Data Communication and Computer Networks

5. Transport Layer
PART-B

Dr. Aiman Hanna
Department of Computer Science & Software Engineering
Concordia University, Montreal, Canada

These slides have mainly been extracted, modified and updated from original slides of :
Computer Networking: A Top Down Approach, 6th edition Jim Kurose, Keith Ross
Addison-Wesley, 2013

Additional materials have been extracted, modified and updated from:
Understanding Communications and Networking, 3e by William A. Shay 2005

TCP: Overview RFCs: 793,1122,1323, 2018, 2581

- point-to-point:
  - only one sender, one receiver
- reliable, in-order *byte steam:*
  - no loss or alteration of data
- pipelined:
  - TCP congestion and flow control set window size
- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size; controls amount of sent data (excluding headers)
- connection-oriented:
  - 3-way handshake (exchange of control msgs) before data exchange
- flow controlled:
  - sender will not overwhelm receiver

TCP segment structure

URG: urgent data
(generally not used)

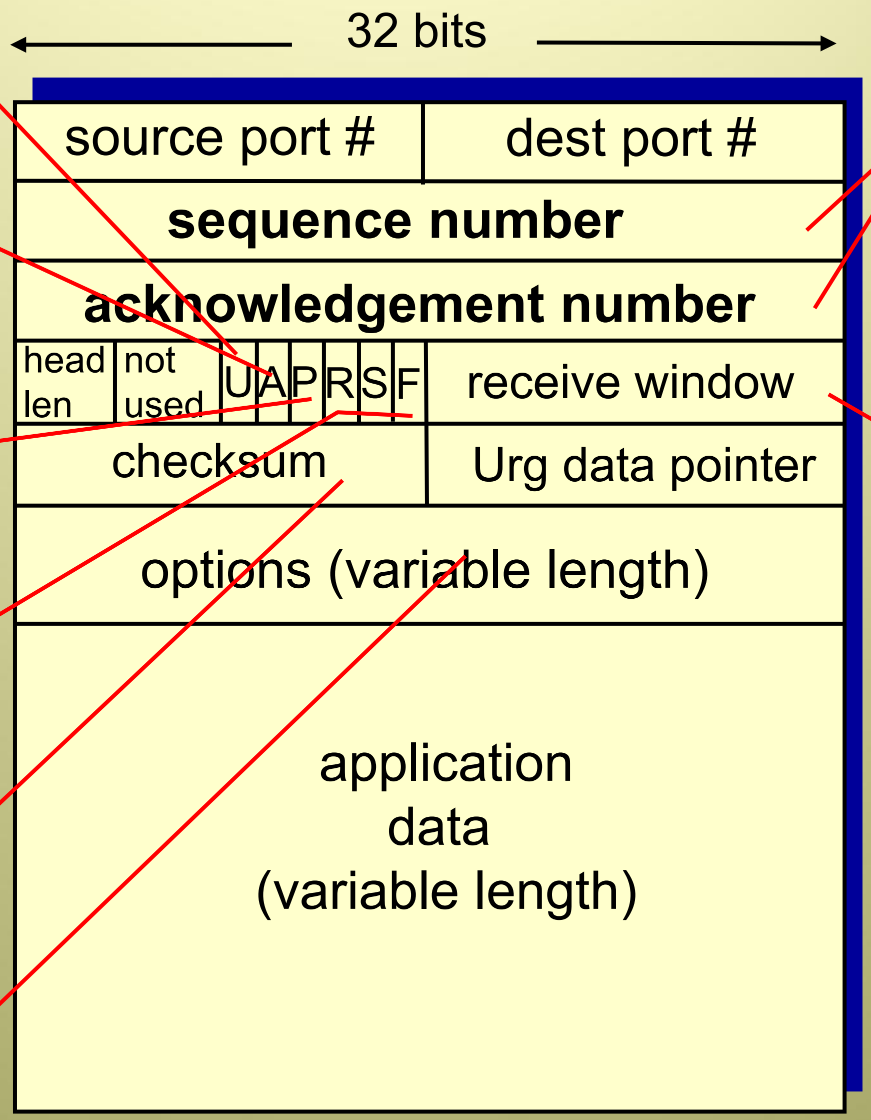
ACK: ACK #
valid indicating segment
successfully received

PSH: for receiver to
push data immediately
to upper layer
(generally not used)

RST, SYN, FIN:
Connection estbl.
(setup, teardown
commands)

Internet checksum
(as in UDP)

used for sender and
receiver negotiation
of MSS



counting
by bytes
of data
(not segments!)

bytes
rcvr willing
to accept;
Needed for flow
control

TCP seq. numbers, ACKs

sequence numbers:

- byte stream "number" of first byte in segment's data
- Ex: sender wishes to send 500,000 bytes, where MSS is 1000

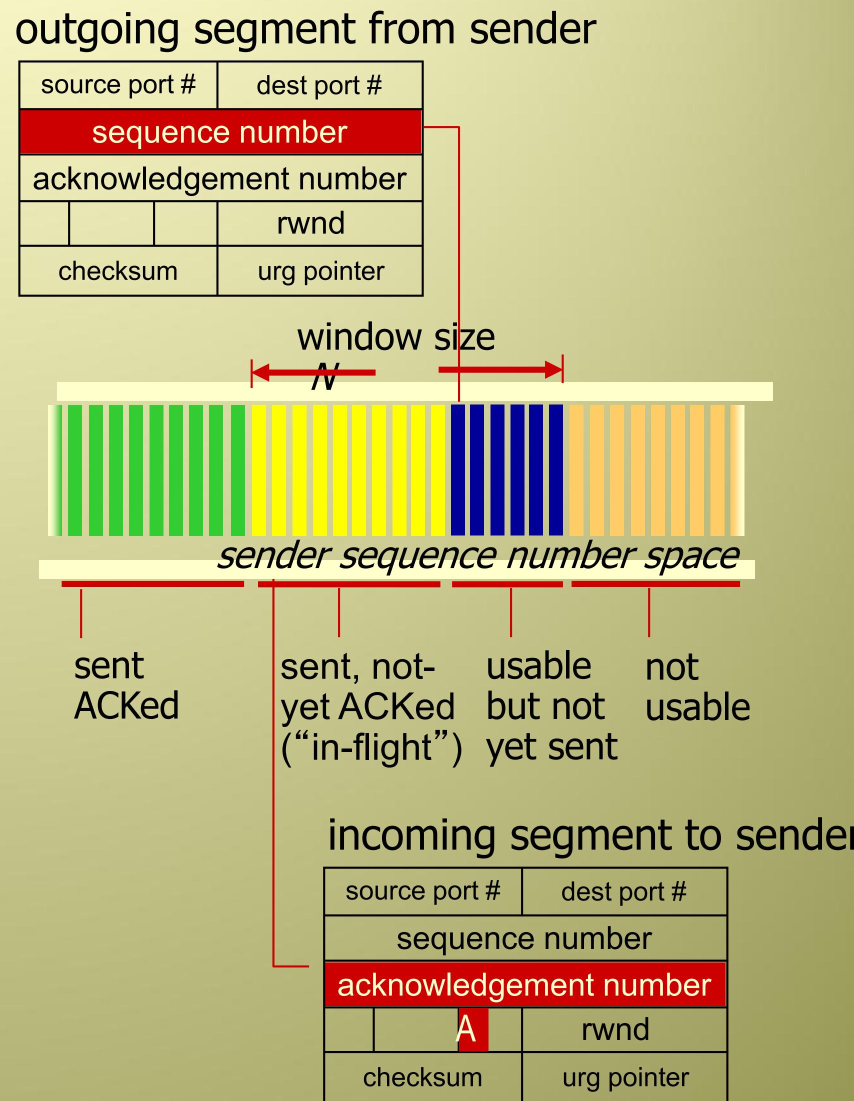
  →Sequence #s are: 0, 1000, 2000, …, 499,000

→acknowledgements:

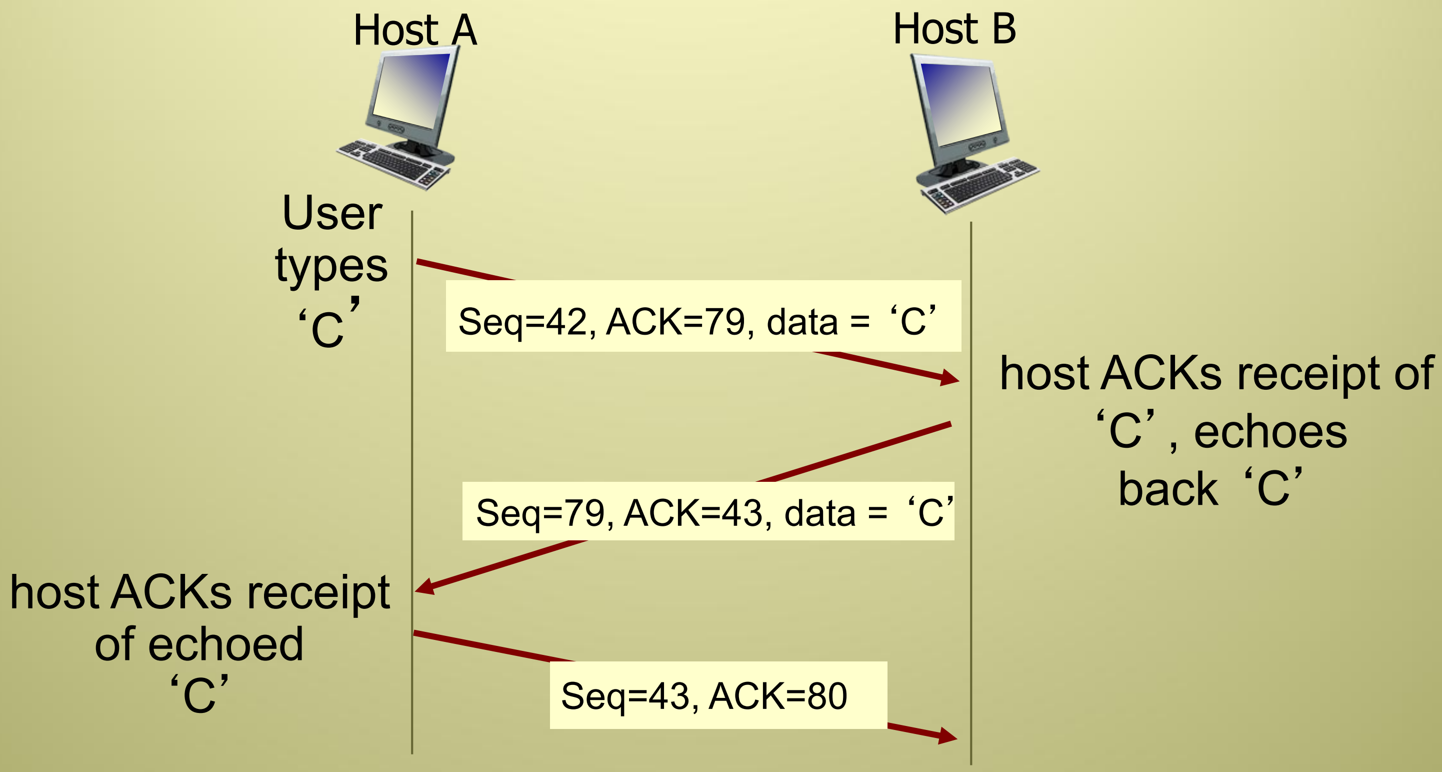
- seq # of **next** byte expected from other side
- **cumulative** ACK

Q: how receiver handles out-of-order segments

- A: TCP spec doesn't say, - up to implementer



TCP seq. numbers, ACKs



simple *Telnet* scenario

TCP round trip time, timeout

Q: how to set TCP timeout value?

- must be longer than RTT
  - but RTT varies
- *too short:* premature timeout, unnecessary retransmissions
- *too long:* slow reaction to segment loss

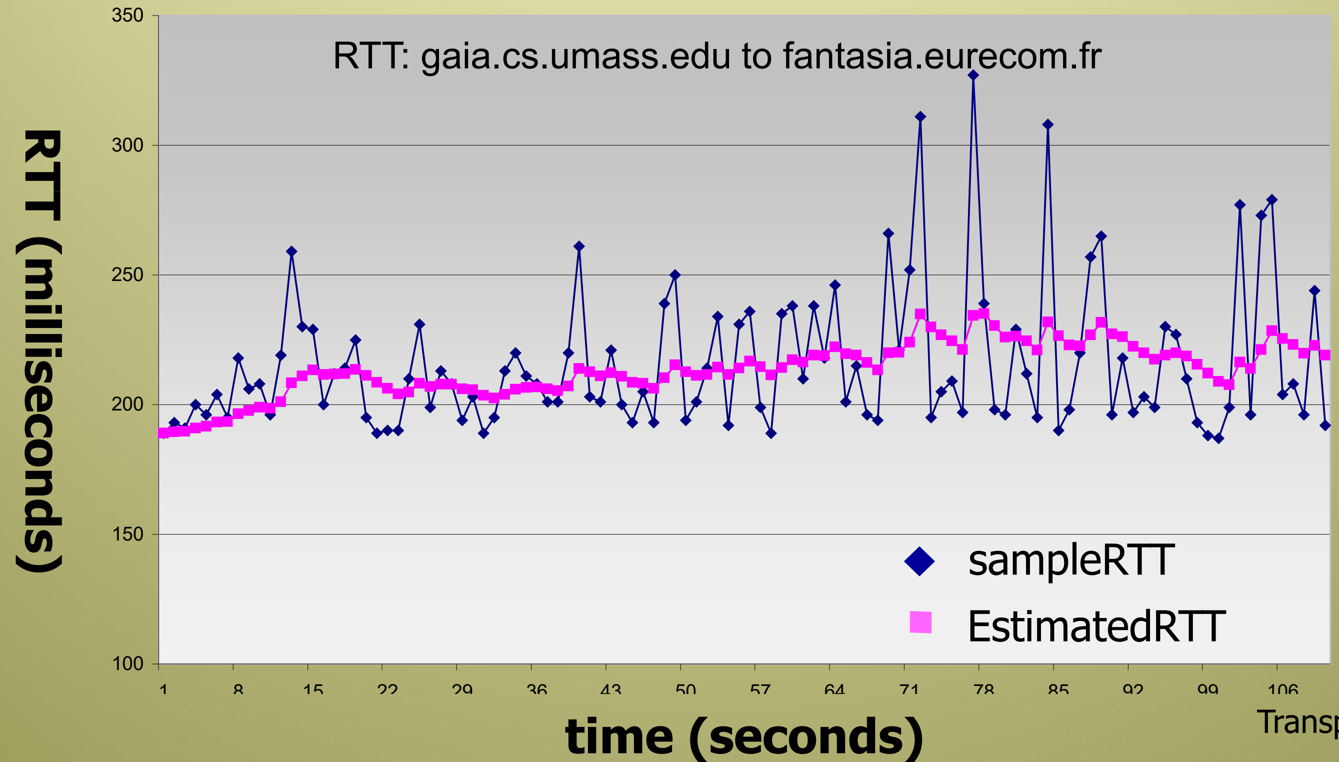
Q: how to estimate RTT?

- **`SampleRTT`**: measured time from segment transmission until ACK receipt
  - ignore retransmissions
- **`SampleRTT`** will vary, want estimated RTT "smoother"
  - average several *recent* measurements, not just current **`SampleRTT`**

TCP round trip time, timeout

`EstimatedRTT = (1- α)* EstimatedRTT + α * SampleRTT`

- exponential weighted moving average
  - weight of a given sample decays exponentially fast as updates proceed
- typical value: $\alpha = 0.125$; hence much more load is given to last sample



TCP round trip time, timeout

- timeout interval: **`EstimatedRTT`** plus "safety margin"
  - large variation in **`EstimatedRTT`** -> larger safety margin
- estimate SampleRTT **deviation** from EstimatedRTT:

$$\texttt{DevRTT} = (1-\beta)*\texttt{DevRTT} + \beta*|\texttt{SampleRTT}-\texttt{EstimatedRTT}|$$

(typically, $\beta = 0.25$)

`TimeoutInterval = EstimatedRTT + 4*DevRTT`

estimated RTT

"safety margin"

TCP reliable data transfer

- TCP creates rdt service on top of IP's unreliable service
  - pipelined segments
  - cumulative acks
  - single retransmission timer
- retransmissions triggered by:
  - timeout events
  - duplicate acks

let's initially consider simplified TCP sender:

- ignore duplicate acks
- ignore flow control and congestion control

TCP sender events:

data rcvd from app:

- create segment with seq #
- seq # is byte-stream number of first data byte in segment
- start timer if not already running
  - think of timer as for oldest unacked segment
  - expiration interval: **`TimeOutInterval`**

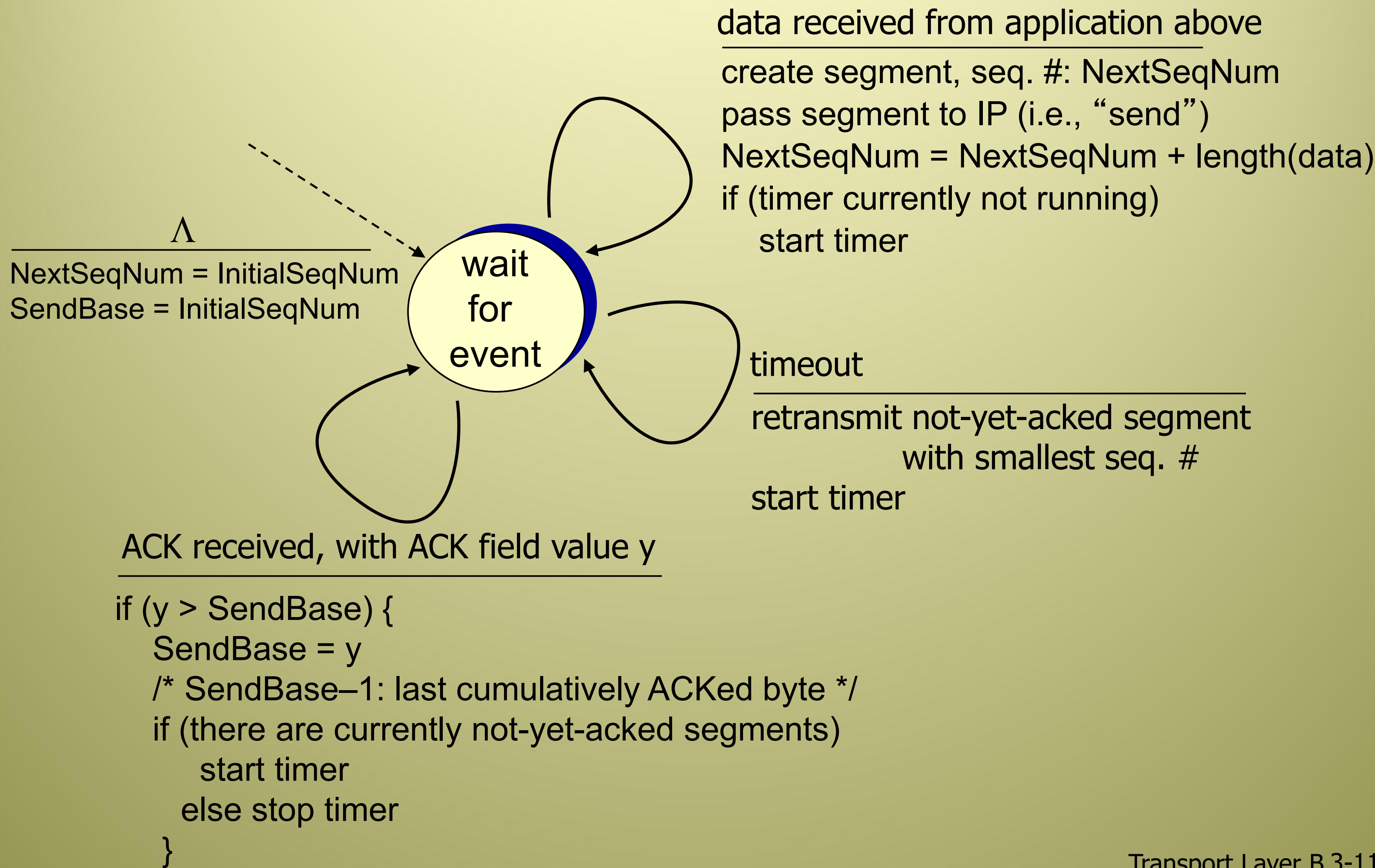
timeout:

- retransmit segment that caused timeout
- restart timer

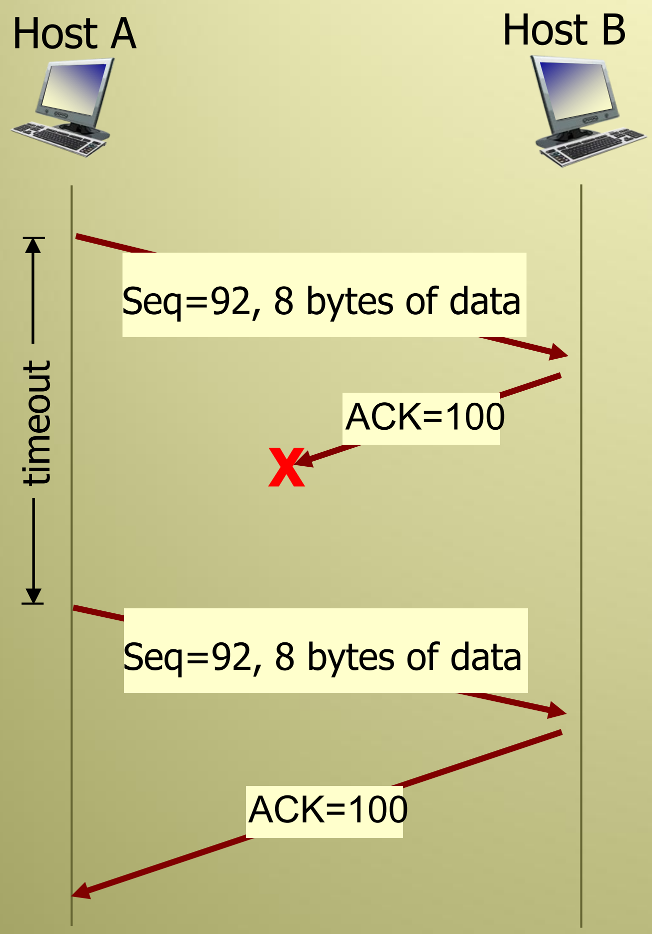
ack rcvd:

- if ack acknowledges previously unacked segments
  - update what is known to be ACKed
  - start timer if there are still unacked segments

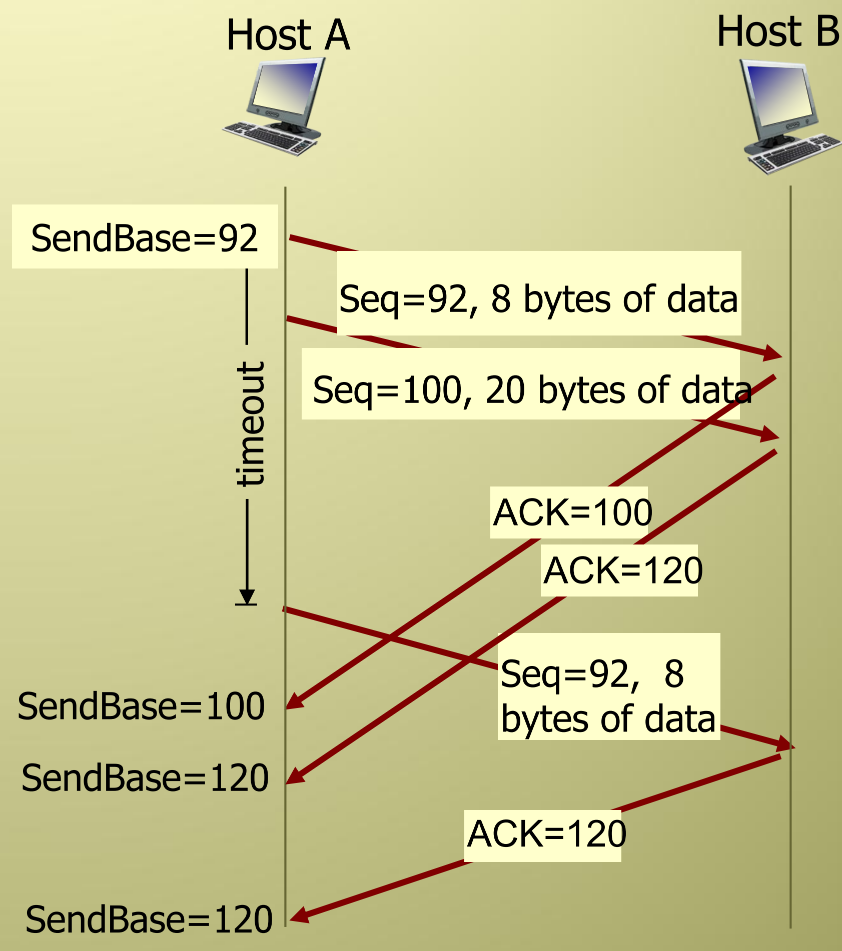
TCP sender (simplified)



TCP: retransmission scenarios

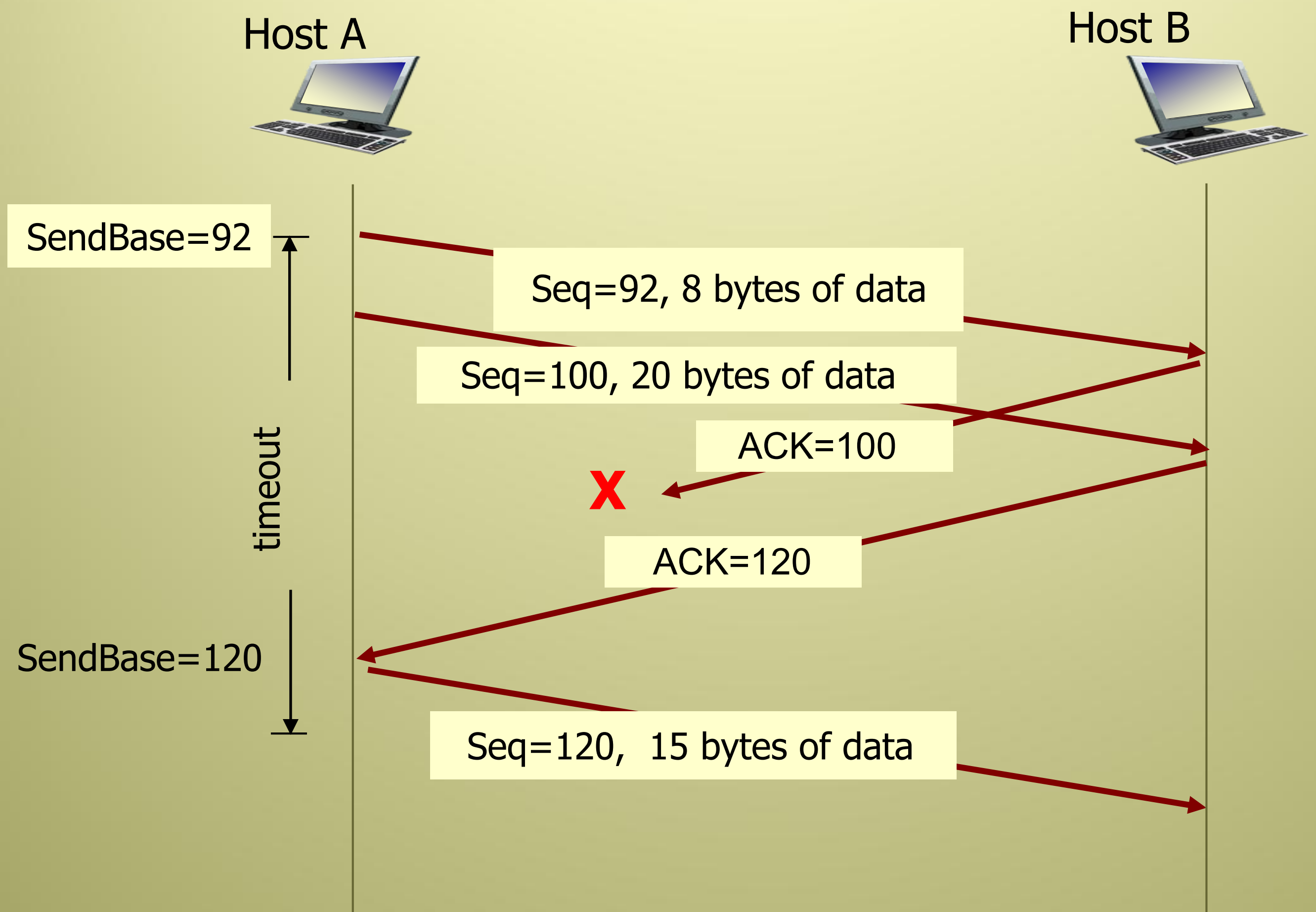


lost ACK scenario



premature timeout

TCP: retransmission scenarios



cumulative ACK

TCP ACK generation [RFC 1122, RFC 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

TCP fast retransmit

- time-out period often relatively long:
  - long delay before resending lost packet
- 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.

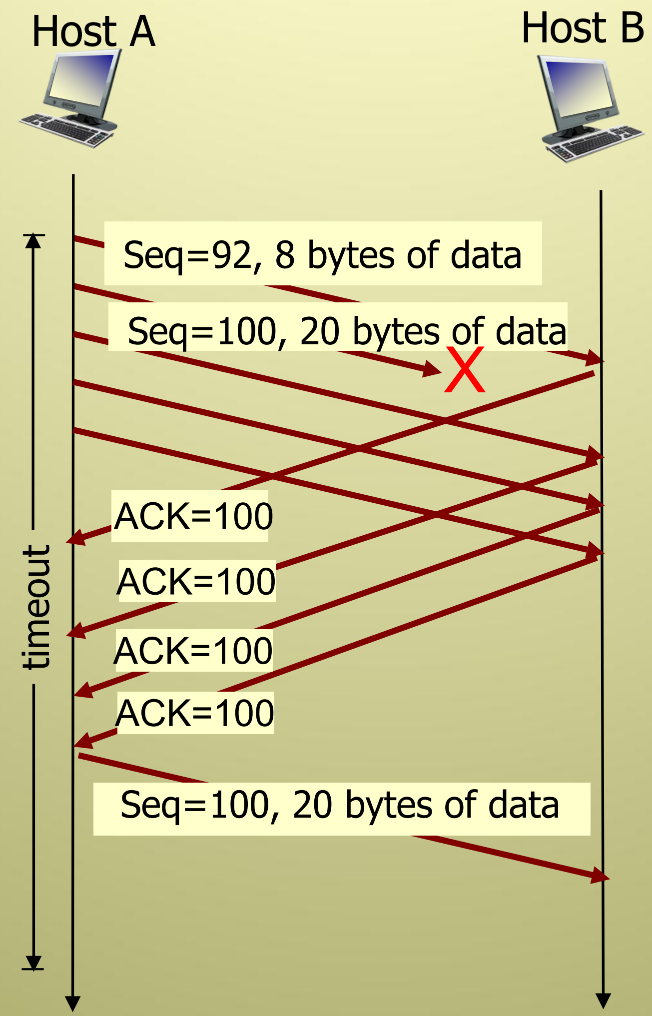
TCP fast retransmit

if sender receives 3 additional ACKs for the same data

("triple duplicate ACKs"), immediately resend unacked segment with smallest seq #

- likely that unacked segment lost, so don't wait for timeout

TCP fast retransmit



fast retransmit after sender receipt of triple duplicate ACK

TCP: Go-Back-N or Selective Repeat?

- "seems" like a GBN since sender maintains the smallest unacked seq # (SendBase), but there are striking differences between TCP operations and GBN
  - most TCP implementation would buffer out-of-order segments
  - Assume sender sent segments 1 to N, and all arrived correctly, but ACK_i for $i < N$ is lost, and the rest of the ACKs are received afterwards:
    - GBN: retransmits ALL segments from i onward
    - TCP: retransmits at most one segment (seg # i)

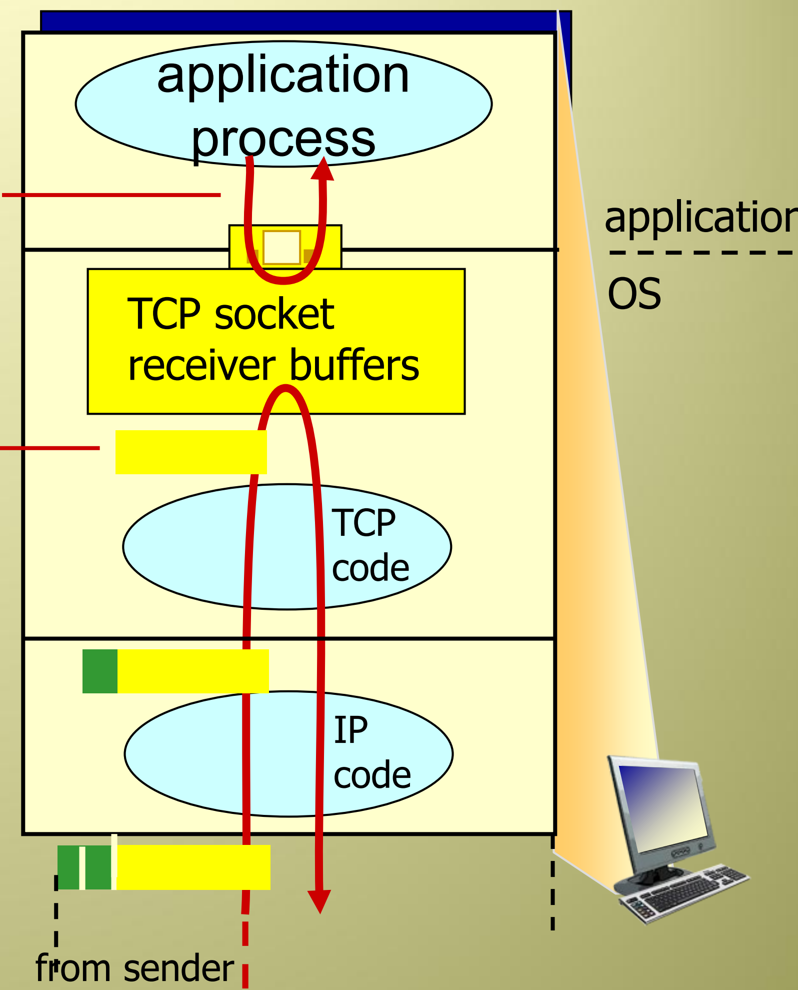
      ➔ In fact, TCP may not even retransmit seg # i, if ACK for any segment with a sequence # > i arrives before timer times-out for segment i

- proposed modifications to TCP: **Selective Acknowledgment**

TCP flow control

application may remove data from TCP socket buffers ….

… slower than TCP receiver is delivering (sender is sending)



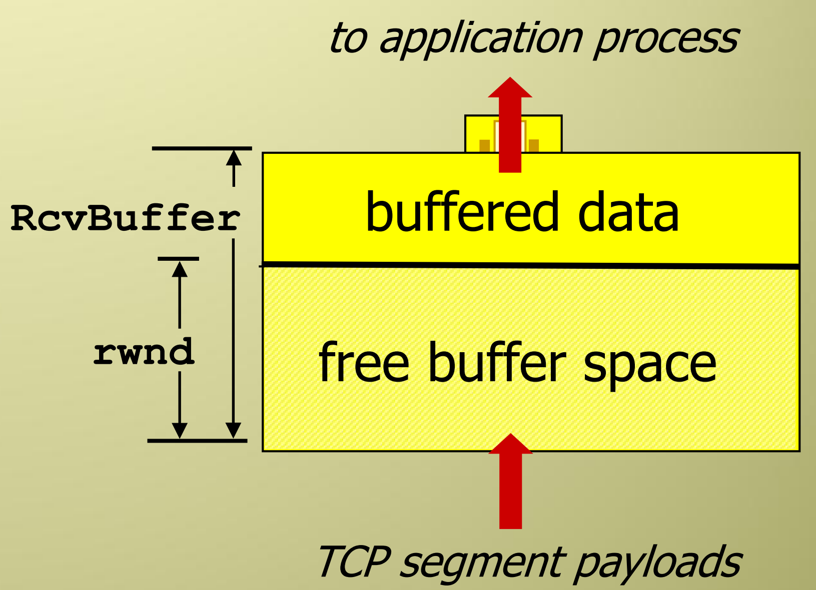
receiver protocol stack

flow control

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

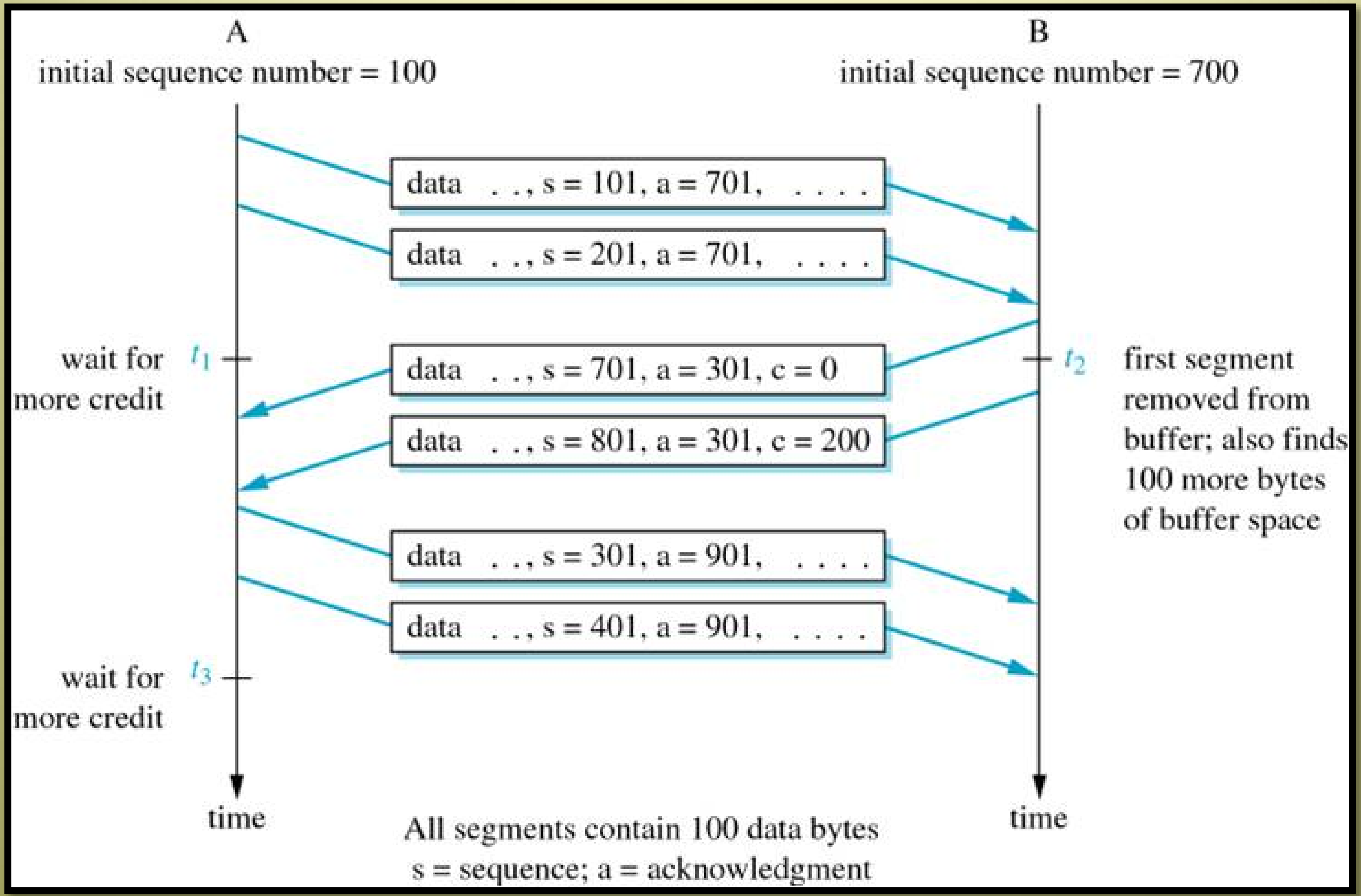
TCP flow control

- receiver "advertises" free buffer space by including **`rwnd`** value in TCP header of receiver-to-sender segments
  - **`RcvBuffer`** size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust **`RcvBuffer`**
- sender limits amount of unacked ("in-flight") data to receiver's **`rwnd`** value
- guarantees receive buffer will not overflow
- Is there a technical problem here?



receiver-side buffering

TCP flow control

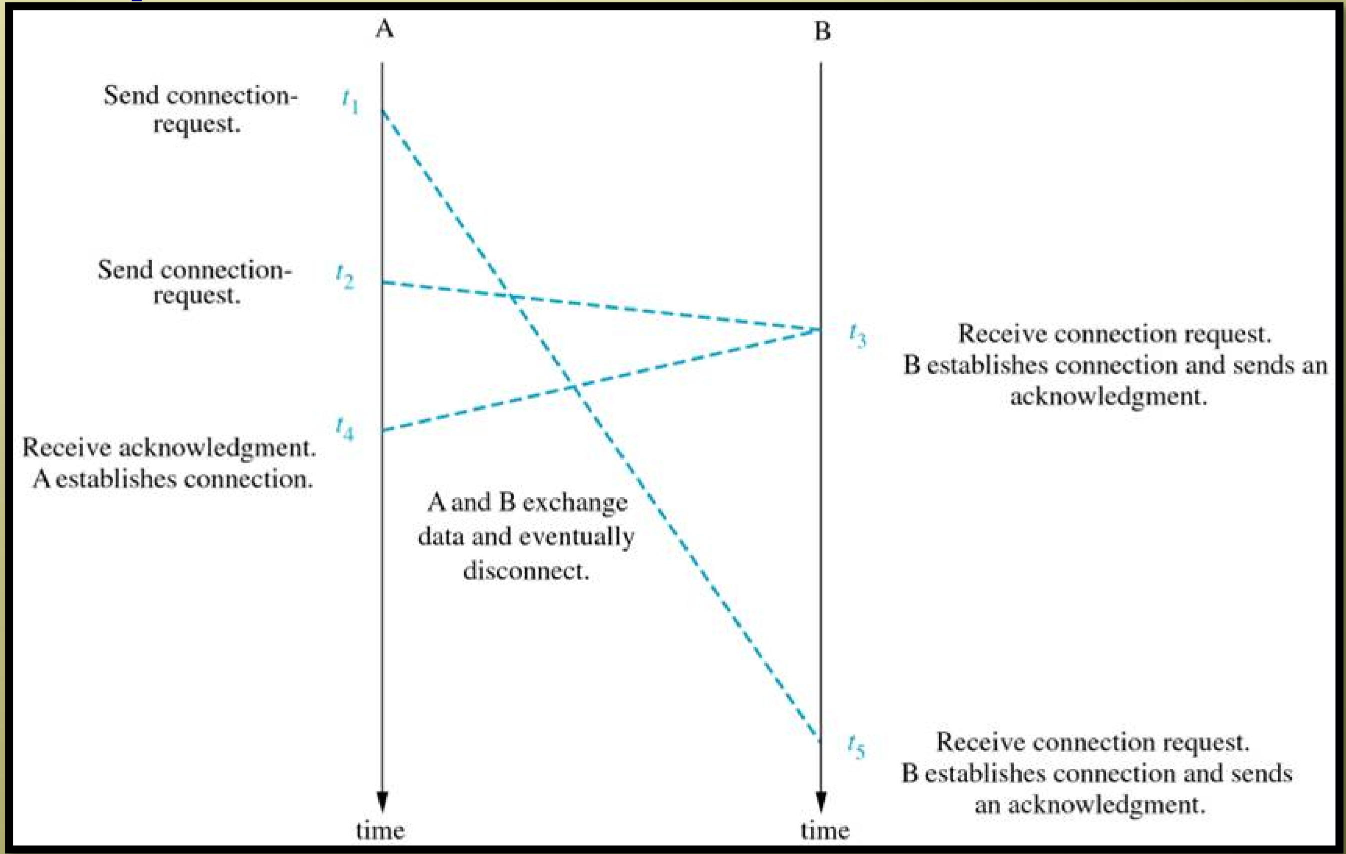


Flow Control Using a Credit Mechanism

TCP connection management

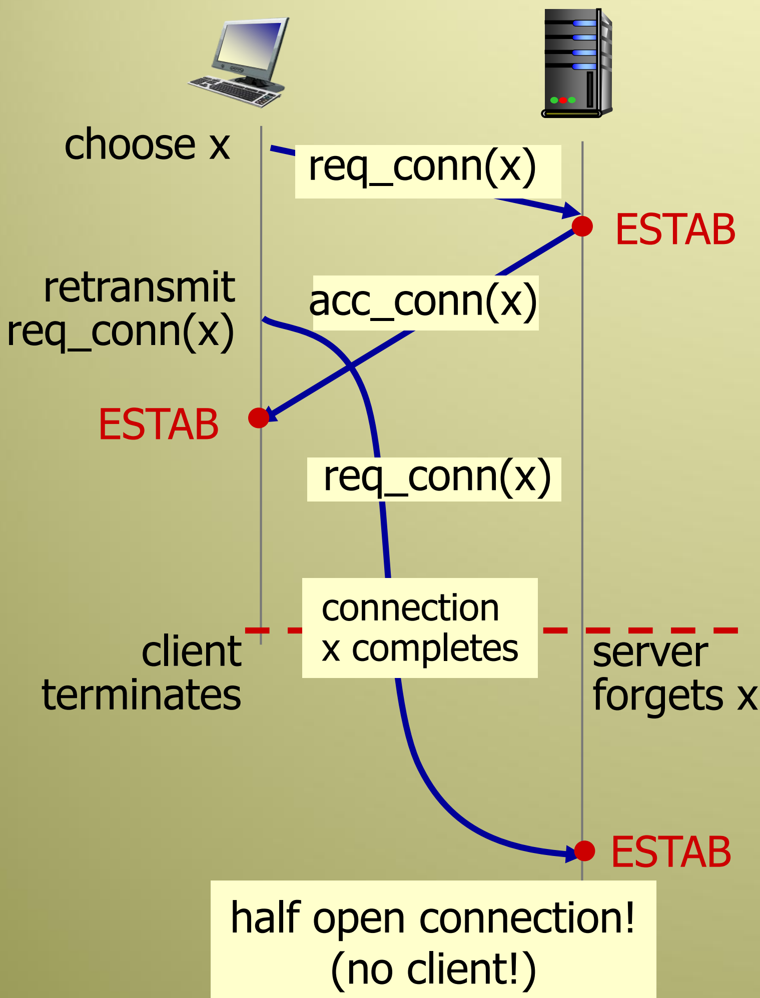
- Establishing a connection may seem easy; one end request a connection, the other end accepts; that is ***two-way handshaking***
  - ***Why does it fail?***

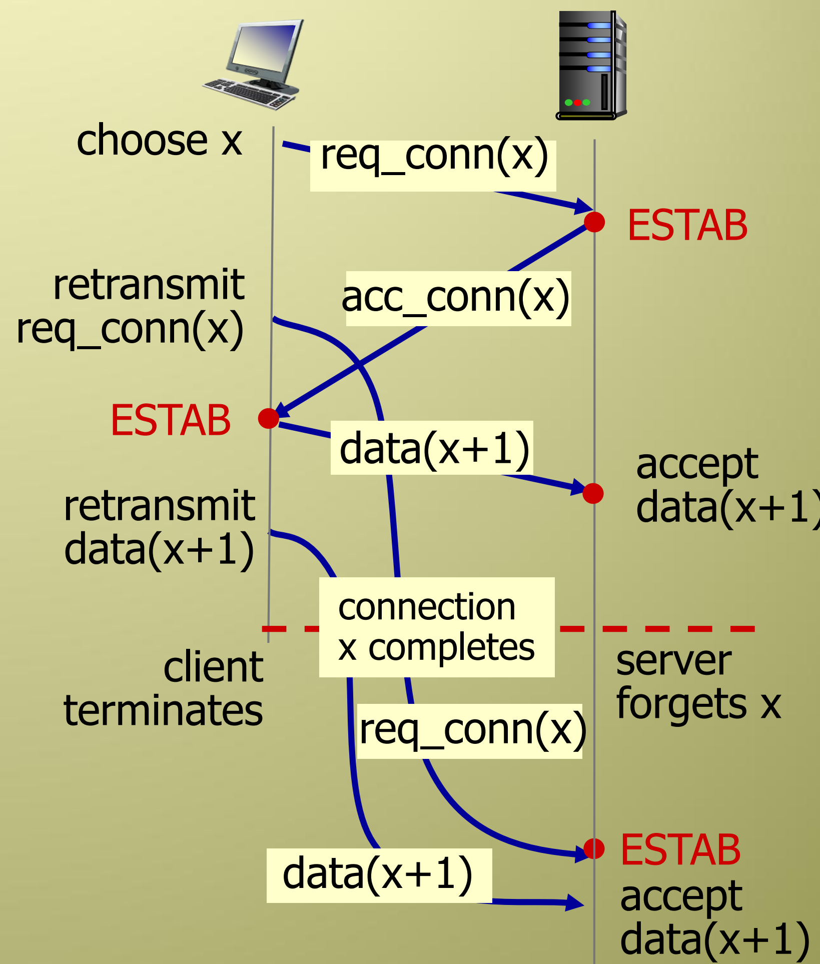
Failure of 2-way Handshake Protocol



Agreeing to establish a connection

2-way handshake failure scenarios:

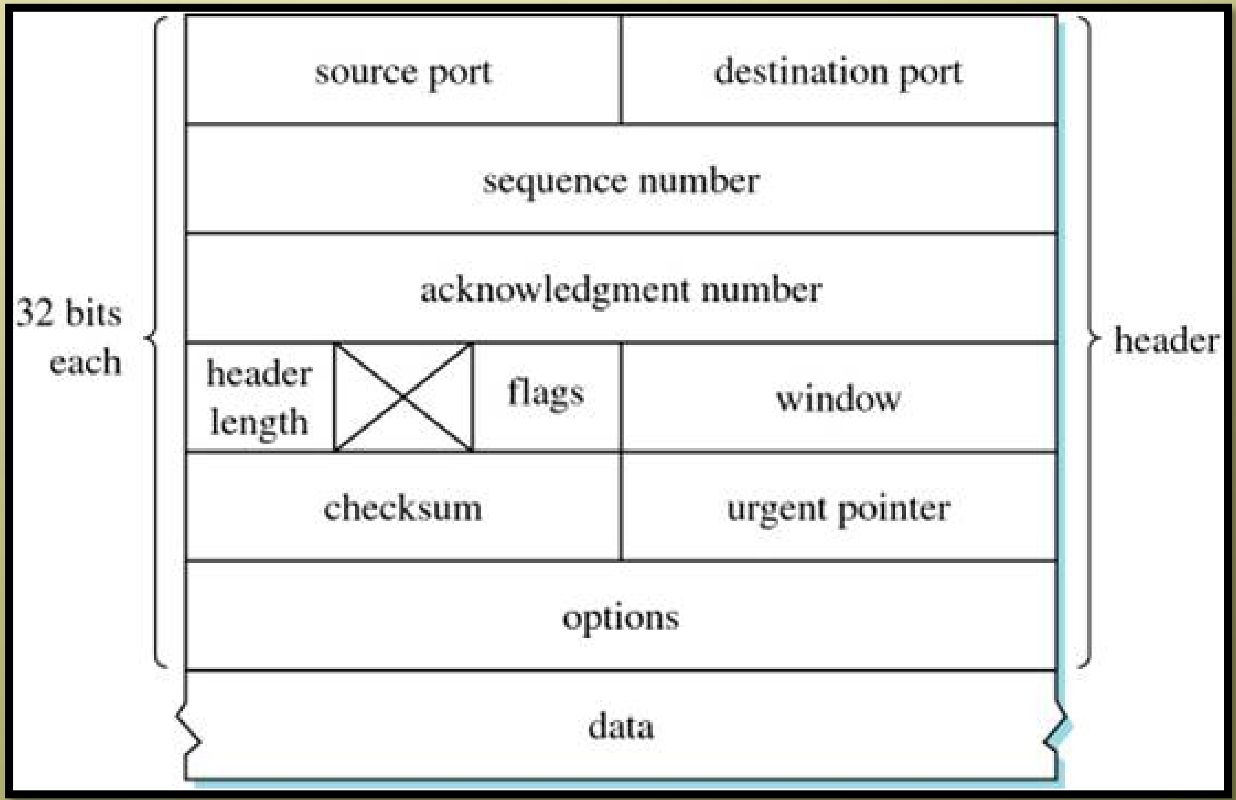




TCP connection management

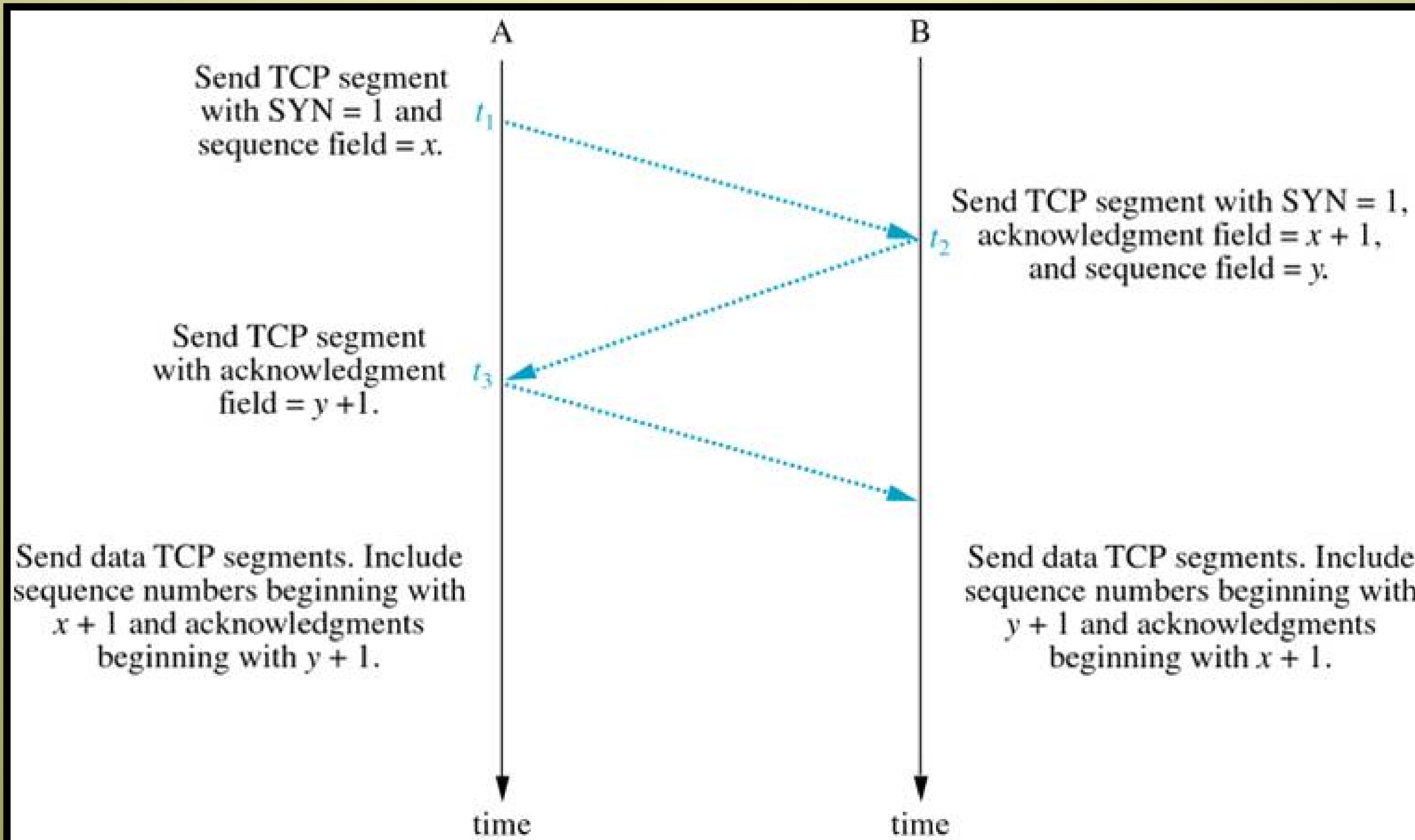
- Recall TCP segment structure
- 32-bit for sequence # field, so it allows more than 4 billion sequence numbers

TCP Segment



TCP connection management

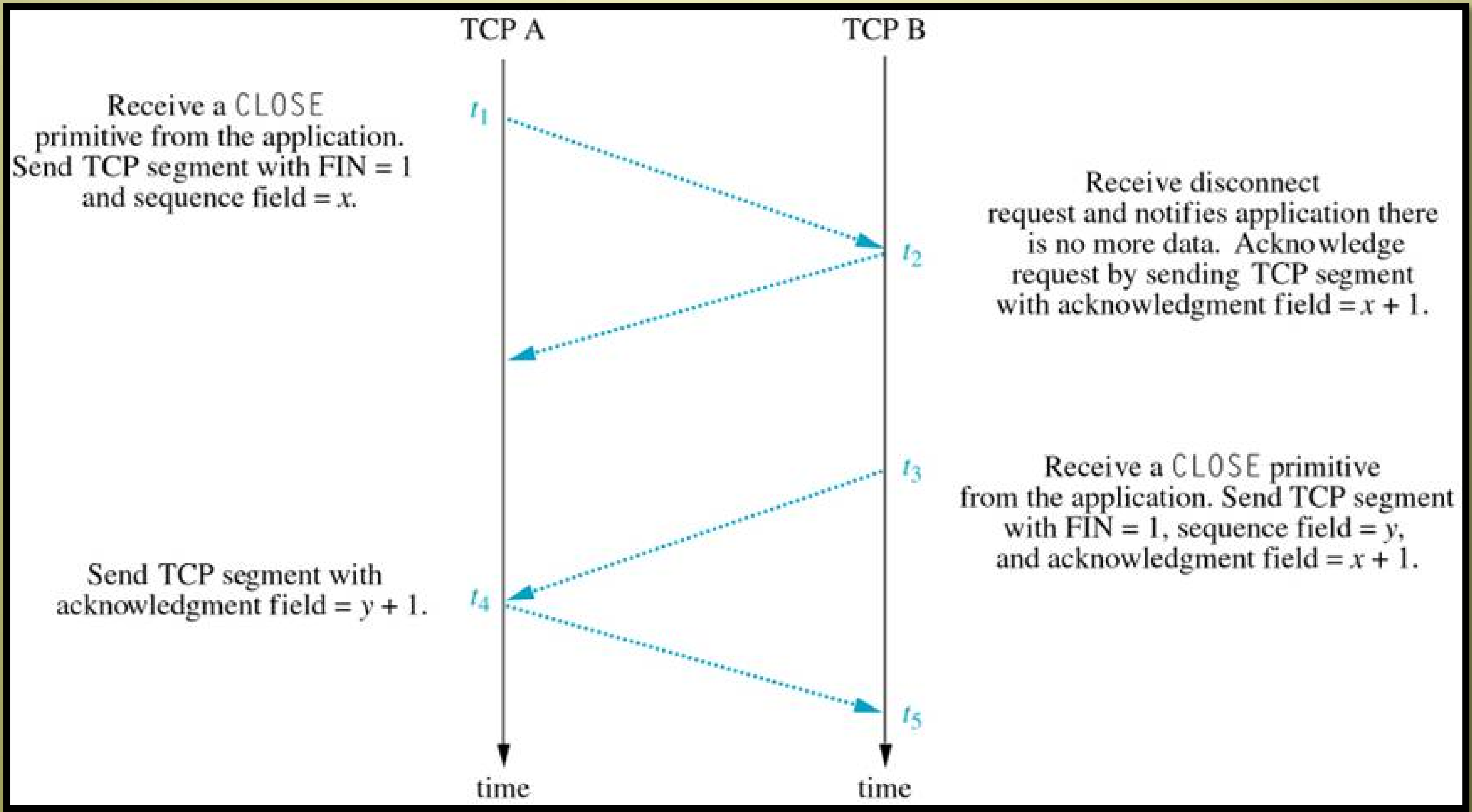
- TCP uses ***three-way handshaking*** instead



3-way Handshake Protocol

TCP connection management

- TCP again uses ***three-way handshaking*** to disconnect



TCP Disconnect Protocol

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 (queuing in router buffers)
- a top-10 problem!

Principles of congestion control

congestion:

- Reasons behind congestion include:
  - failed link,
  - number of transmitted packets exceed network capacity,
  - larger number of nodes than expected,
  - etc.

- Once congested, network suffers delays, possible retransmissions, incapability of a node to receive quickly due to being busy sending (or attempting to send), ..etc.

Approaches towards congestion control

- two broad approaches towards congestion control:

end-end congestion control:

- no explicit feedback from network
- congestion inferred from end-system observed loss, delay
- approach taken by TCP

network-assisted congestion control:

- routers provide feedback to end systems
  - single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
  - explicit rate for sender to send at

congestion control

- there are several ways to handle congestion:
  - Packet elimination
  - Flow control
  - Buffer allocation
  - Choke packets

congestion control

Congestion – Packet Elimination

- if excessive buildup of packets occur at a node, eliminate some of them
- this reduces the network load but suffer loss of packets
- eventually, the higher level protocol will handle this loss by retransmitting the packets, hoping for sure that the congestion has subsided

congestion control

Congestion – Flow Control

- designed to control the number of packets sent, however it is **not** really a congestion control approach
- flow control limits the number of packets between two points, whereas congestion often involves packets coming into a node from many sources
- one solution is to limit the number that can be sent by a node, so the total may not congest the network; this has a bad side effect however; what is it?
- let the nodes communicate with each others so that one reduces its traffic if total traffic is high; this won't work either, why?

congestion control

Congestion – Buffer Allocation

- very suitable for virtual circuit connections
- when the circuit is reserved, a specific amount of buffer is allocated to this communication
- further requests for the same circuit will consume other portions of the available buffers
- if there is no more buffers, requests to use that circuit will be rejected and the higher level protocol must then find an alternative route

congestion control

Congestion – Choke Packets

- more dynamic way to handle congestion
- each node monitors the activity on its outgoing links and traces the utilization of the links
- an increased utilization indicates higher risk of congestion
- if the utilization exceeds specific threshold, the node is put into a warning state
- while in the warning state, if the node receives a packet for further forwarding, it will respond by sending special choke packet to the sender
- when the sender receives a choke packet, it knows that congestion risk is high and hence reduces the number of sent packets for a period of time
- if the time expires without receiving any further choke packets, the sender goes back to its normal transmission rate; otherwise, it reduces the number of sent packets even further

TCP congestion control

- TCP uses **end-to-end** congestion control since the IP layer provides no explicit feedback regarding congestion
- three important questions:
  - how can TCP detect network congestion?
  - what can it do to limit transmission into congested connection?
  - how can transmission be adjusted as perceived congestion changes?

TCP congestion control

- how can TCP detect network congestion?
  - if loss occurs (or suspected) then:
    - sender times out, or
    - three duplicate ACKs are received

- what does TCP do to limit transmission?
  - maintains a **congestion-window** (**`cwnd`**) that imposes constraints on the sender's transmission rate
  - this window decreases when congestion is suspected and increases when network is healthy

TCP congestion control: additive increase multiplicative decrease (**AIMD**)

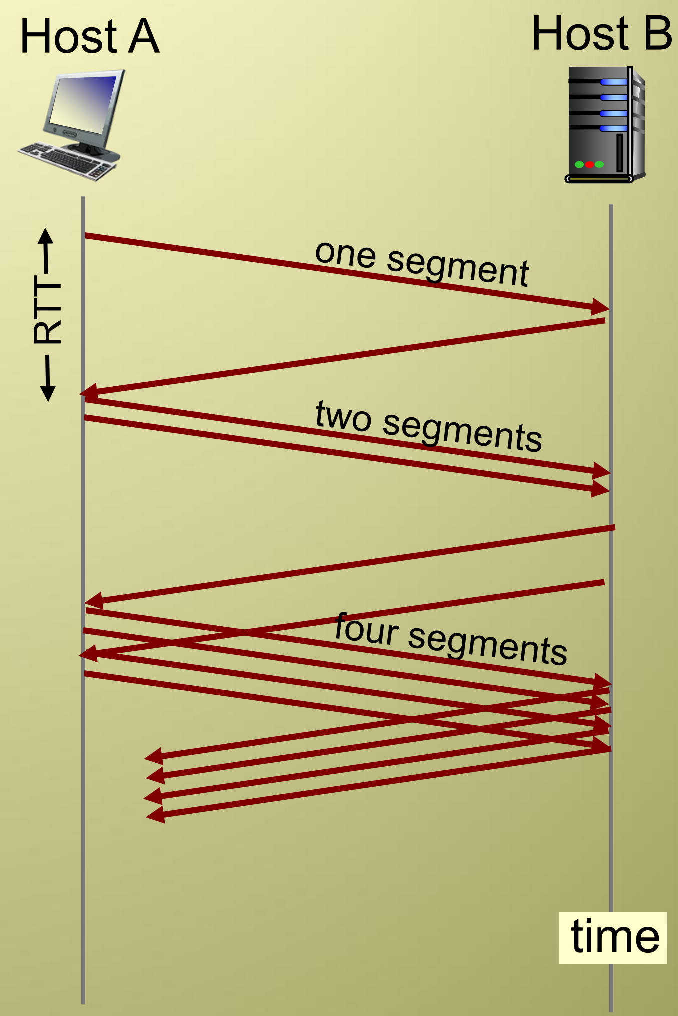
- how can transmission be adjusted as perceived congestion changes?
  - transmitting too fast has the potential of congesting the network
  - transmitting too slow has the potential of under-utilizing the network
- *approach:* sender increases transmission rate, probing for usable bandwidth, until loss occurs

  - *additive increase:* increase transmission additively (i.e. add 1MSS, or double MSS, every RTT) until loss detected

  - *multiplicative decrease:* cut **`transmission rate (i.e.`** in half, or put it down to 1MSS) after loss

TCP Slow Start

- when connection begins, increase rate exponentially until first loss event:
  - initially **`cwnd`** = 1 MSS
  - double **`cwnd`** every RTT
  - done by incrementing **`cwnd`** for every ACK received
- *summary:* initial rate is slow but ramps up exponentially fast



TCP: detecting, reacting to loss

- loss indicated by timeout:
  - **`cwnd`** set to 1 MSS;
  - window then grows exponentially (as in slow start) to threshold, then grows linearly

- loss indicated by 3 duplicate ACKs: **TCP RENO**
  - dup ACKs indicate network capable of delivering some segments
  - **`cwnd`** is cut in half window then grows linearly

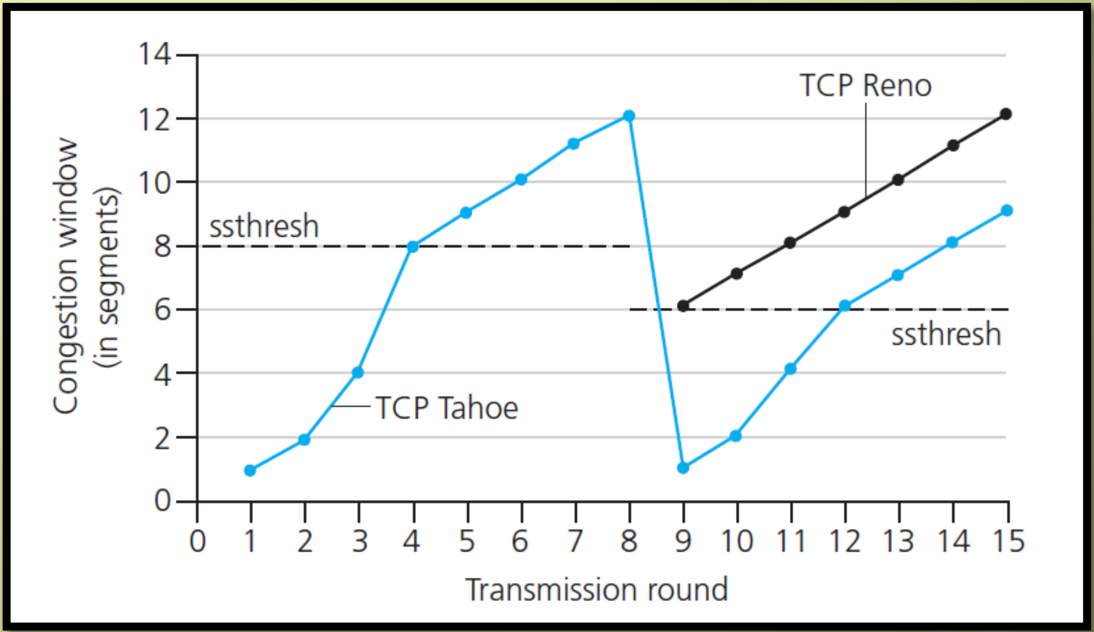
- **TCP Tahoe** always sets **`cwnd`** to 1 (timeout or 3 duplicate acks)

TCP: congestion avoidance

- switching from slow-start to congestion avoidance

Q: when should the exponential increase ends?

A: when **`cwnd`** gets to 1/2 of its value before timeout. Why?



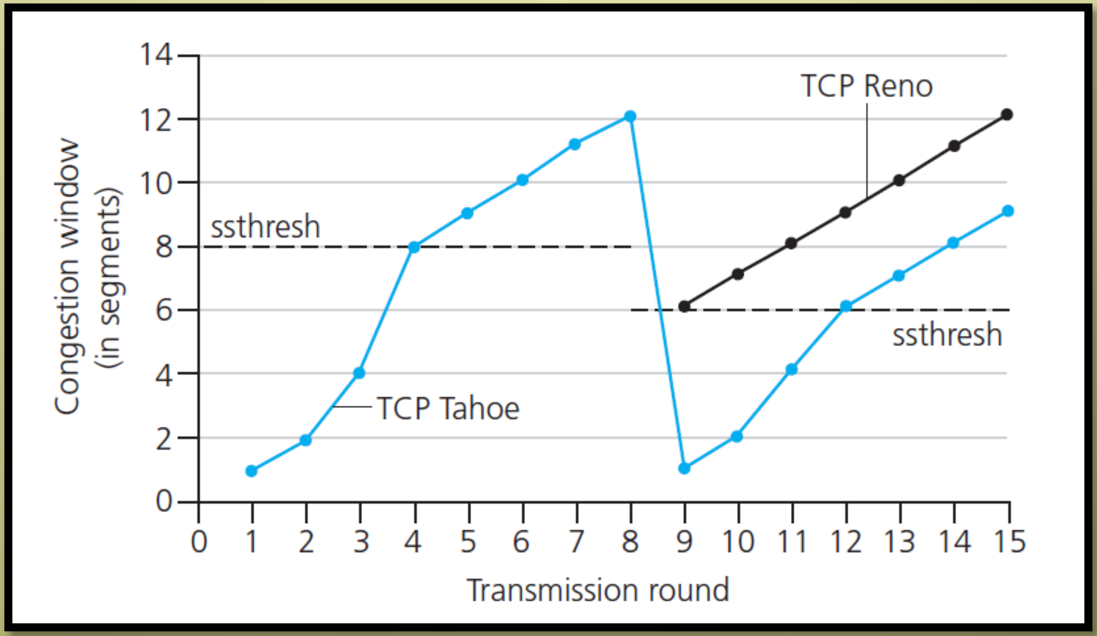
➔ assume timeout occurs, and the **`cwnd`** was set back to 1. when the increase is made again, it will be reckless to double **`cwnd`** size again once your reach ½ of its value since this has high potential of causing congestion again

TCP: congestion avoidance

Implementation:

- maintain a variable **`ssthresh`** for "**s**low **s**tart **thresh**old"

- on loss event, **`ssthresh`** is set to 1/2 of the value of **`cwnd`**



- both TCP RENO, which provides **fast recovery**, and TCP Tahoe follow this congestion avoidance scheme

TCP throughput

- avg. TCP throughput as function of window size, RTT?
  - ignore slow start, assume always data to send
- W: window size (measured in bytes) where loss occurs
  - transmission ranges, as a highly-simplified estimation, from ½ W to W
  - avg. window size (# in-flight bytes) is ¾ W
  - avg. throughput is 3/4W per RTT

→ avg TCP throughput = $\frac{3}{4}\frac{W}{RTT}$ **bytes/sec**

TCP over high-bandwidth paths: *Do we need a new version of TCP?*

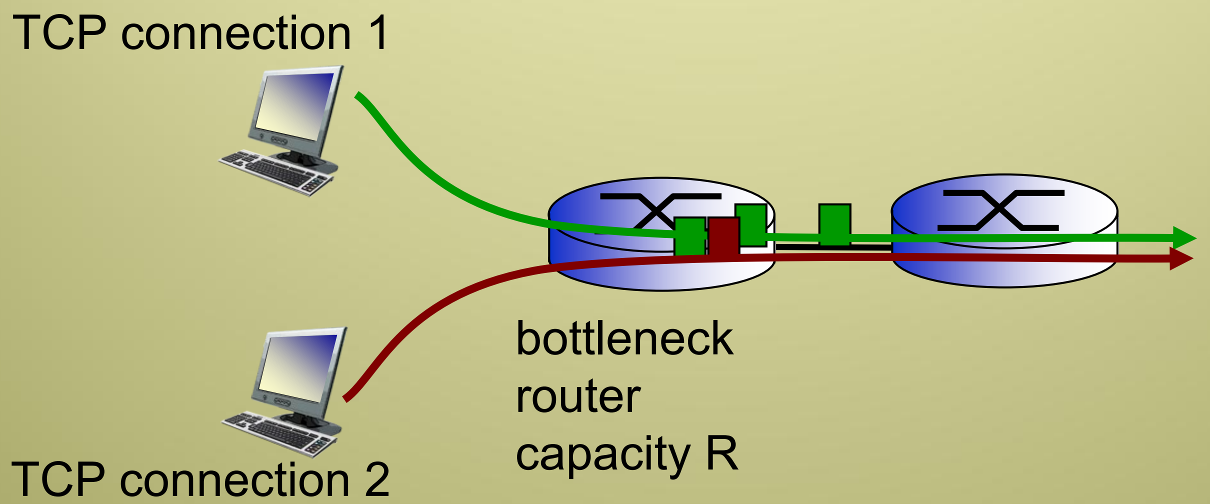
- example: 1500 byte segments, 100ms RTT, want to achieve 10 Gbps throughput
- requires W = 83,333 in-flight segments
- but some segments may be lost. The question is: how much loss can we afford in order to still achieve 10Gbps rate?
- throughput in terms of segment loss probability, L: [Mathis 1997] (*NOTE: Details of the equation is irrelevant & out of our scope here):

$$\text{TCP throughput} = \frac{1.22 \cdot MSS}{RTT\sqrt{L}}$$

- ➜ to achieve 10 Gbps throughput, need a loss rate of $L = 2 \cdot 10^{-10}$ *(one loss event per 5 billion segment; a very small loss rate!*
- new versions of TCP for high-speed paths is needed!

TCP Fairness

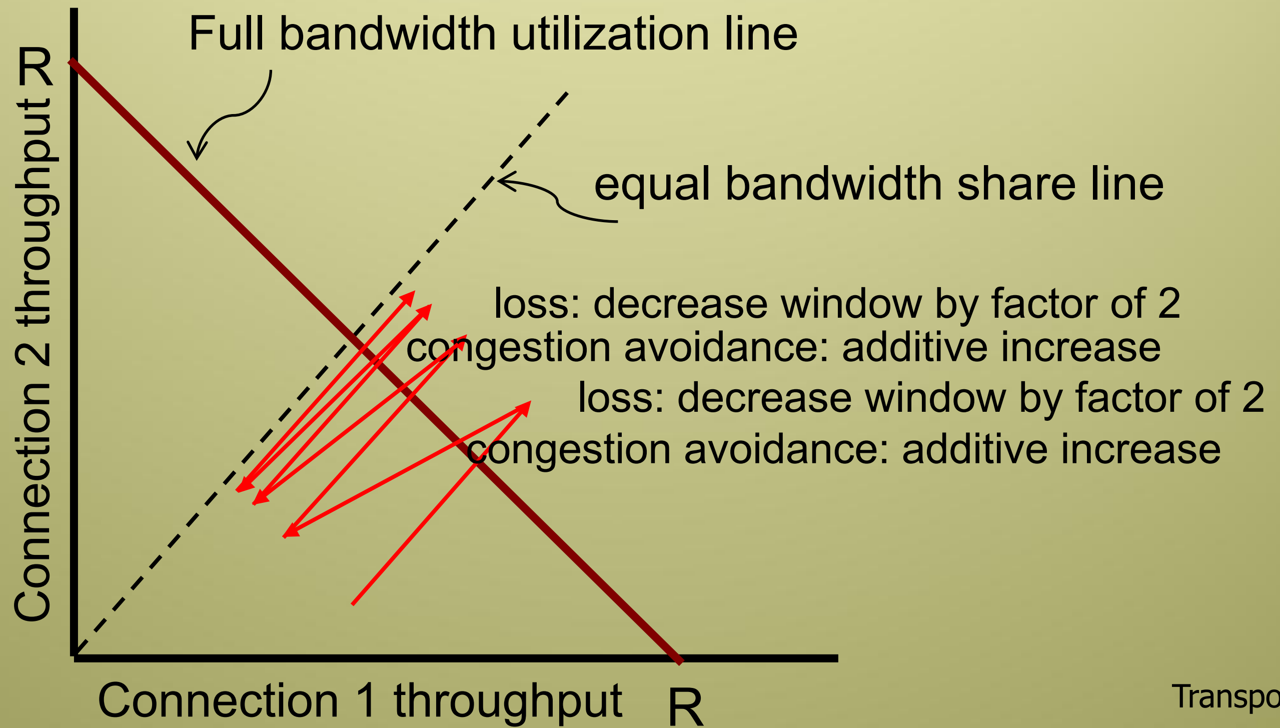
fairness goal: if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K



Why is TCP fair? Or is it!!

Two competing sessions (assume no other connections and no UDP traffic is passing through that link):

- AIMD attempts to be fair, and roughly it is!
- additive increase gives slope of 1, as throughout increases
- multiplicative decrease decreases throughput proportionally



is TCP actually fair?

Fairness and UDP

- multimedia apps often do not use TCP
  - do not want rate throttled by congestion control
- instead use UDP:
  - send audio/video at constant rate, tolerate packet loss

Fairness, parallel TCP connections

- application can open multiple parallel connections between two hosts
- web browsers do this
- e.g., link of rate R with 9 existing connections:
  - new app asks for 1 TCP, gets rate R/10
  - new app asks for 11 TCPs, gets more than R/2, while each of the other ones gets roughly R/20!