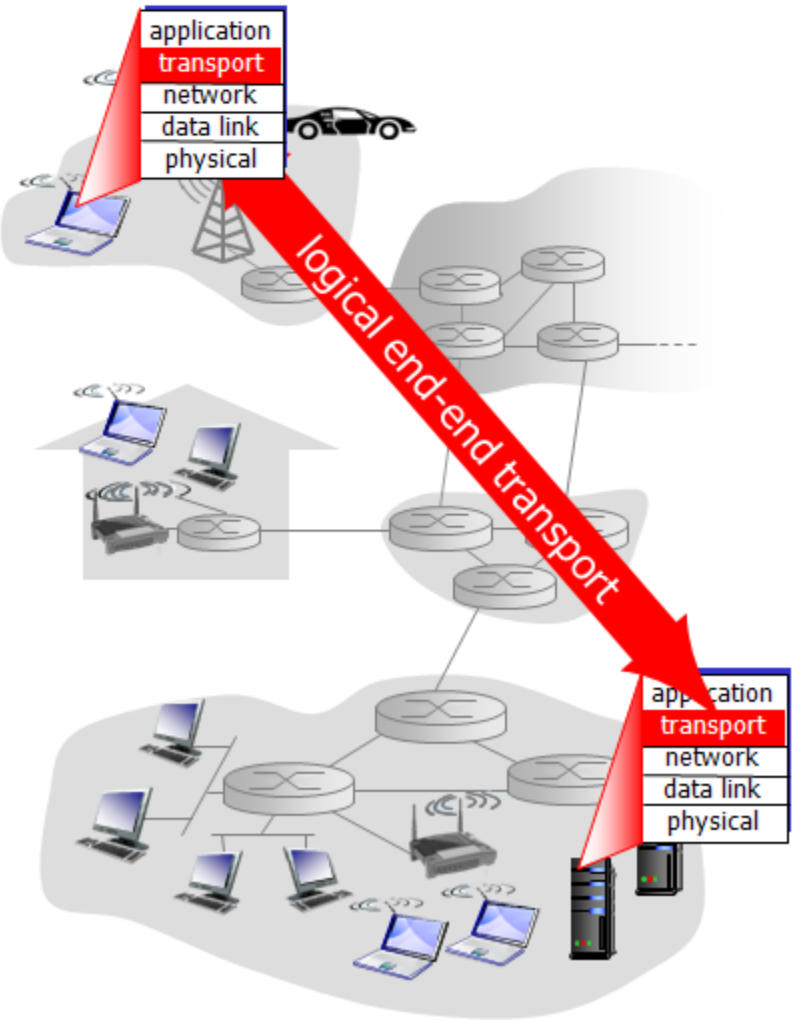
Transport Layer Protocols

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

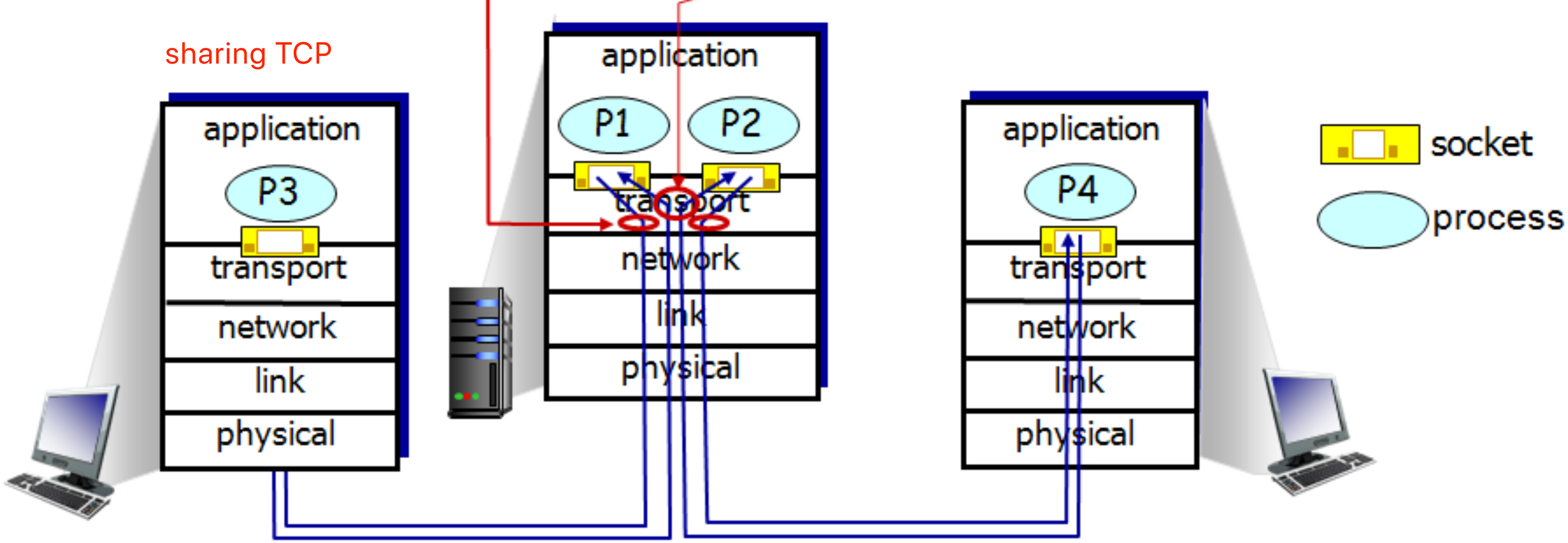
- Functions of the transport layer protocols include:
  - Provide for Process-to-Process communications. To accomplish this task, Port Numbers are used to identify the process, at both the client and at the server side
  - Provide for end-to-end Error Checking (both TCP and UDP), Error Control and Flow 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"

TCP only guarantee no error, because TCP does not run in network,so he cannot guarantee delay or bandwidth(考判断题)

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



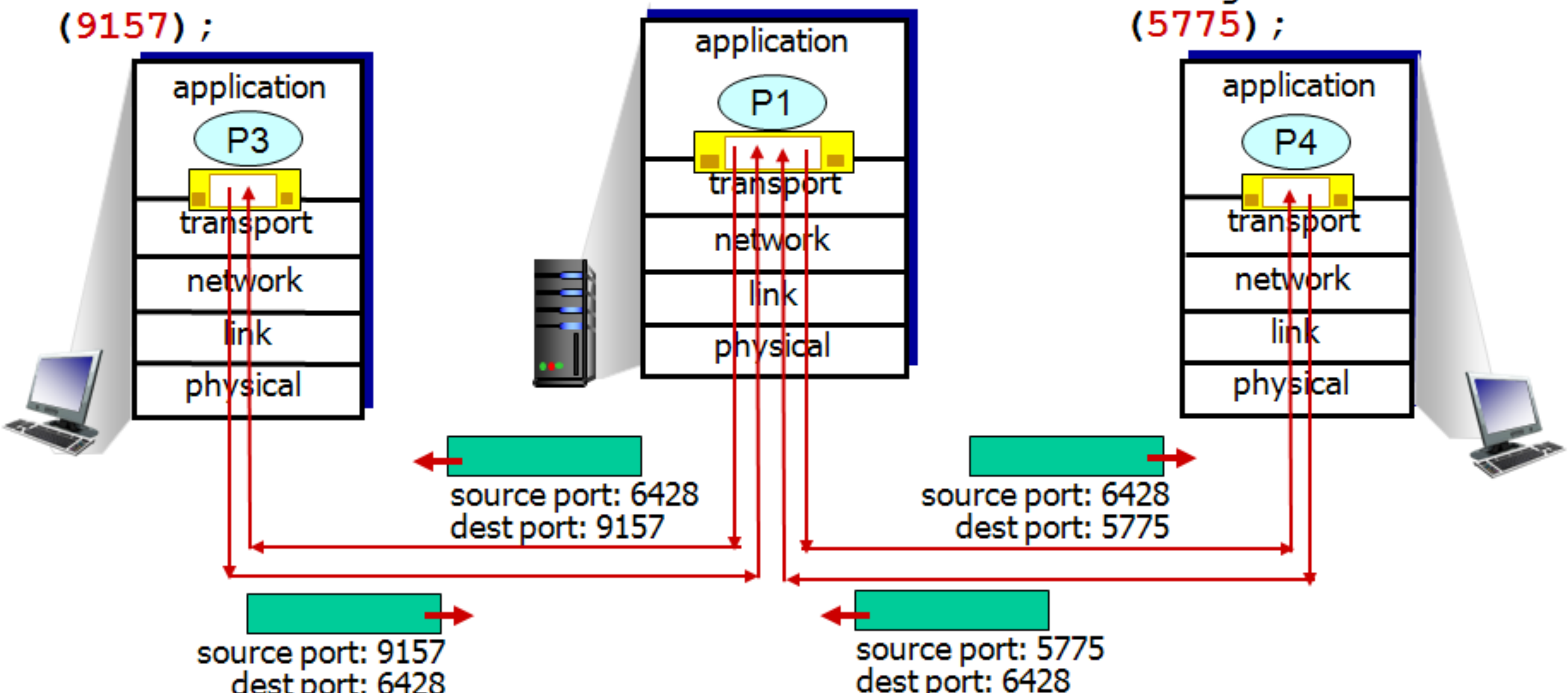
child sockets to transimition
parent sockets to shake hands

Connectionless (UDP) Demultiplexing

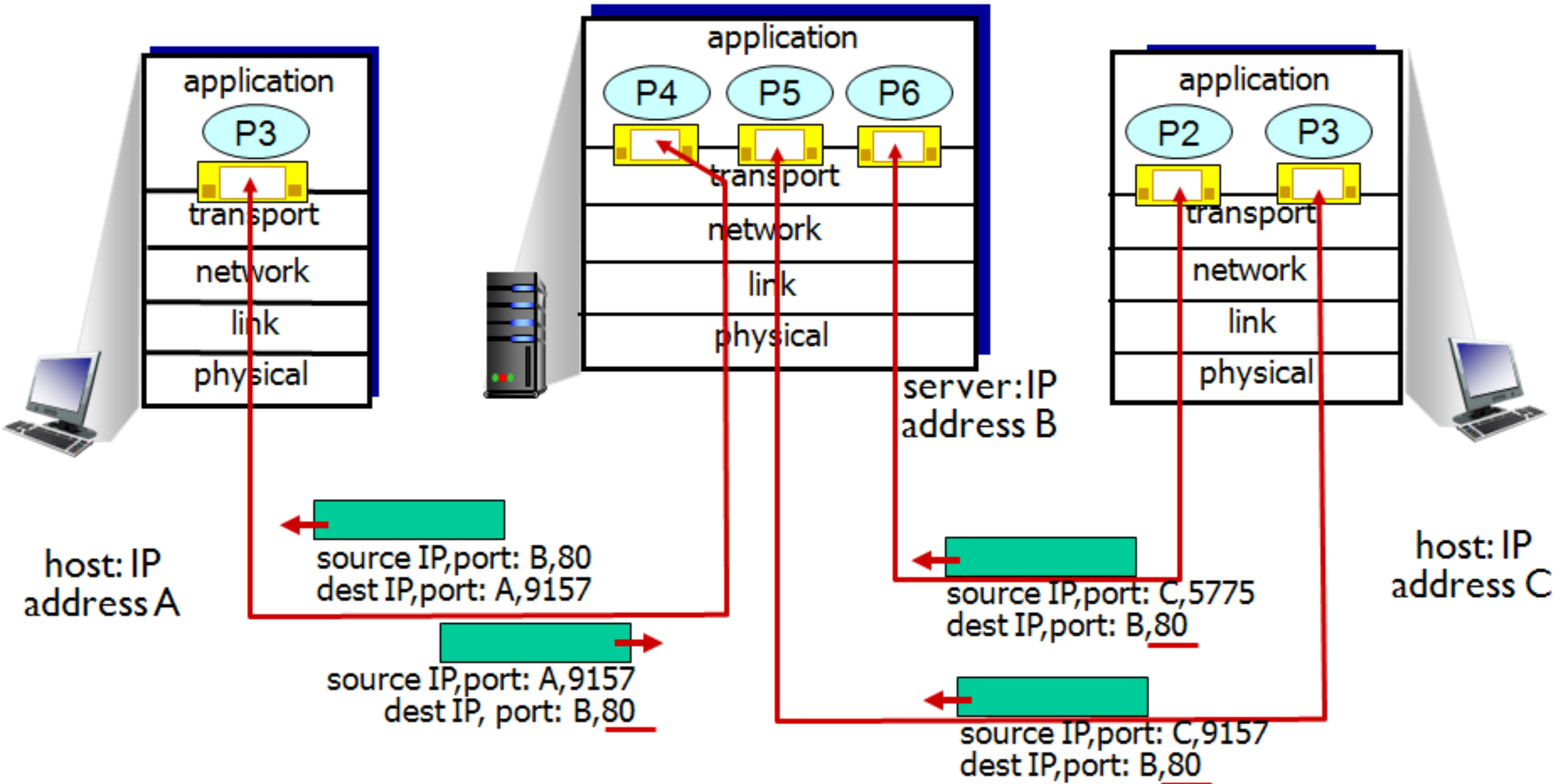
```
DatagramSocket
 mySocket2 = new
 DatagramSocket
 (9157);
```

```
DatagramSocket
 serverSocket = new
 DatagramSocket
 (6428);
```

```
DatagramSocket
 mySocket1 = new
 DatagramSocket
 (5775);
```

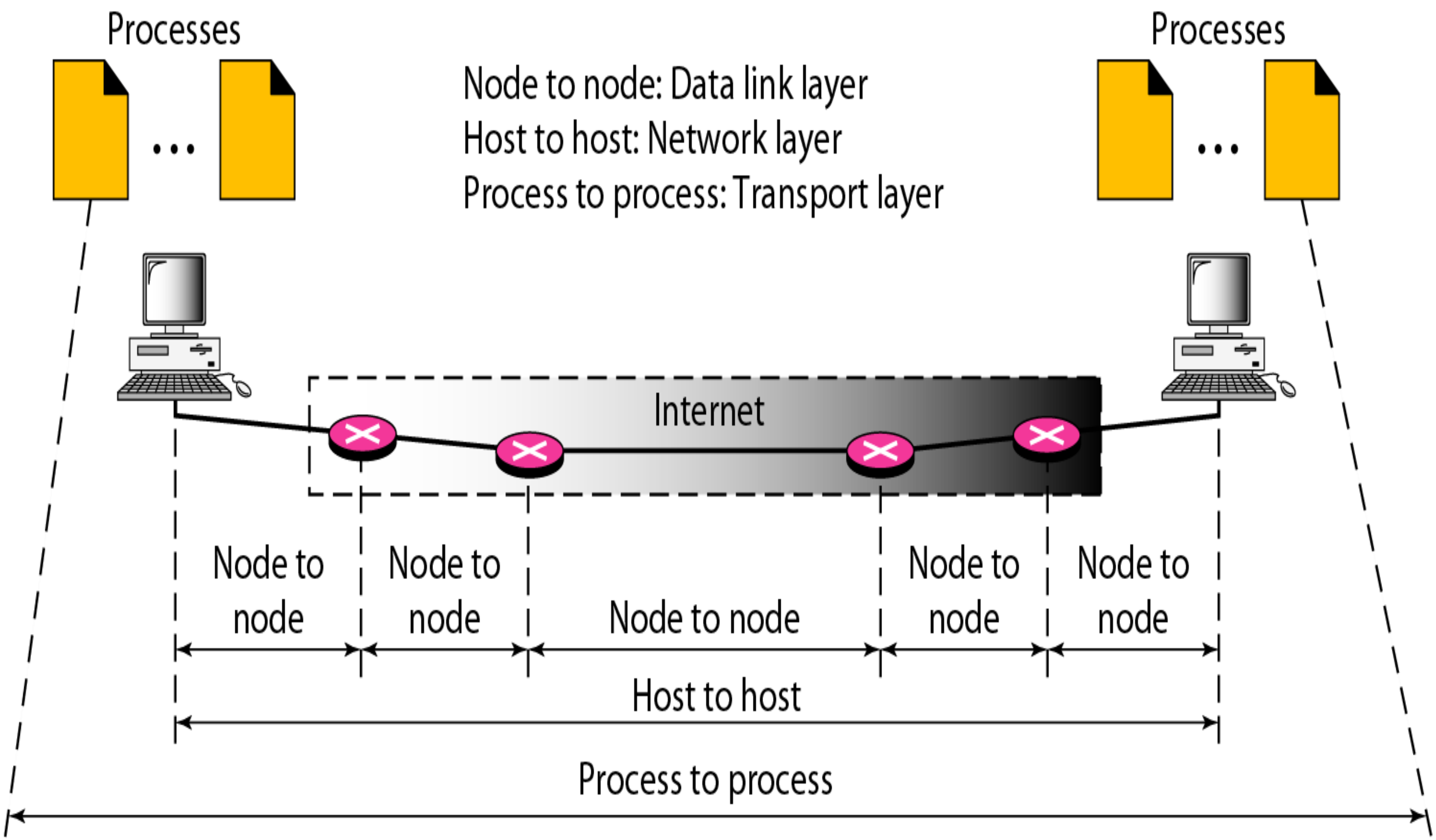


Connection-Oriented (TCP) Demultiplexing



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

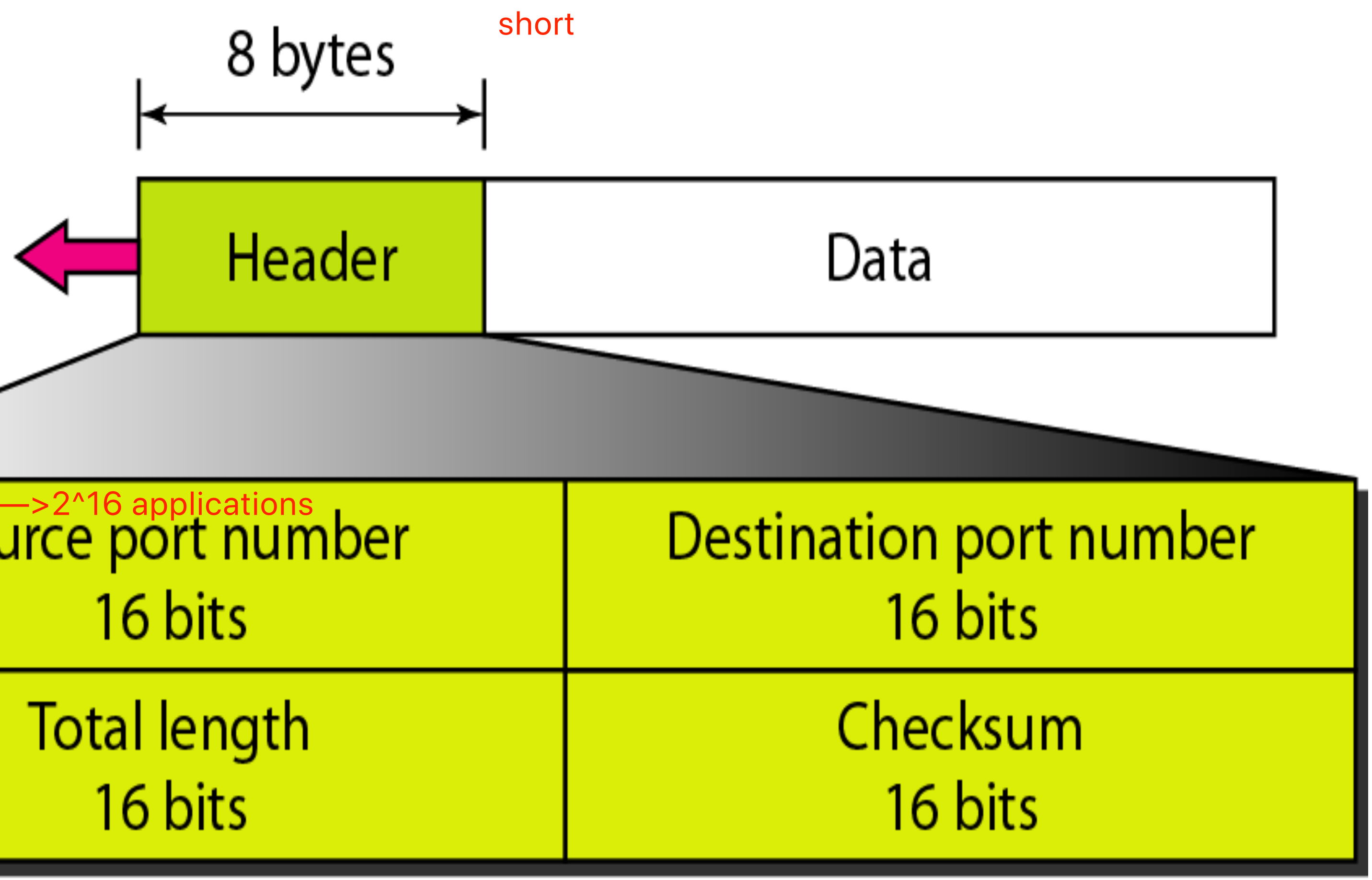
Types of Data Delivery



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 detect error： drop and do nothing
- 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

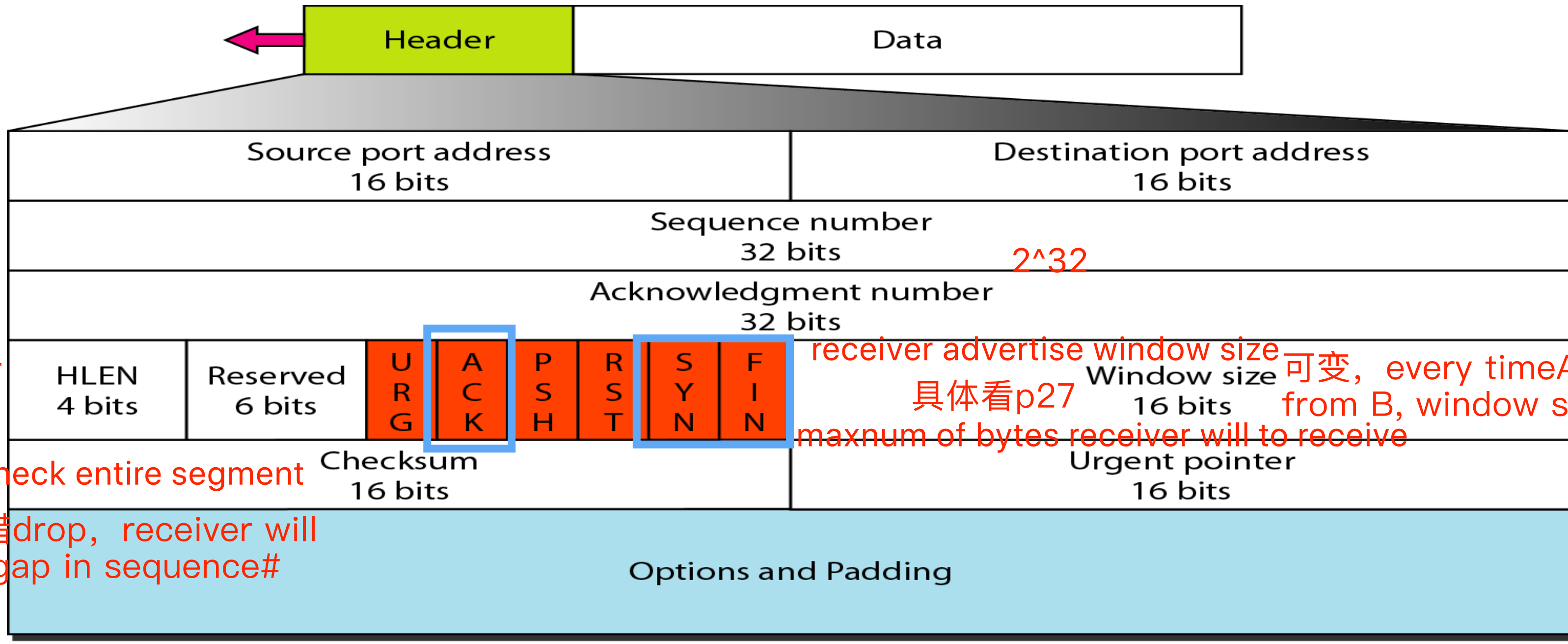
- TCP is a point-to-point, connection-oriented, not running in router reliable, end-to-end protocol that provides
  - Process-to-process communications port number
  - End-to-end error, flow and congestion control
  - FDX service 建立连接之后可以不停传信息
- TCP services is used by
  - Applications that ~~can~~ cannot 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)

ISN: initial sequence number;chosen randomly by NOS
不一定是0，这里的例子是0，ANS确认的是50之前的已经收到

eg: payload:0-49
sequence #: 0

20-40bytes

TCP Segment Format



2^32

receiver advertise window size
可变，every timeA receive from B, window size 变小
具体看p27
maxnum of bytes receiver will to receive

header length

check entire segment

如果有错drop，receiver will detect gap in sequence#

URG: Urgent pointer is valid
ACK: Acknowledgment is valid
PSH: Request for push

只关注画框的3个

RST: Reset the connection
SYN: Synchronize sequence numbers
FIN: Terminate the connection

hand shaking

final

URG	ACK	PSH	RST	SYN	FIN

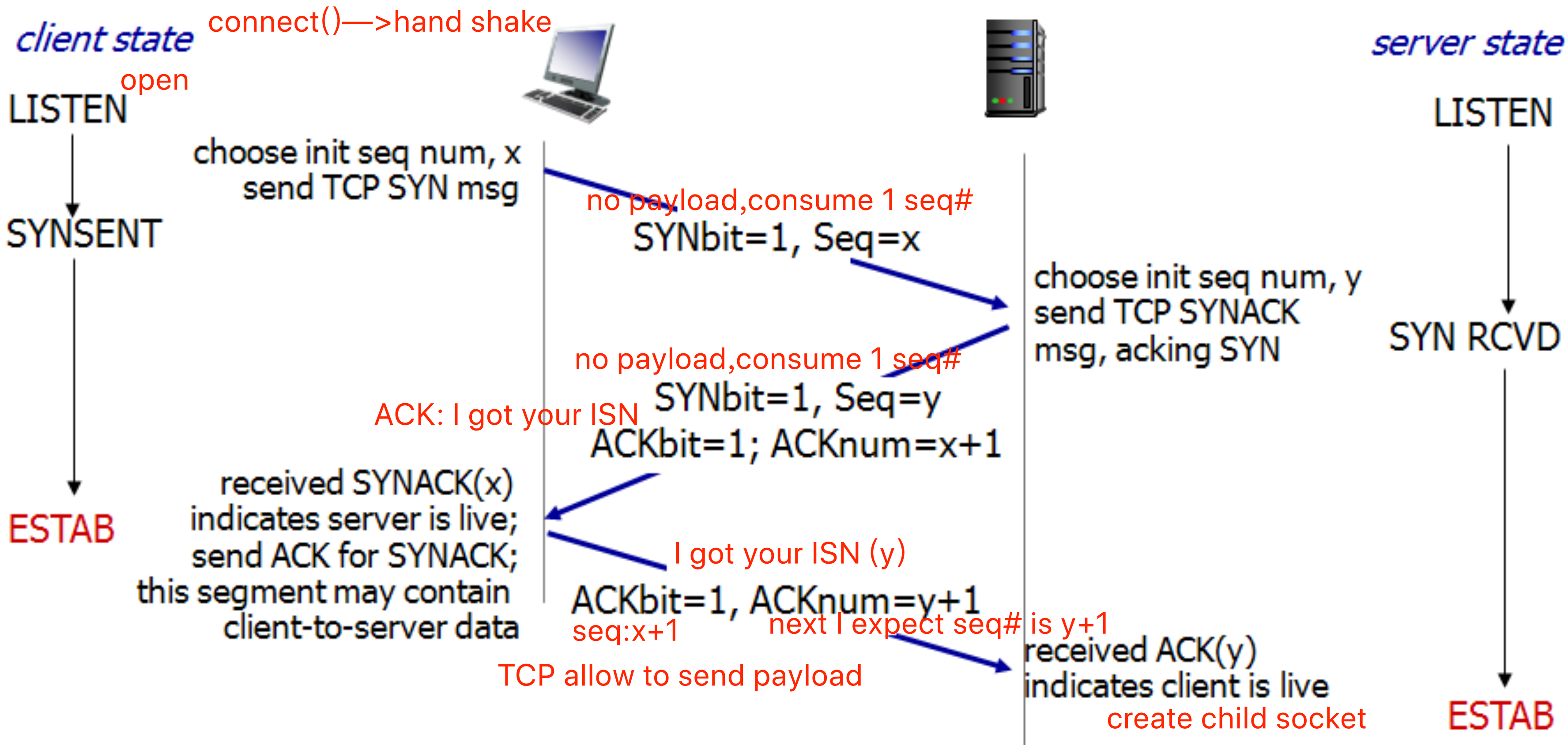
numbering the bytes

- 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

logical connection

在HTTP的时候看作1RTT，实际是1.5RTT，但是最后一个可以send data所以connection可以看作1个RTT

TCP Connection Set-up

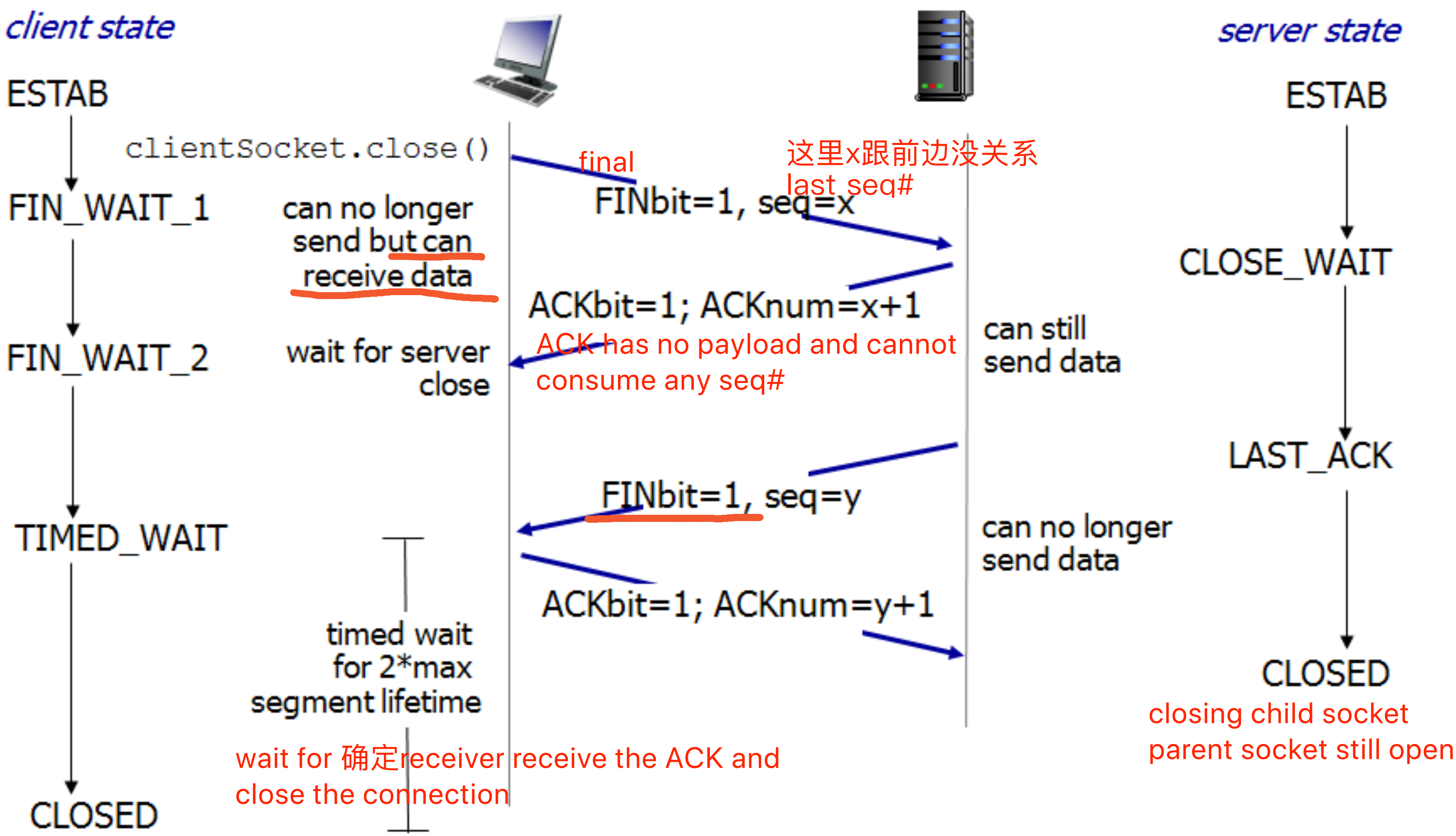


think as an empty bytes

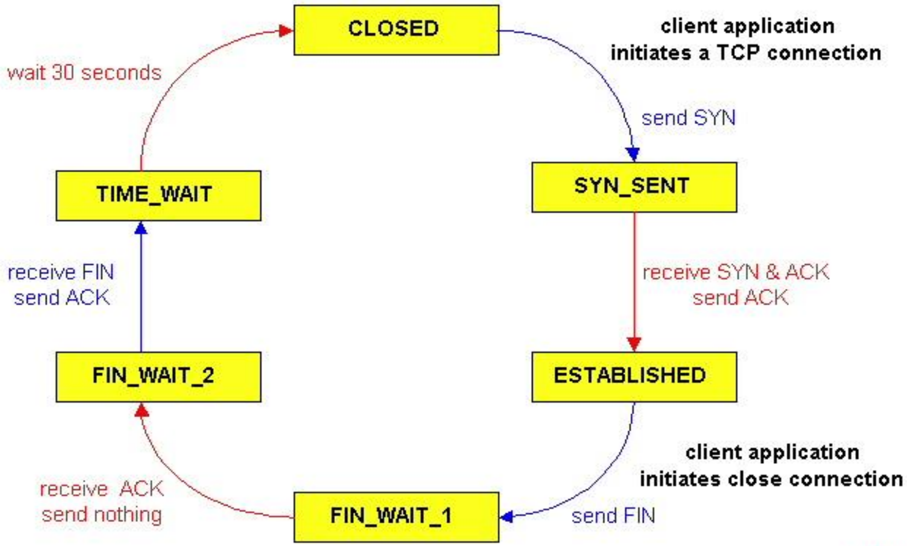
A SYN segment doesn't carry data, but it consumes one sequence number.
A SYN/ACK segment doesn't carry data, but it consumes one sequence number
An ACK segment can carry data

4 ways hand shake结束TCP

TCP Connection Termination

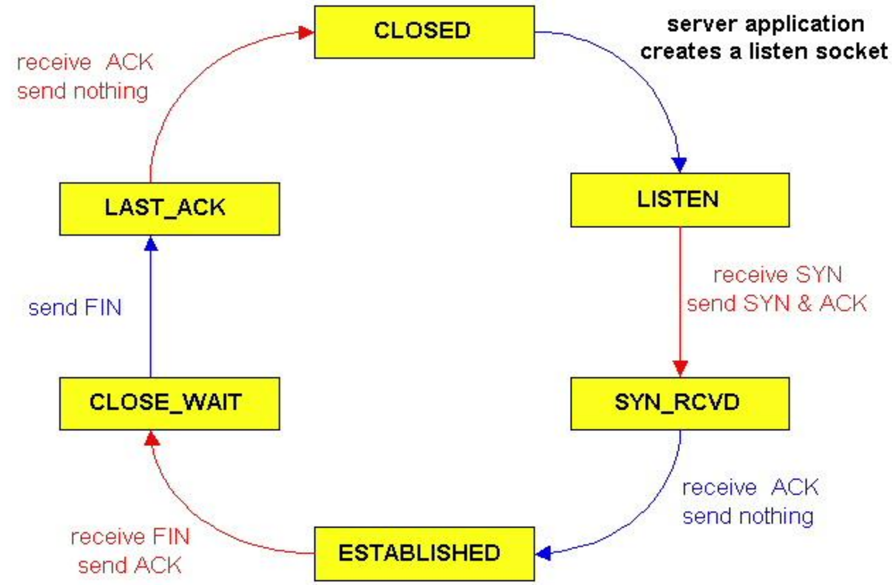


TCP Connection Management



TCP client lifecycle

TCP server lifecycle



Reliability in TCP

- Components of Reliability

TCP able to 1.reorder 2.detect any gap

  - Sequence Numbers/Acknowledgements
  - Retransmissions

if receiver dectect error or gap,he will ask for retransmission

loss or delay->
RTO(retransmission timeout)

  - Timeout Mechanism(s): function of the round trip time (RTT) between the two hosts (is it static?). How to set TCP timeout value?

RTO at least == RTT

    - 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 is random->so RTO need to dynimcally change

if send 10 segments, set only one timeout for the oldest one how sender know the RTT: send segment has the timer,when he receive the ACK of that segment there will be a time too, then he can caculate RTT

不考细节

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

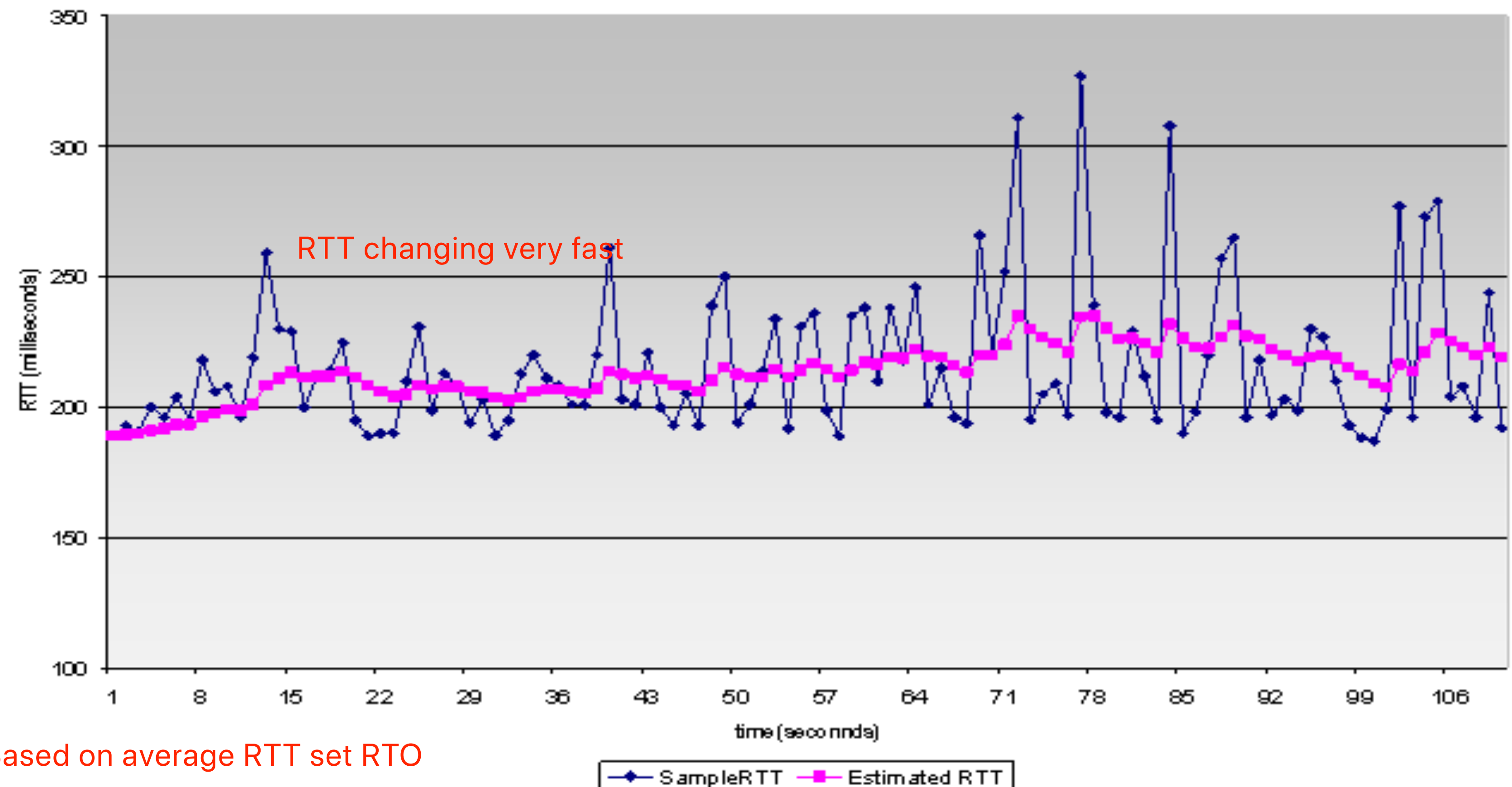
Influence of past sample decreases exponentially fast

Typical value: $\alpha = 0.125$

Set:

TimeoutInterval = EstimatedRTT + 4*DevRTT

Estimation of RTT



Based on average RTT set RTO

take average

TCP Reliable Data Transfer

- TCP creates reliable service on top of IP's unreliable service

can send mutiple segment at the same time

- Pipelined segments
- Cumulative ACKs
- TCP uses single retransmission timer
- Retransmissions are triggered by:
  - Timeout events
  - Duplicate ACKs 3 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:

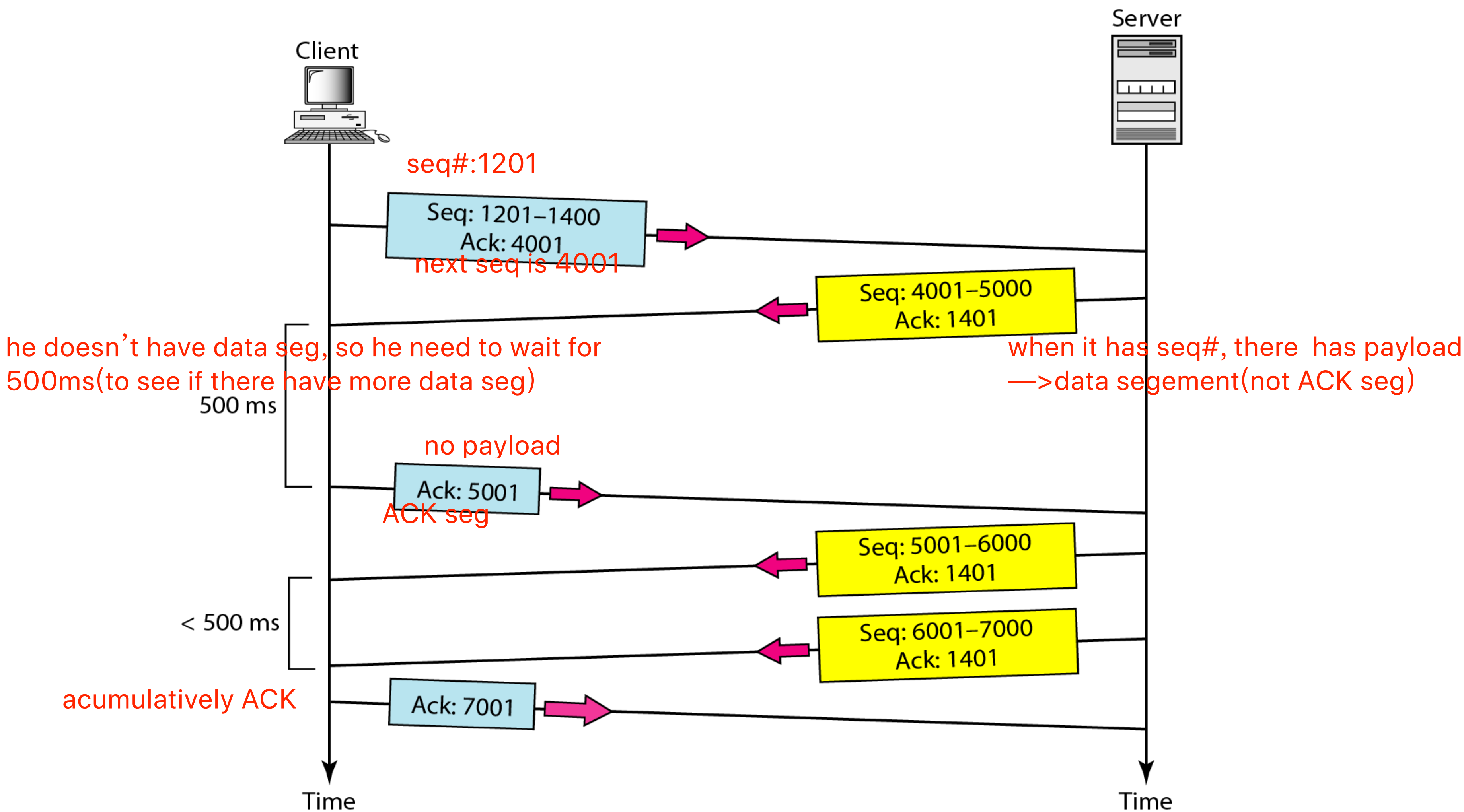
- Retransmit segment that caused timeout
- Restart timer

Ack rcvd:

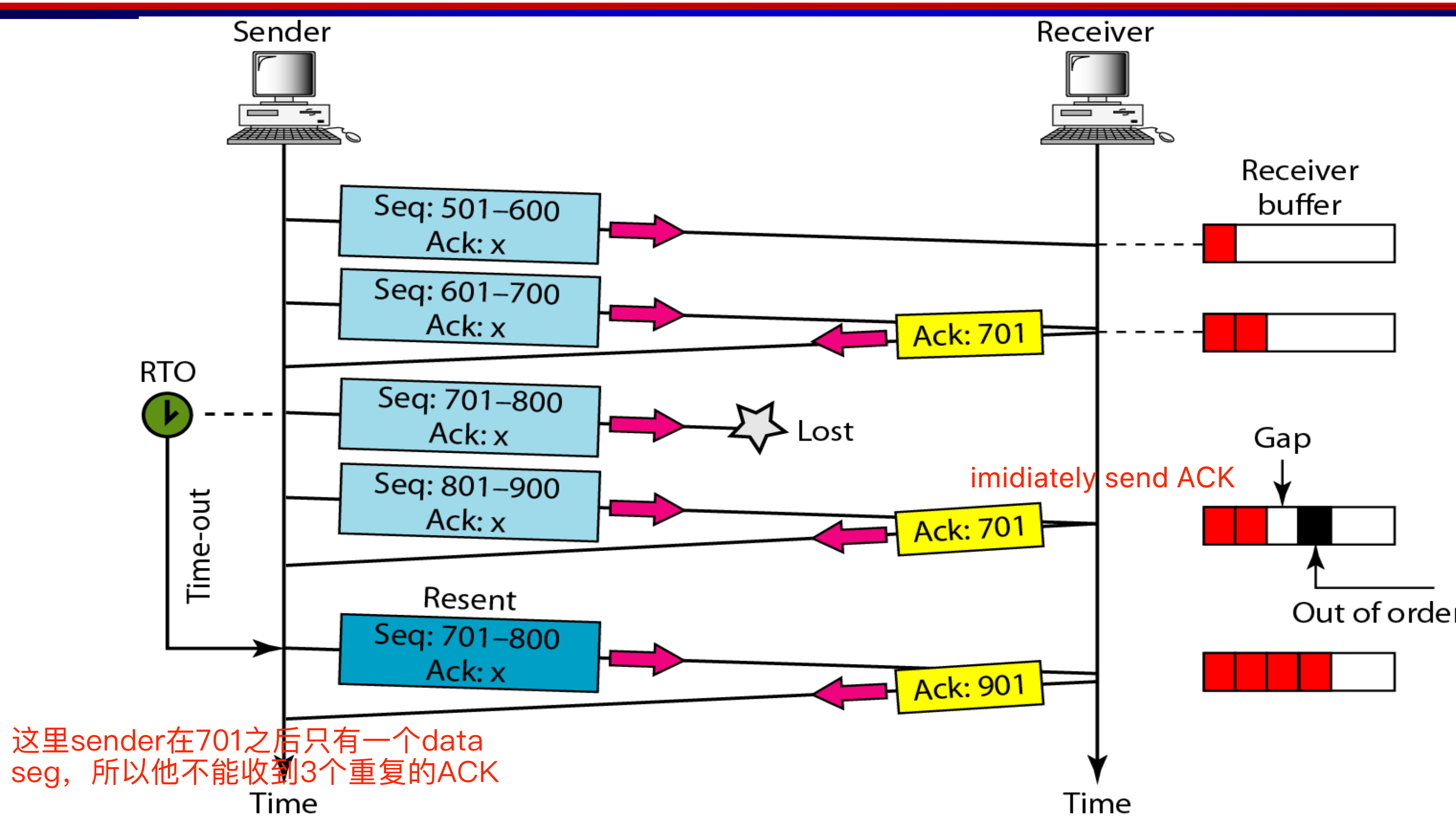
- If acknowledges previously un-Acked segments
  - Update what is known to be Acked
  - Start timer if there are outstanding segments

Anything in order

Normal Operation



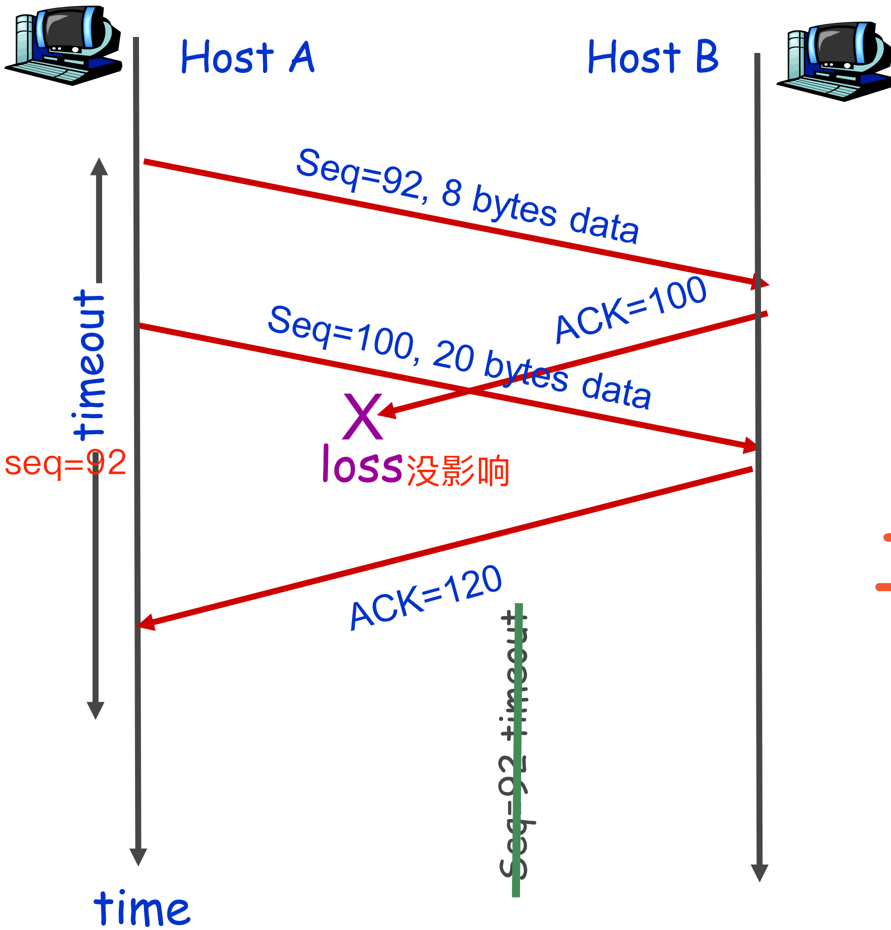
Lost TCP Segment Scenario



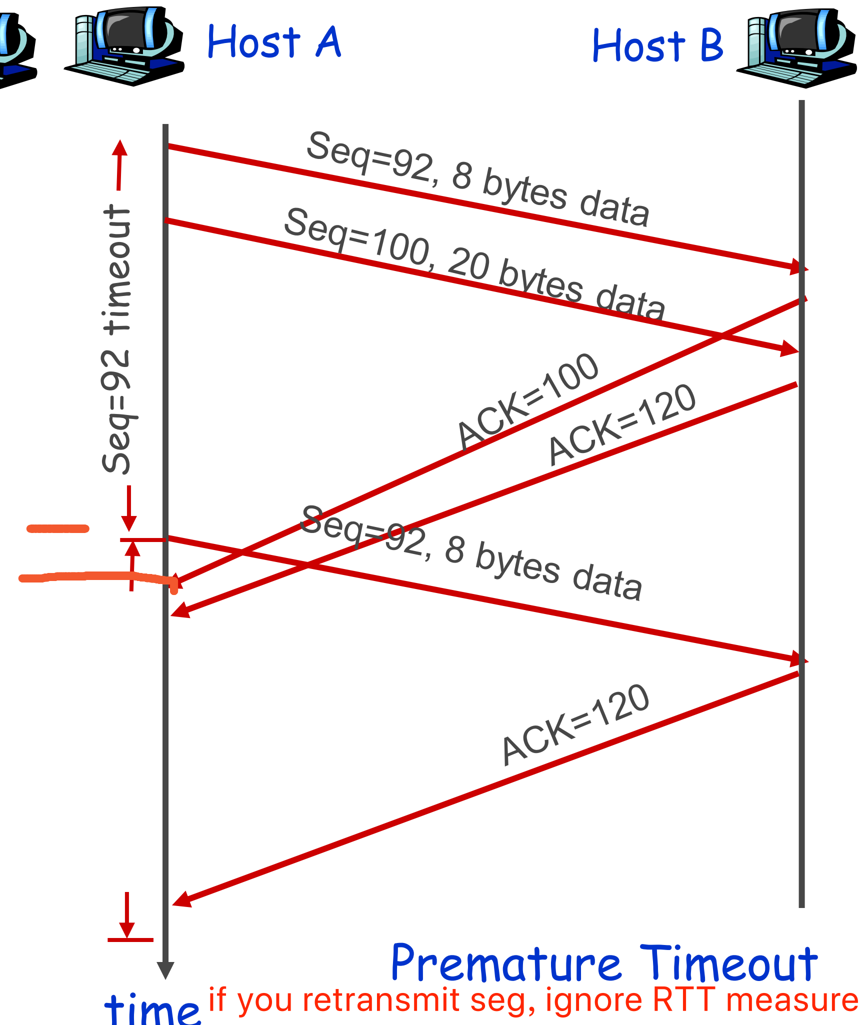
The receiver TCP delivers only ordered data to the process.

No retransmission timer is set for an ACK segment.

Other Scenarios



Cumulative ACK scenario



Premature Timeout

if you retransmit seg, ignore RTT measurement because it will be misleading
（这里sender以为的RTT是从第二个seq92到第一个ACK100）

TCP ACK Generation (RFC 1122, 2581)

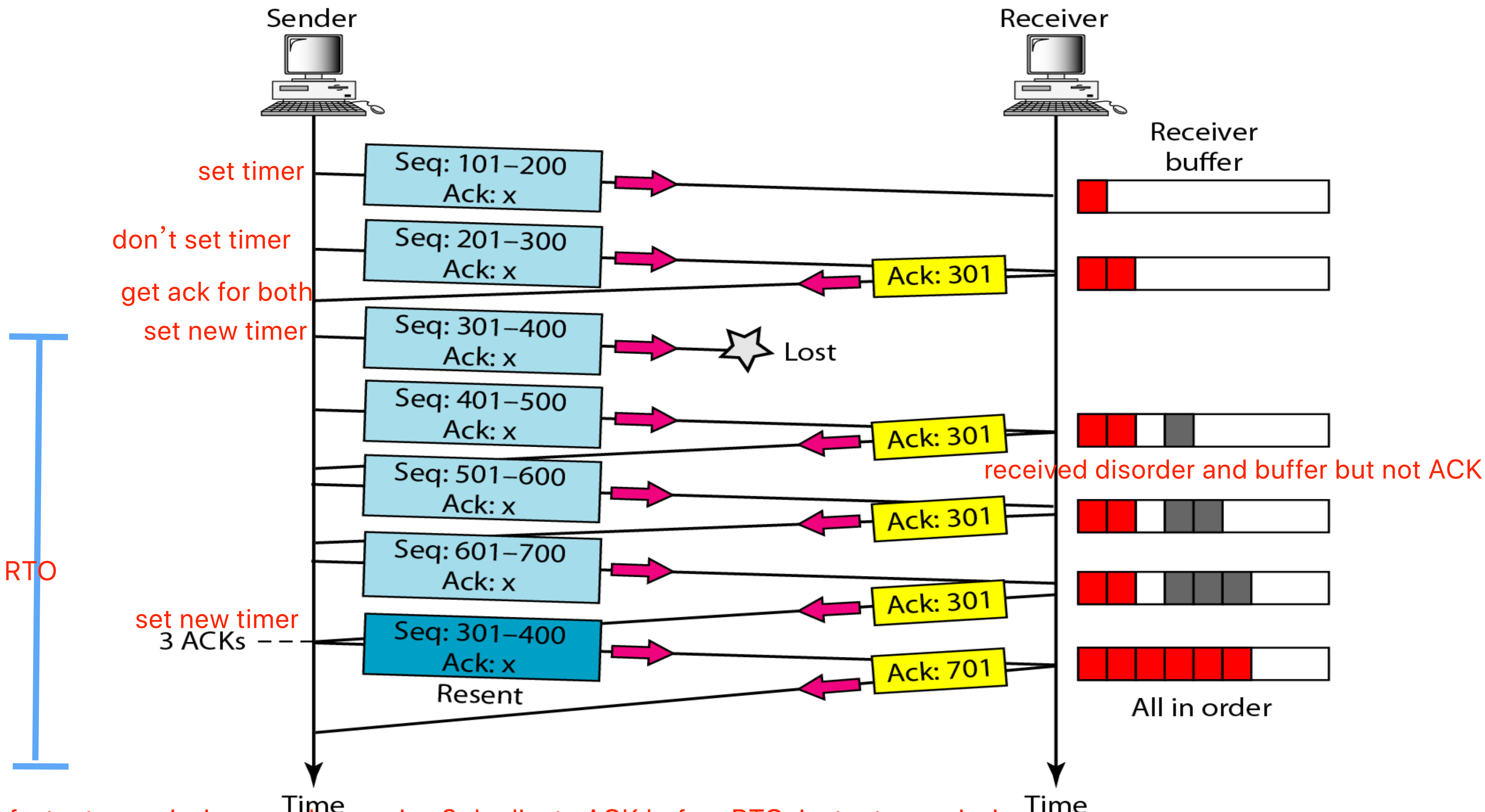
event at receiver	*TCP receiver action*
P20 arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK only when receiver have no data seg
arrival of in-order segment with expected seq #. One other segment has ACK pending	immediately send single cumulative ACK, ACKing both in-order segments
P21 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 gap start place

eg:收到125，gap是3，4，ACK（3）不ACK（4）

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

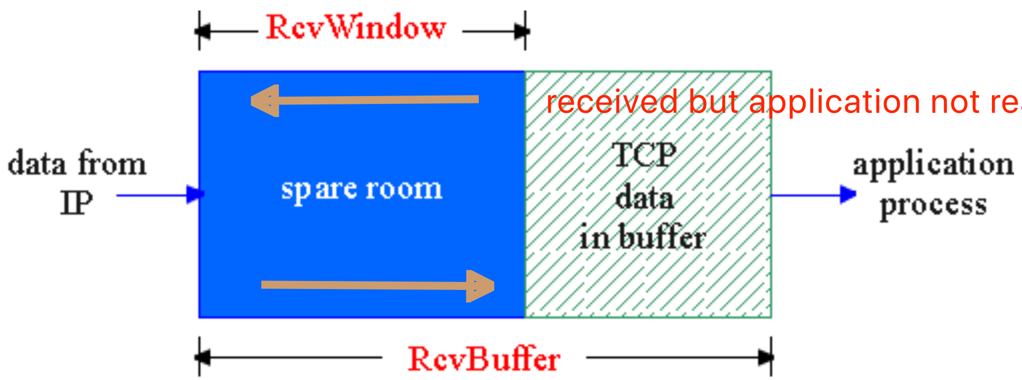


fast retransmission: sender receive 3 duplicate ACK before RTO, just retransmission

与SR ARQ（layer2）区别：
layer2是link to link
layer 4: end to end control

Flow Control in TCP

- Receive side of TCP connection has a receive buffer:



- application process may be slow at reading from buffer

flow control

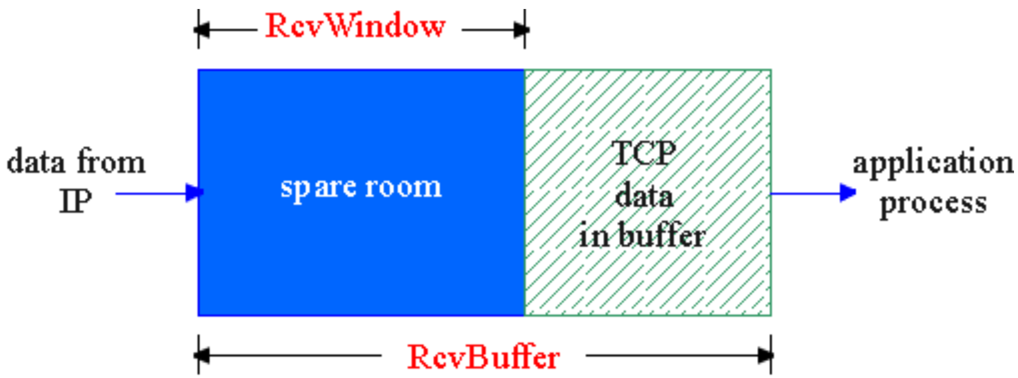
> sender won't overflow receiver's buffer by transmitting too much, too fast

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

receiver advertise window size =
spare room=recv buffer - last byte recv - last byte read

explicit notification from recver to sender

Flow Control in TCP (Cont.)



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

- Spare room in buffer

= **RcvWindow**

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

如果所有sender发的seg都被ACKed，RWS=SWS

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

SWS = max # of bytes the sender can send
SWS<=RWS-[last bytes sent-last bytes ACKed]
inside the bracket represent # of bytes that have already sent but not been ACKed yet
these are called the "inflight" bytes

Principles of Congestion Control

Congestion:

- Informally: “too many sources sending too much data too fast for network to handle”
- Different from flow control!
- Manifestations:

flow control: prevent recver buffer overflow
congestion control: prevent network overflow(for routers)

sender will timeout and send again

  - Lost packets (Buffer overflow at routers)
  - Long delays (queueing in router buffers)
- A top-10 problem in Network Research!

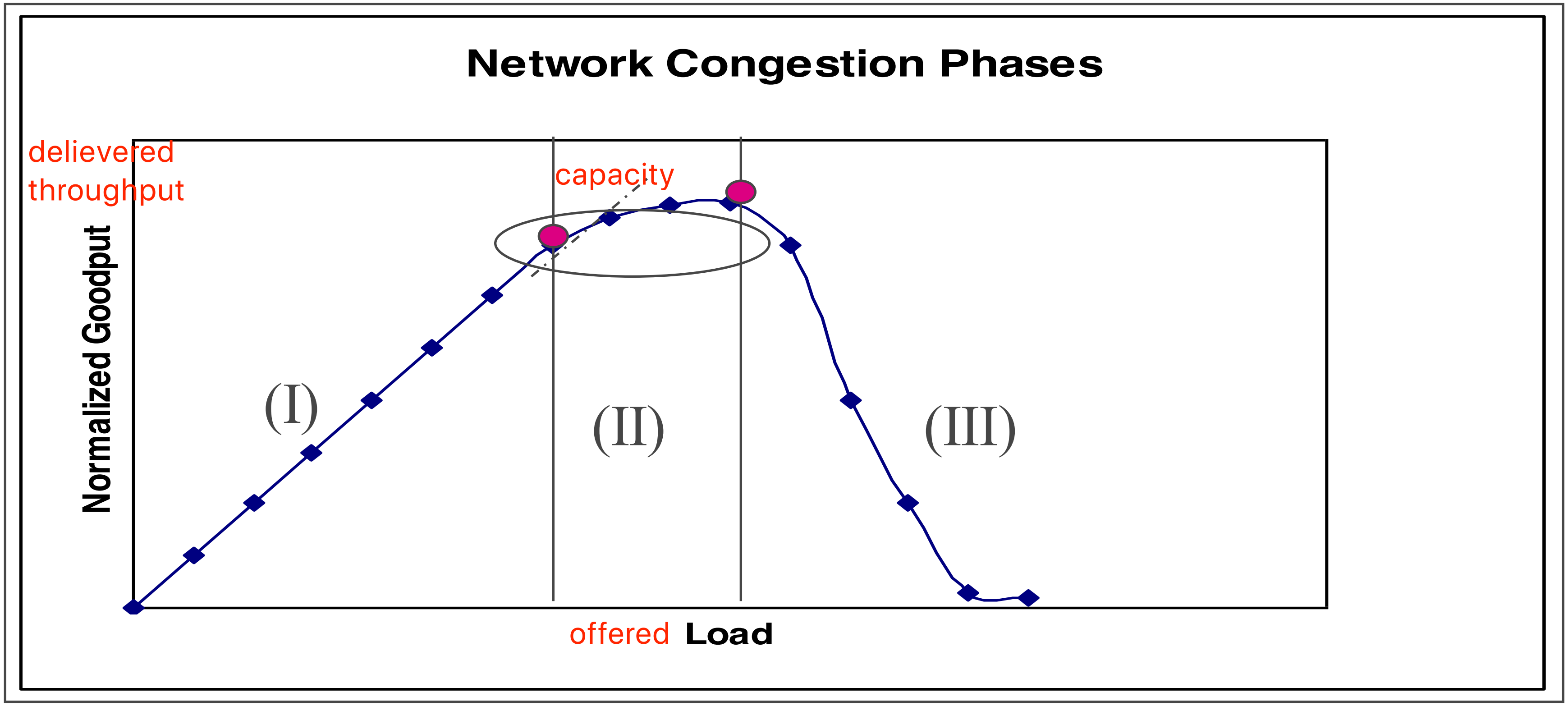
congestion occur:when the incoming rate exceed average service rate

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 sent without congesting the network
- Max # of bytes that can be sent = min (w_a , w_c)

w-# inflight bytes

Network Congestion Phases

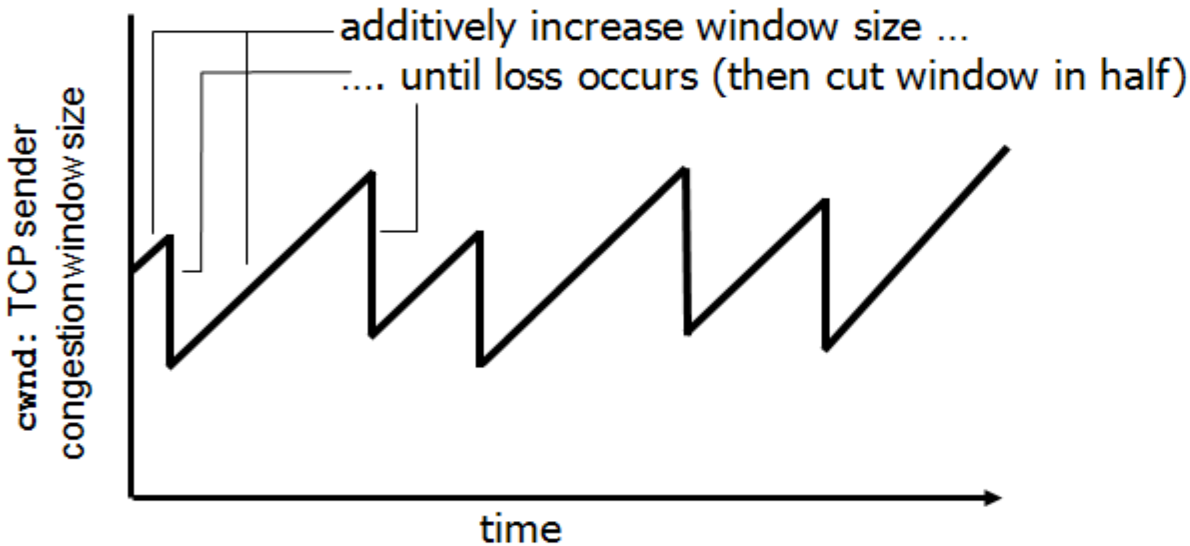


(I) No Congestion
(II) Moderate Congestion
(III) Severe Congestion (Collapse)

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



during the handshake ,both client
and server will agree on MSS

TCP congestion control (tahoe)

~~Slow Start Approach~~ (Tahoe)

set wc=1MSS(Maximum segment size in bytes)

- 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

an ACK is received in one RTT, TCP sender will double Wc

try to test the network

SS threshold(slow start)

- 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

Wc=Wc+1(for every RTT)

Congestion Control (Cont.)

TCP sender will set Wc=1 and set a new SS threshold(Wc/2)

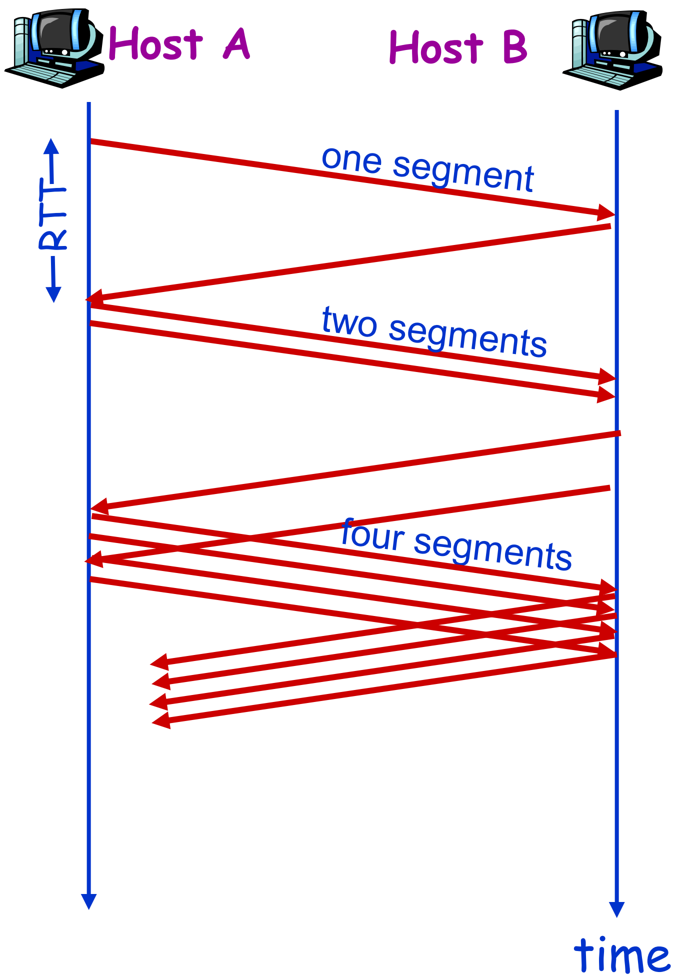
- Phase 3: The congestion window stops increasing when the client TCP detects the network is congested. This happens when an ACK doesn't arrive before the time-out expires. In this phase the congestion threshold is set to 1/2 the current window size which is the min (w_a , w_c). The congestion window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

Slow Start Phase

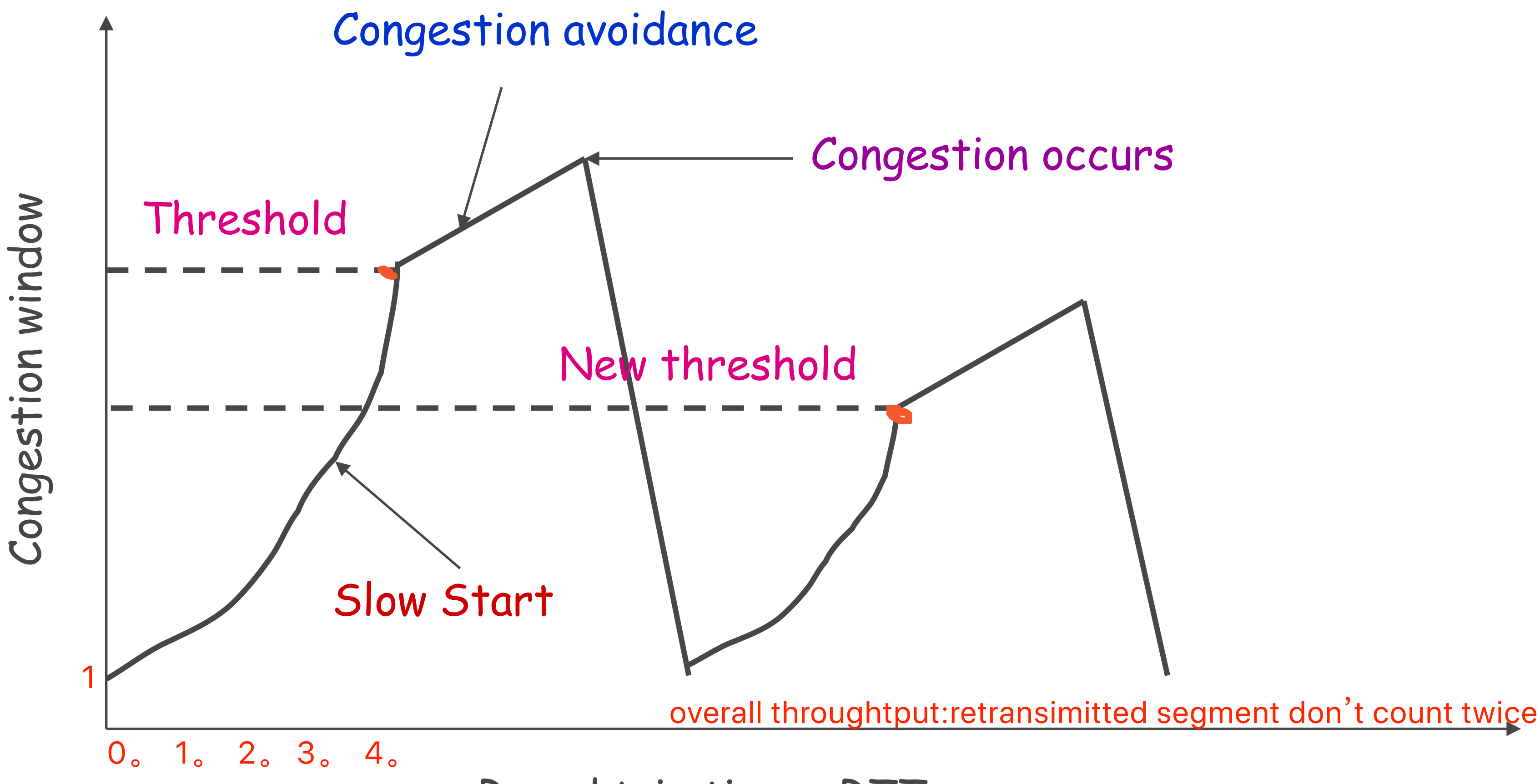
- When connection begins, increase rate exponentially
  - double **CongWin** every RTT
  - done by incrementing **CongWin** for every ACK received
- <u>Summary</u> initial rate is slow but ramps up exponentially fast

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



1 means the end of first RTT and the begining of the second RTT

final 要画这个图

Reno doesn't treat the

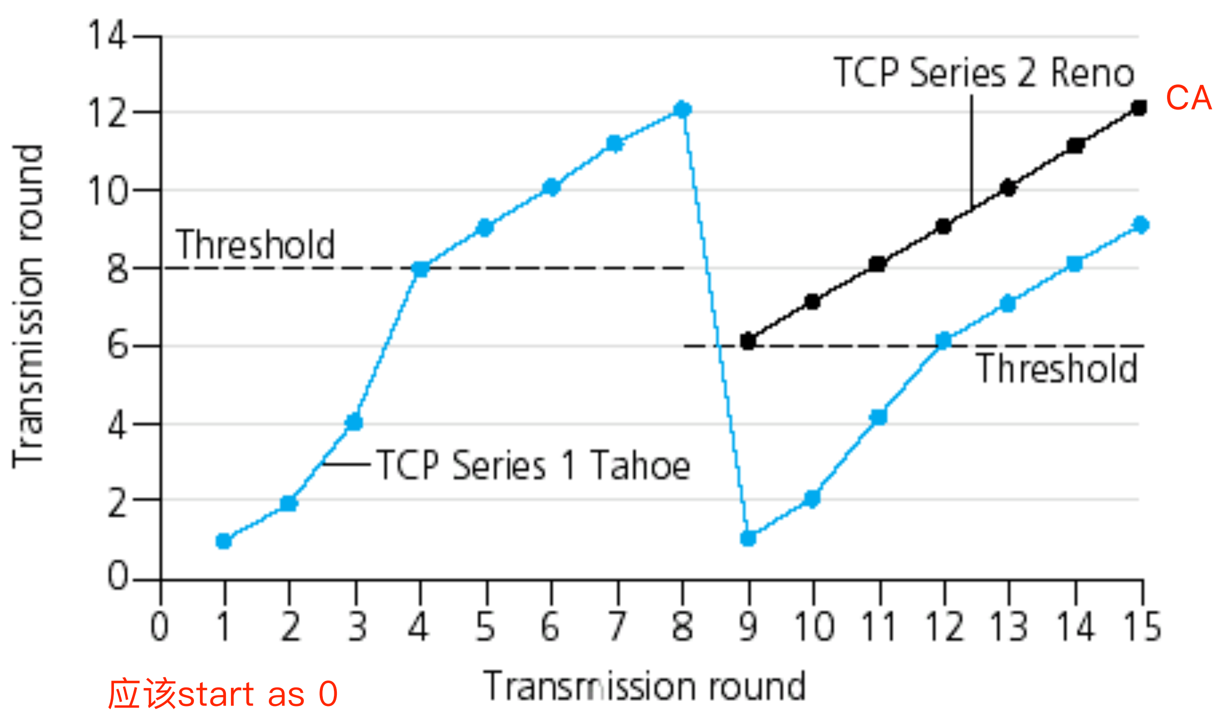
Fast Retransmission/Recovery

Reno

- 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

if timeout====>Reno behaveior is simliar to Tahoe
but if the sender received 3 duplicate ACK, he will set his Wc=Wc/2+3(这里是之前disorder的3个已经发的segment)

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