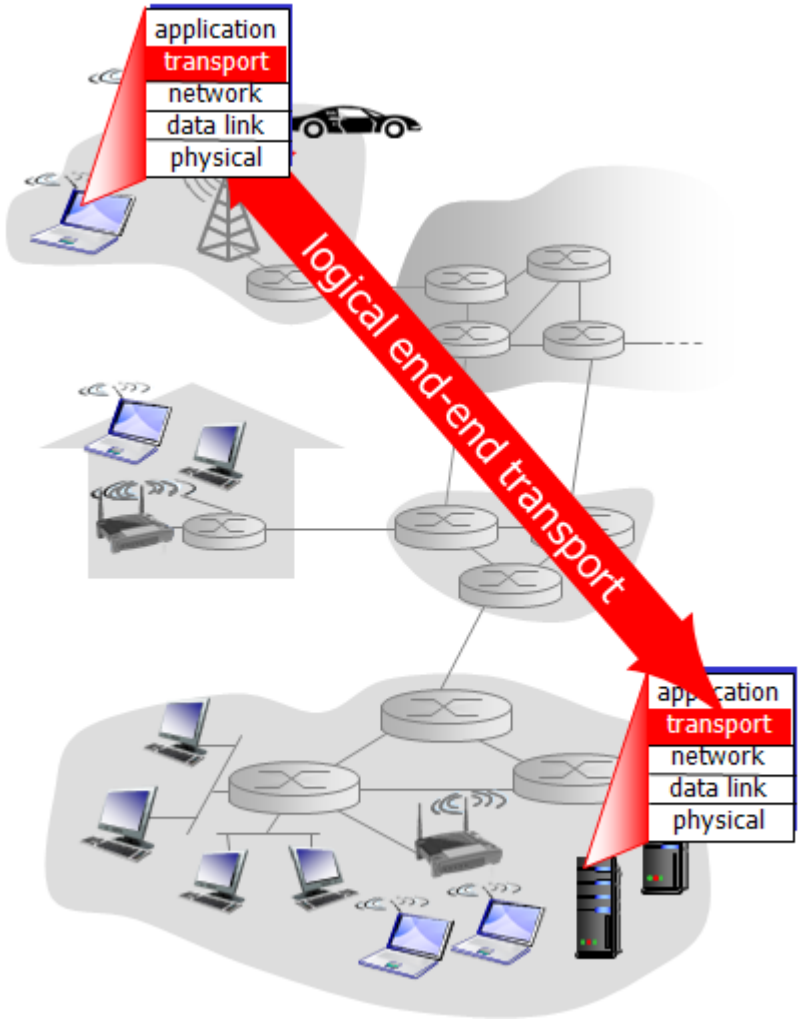
Transport Layer Protocols

EE450: Introduction to Computer Networks

Professor A. Zahid

Transport Layer

- Provide logical Communications between app processes running on different hosts
- Transport protocols run in end systems
  - Send side: Breaks app messages into segments, passes to network layer
  - Recv side: reassembles segments into messages, passes to app layer
- Several Transport Protocols, ex. TCP, UDP, SCTP, etc…



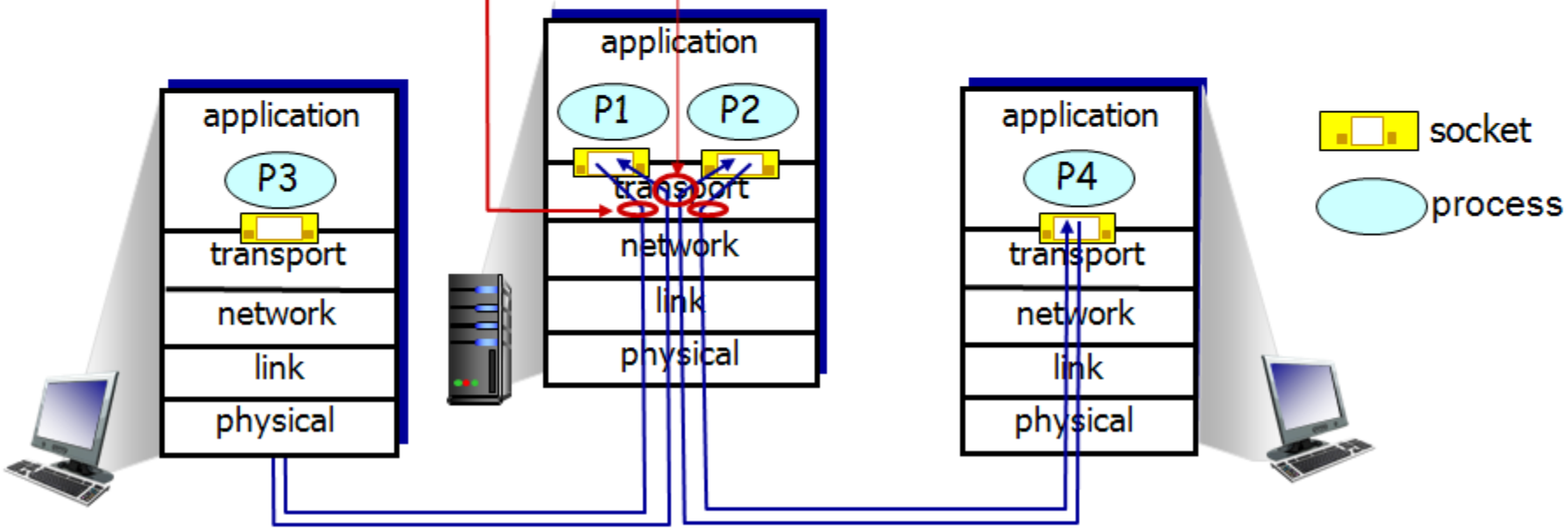
Functions of Transport Protocols

- Functions of the transport layer protocols include:
  - Provide for Process-to-Process communications. To accomplish this task, Port Numbers are used to identify the process, at both the client and at the server side
  - Provide for end-to-end Error Checking (both TCP and UDP), Error Control and Flow and Congestion control (only TCP)
  - TCP is a reliable protocol, UDP is an unreliable Protocol
- Neither TCP nor UDP provides for "Guaranteed Delay" or Guaranteed Bandwidth"

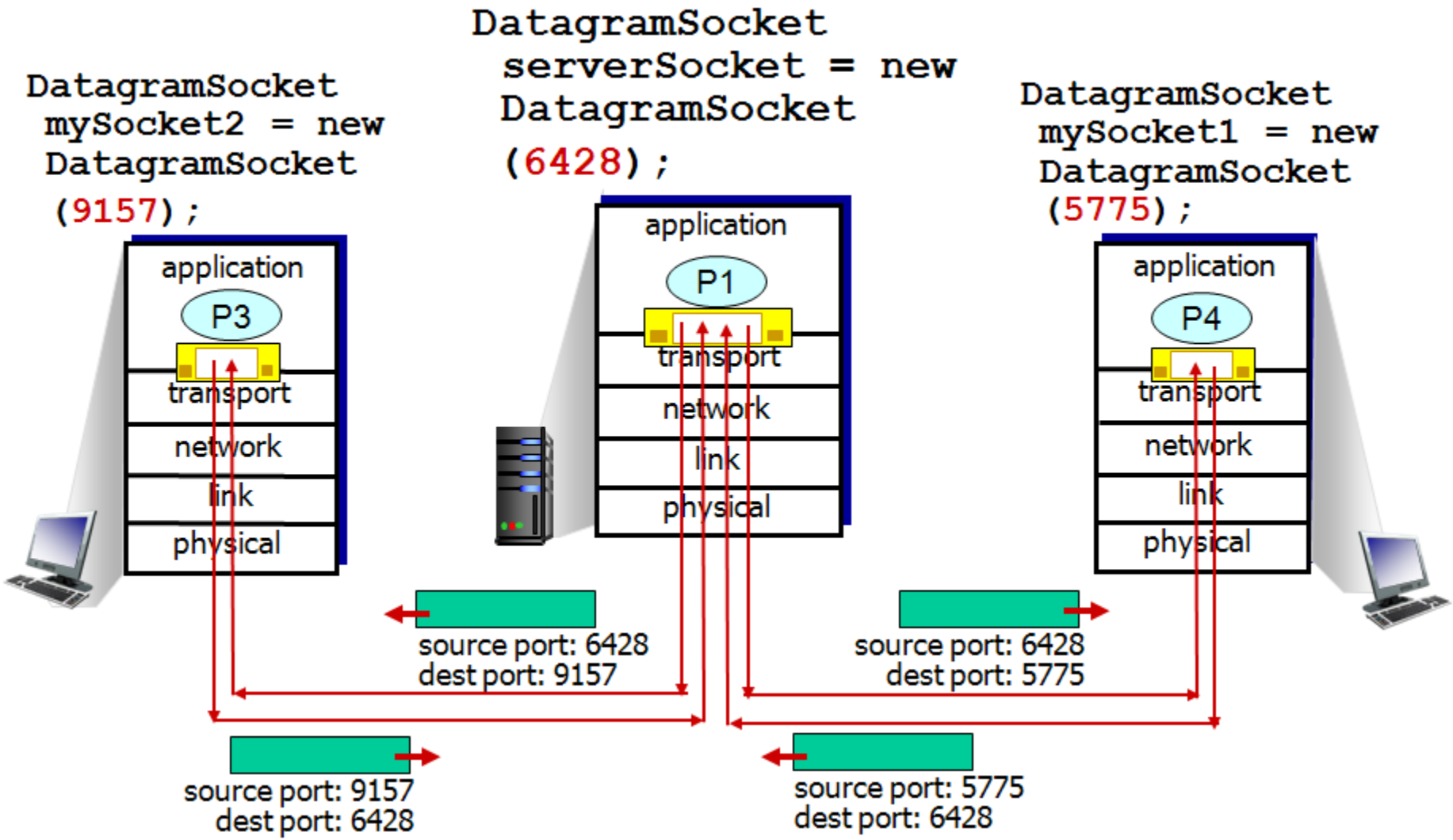
Multiplexing/Demultiplexing

multiplexing at sender:
handle data from multiple sockets, add transport header (later used for demultiplexing)

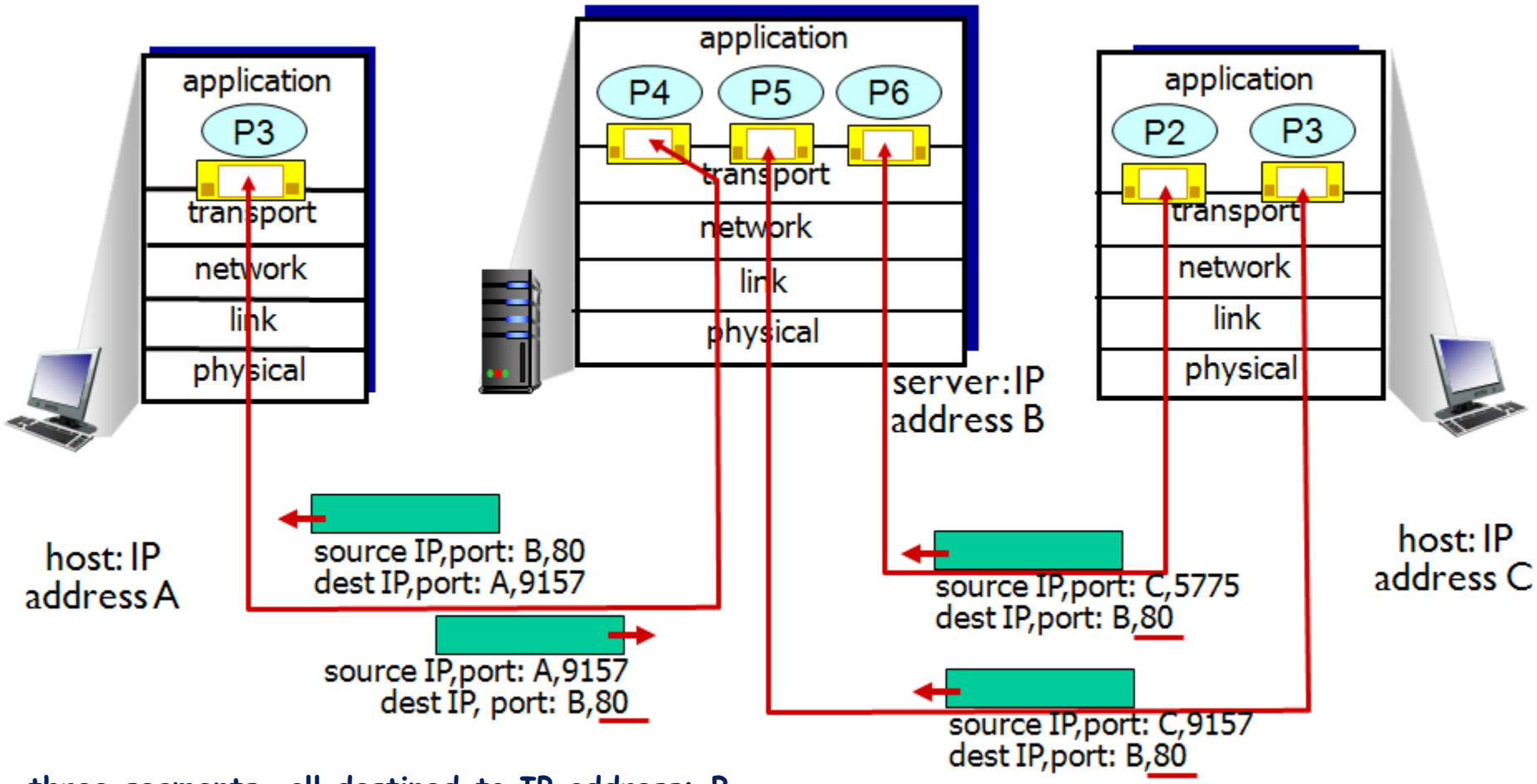
demultiplexing at receiver:
use header info to deliver received segments to correct socket



Connectionless (UDP) Demultiplexing

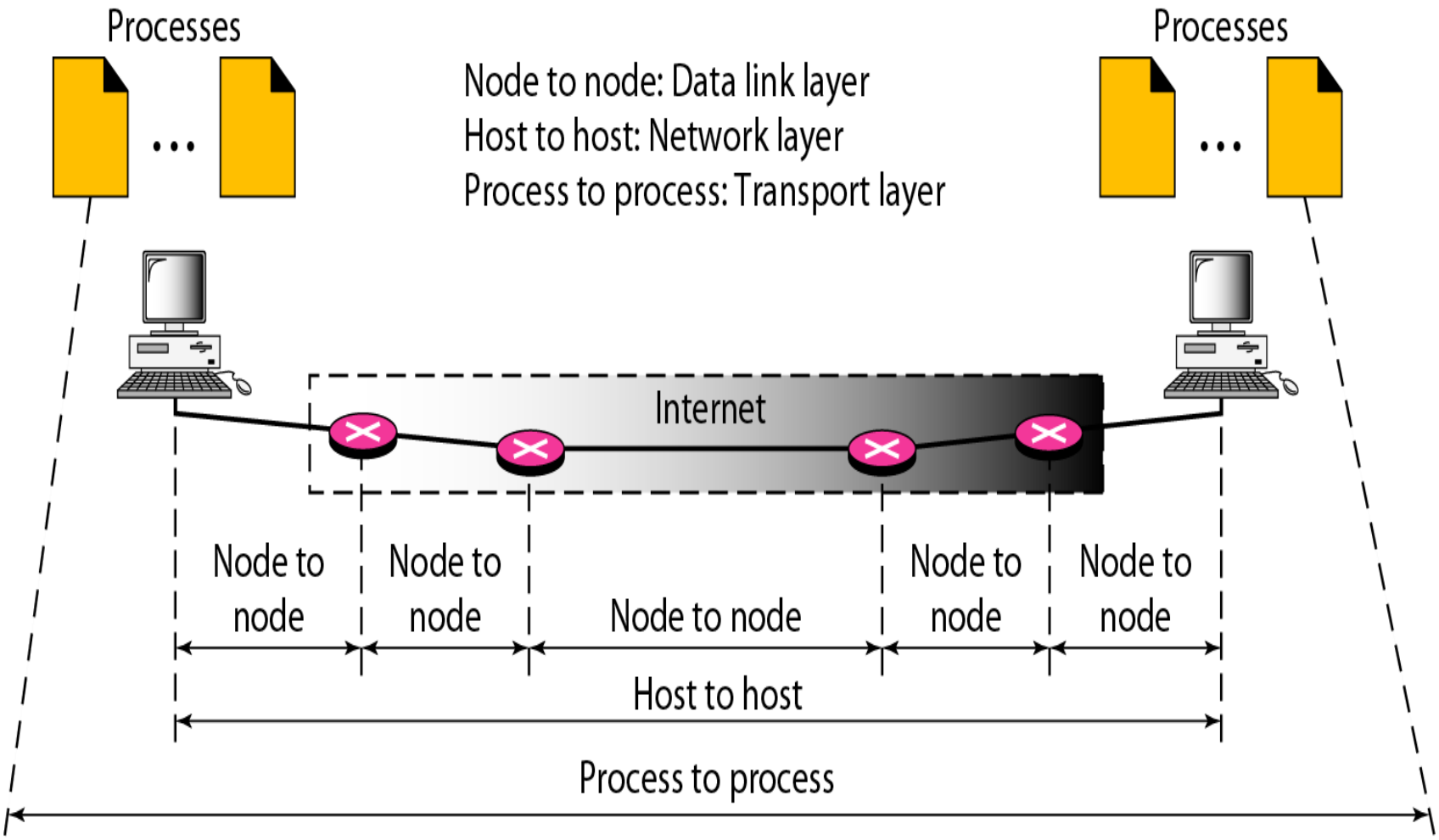


Connection-Oriented (TCP) Demultiplexing



three segments, all destined to IP address: B, dest port: 80 are demultiplexed to *different* sockets

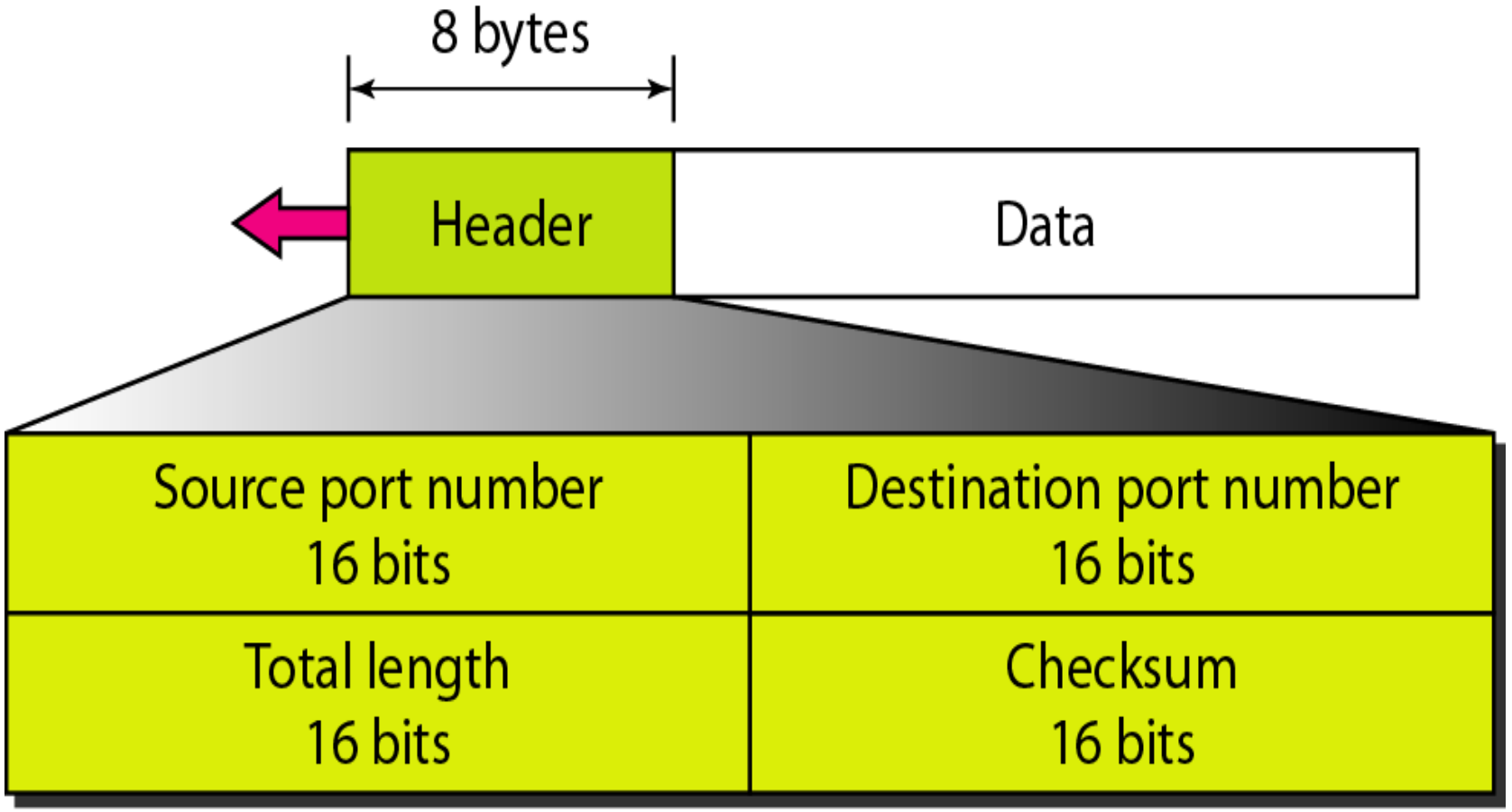
Types of Data Delivery



User Datagram Protocol

- UDP is a connection-less, unreliable end-to-end transport layer protocol that provides
  - Process-to-process communications
  - End-to-end error checking only
- UDP does not provide for end-to-end error or flow control
- UDP services is used by
  - Applications that involves short request/response such as DNS, SNMP, RIP, etc…
  - Applications that can't tolerate connection-setup delay such as multimedia applications, internet telephony, streaming audio/video, etc…

UDP Datagram Format

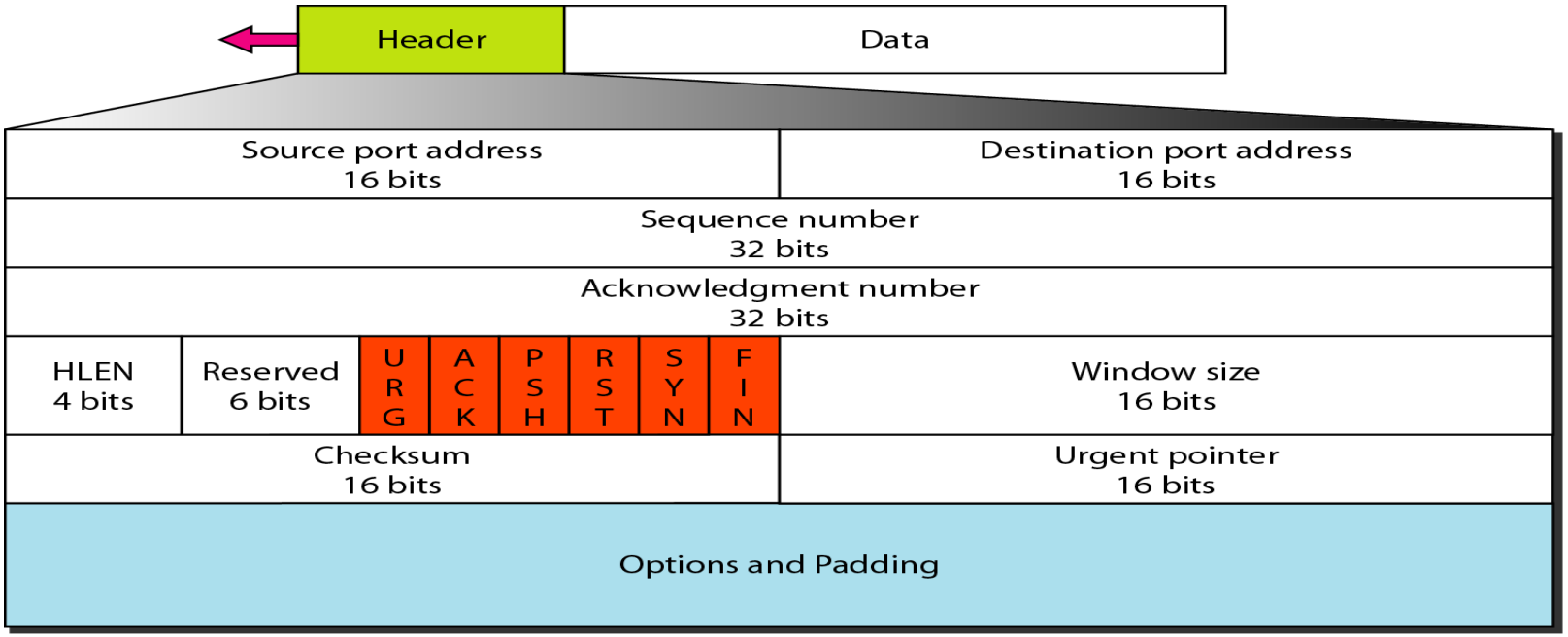


Checksum: checks entire UDP datagram for errors

TCP: Transport Control Protocol

- TCP is a point-to-point, connection-oriented, reliable, end-to-end protocol that provides
  - Process-to-process communications
  - End-to-end error, flow and congestion control
  - FDX service
- TCP services is used by
  - Applications that can tolerate packet losses but can tolerate the additional delay required to set up the logical connection. Such applications include HTTP, SMTP, FTP, TELNET, etc…
- The unit of data using TCP is called a Segment
- TCP is a Byte-Oriented Protocol (No message boundary)

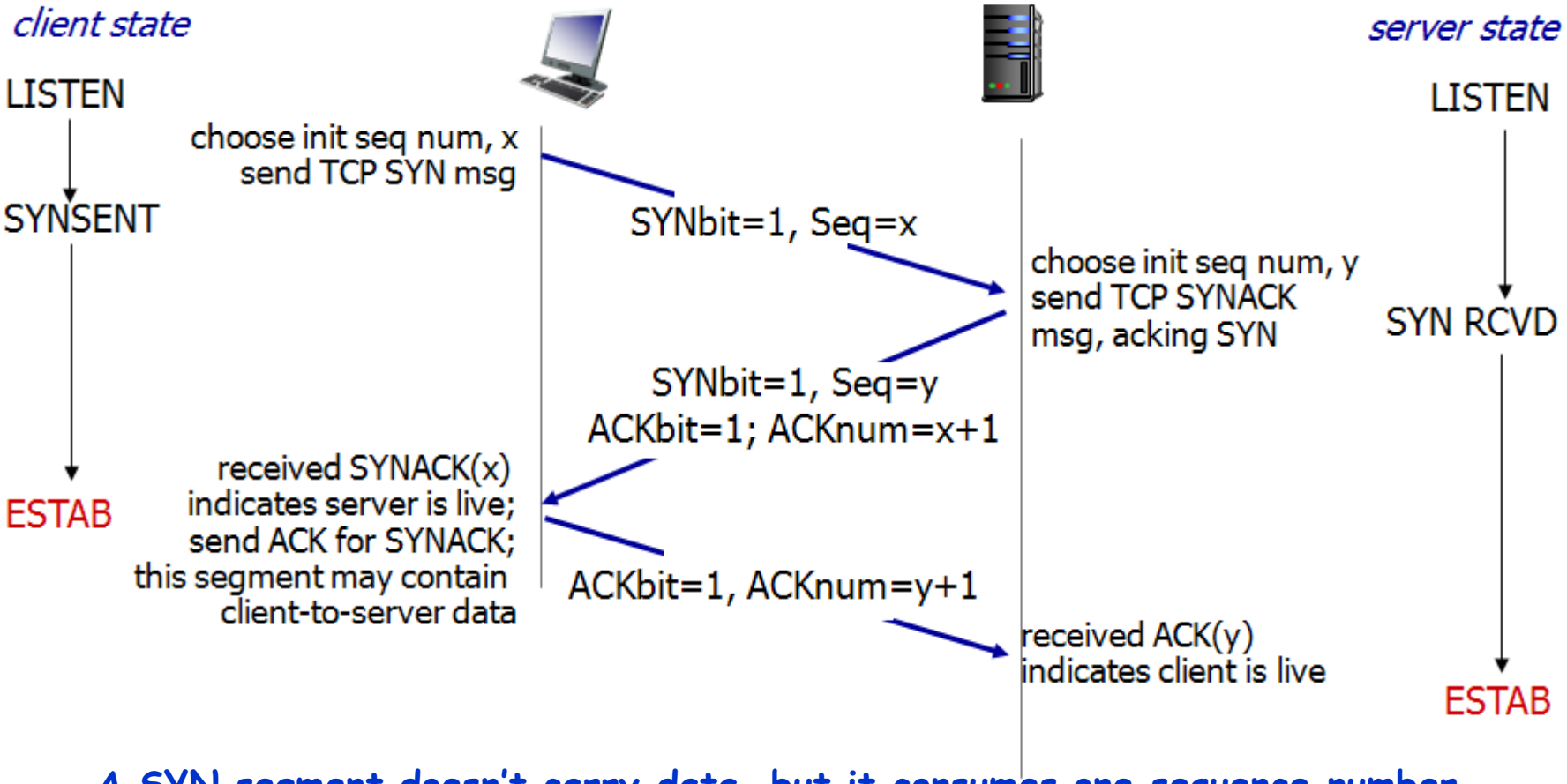
TCP Segment Format



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

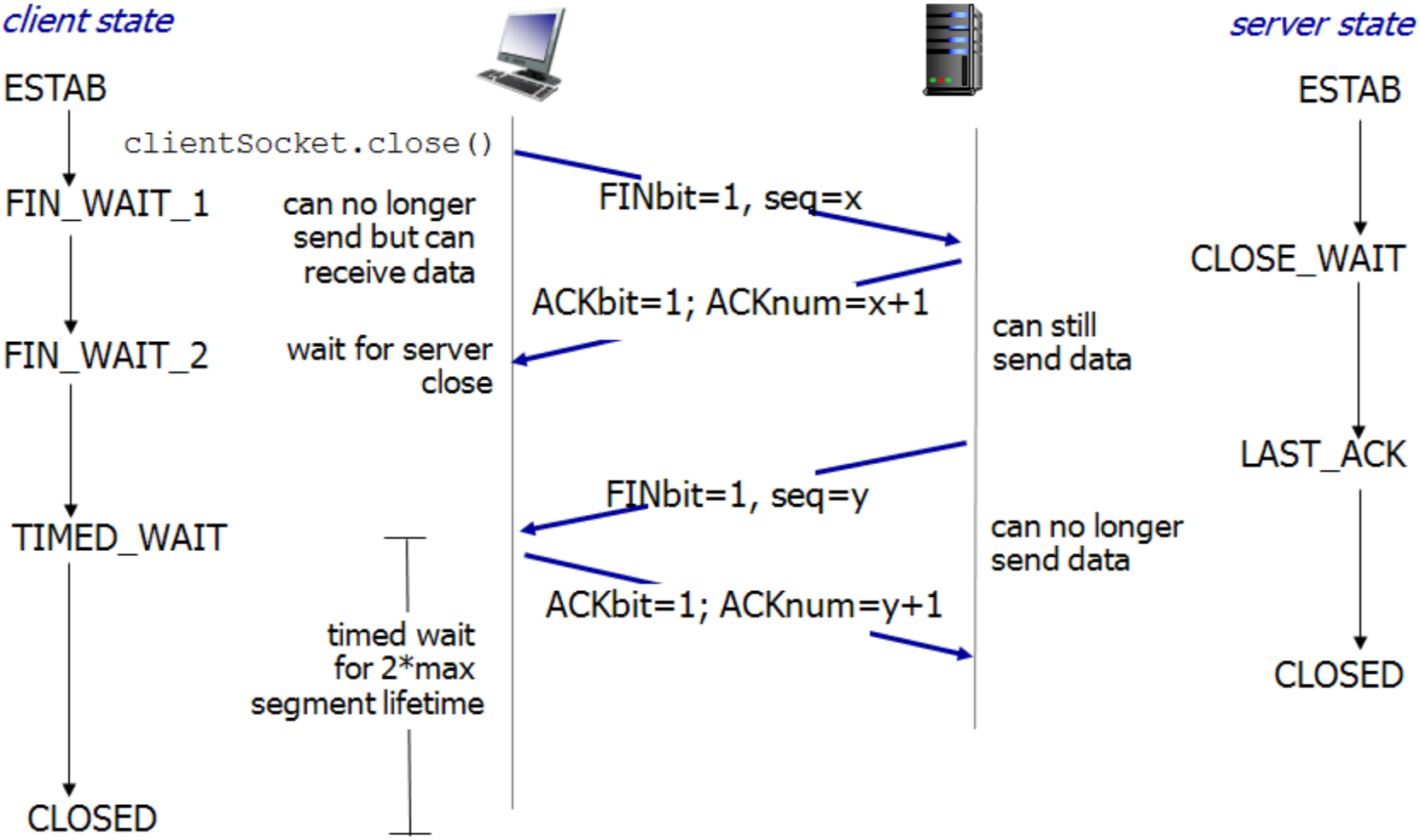
- The sequence number identifies the number of the first byte in the payload
- The Acknowledgement number is the number of the next byte expected to be received
- The receiver window size indicates the number of bytes the receiver is willing to accept

TCP Connection Set-up

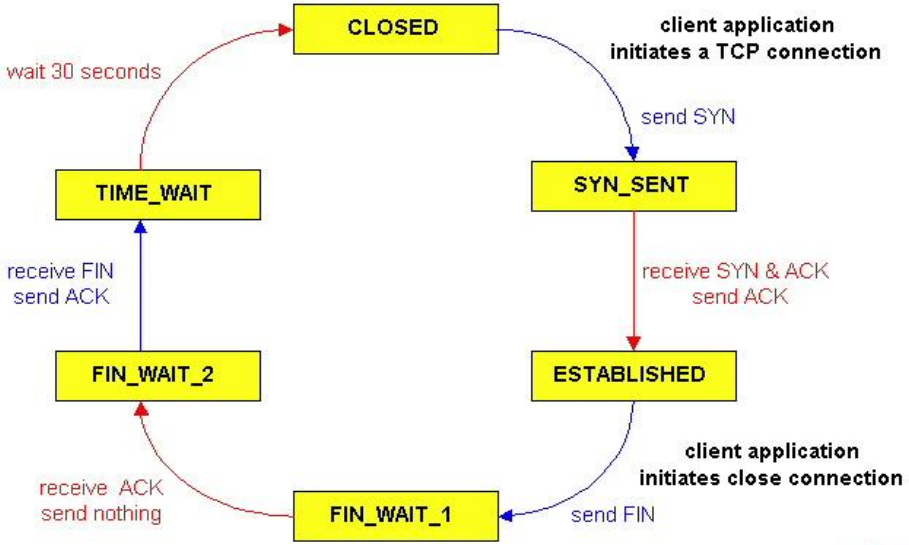


A SYN segment doesn't carry data, but it consumes one sequence number.
A SYN/ACK segment doesn't carry data, but it consumes one sequence number
An ACK segment can carry data

TCP Connection Termination

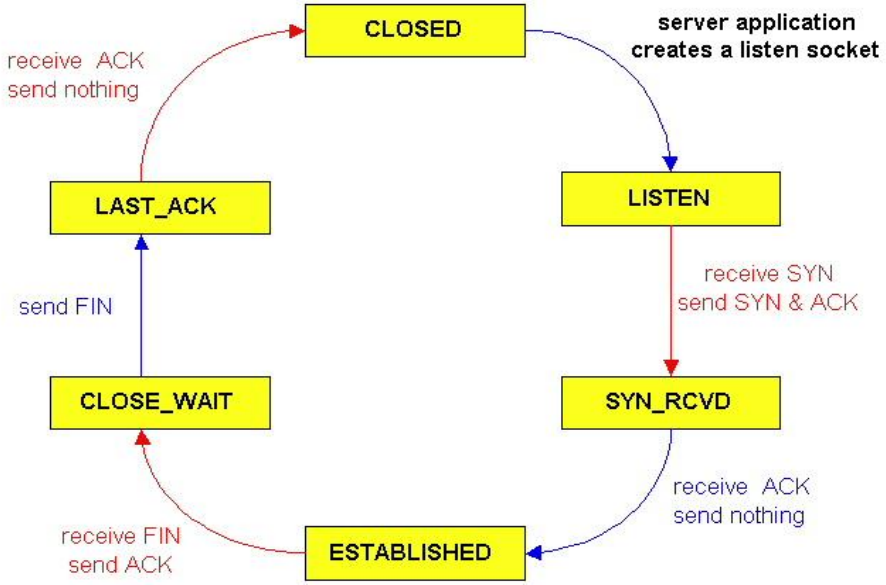


TCP Connection Management



TCP client lifecycle

TCP server lifecycle



Reliability in TCP

- Components of Reliability
  - Sequence Numbers/Acknowledgements
  - Retransmissions
  - Timeout Mechanism(s): function of the round trip time (RTT) between the two hosts (is it static?). How to set TCP timeout value?
    - Longer than RTT, but RTT varies???
    - Too short, but that may mean premature timeouts and hence unnecessary retransmissions
    - Too long, but that means slow reaction to segment losses

RTT and RTO Estimates (EE555)

- Calculate SampleRTT: measured time from segment transmission until ACK receipt. Ignore calculating SampleRTT for retransmitted segments
- Calculate a "Smoothed RTT" based on current and previous SampleRTTs

$$\text{EstimatedRTT}(k) = (1-\alpha)*\text{EstimatedRTT}(k-1) + \alpha*\text{SampleRTT}(k)$$
$$= (1-\alpha)^k*\text{SampleRTT}(0) + \alpha(1-\alpha)^{k-1}*\text{SampleRTT}(1) + \ldots + \alpha*\text{SampleRTT}(k)$$

Exponential weighted moving average

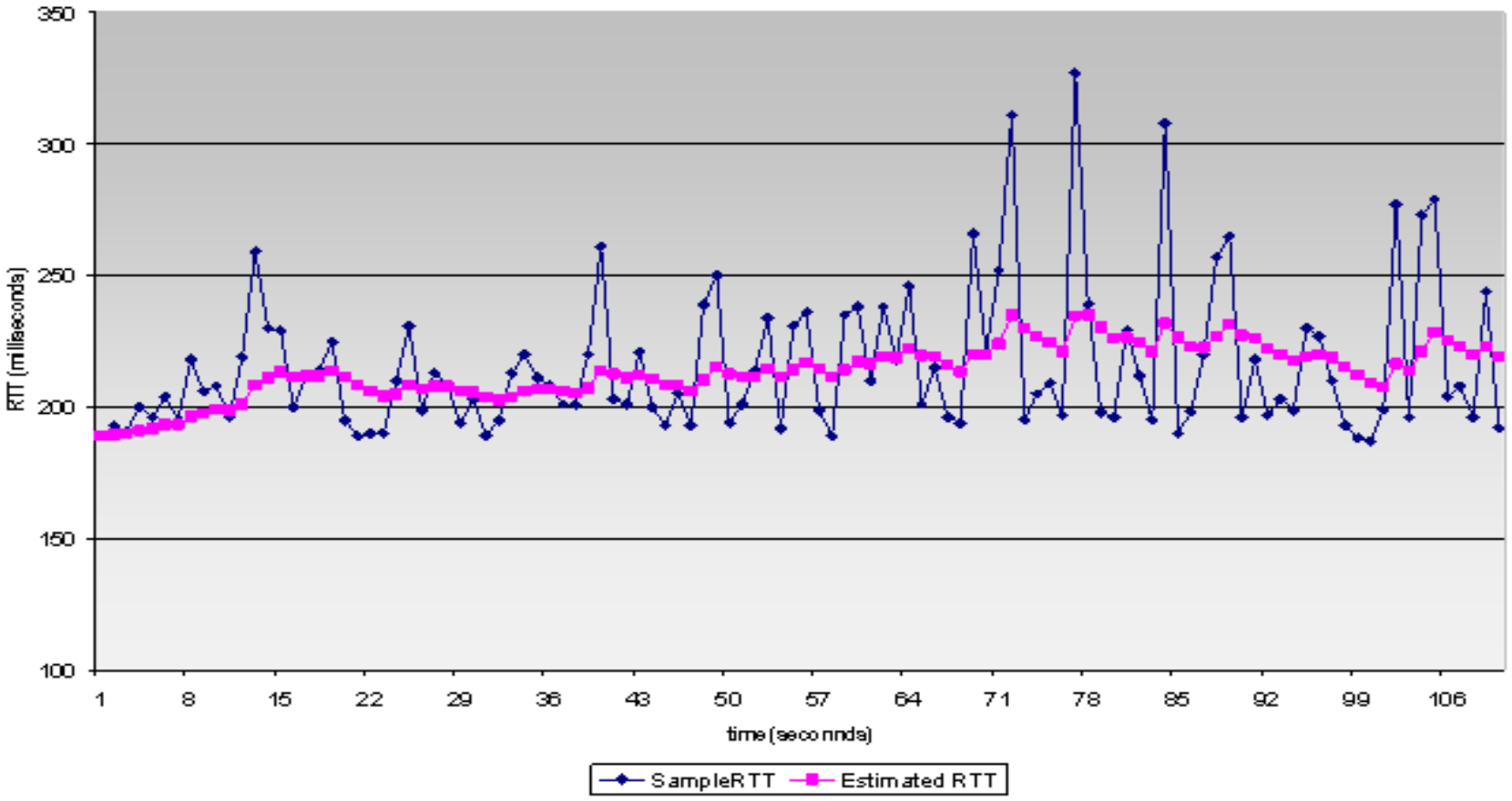
Influence of past sample decreases exponentially fast

Typical value: $\alpha = 0.125$

Set:

$$\text{TimeoutInterval} = \text{EstimatedRTT} + 4*\text{DevRTT}$$

Estimation of RTT



TCP Reliable Data Transfer

- TCP creates reliable service on top of IP's unreliable service
- Pipelined segments
- Cumulative ACKs
- TCP uses single retransmission timer
- Retransmissions are triggered by:
  - Timeout events
  - Duplicate ACKs
- Initially consider simplified TCP sender:
  - Ignore duplicate ACKs
  - Ignore flow control, congestion control

TCP Sender Events

data rcvd from app:

- Create segment w/seq #
- Seq # is byte-stream number of first data byte in segment
- Start timer if not already running (think of timer as for oldest un-Acked seg.)
- Expiration interval: Time-Out-Interval

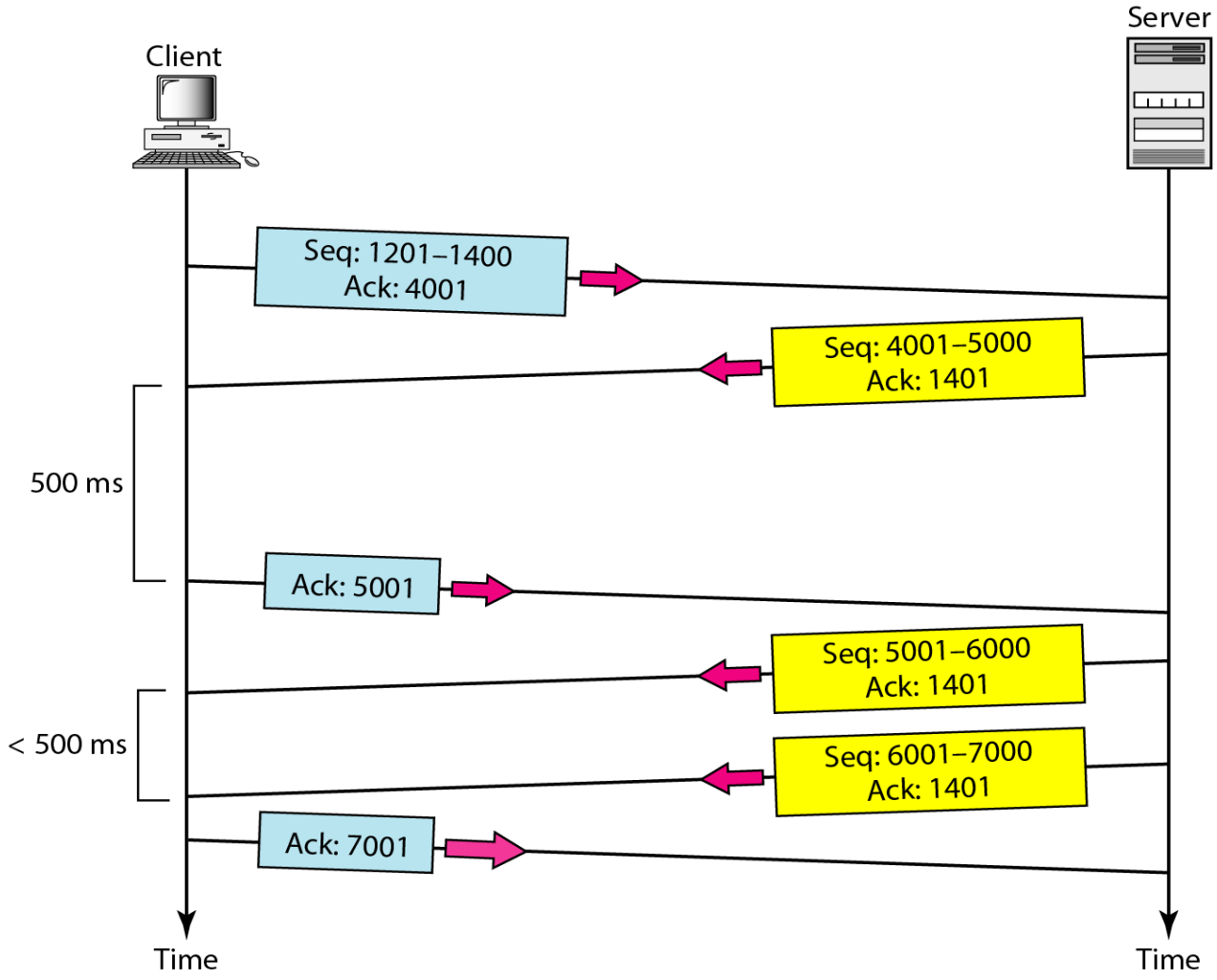
timeout:

- Retransmit segment that caused timeout
- Restart timer

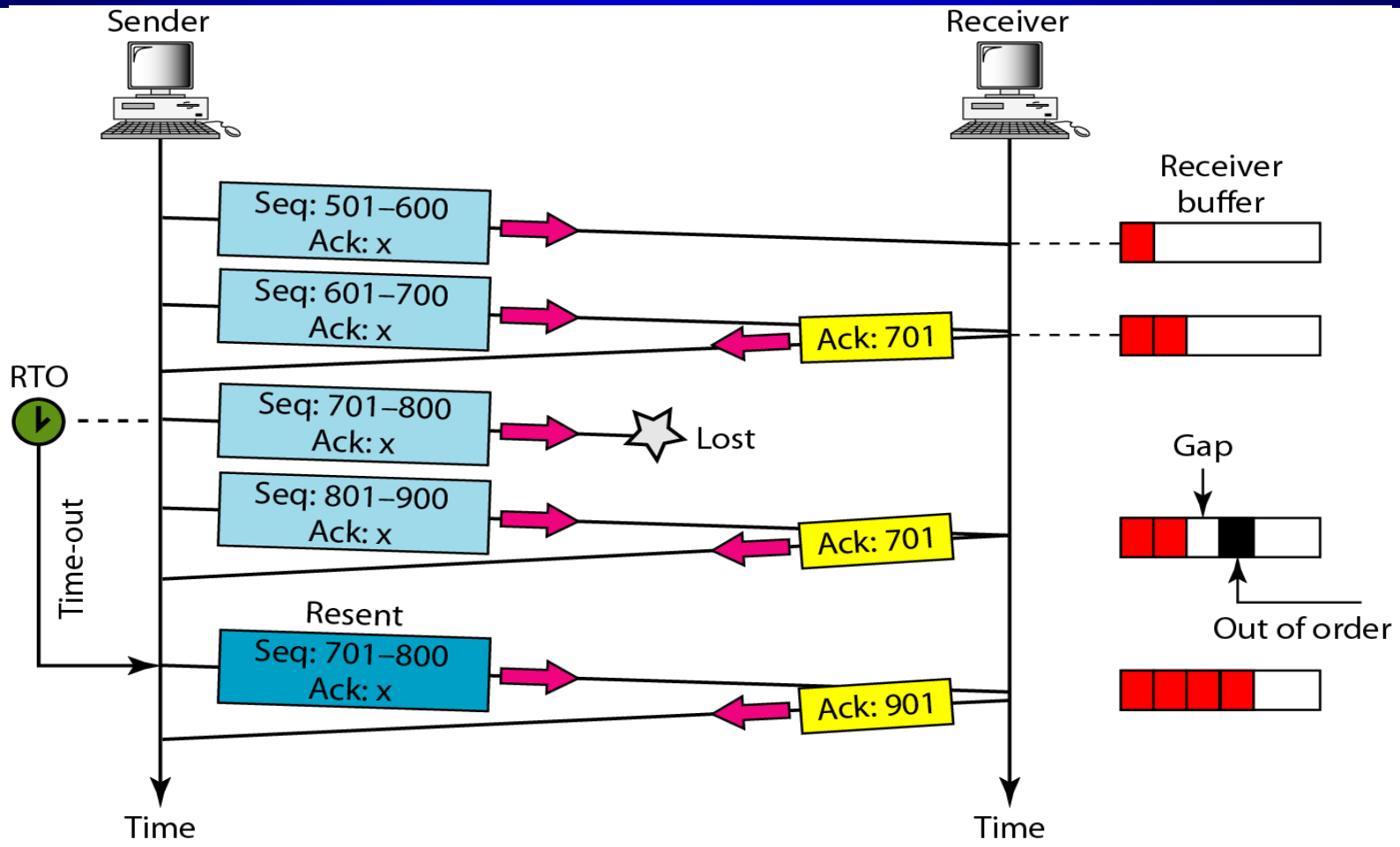
Ack rcvd:

- If acknowledges previously un-Acked segments
  - Update what is known to be Acked
  - Start timer if there are outstanding segments

Normal Operation



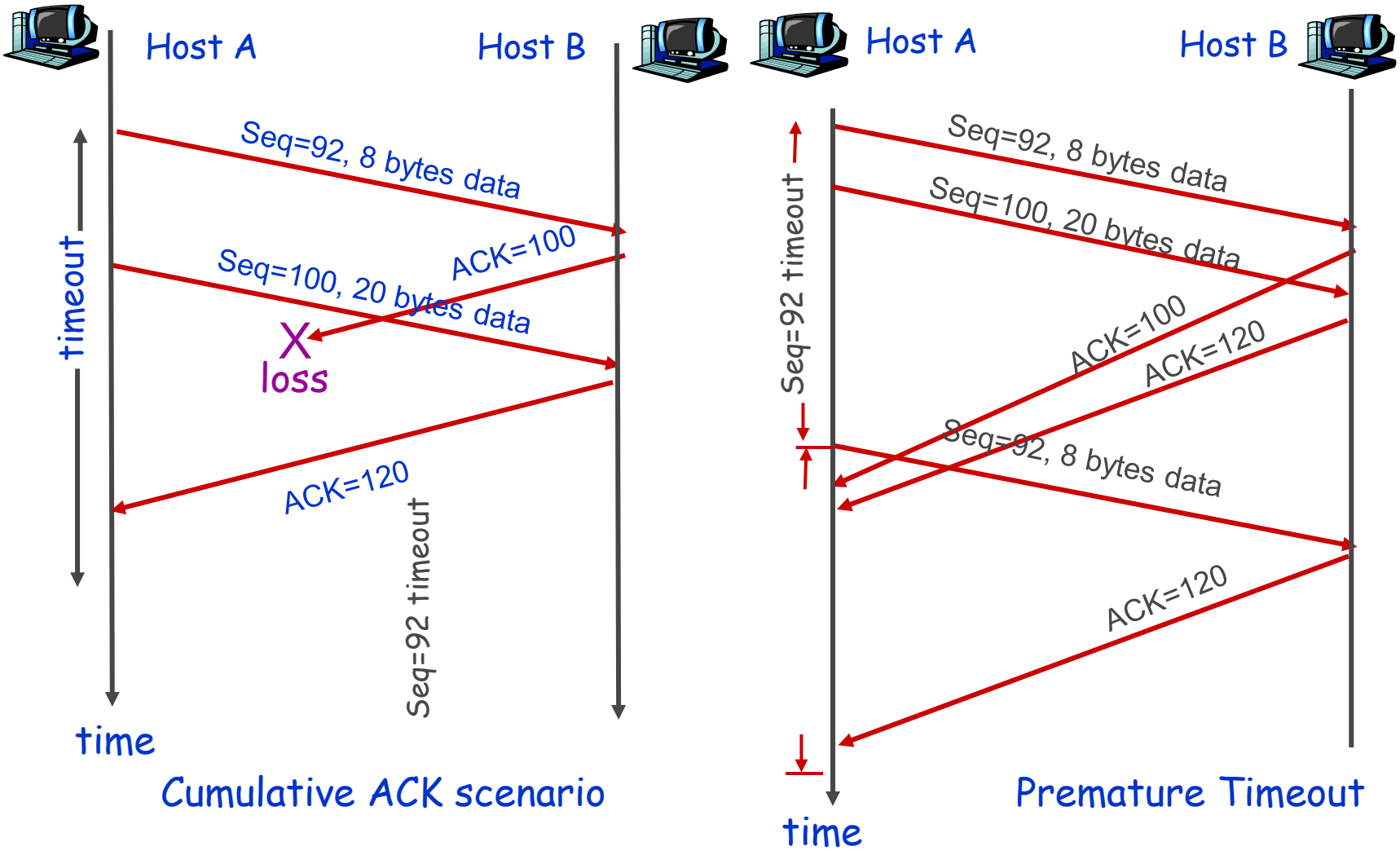
Lost TCP Segment Scenario



The receiver TCP delivers only ordered data to the process.

No retransmission timer is set for an ACK segment.

Other Scenarios



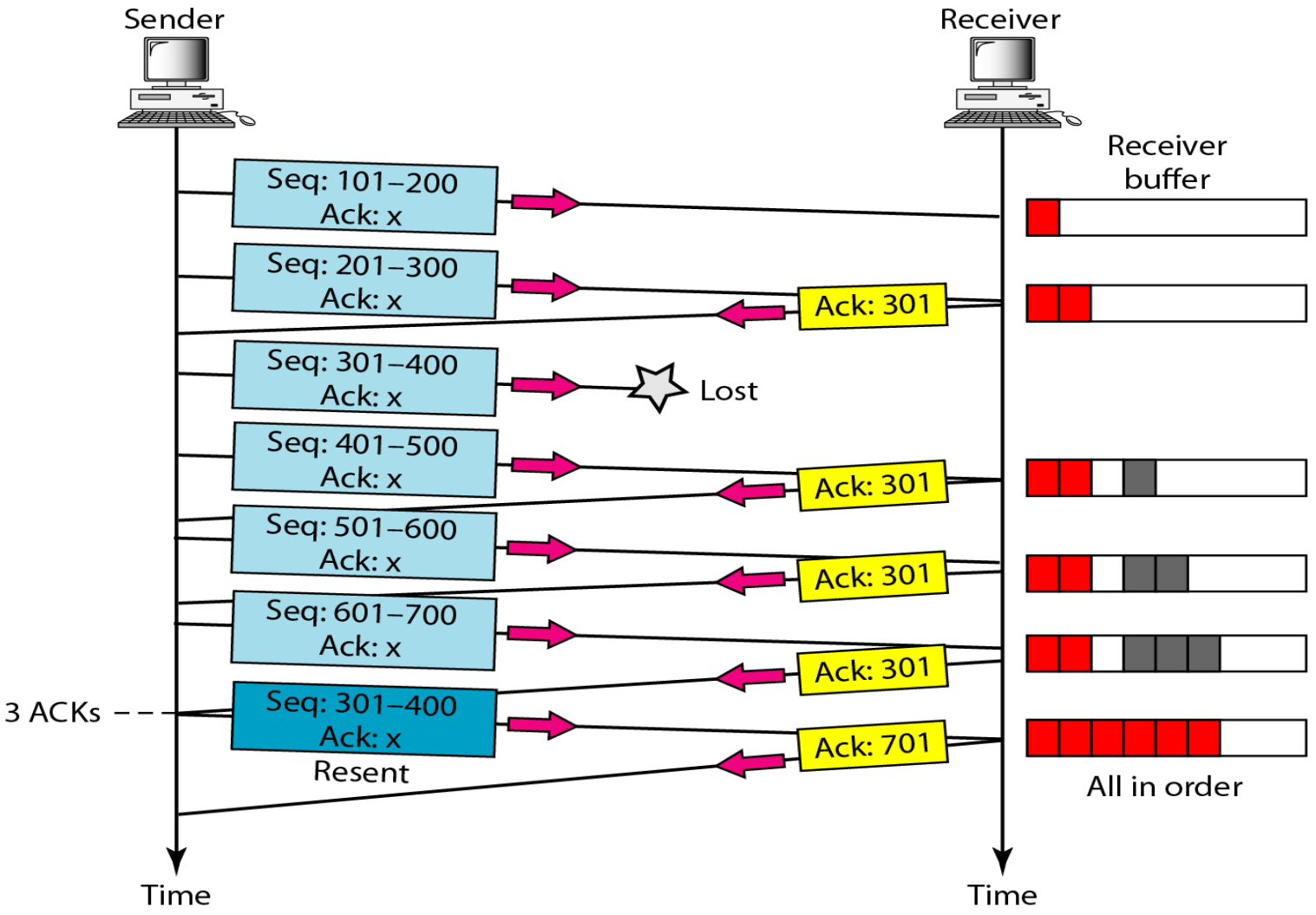
TCP ACK Generation (RFC 1122, 2581)

event at receiver	*TCP receiver action*
arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
arrival of in-order segment with expected seq #. One other segment has ACK pending	immediately send single cumulative ACK, ACKing both in-order segments
arrival of out-of-order segment higher-than-expect seq. # . Gap detected	immediately send *duplicate ACK*, indicating seq. # of next expected byte
arrival of segment that partially or completely fills gap	immediate send ACK, provided that segment starts at lower end of gap

Fast Retransmission

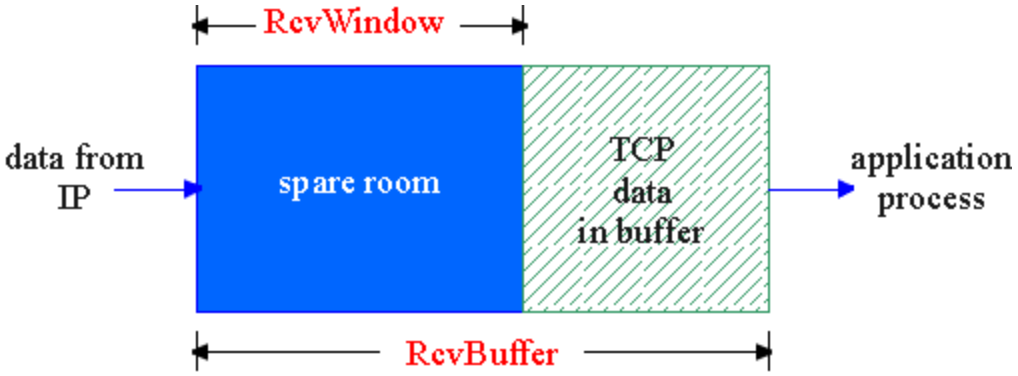
- Time-out period often relatively long:
  - long delay before resending lost segment
- Detect lost segments via duplicate ACKs.
  - Sender often sends many segments back-to-back
  - If segment is lost, there will likely be many duplicate ACKs.
- If sender receives 3 ACKs for the same data, it supposes that segment after ACKed data was lost:
  - fast retransmit: resend segment before timer expires

Fast Retransmission Strategy



Flow Control in TCP

- Receive side of TCP connection has a receive buffer:



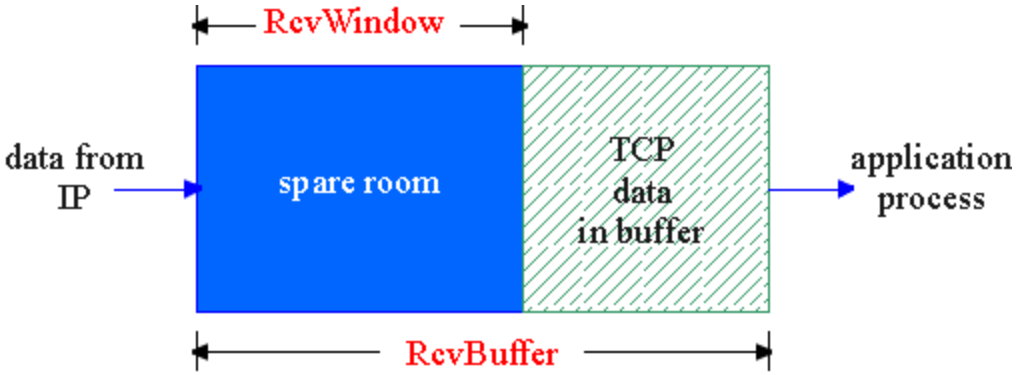
- application process may be slow at reading from buffer

flow control

sender won't overflow receiver's buffer by transmitting too much, too fast

- speed-matching service: matching the send rate to the receiving app's drain rate

Flow Control in TCP (Cont.)



(Suppose TCP receiver discards out-of-order segments)

- Spare room in buffer

= **RcvWindow**

= **RcvBuffer-[LastByteRcvd - LastByteRead]**

- Receiver advertise spare room by including value of **RcvWindow** in segments
- Sender limits un-ACKed data to **RcvWindow** ⇒ No overflow

Principles of Congestion Control

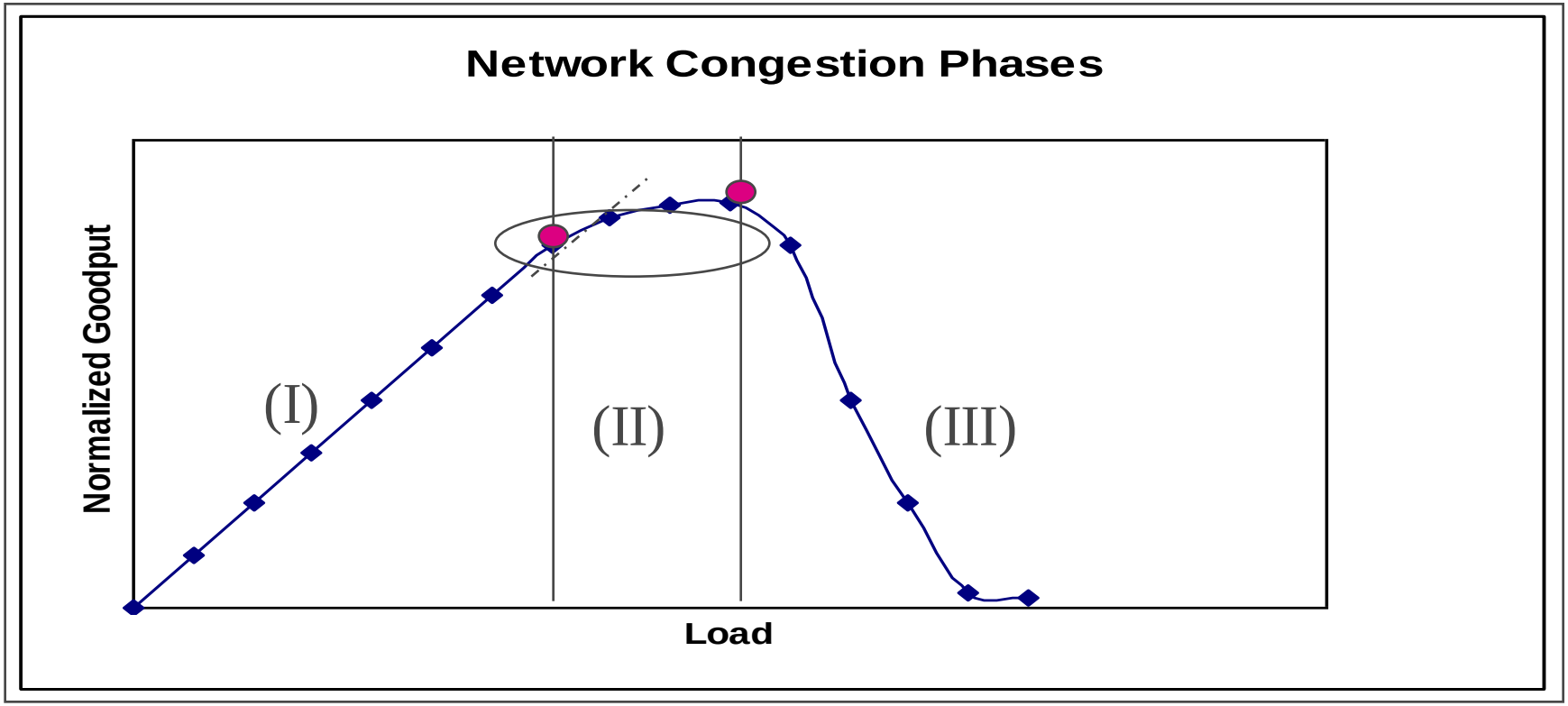
Congestion:

- Informally: "too many sources sending too much data too fast for network to handle"
- Different from flow control!
- Manifestations:
  - Lost packets (Buffer overflow at routers)
  - Long delays (queueing in router buffers)
- A top-10 problem in Network Research!

Congestion Control (CS551/EE555)

- The receiver window (advertised window, w_a) ensures that receiver buffer will never overflow, however it does not guarantee that buffers in intermediate routers will not overflow (congestion)
- IP does not provide any mechanism for congestion control. It is up to TCP to detect congestion
- Define another window, called congestion window, w_c that determines the maximum number of bytes that can be transmitted without congesting the network
- Max # of bytes that can be sent = min (w_a , w_c)

Network Congestion Phases

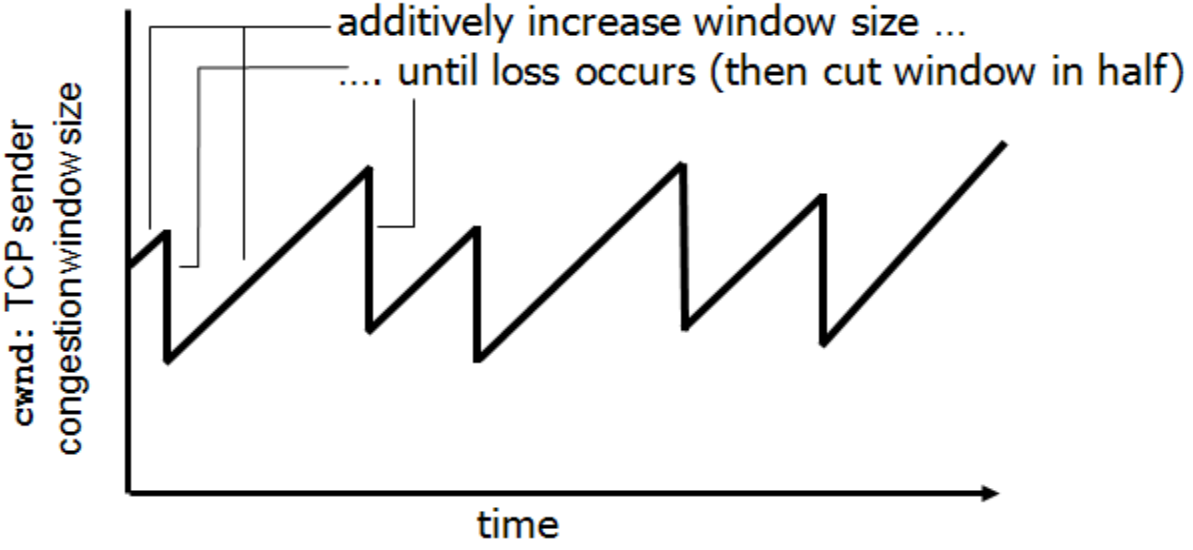


(I) No Congestion
(II) Moderate Congestion
(III) Severe Congestion (Collapse)

AIMD Approach

- *approach:* sender increases transmission rate (window size), probing for usable bandwidth, until loss occurs
  - *additive increase:* increase `cwnd` by 1 MSS every RTT until loss detected
  - *multiplicative decrease:* cut `cwnd` in half after loss

AIMD saw tooth behavior: probing for bandwidth



Slow Start Approach (Tahoe)

- Phase 1: Start by setting the congestion window, w_c to one MSS. Each time the sender receives an ACK it increases its congestion window by one and so on. Hence, $w_c = w_c+1$ for every ACK received. This phase is referred to as the "Slow Start Phase". In SS the congestion window increases exponentially
- Phase 2: As the congestion window reaches a threshold value, the congestion window starts to increase linearly. This phase is referred to Congestion Avoidance Phase. In this phase the congestion window is increased by one segment every RTT, i.e. $w_c = w_c+(1/w_c)$ for every ACK received

Congestion Control (Cont.)

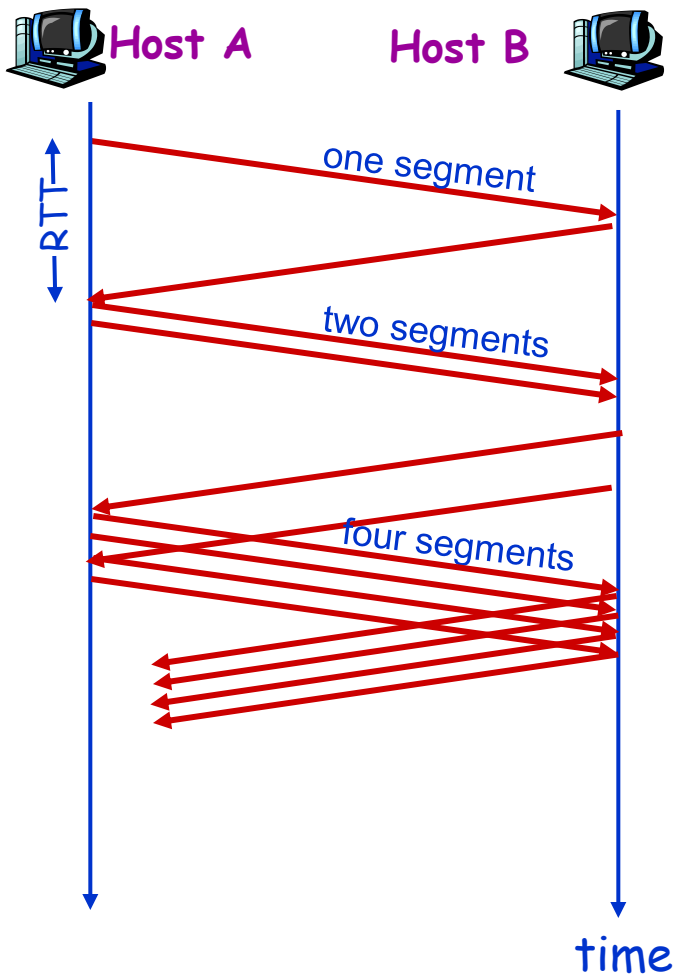
- Phase 3: The congestion window stops increasing when the client TCP detects the network is congested. This happens when an ACK doesn't arrive before the time-out expires. In this phase the congestion threshold is set to 1/2 the current window size which is the min (w_a , w_c). The congestion window is then reset to one segment and the slow start phase is repeated. This phase is referred to as Congestion Control

Slow Start Phase

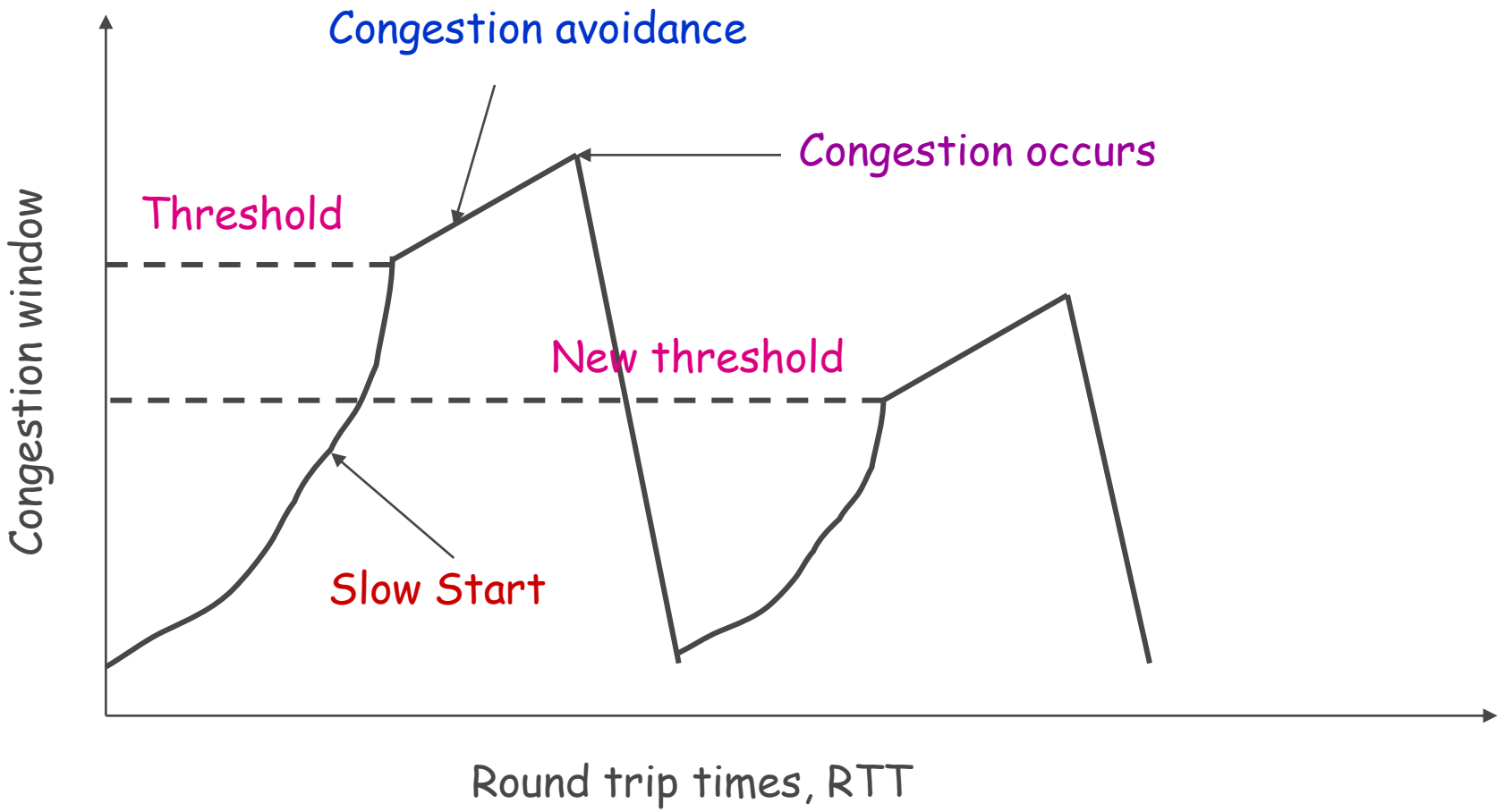
- When connection begins, increase rate exponentially
  - double **CongWin** every RTT
  - done by incrementing **CongWin** for every ACK received
- Summary initial rate is slow but ramps up exponentially fast

Start with CongWin=1, then CongWin=CongWin+1 with every 'Ack'

This leads to 'doubling' of the CongWin with RTT; i.e., exponential increase



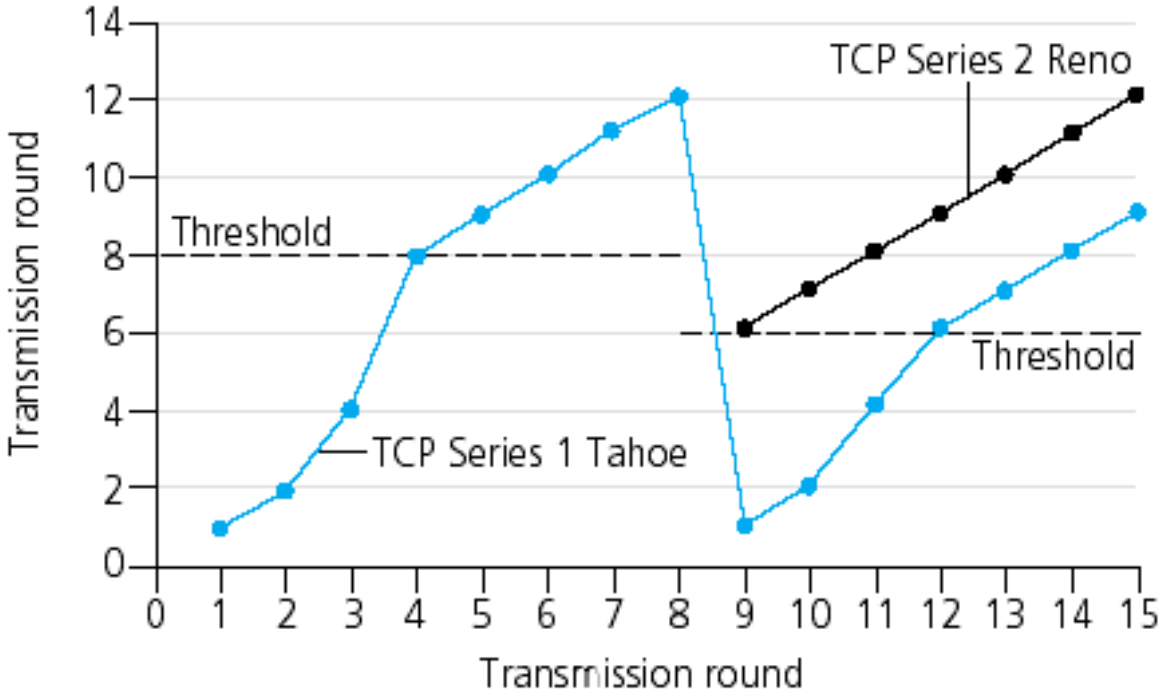
Time Trajectory of CC Phases



Fast Retransmission/Recovery

- Fast retransmit:
  - receiver sends Ack with last in-order segment for every out-of-order segment received
  - when sender receives 3 duplicate ACKs it retransmits the missing/expected segment
- Fast recovery: when 3rd dup Ack arrives
  - ssthresh=CongWin/2
  - retransmit segment, set CongWin=ssthresh
  - Enter congestion avoidance phase, i.e. skip SS

Fast Recovery (Reno Implementation)



3 dup ACKs indicates network capable of delivering some segments, Network is not that badly congested
Timeout indicates a "more alarming" congestion scenario

Summary of TCP Congestion Control