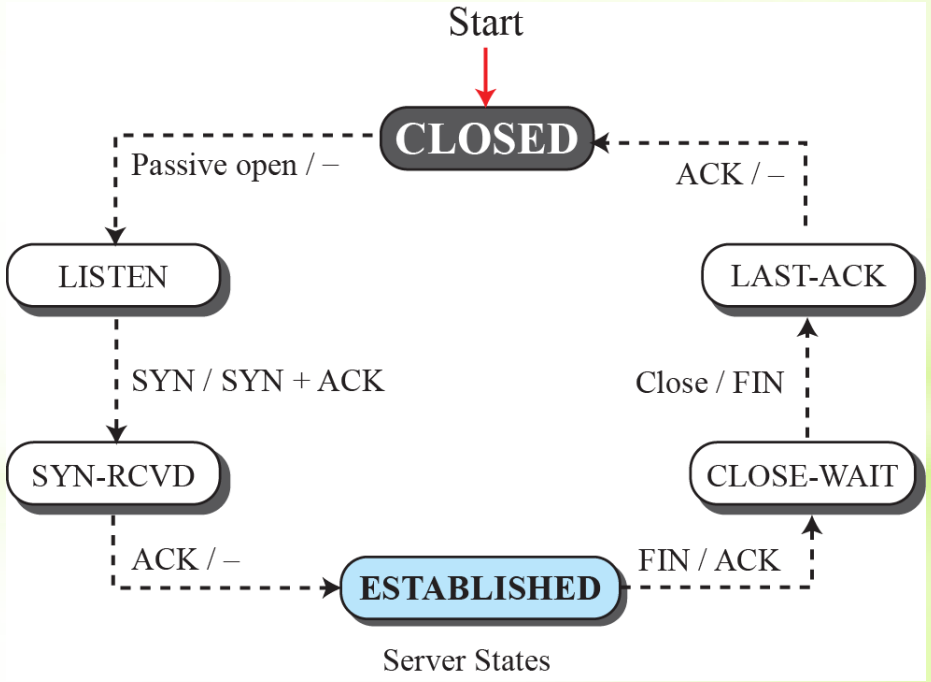


State transition diagram



Transition diagram with half-close connection termination





Windows in TCP

TCP uses two windows (send window and receive window) for each direction of data transfer, which means four windows for a bidirectional communication. To make the discussion simple, we make an unrealistic unidirectional; the bidirectional communication can be inferred using two unidirectional communications with piggybacking.

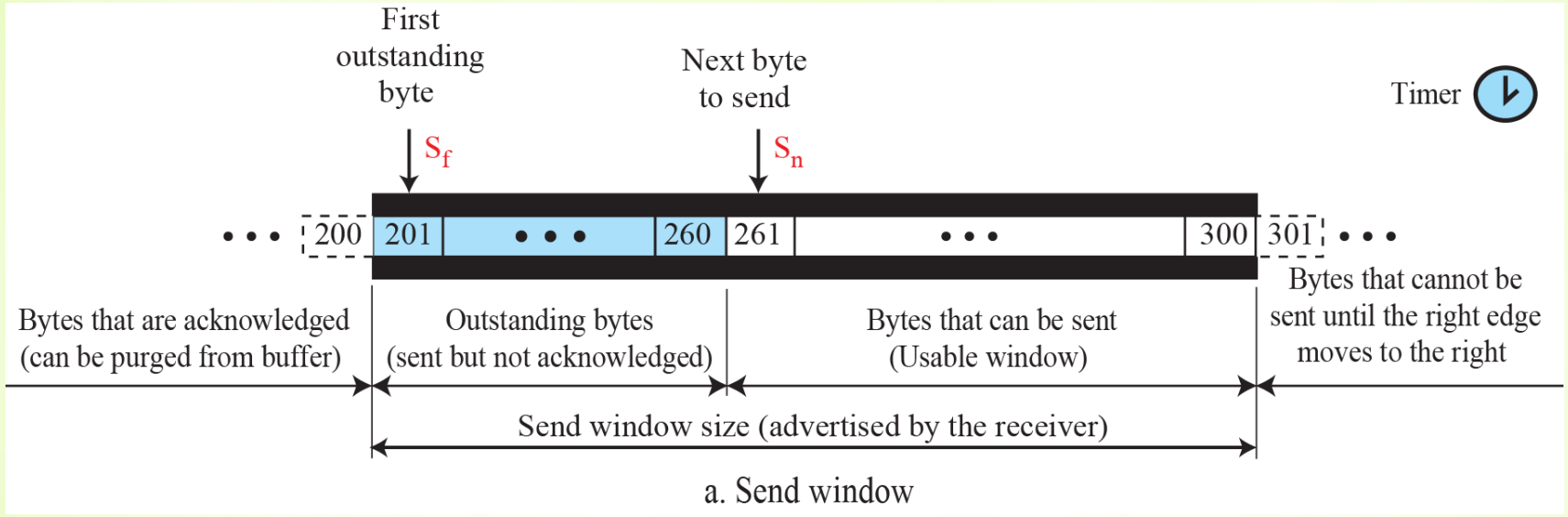
- Send Window

- Receive Window

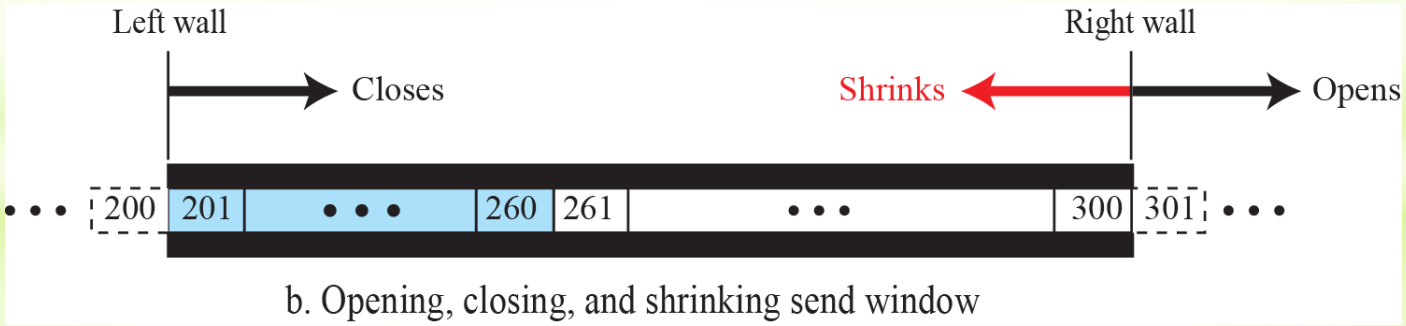
Send window in TCP

- The send window size is dictated by the receiver (flow control) and the congestion in the underlying network (congestion control).
- A send window *opens*, *closes*, or *shrinks*
- The send window in TCP is similar to the one used with the Selective-Repeat protocol, but with some differences:
  - The window size in SR is the number of packets, but the window size in TCP is the number of bytes. Although actual transmission in TCP occurs segment by segment, the variables that control the window are expressed in bytes.
  - TCP can store data received from the process and send them later
  - The theoretical Selective-Repeat protocol may use several timers for each packet sent, but as mentioned before, the TCP protocol uses only one timer.

Send window in TCP



a. Send window



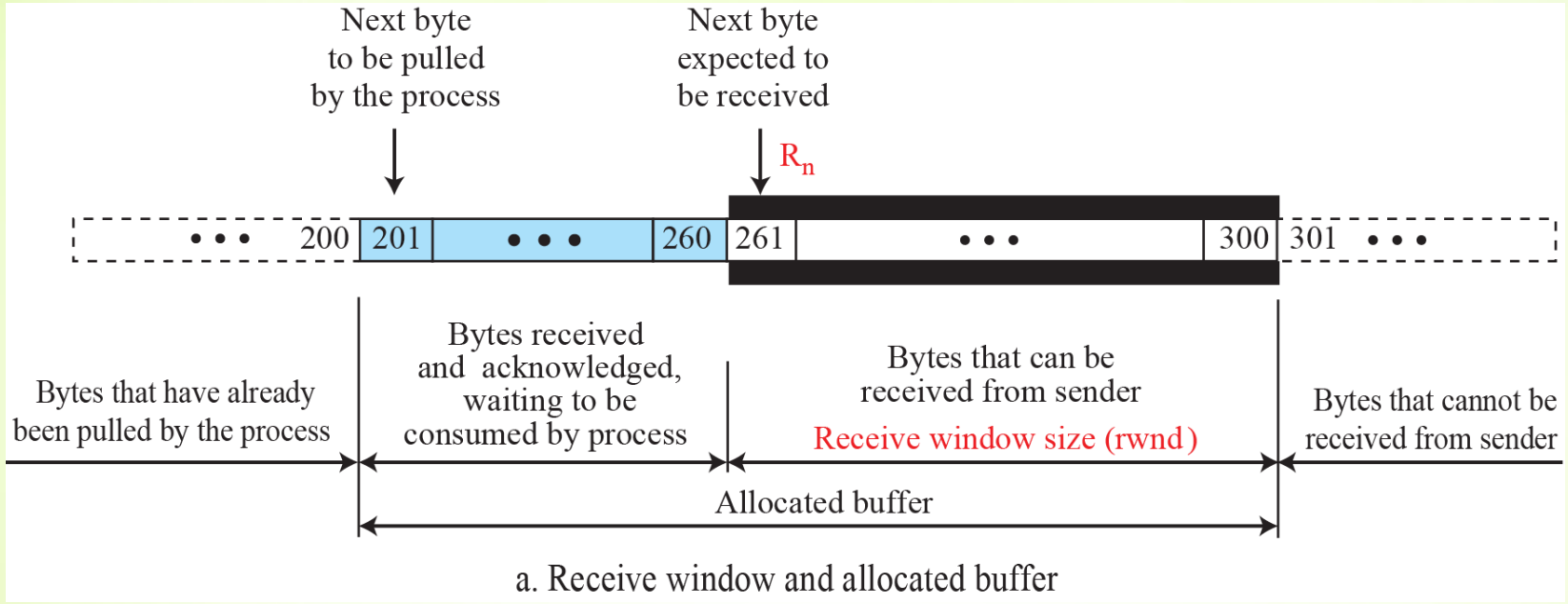
b. Opening, closing, and shrinking send window

Receive window in TCP

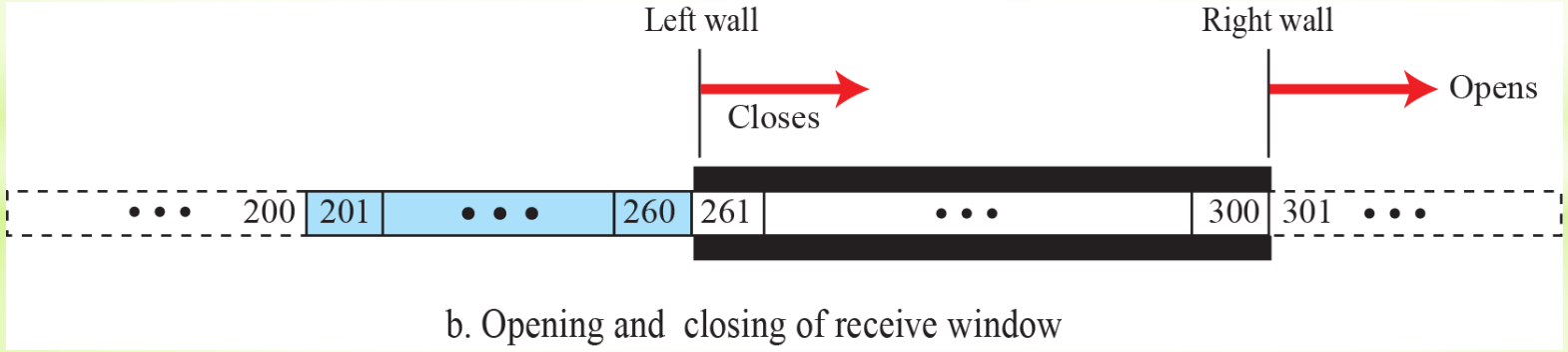
- The receive window opens and closes; in practice, the window should never shrink.
- There are two differences between the receive window in TCP and the one we used for SR.
  - The first difference is that TCP allows the receiving process to pull data at its own pace. The receive window size is then always smaller than or equal to the buffer size

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

Receive window in TCP



a. Receive window and allocated buffer



b. Opening and closing of receive window

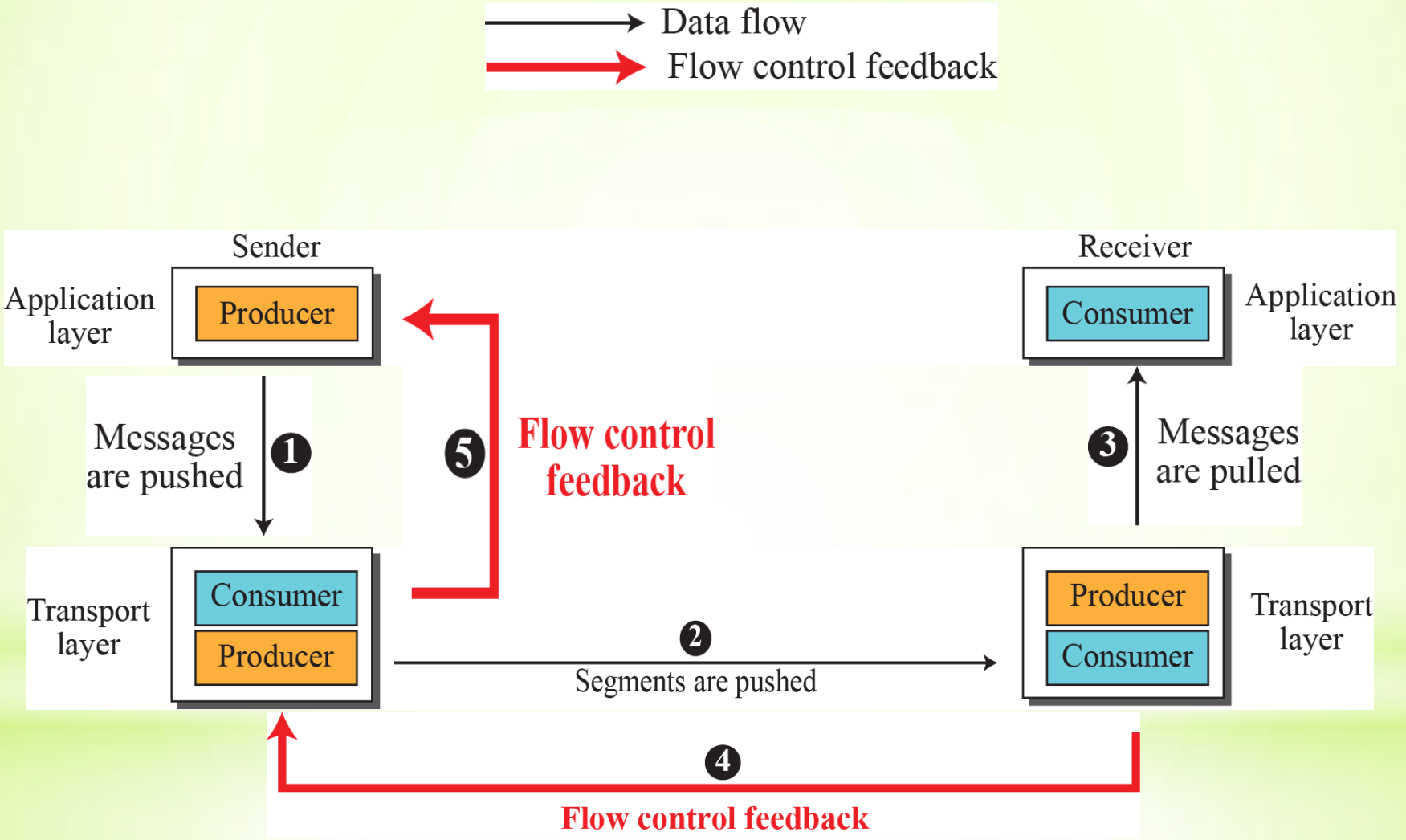
Flow Control

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.

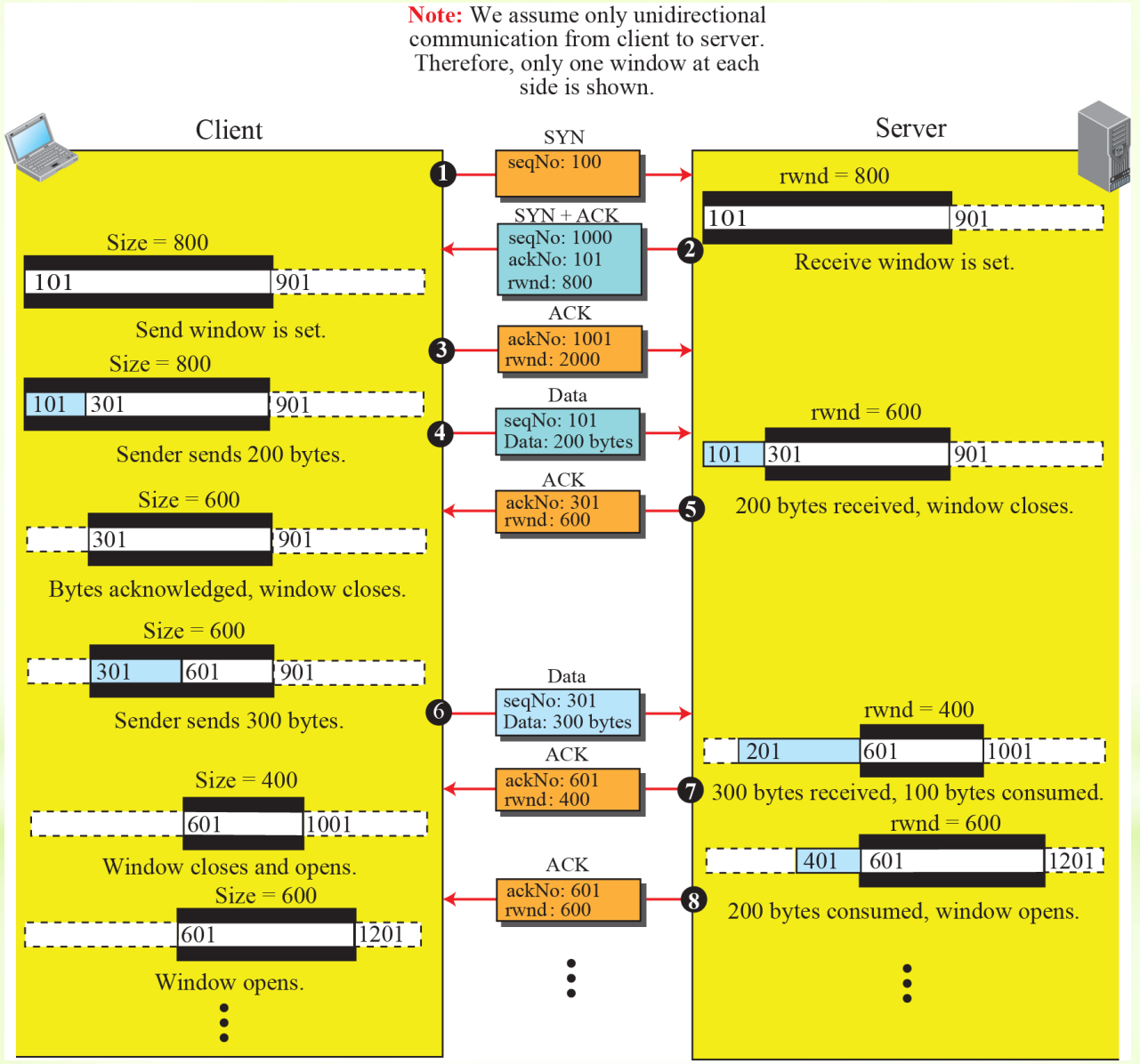
(continued)

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

Data flow and flow control feedbacks in TCP



An example of flow control



Example

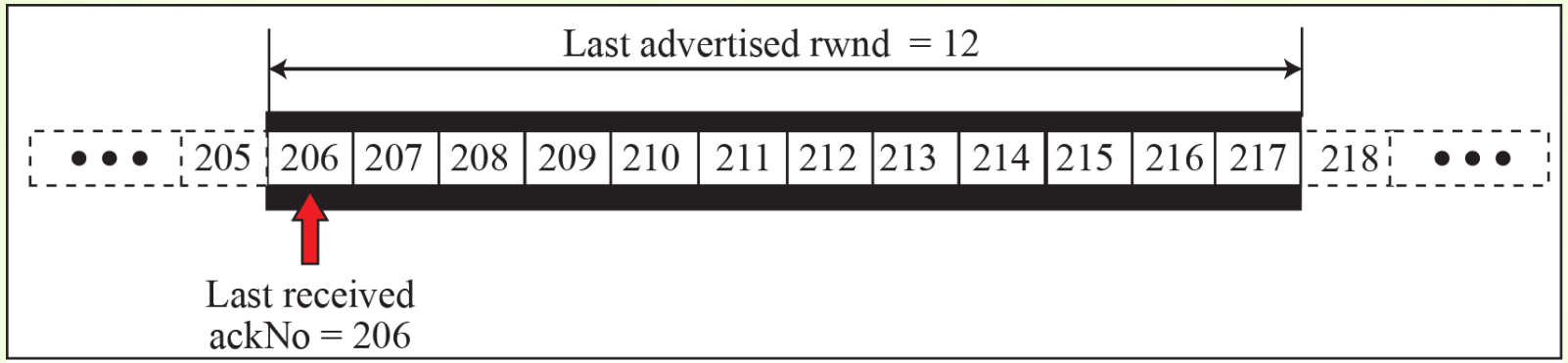
Figure shows the reason for this mandate.

Part a of the figure shows the values of the last acknowledgment and *rwnd*. Part b shows the situation in which the sender has sent bytes 206 to 214. Bytes 206 to 209 are acknowledged and purged. The new advertisement, however, defines the new value of *rwnd* as 4, in which $210 + 4 < 206 + 12$. When the send window shrinks, it creates a problem: byte 214, which has already been sent, is outside the window. The relation discussed before forces the receiver to maintain the right-hand wall of the window to be as shown in part a, because the receiver does not know which of the bytes 210 to 217 has already been sent. described above.

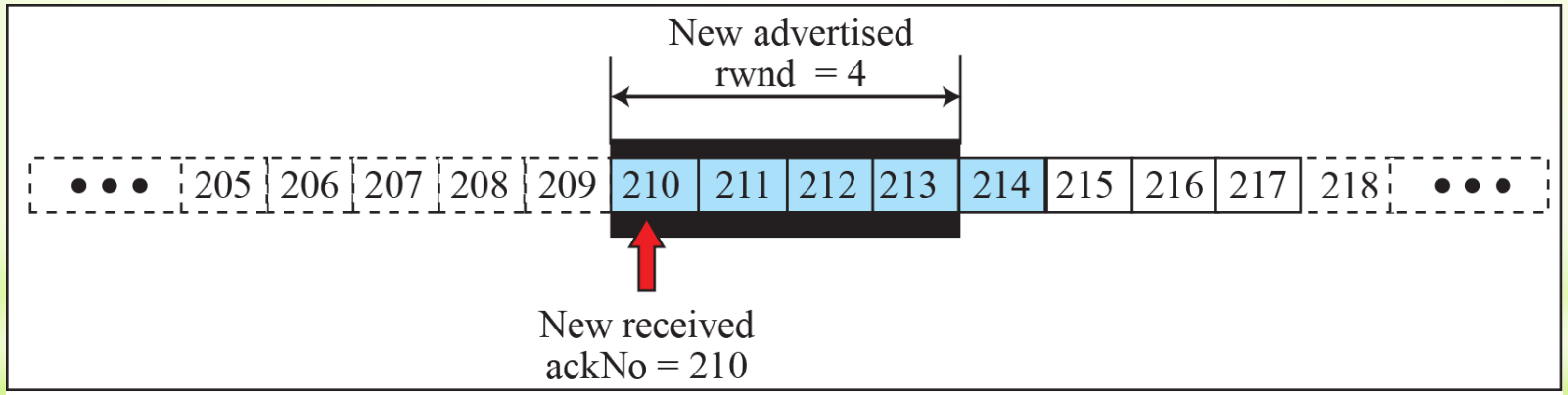
Shrinking of Windows

new ackNo + new *rwnd* ≥ last ackNo + last *rwnd*

Example



a. The window after the last advertisement



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

Window Shutdown - the receiver can temporarily shut down the window by sending a *rwnd* of 0.

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.

❖ Syndrome Created by the Sender

- TCP may create a silly window syndrome if it is serving an application program that creates data slowly, for example, 1 byte at a time.
- The result is a lot of 41-byte segments that are traveling through an internet.
- 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. **Nagle** found an elegant solution.

Nagle's algorithm is simple:

1. The sending TCP sends the first piece of data it receives from the sending application program even if it is only 1 byte.

2. 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 have accumulated to fill a maximum-size segment. At this time, the sending TCP can send the segment.

3. 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.

- If the application program is faster than the network, the segments are larger (maximum-size segments). If the application program is slower than the network, the segments are smaller (less than the maximum segment size).

❖ Syndrome Created by the Receiver

Suppose that the sending application program creates data in blocks of 1 kilobyte, but the receiving application program consumes data 1 byte at a time. Two solutions have been proposed:

- **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.
- The second solution is to delay sending the acknowledgment. This means that when a segment arrives, it is not acknowledged immediately. The receiver waits until there is a decent amount of space in its incoming buffer before acknowledging the arrived segments.
  - Delayed acknowledgment also has another advantage: it reduces traffic. The receiver does not have to acknowledge each segment.
  - However, there also is a disadvantage in that the delayed acknowledgment may result in the sender unnecessarily retransmitting the unacknowledged segments.

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.

(continued)

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

(continued)

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

❑ Checksum

- Each segment includes a checksum field, which is used to check for a corrupted segment.
- If a segment is corrupted, as detected by an invalid checksum, the segment is discarded by the destination TCP and is considered as lost.

❑ Acknowledgment

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

❖ Cumulative Acknowledgment (ACK): The receiver advertises the next byte it expects to receive, ignoring all segments received and stored out of order. This is sometimes referred to as *positive cumulative acknowledgment*, or ACK and is specified as 32-bit ACK field in the TCP header

❖ Selective Acknowledgment (SACK): A SACK reports a block of bytes that is out of order, and also a block of bytes that is duplicated, i.e., received more than once. SACK is implemented as an option at the end of the TCP header.

❑ Retransmission

- Retransmission after RTO: The sending TCP maintains one **retransmission time-out (RTO)** for each connection. When the timer matures, i.e. times out, TCP resends the segment in the front of the queue (the segment with the smallest sequence number) and restarts the timer. RTO is dynamic in TCP and is updated based on the **round-trip time (RTT)** of segments.

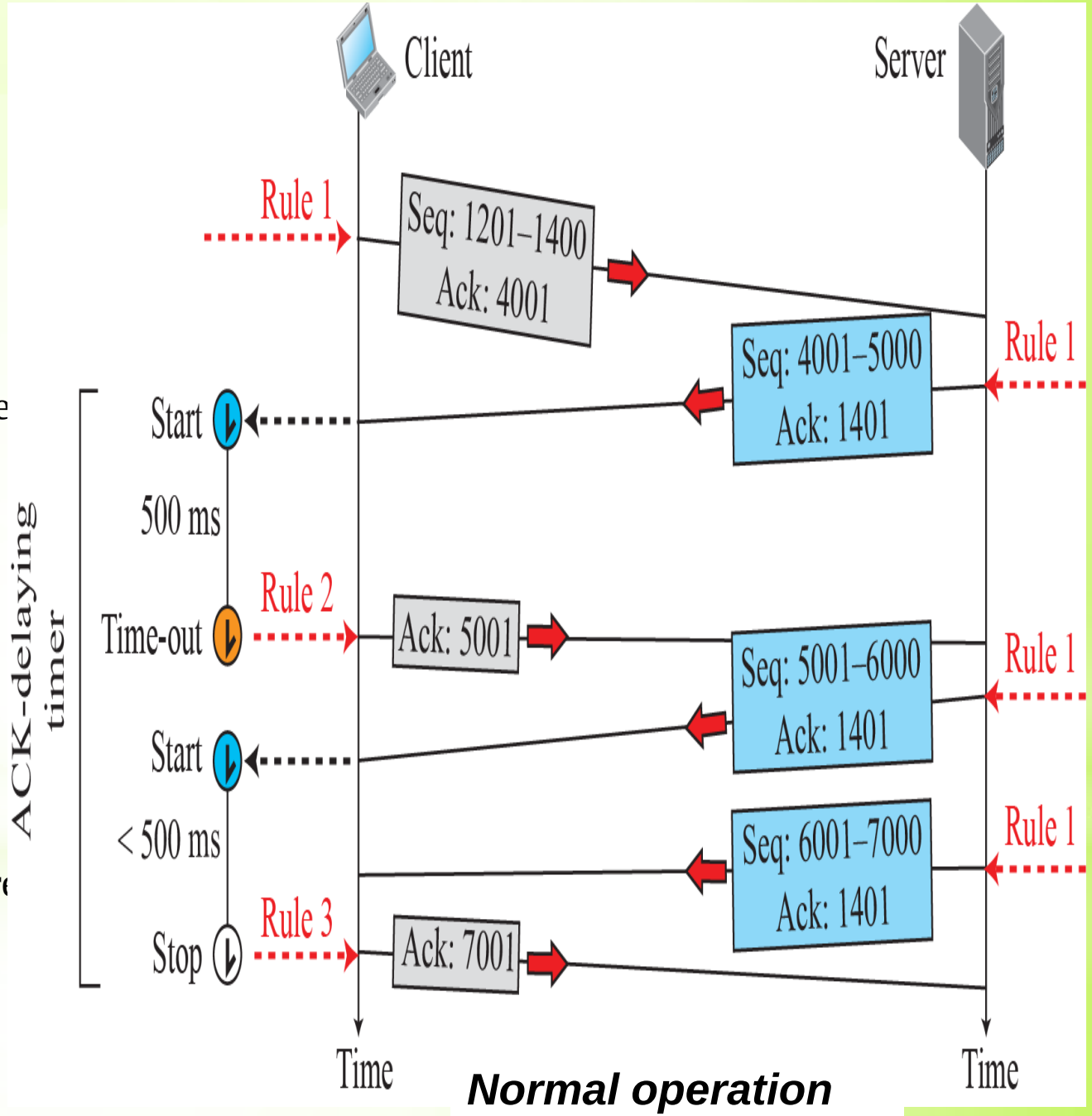
- Retransmission after Three Duplicate ACK: To expedite service throughout the Internet by allowing senders to retransmit without waiting for a time out, most implementations today follow the three duplicate ACKs rule and retransmit the missing segment immediately. This feature is called ***fast retransmission.***

❑ Out-of-Order Segments

- TCP implementations today do not discard out-of-order segments. They store them temporarily and flag them as out-of-order segments until the missing segments arrive.
- **TCP guarantees that no out-of-order data are delivered to the process.**

Generating Acknowledgments

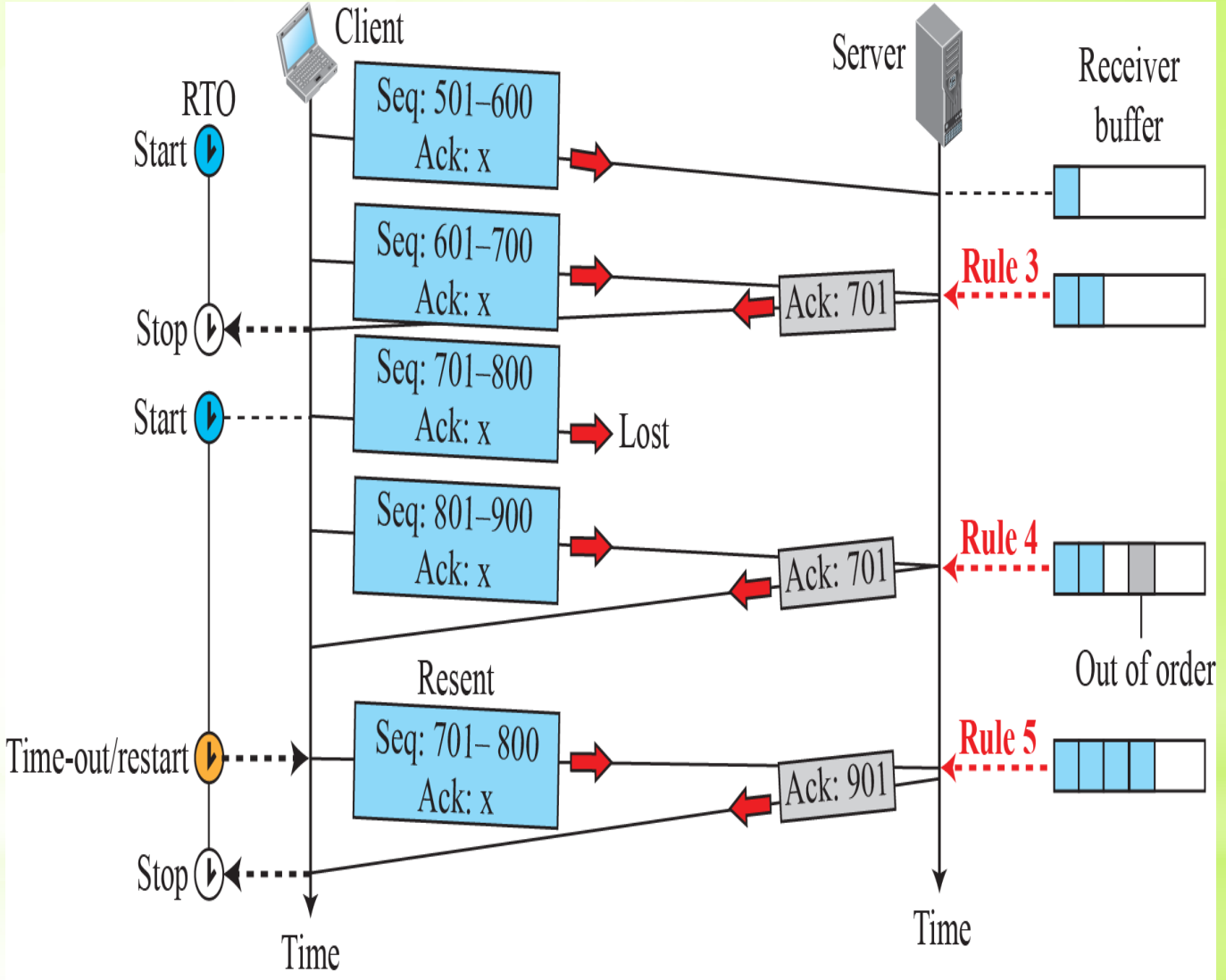
1. When end A sends a data segment to end B, it must include (piggyback) an acknowledgment that gives the next sequence number it expects to receive. This rule decreases the number of segments needed and therefore reduces traffic.
2. When the receiver has no data to send and it receives an in-order segment (with expected sequence number) and the previous segment has already been acknowledged, the receiver delays sending an ACK segment until another segment arrives or until a period of time (normally 500 ms) has passed. In other words, the receiver needs to delay sending an ACK segment if there is only one outstanding in-order segment. This rule reduces ACK segments.
3. When a segment arrives with a sequence number that is expected by the receiver, and the previous in-order segment has not been acknowledged, the receiver immediately sends an ACK segment. In other words, there should not be more than two in-order unacknowledged segments at any time. This prevents the unnecessary retransmission of segments that may create congestion in the network.



Generating Acknowledgments

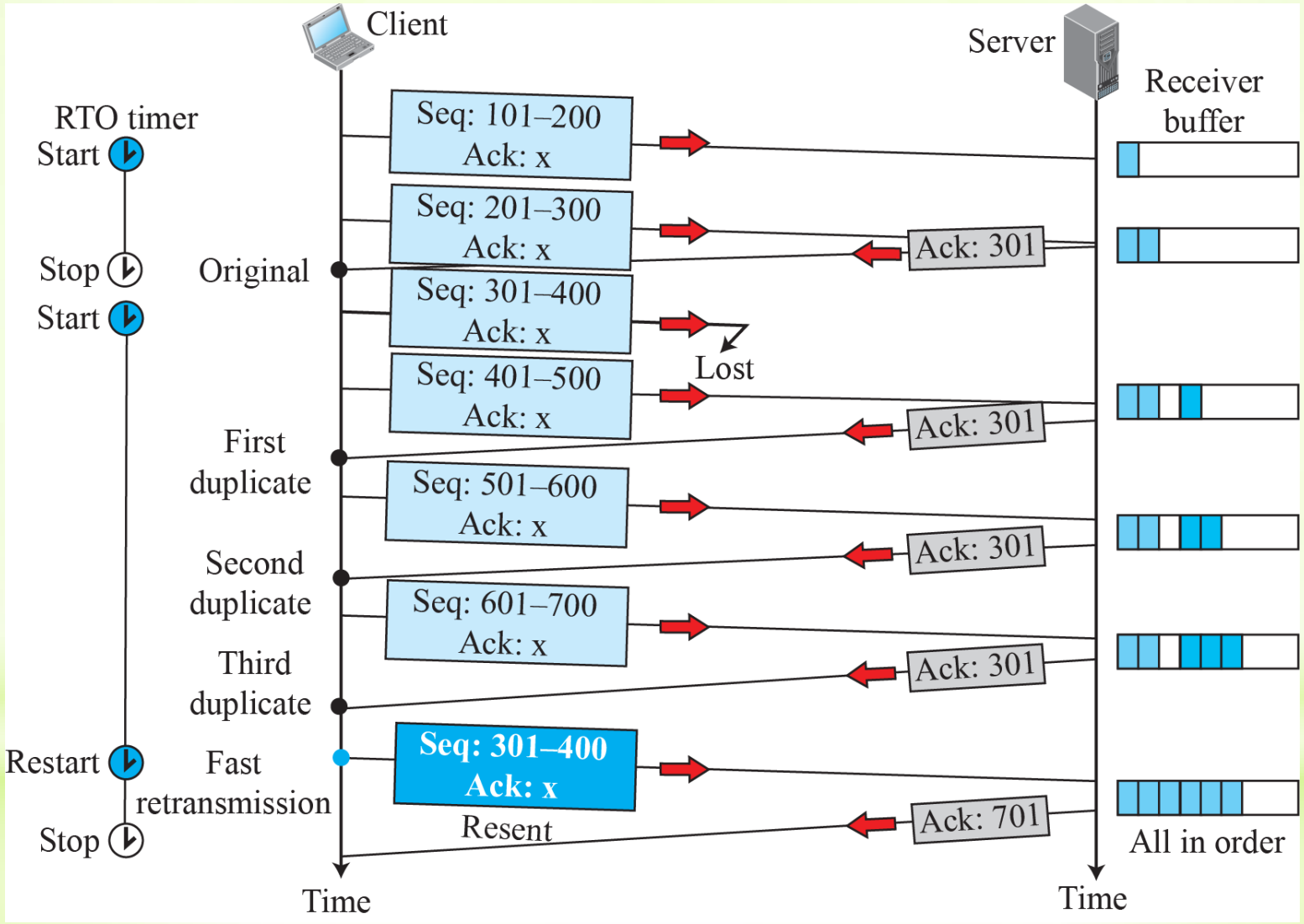
4. When a segment arrives with an out-of-order sequence number that is higher than expected, the receiver immediately sends an ACK segment announcing the sequence number of the next expected segment. This leads to the **fast retransmission** of missing segments.

5. When a missing segment arrives, the receiver sends an ACK segment to announce the next sequence number expected. This informs the receiver that segments reported missing have been received.

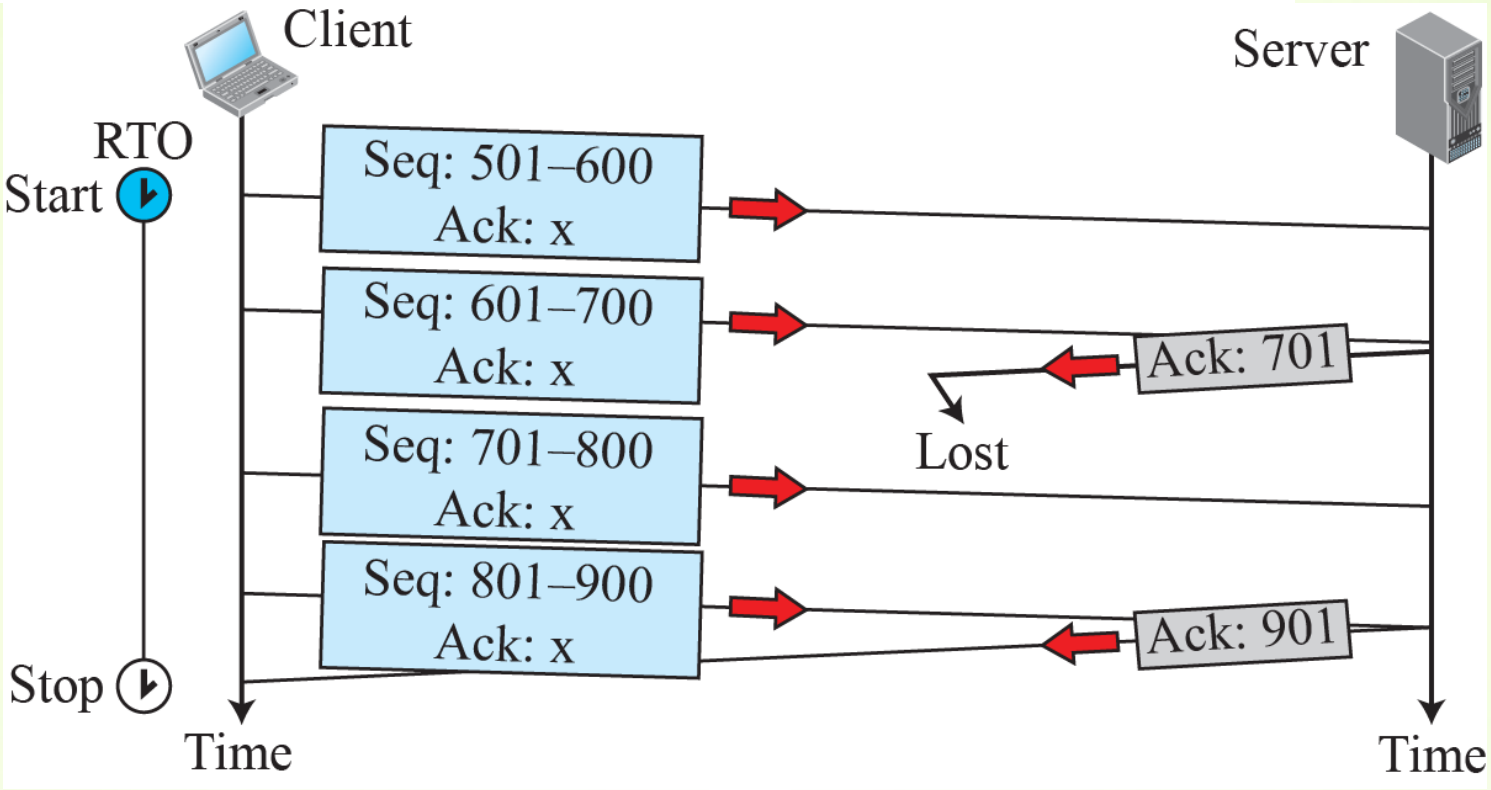


Lost segment

Fast retransmission (Rule-4)



Lost acknowledgment



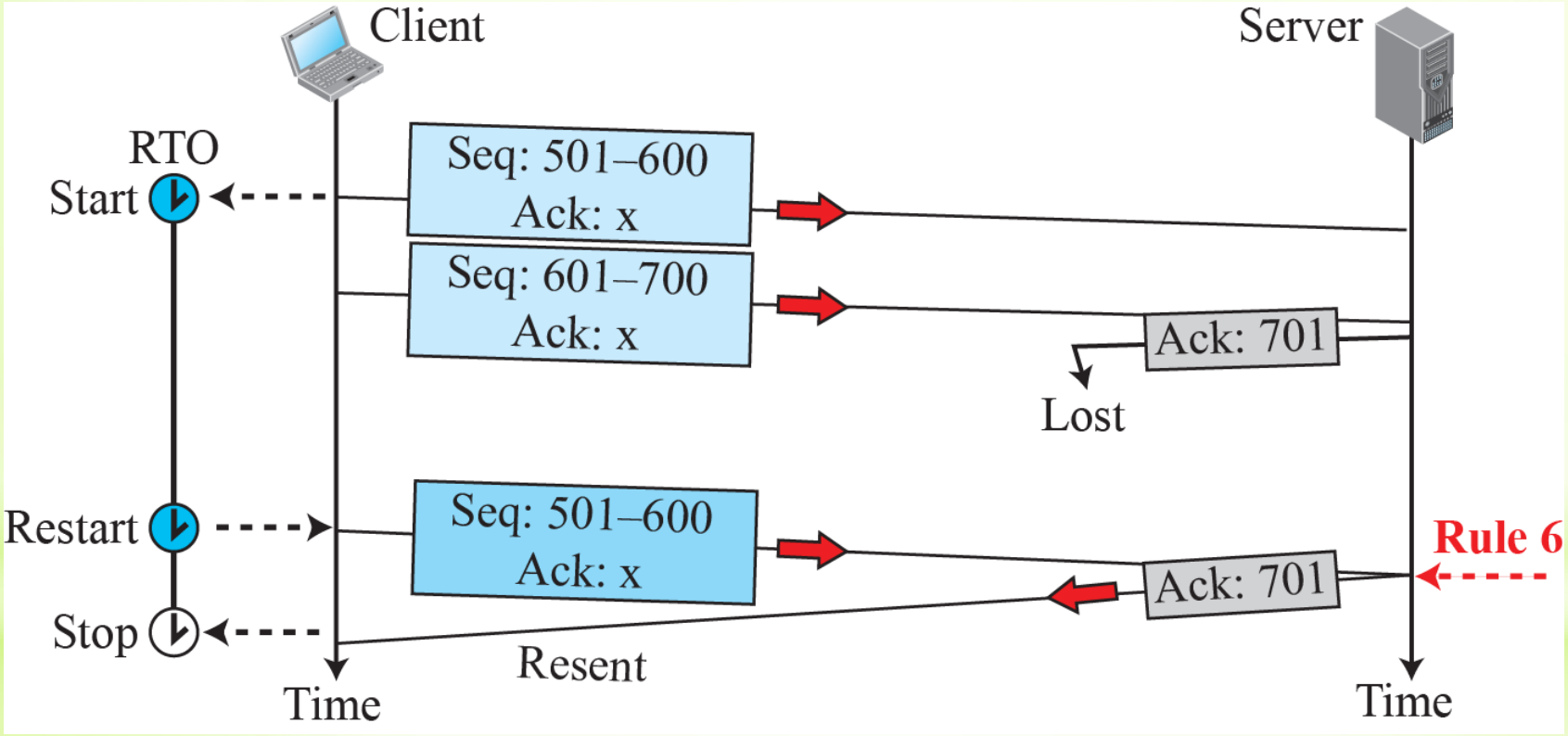
Deadlock Created by Lost Acknowledgment

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

Eg: ack(wsize=0)->sender shutdown Window :: ack(wsize!=0) :: Ack Lost :: Sender still in shutdown mode

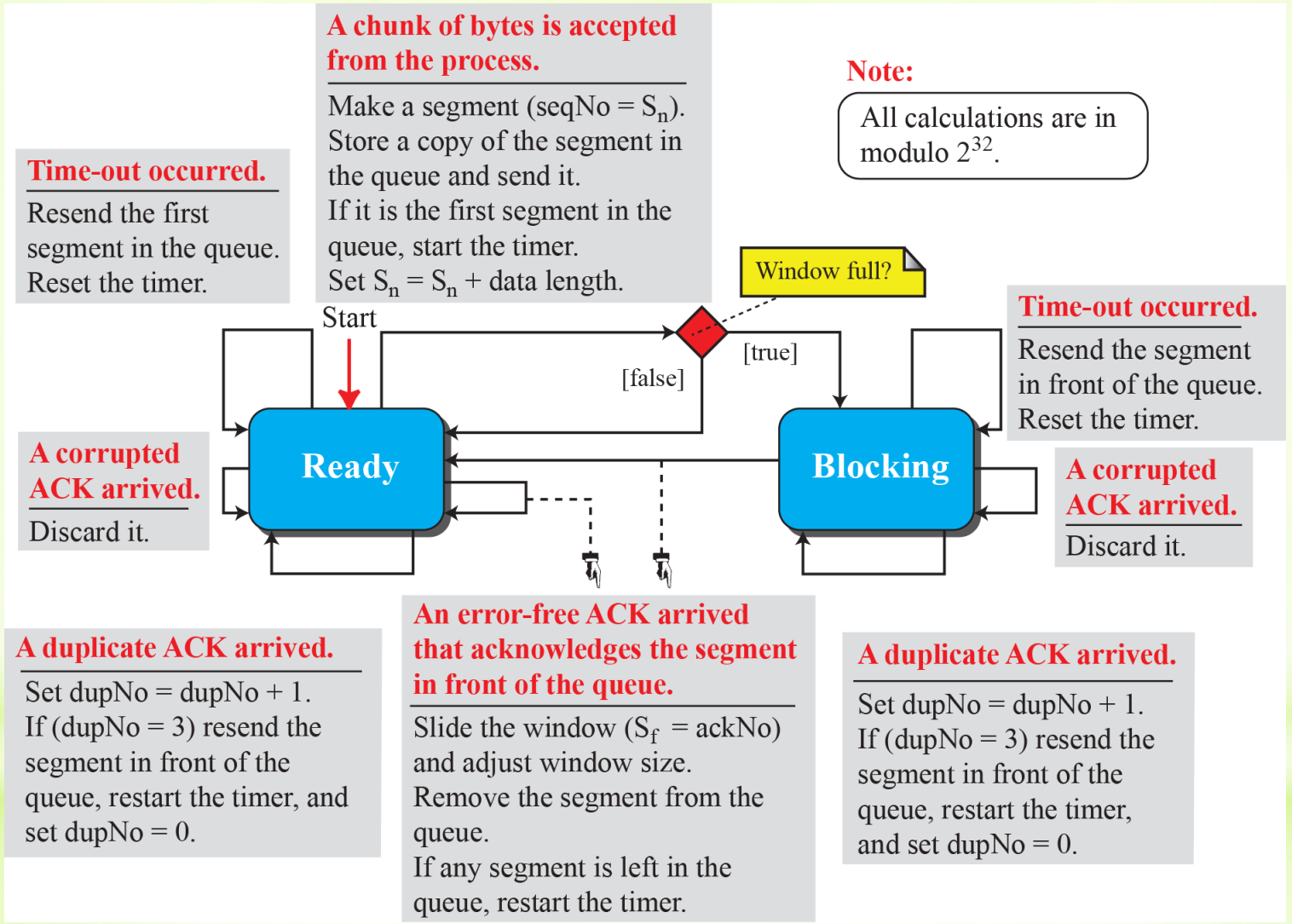
Generating Acknowledgments

6. If a duplicate segment arrives, the receiver discards the segment, but immediately sends an acknowledgment indicating the next in-order segment expected. This solves some problems when an ACK segment itself is lost.

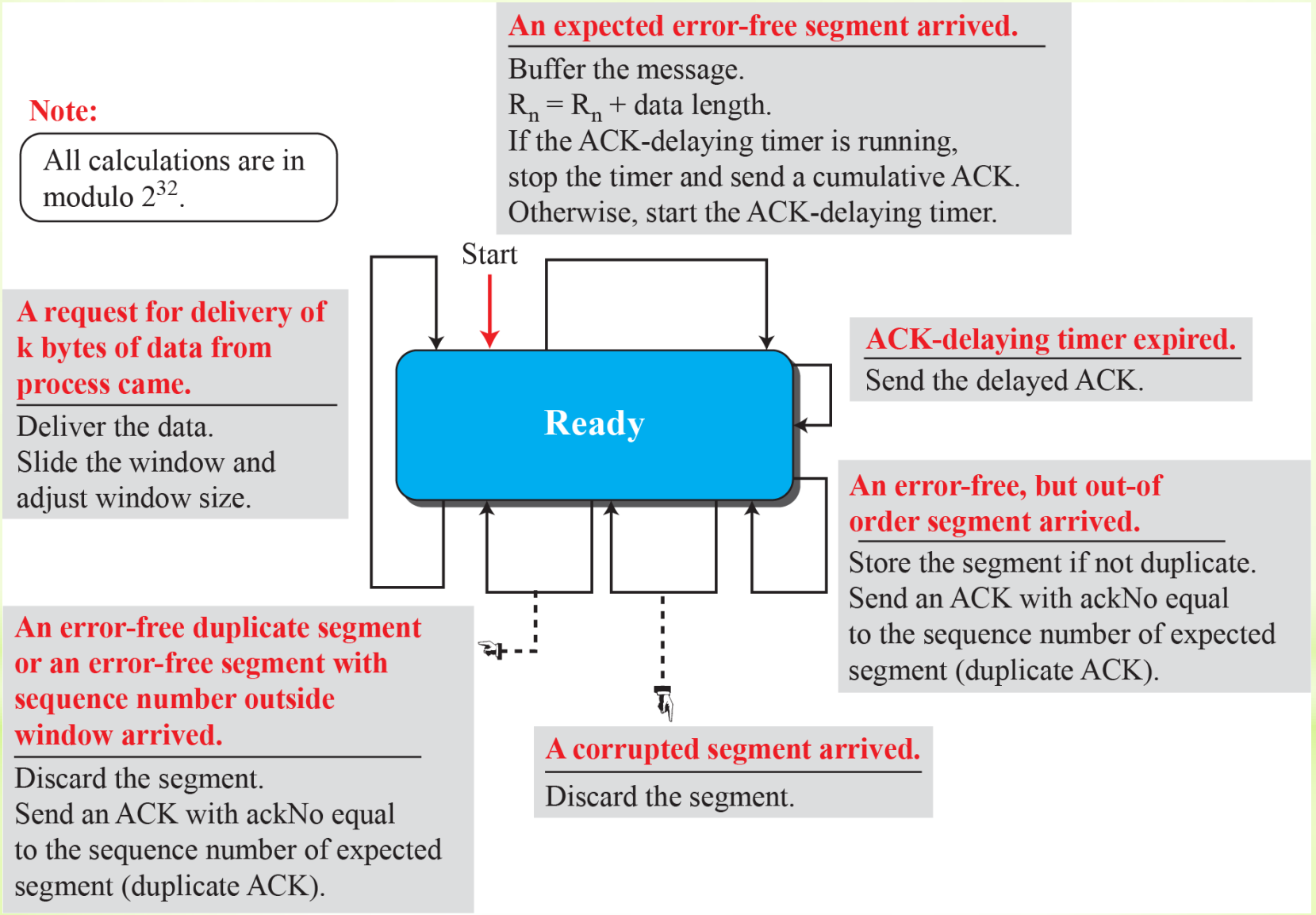


Lost acknowledgment corrected by resending a segment

Simplified FSM for the TCP sender side



Simplified FSM for the TCP receiver side



TCP Congestion Control

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

- Congestion Window & Receiver window
- Congestion Detection
  - Time out
  - Three Selective Acknowledgements
- Congestion Policies
  - Slow Start: Exponential Increase
  - Congestion Avoidance: Additive Increase
  - Fast Retransmission/Fast Recovery

(continued)

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

❑ Congestion Window & Receiver window

TCP is an end-to-end protocol that uses the service of IP. The congestion in the router is in the IP territory and should be taken care of by IP. IP is a simple protocol with no congestion control. TCP, itself, needs to be responsible for this problem. TCP cannot ignore the congestion in the network; it cannot aggressively send segments to the network. The result of such aggressiveness would hurt the TCP itself

TCP cannot be very conservative, either, sending a small number of segments in each time interval, because this means not utilizing the available bandwidth of the network. TCP needs to define policies that accelerate the data transmission when there is no congestion and decelerate the transmission when congestion is detected.

To control the number of segments to transmit, TCP uses another variable called a *congestion window, cwnd*, whose size is controlled by the congestion situation in the network (as we will explain shortly). The *cwnd* variable and the *rwnd* variable together define the size of the send window in TCP.

Actual window size = minimum (*rwnd*, *cwnd*)

❑ Congestion Detection

The TCP sender uses the occurrence of two events as signs of congestion in the network: time-out and receiving three duplicate ACKs.

- **Time out :** If a TCP sender does not receive an ACK for a segment or a group of segments before the time-out occurs, it assumes that the corresponding segment or segments are lost and the loss is due to congestion.

- **Three Selective Acknowledgements:** Recall that when a TCP receiver sends a duplicate ACK, it is the sign that a segment has been delayed, but sending three duplicate ACKs is the sign of a missing segment, which can be due to congestion in the network. When a receiver sends three duplicate ACKs, it means that one segment is missing, but three segments have been received. (slightly congested)

Maximum segment size (MSS)

- The MSS is a value negotiated during the connection establishment, using an option of the same name

- Each segment is of the same size and carries MSS bytes

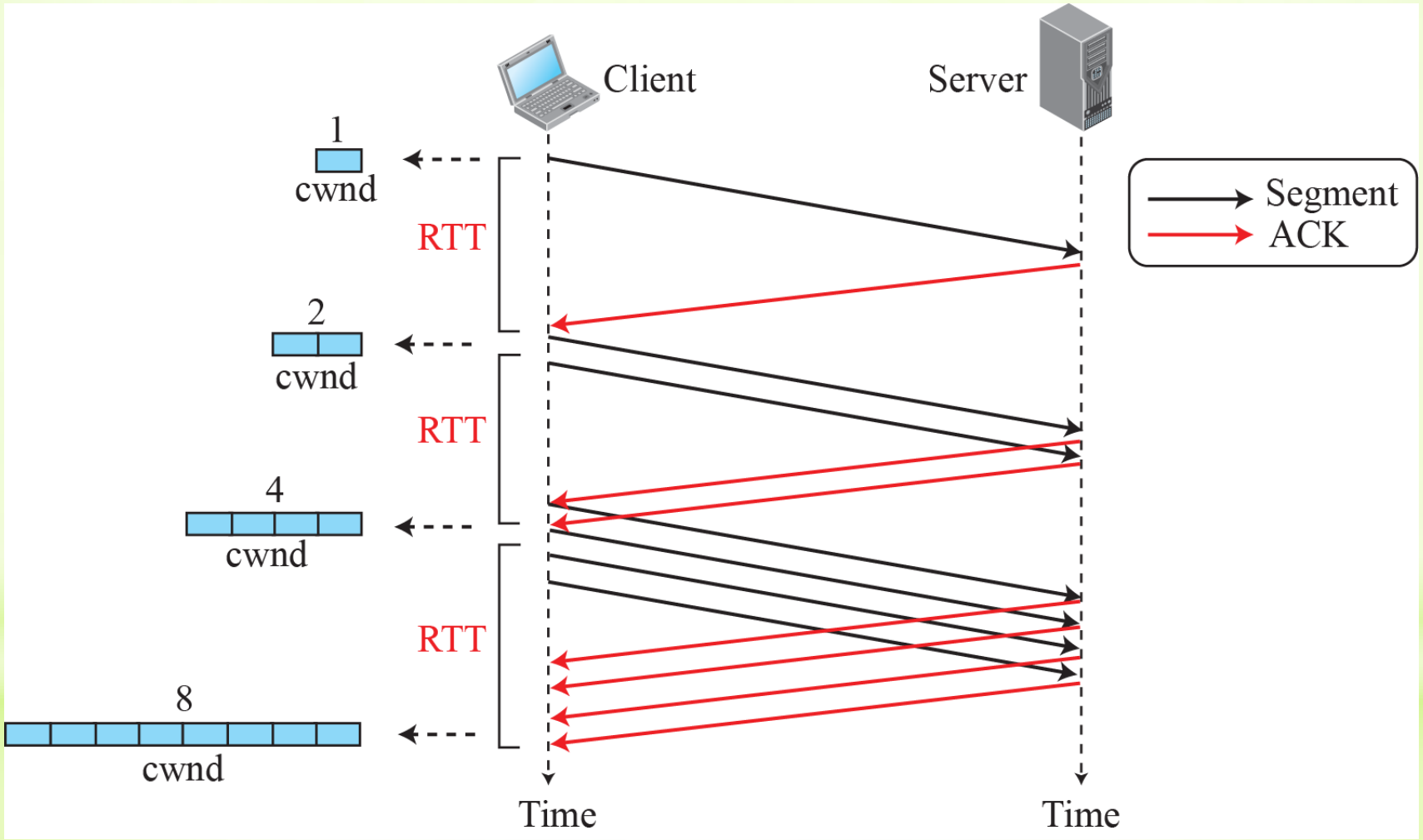
❑ Congestion Policies

❖ Slow Start: Exponential Increase

- The **slow-start algorithm** is based on the idea that the size of the congestion window (*cwnd*) starts with one maximum segment size (MSS), but it increases one MSS each time an acknowledgment arrives. i.e. window size grows exponentially.

- There must be a threshold to stop this phase. The sender keeps track of a variable named *ssthresh* (slow-start threshold). When the size of the window in bytes reaches this threshold, slow start stops and the next phase starts.

Slow start, exponential increase



❑ Congestion Policies

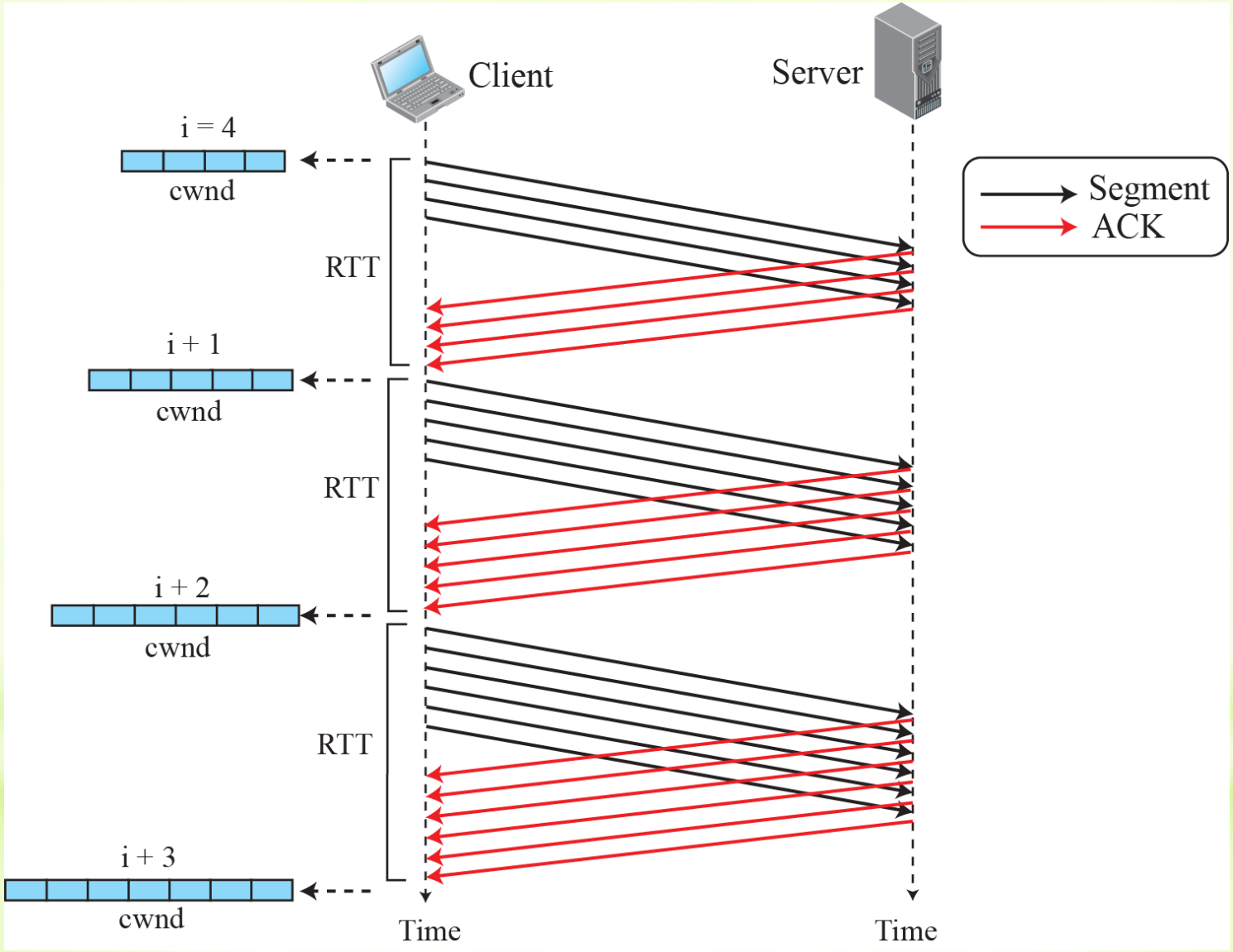
❖ Congestion Avoidance: Additive Increase

- When the size of the congestion window reaches the slow-start threshold, the slow-start phase stops and the additive phase begins. In this algorithm, each time the whole "window" of segments is acknowledged, the size of the congestion window is increased by one.
- **In the congestion-avoidance algorithm, the size of the congestion window increases additively until congestion is detected.**

❖ Fast Retransmission/Fast Recovery

- The **fast-recovery** algorithm is optional in TCP. The old version of TCP did not use it, but the new versions try to use it.

Congestion avoidance, additive increase



Three versions of TCP with different congestion policies

- Taho TCP
- Reno TCP
- NewReno TCP

Taho TCP

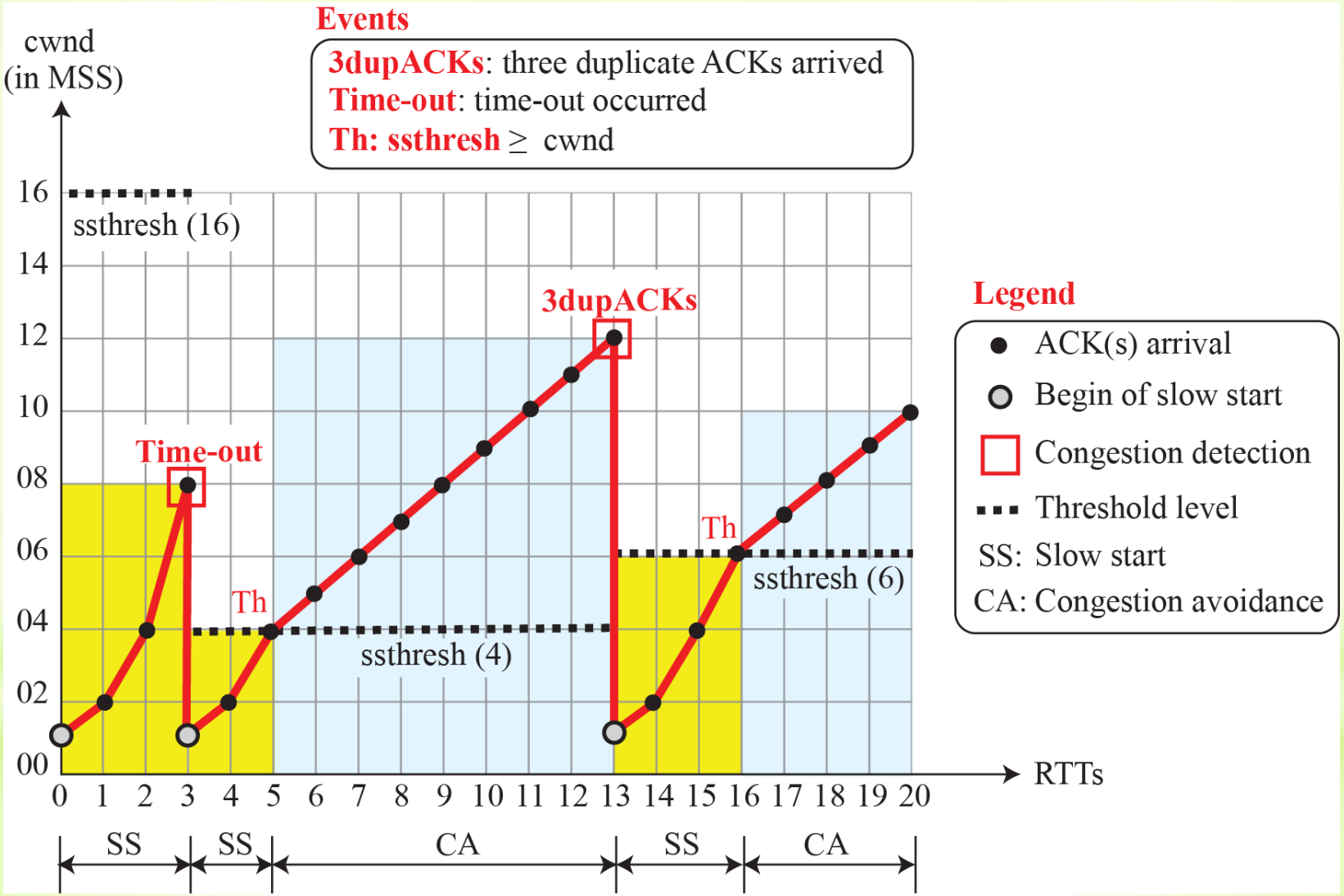
- The early TCP, known as *Taho TCP*, used only two different algorithms in their congestion policy: *slow start* and *congestion avoidance*

- Taho TCP treats the two signs used for congestion detection, time-out and three duplicate ACKs, in the same way.

- When the connection is established, TCP starts the slow-start algorithm and sets the *ssthresh* variable to a pre-agreed value (normally a multiple of MSS) and the *cwnd* to 1 MSS. Then it continues to congestion avoidance phase.

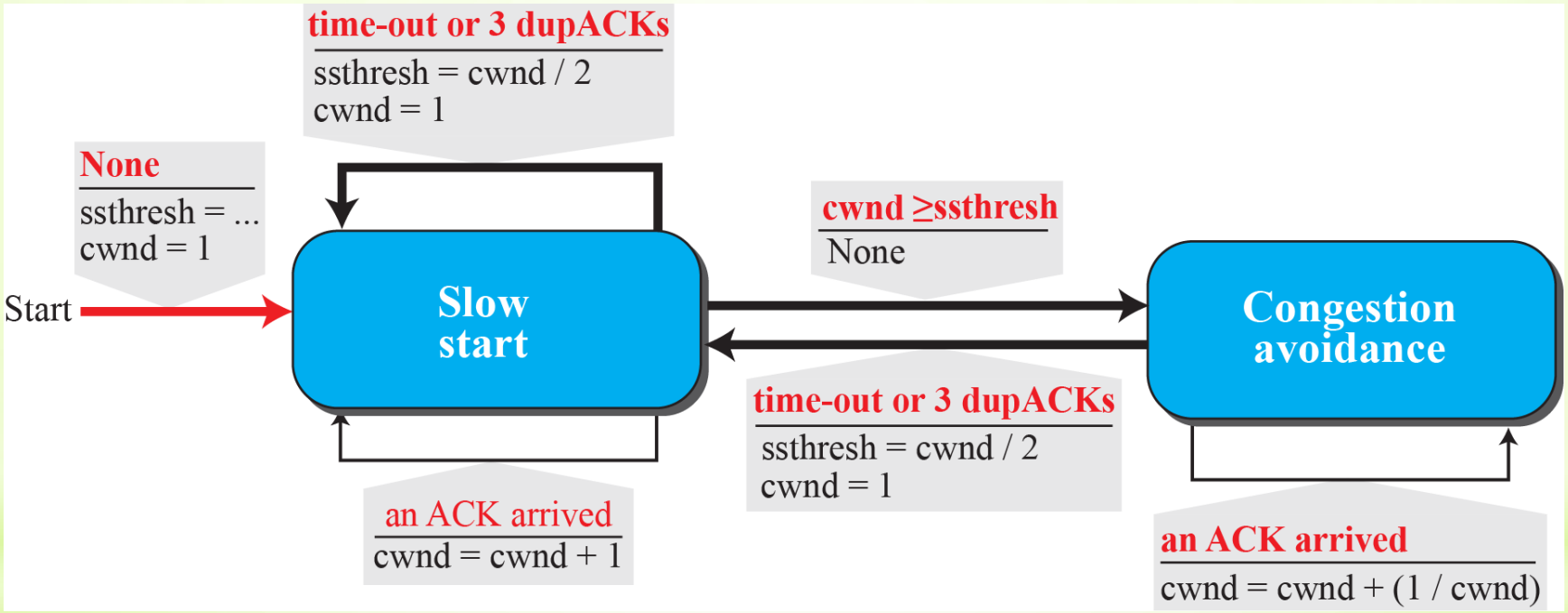
- If congestion is detected (occurrence of time-out or arrival of three duplicate ACKs), TCP immediately interrupts this aggressive growth and restarts a new slow start algorithm by **limiting the threshold to half of the current *cwnd*** and **resetting the congestion window to 1**.

Example of Taho TCP



Taho TCP

FSM for Taho TCP



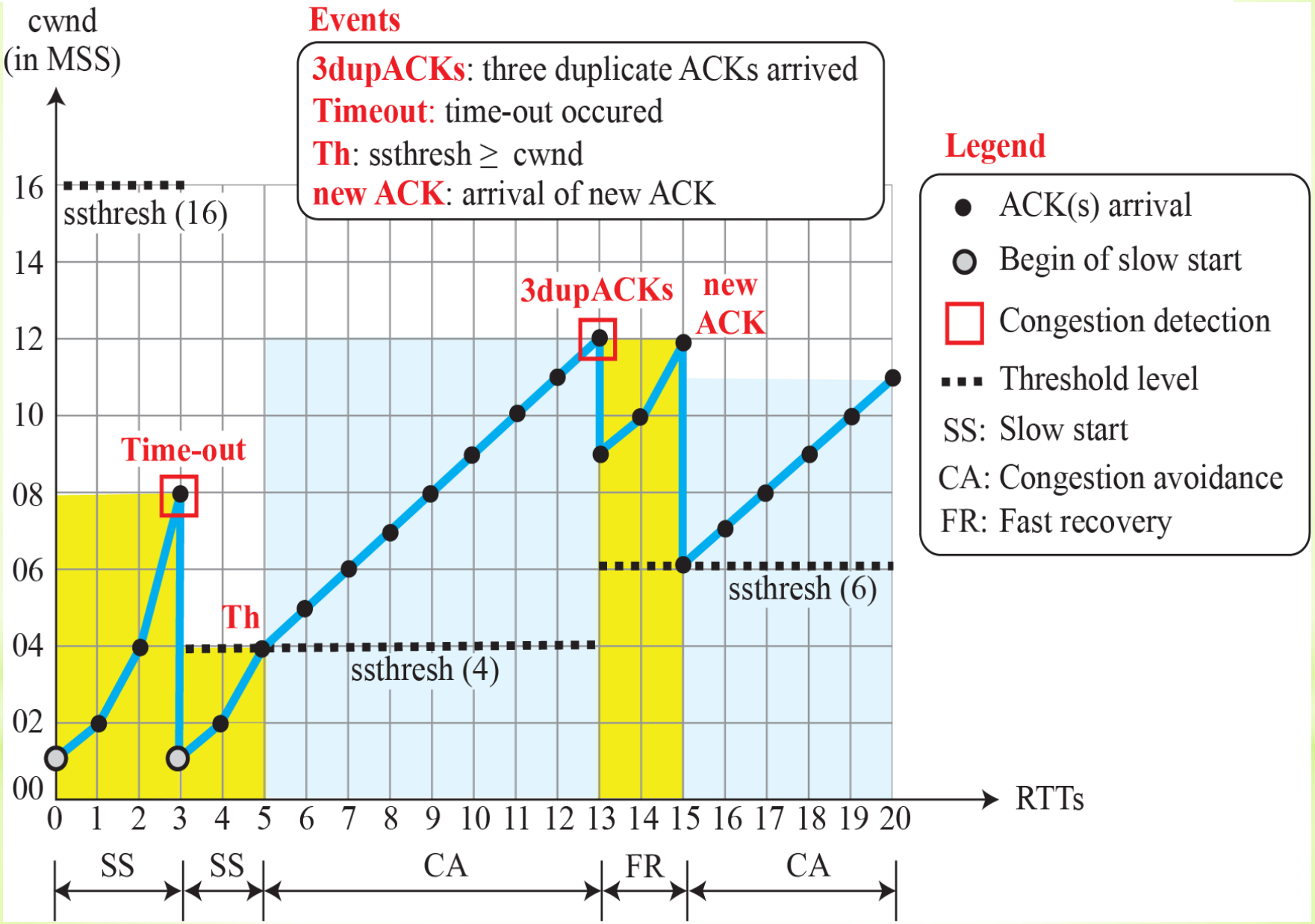
Reno TCP

- A newer version of TCP, called *Reno TCP*, added a new state to the congestion-control FSM, called the fast-recovery state.
- This version treated the two signals of congestion, time-out and the arrival of three duplicate ACKs, differently.
- In this version, **if a time-out occurs, TCP moves to the slow-start state** (or starts a new round if it is already in this state)
- **If three duplicate ACKs arrive, TCP moves to the fast-recovery state and remains there as long as more duplicate ACKs arrive**. The fast-recovery state is a state somewhere between the slow-start and the congestion-avoidance states.

- In the fast-recovery state, it behaves like the slow start, in which the ***cwnd* grows exponentially, but the *cwnd* starts with the value of *ssthresh* plus 3 MSS (instead of 1)**.

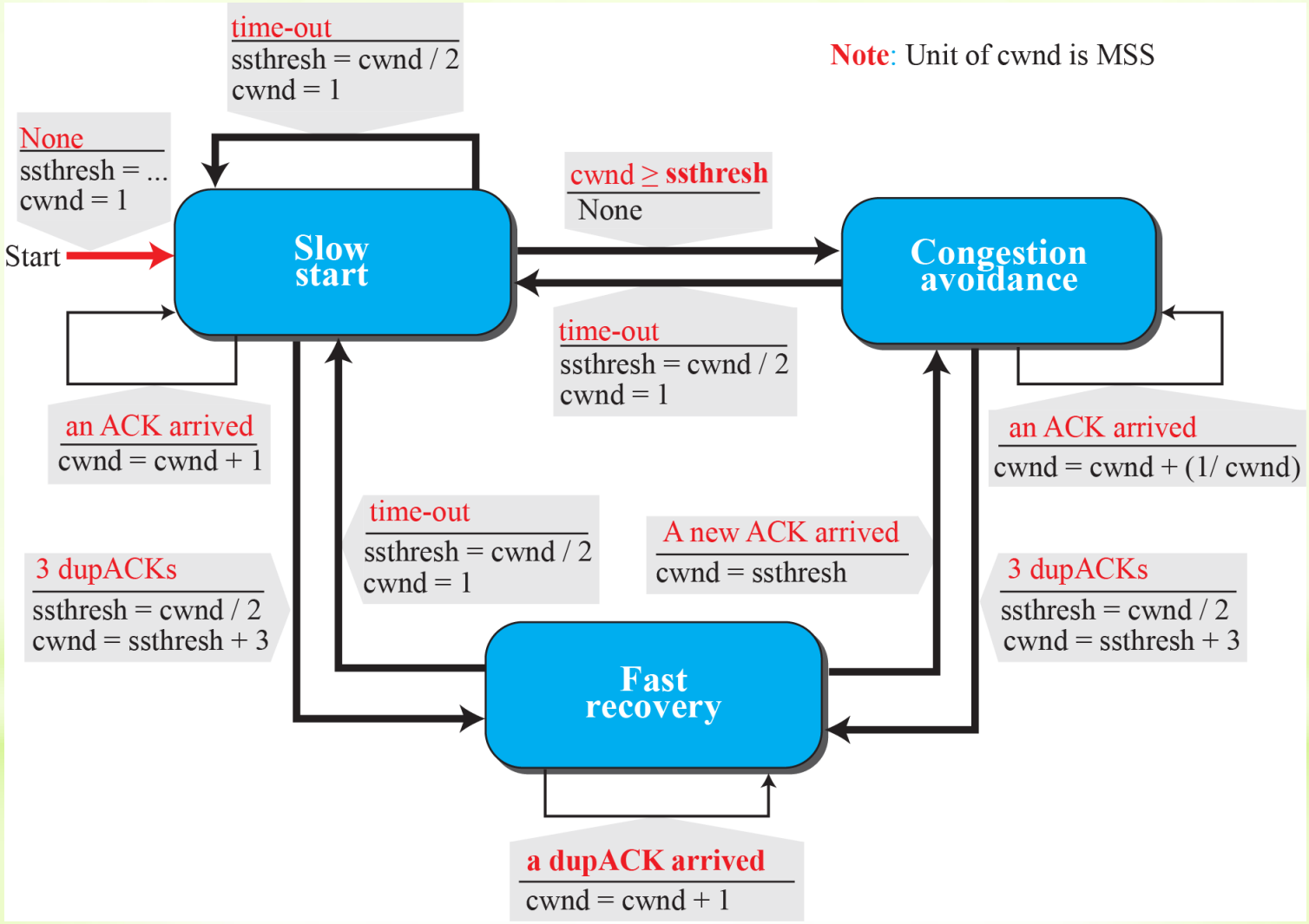
- When TCP enters the fast-recovery state, three major events may occur.
  - If duplicate ACKs continue to arrive, TCP stays in this state, but the *cwnd* grows exponentially.
  - If a time-out occurs, TCP assumes that there is real congestion in the network and moves to the slow-start state.
  - If a new (nonduplicate) ACK arrives, TCP moves to the congestion-avoidance state, but deflates the size of the *cwnd* to the *ssthresh* value, as though the three duplicate ACKs have not occurred, and transition is from the slow-start state to the congestion-avoidance state.

Example of a Reno TCP



Reno TCP

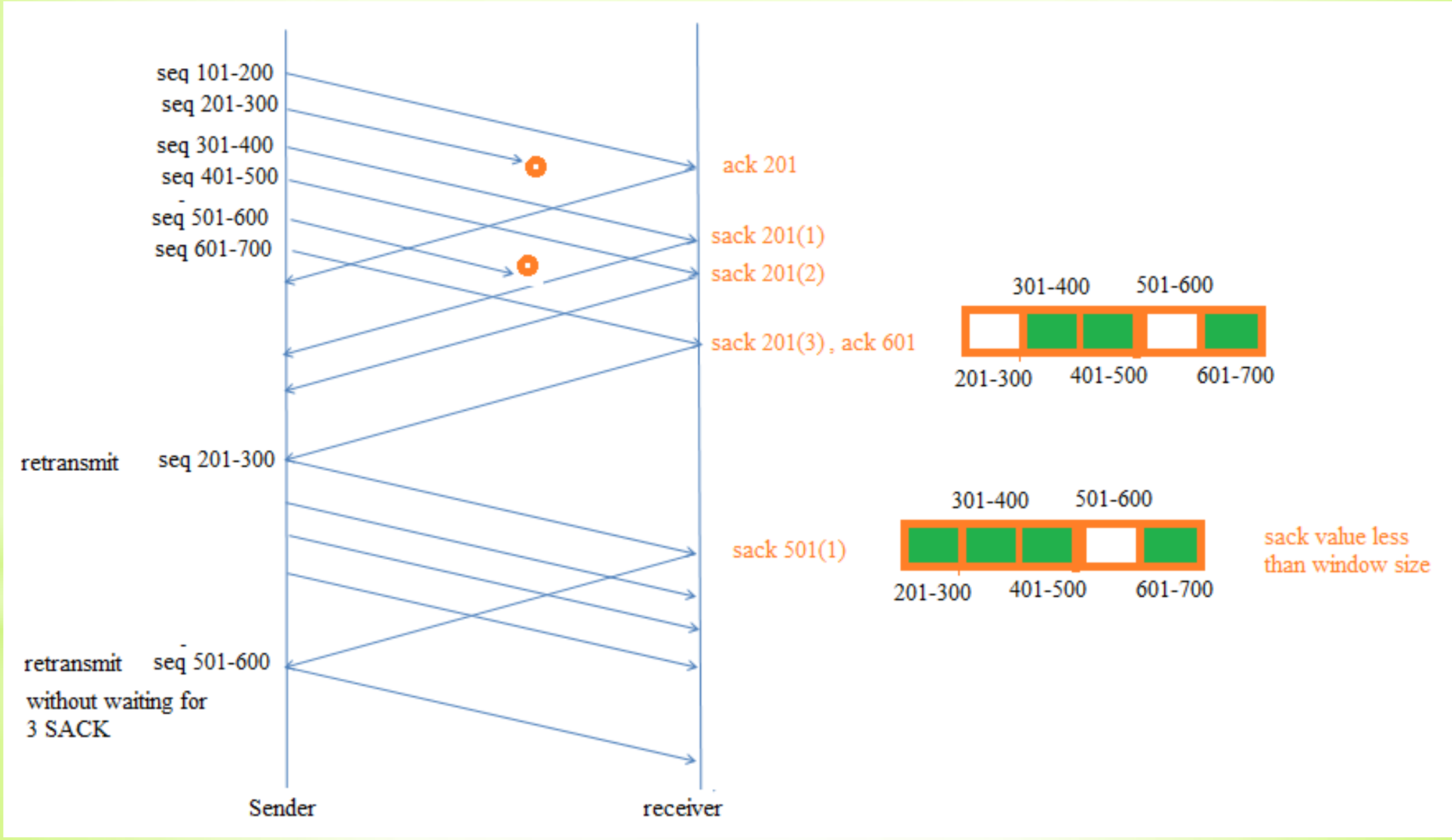
FSM for Reno TCP



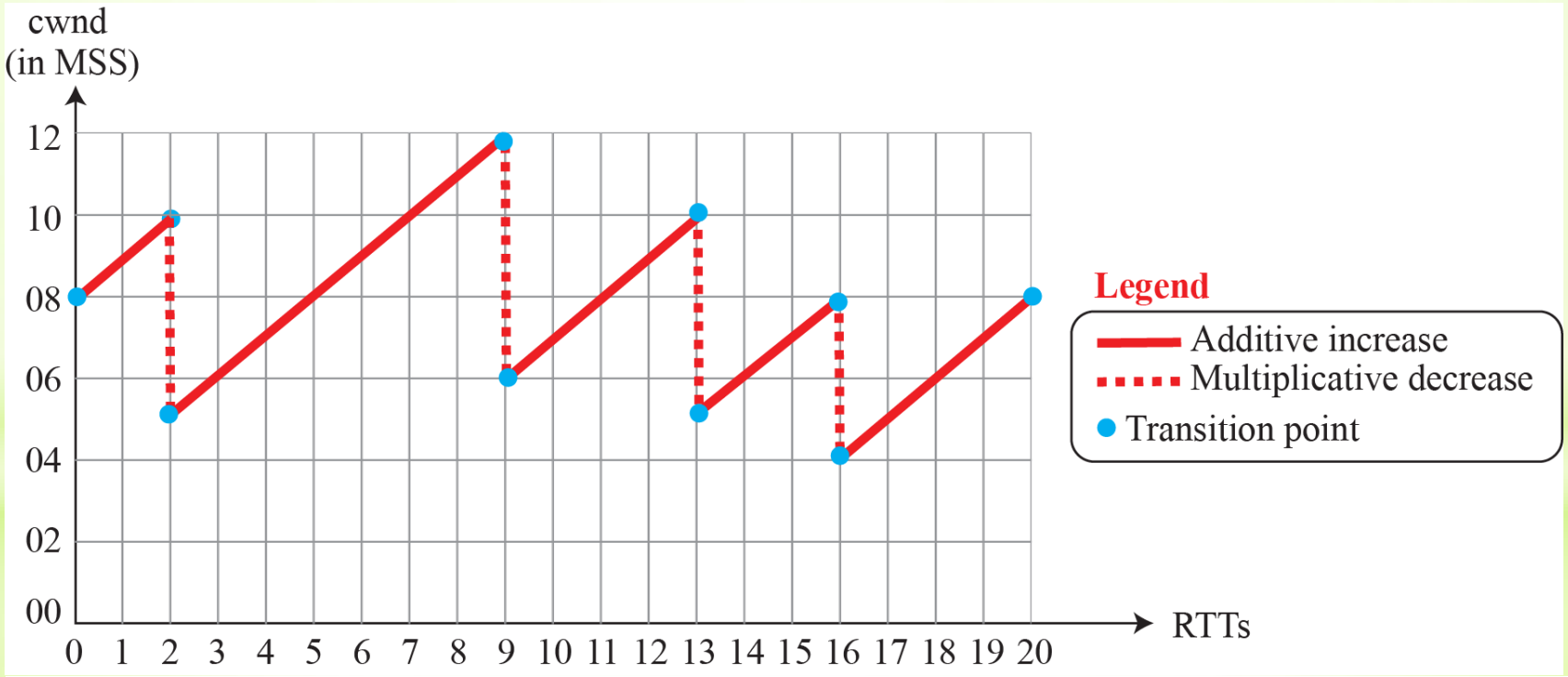
New Reno TCP

- A later version of TCP, called *NewReno TCP*, made an extra optimization on the Reno TCP.
- When TCP receives three duplicate ACKs, it retransmits the lost segment until a new ACK (not duplicate) arrives. (Fast Recovery Sate)
- If the new ACK defines the end of the window when the congestion was detected, TCP is certain that only one segment was lost. However, if the ACK number defines a position between the retransmitted segment and the end of the window, it is possible that the segment defined by the ACK is also lost.
- NewReno TCP retransmits this segment to avoid receiving more and more duplicate ACKs for it.

New Reno TCP


seq 101-200
seq 201-300
seq 301-400
seq 401-500
seq 501-600
seq 601-700
ack 201
sack 201(1)
sack 201(2)
sack 201(3) , ack 601
301-400
501-600
201-300
401-500
601-700
retransmit
seq 201-300
sack 501(1)
301-400
501-600
201-300
401-500
601-700
sack value less than window size
retransmit
seq 501-600
without waiting for 3 SACK
Sender
receiver

- Out of the three versions of TCP, the Reno version is most common today.
- If we ignore the slow-start states and short exponential growth during fast recovery, the TCP congestion window is *cwnd* = *cwnd* + (1 / cwnd) when an ACK arrives (congestion avoidance), and *cwnd* = *cwnd* / 2 when congestion is detected. i.e. Additive increase, multiplicative decrease (AIMD)



Additive increase, multiplicative decrease (AIMD)

TCP Timers

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

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

Retransmission Timer

Round-Trip Time (RTT)

Measured RTT (RTT_M)

- How long it takes to send a segment and receive an acknowledgment for it. This is the measured RTT_M.
- In TCP the segments and their acknowledgments do not have a one-to-one relationship, several segments may be acknowledged together.
- **In TCP, there can be only one RTT_M measurement in progress at any time.**

Smoothed RTT (RTT_S)

- 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	→	$RTT_S = (1 - \alpha)\, RTT_S + \alpha \times RTT_M$

α is implementation-dependent, but it is normally set to 1/8

Retransmission Timer

Deviated RTT

Most implementations do not just use RTTS; they also calculate the RTT deviation, called RTT_D, based on the RTT_S and RTT_M

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

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

Retransmission Time Out (RTO)

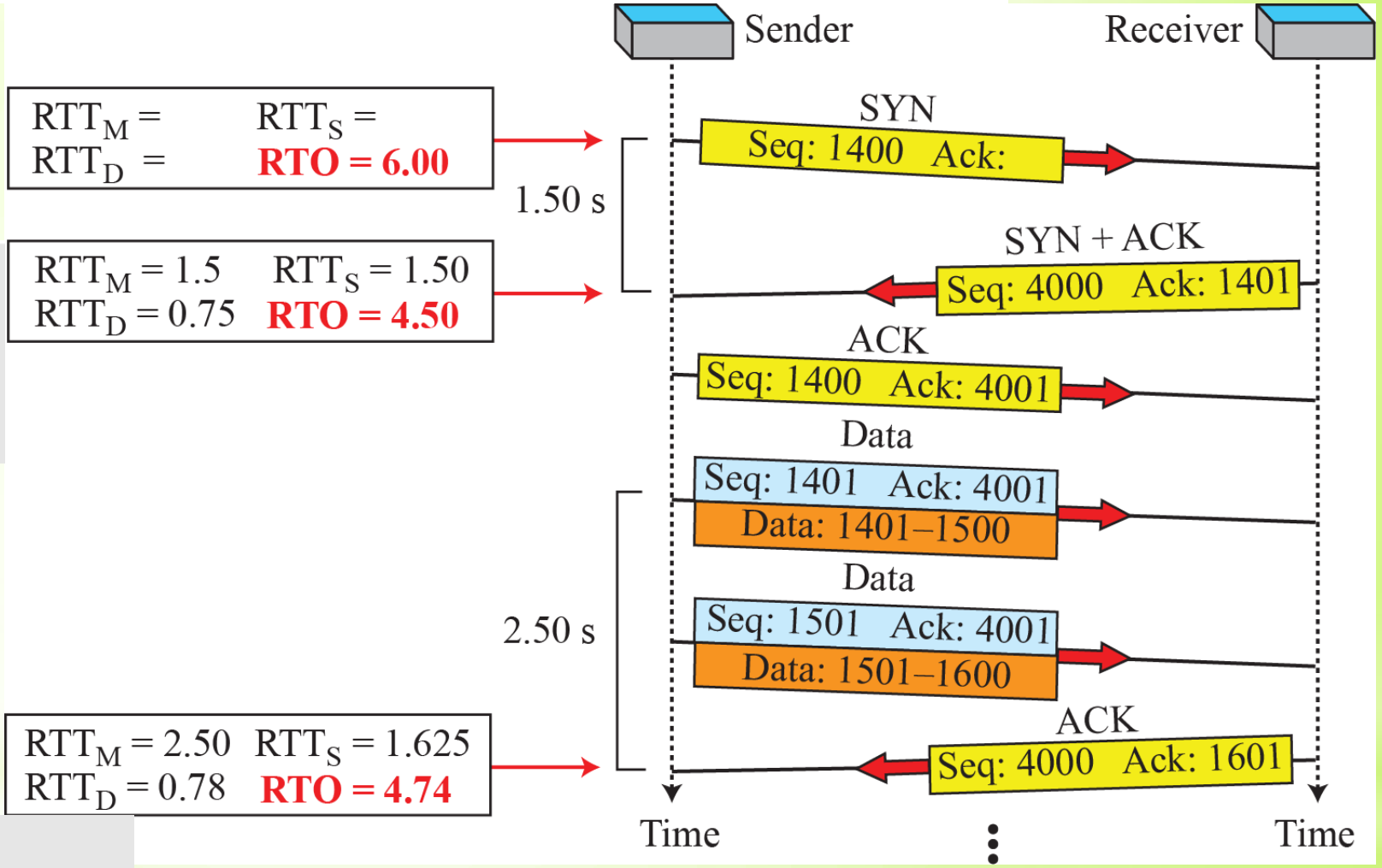
The value of RTO is based on the smoothed round trip time and its deviation.

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

Example

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

$RTT_M = 2.5$
$RTT_S = (7/8) \times (1.5) + (1/8) \times (2.5) = 1.625$
$RTT_D = (3/4) \times (0.75) + (1/4) \times |1.625 - 2.5| = 0.78$
$RTO = 1.625 + 4 \times (0.78) = 4.74$



❖ Karn's Algorithm

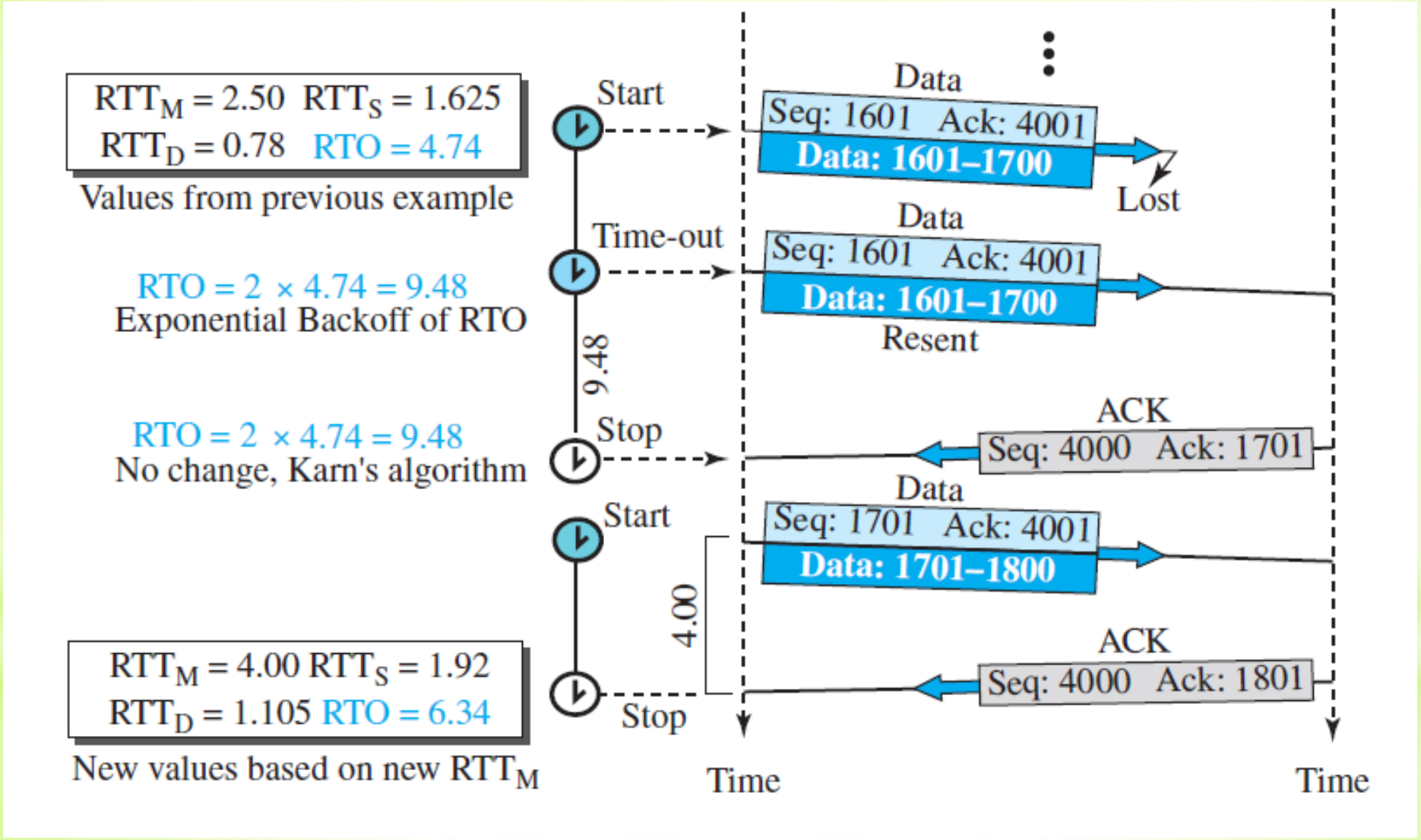
- Suppose that a segment is not acknowledged during the retransmission time-out period and is therefore retransmitted. When the sending TCP receives an acknowledgment for this segment, it does not know if the acknowledgment is for the original segment or for the retransmitted one.

- Karn's algorithm is simple: **TCP does not consider the RTT of a retransmitted segment in its calculation of a new RTO**.

❖ Exponential Backoff

- Most TCP implementations use an exponential backoff strategy. The value of RTO is doubled for each retransmission. So if the segment is retransmitted once, the value is two times the RTO. If it transmitted twice, the value is four times the RTO, and so on.

Retransmission and Karn's algorithm is applied.



Persistence Timer

- To deal with a zero-window-size advertisement, TCP needs another timer. If the receiving TCP announces a window size of zero, the sending TCP stops transmitting segments until the receiving TCP sends an ACK segment announcing a nonzero window size. This ACK segment can be lost. Both TCP's might continue to wait for each other forever (a deadlock).
- To correct this deadlock, TCP uses a persistence timer for each connection. When the sending TCP receives an acknowledgment with a window size of zero, it starts a persistence timer.
- When the persistence timer goes off, the sending TCP sends a special segment called a probe.
- This segment contains only 1 byte of new data. It has a sequence number, but its sequence number is never acknowledged
- The value of the persistence timer is set to the value of the retransmission time. However, if a response is not received from the receiver, another probe segment is sent and the value of the persistence timer is doubled and reset until the value reaches a threshold (usually 60 s). After that the sender sends one probe segment every 60 seconds until the window is reopened.

❑ Keepalive Timer

- A keep alive timer is used in some implementations to prevent a long idle connection between two TCP's.
- The time-out is usually 2 hours. If the server does not hear from the client after 2 hours, it sends a probe segment. If there is no response after 10 probes, each of which is 75 seconds apart, it assumes that the client is down and terminates the connection.

❑ TIME-WAIT Timer

- The TIME-WAIT (2MSL) timer is used during connection termination. The 2MSL timer is used when TCP performs an active close and sends the final ACK. The connection must stay open for 2 MSL amount of time to allow TCP to resend the final ACK in case the ACK is lost.
- Common values are 30 seconds, 1 minute, or even 2 minutes.