Transport Layer

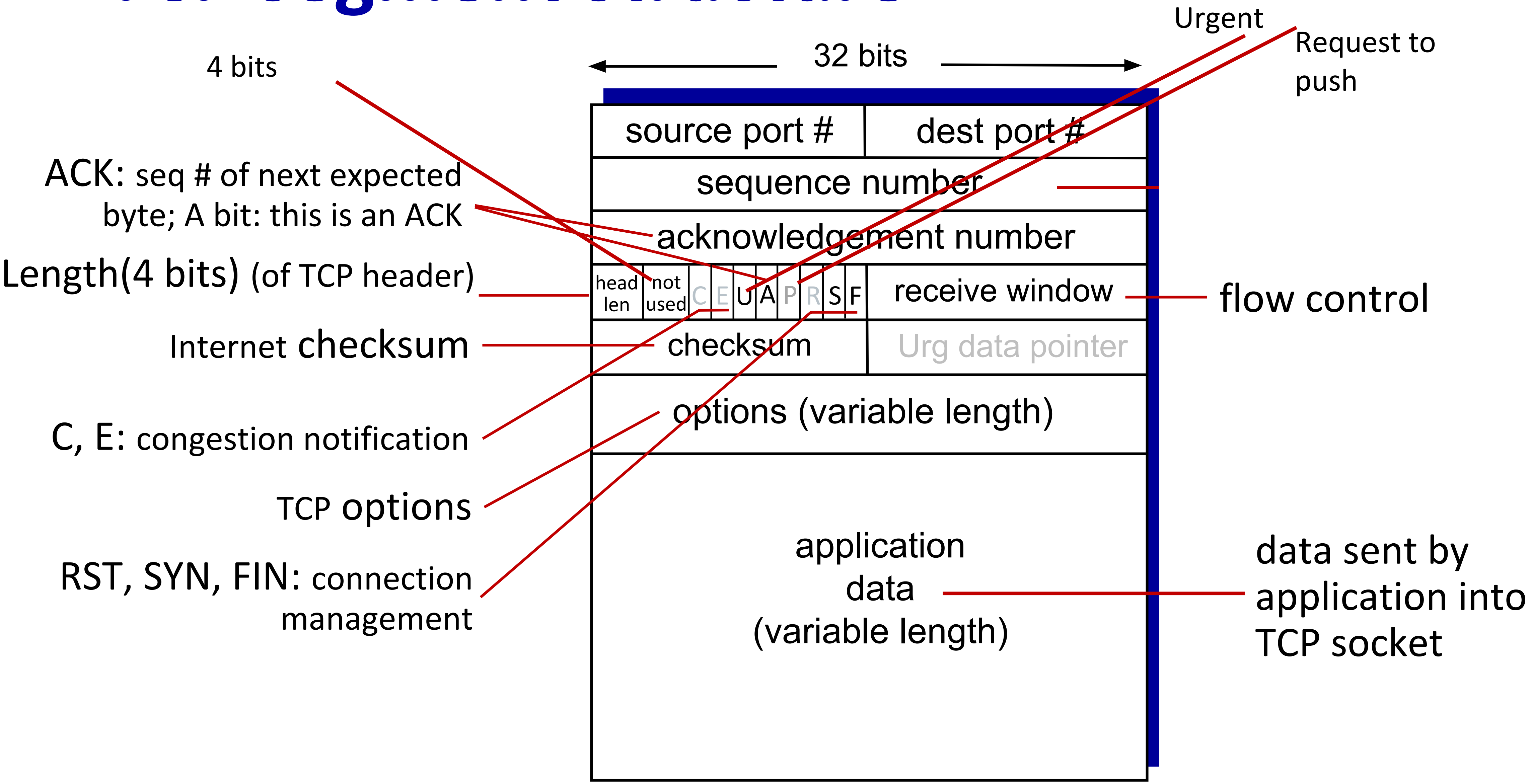
Part 2

TCP: overview RFCs: 793,1122, 2018, 5681, 7323

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte stream:*
- full duplex data:
  - MSS: maximum segment size

  MSS: maximum amount of application layer data in the segment.
- cumulative ACKs
- pipelining:
  - TCP congestion and flow control set window size
- connection-oriented:
  - handshaking (exchange of control messages) initializes sender, receiver state before data exchange

TCP segment structure



TCP sequence numbers, ACKs

Sequence numbers:

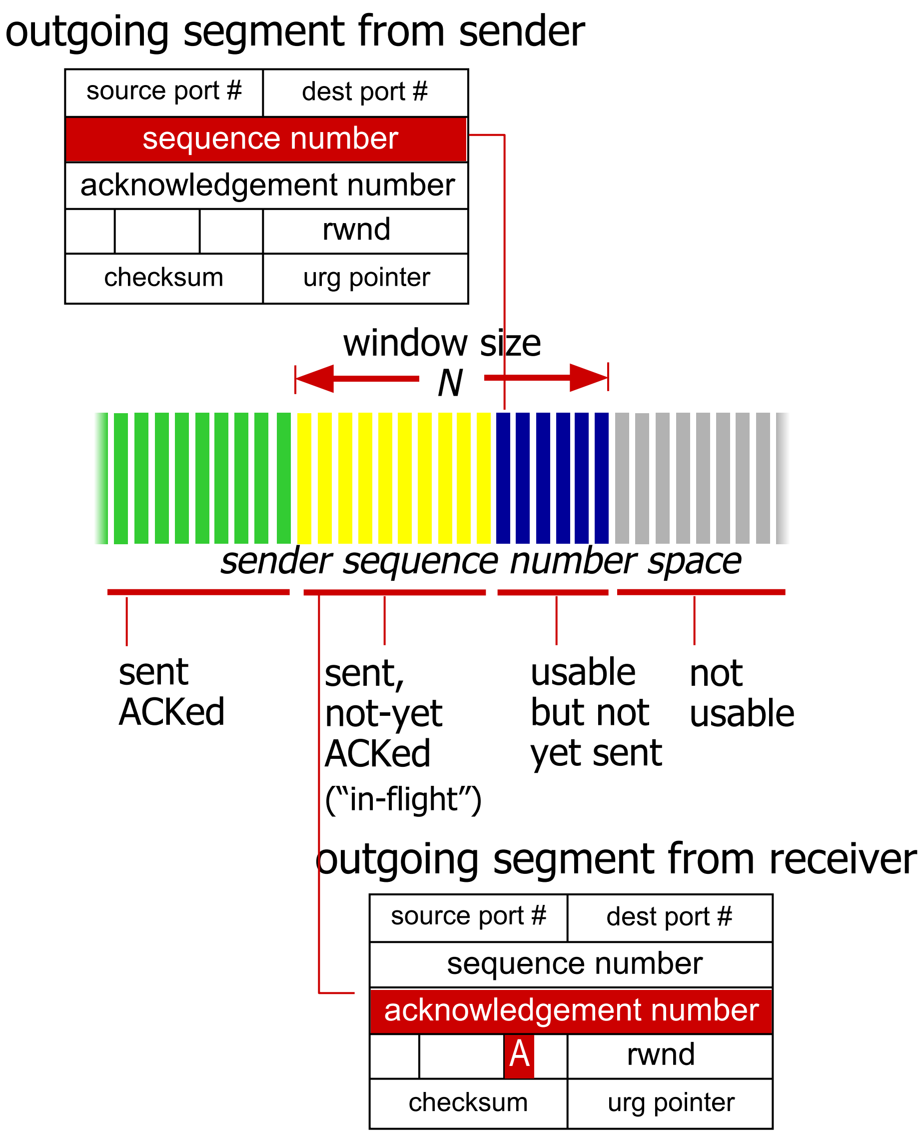
- byte stream "number" of first byte in segment's data

Acknowledgements:

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

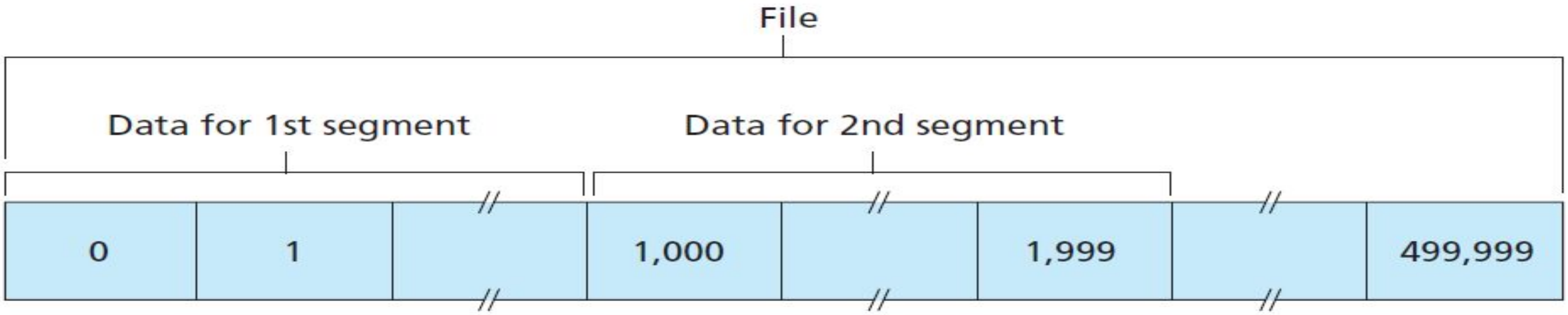
Q: how the receiver handles out-of-order segments

- *A:* Discards out-of-order segments
- *B:* Keeps the out-of-order bytes in the buffer



TCP Seq numbers and Acks

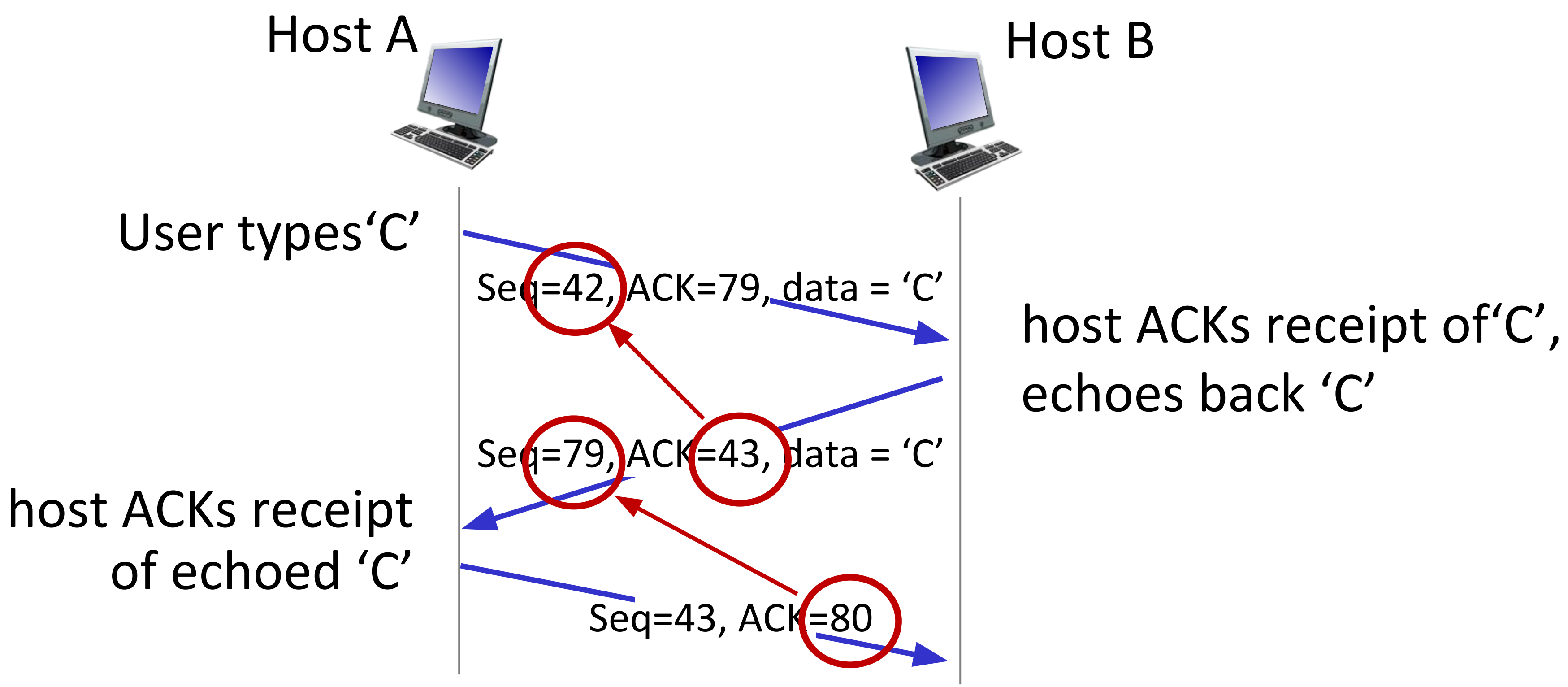
- Suppose Host A wants to send a stream of data to a process in Host B over a TCP connection. Assume that the data stream consists of a file consisting of 500,000 bytes, and that the each MSS is 1,000 bytes. Then the segment looks like this:



TCP sequence numbers, ACKs (Some Scenarios)

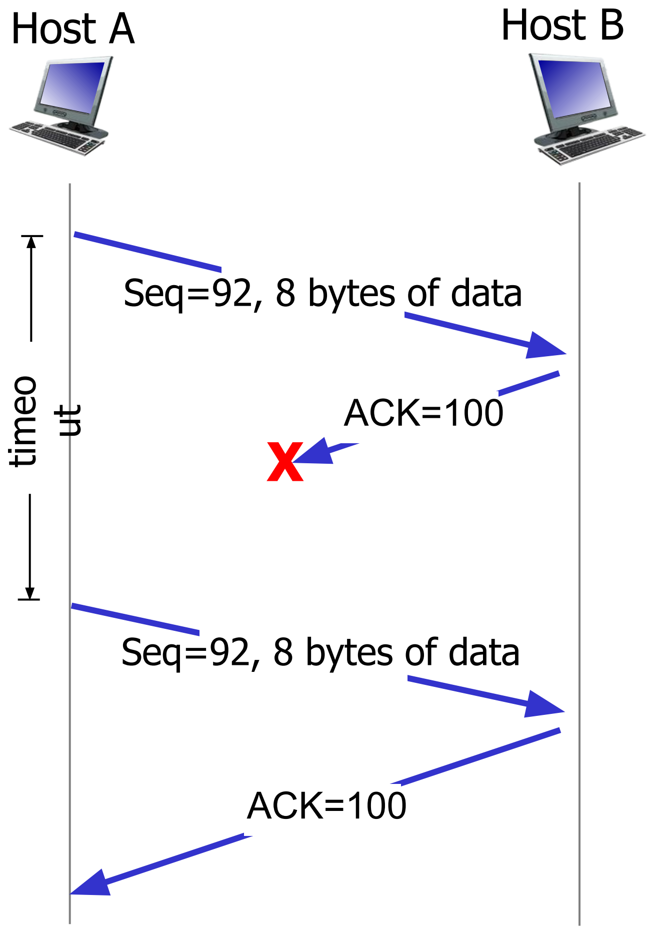
- Host A has received all data from 0 to 535 and Host A is expecting data 536 and all subsequent byte streams from B.
- Host A received one segment from 0 through 535 and another segment from 900 to 1000.
  - It has not received any segment from 536 to 899. Therefore, there has a gap
  - Cumulative acknowledgments.
- Host A received the segment from 900 to 1000 before receiving bytes 536 to 899. Therefore, out of order.

TCP sequence numbers, ACKs

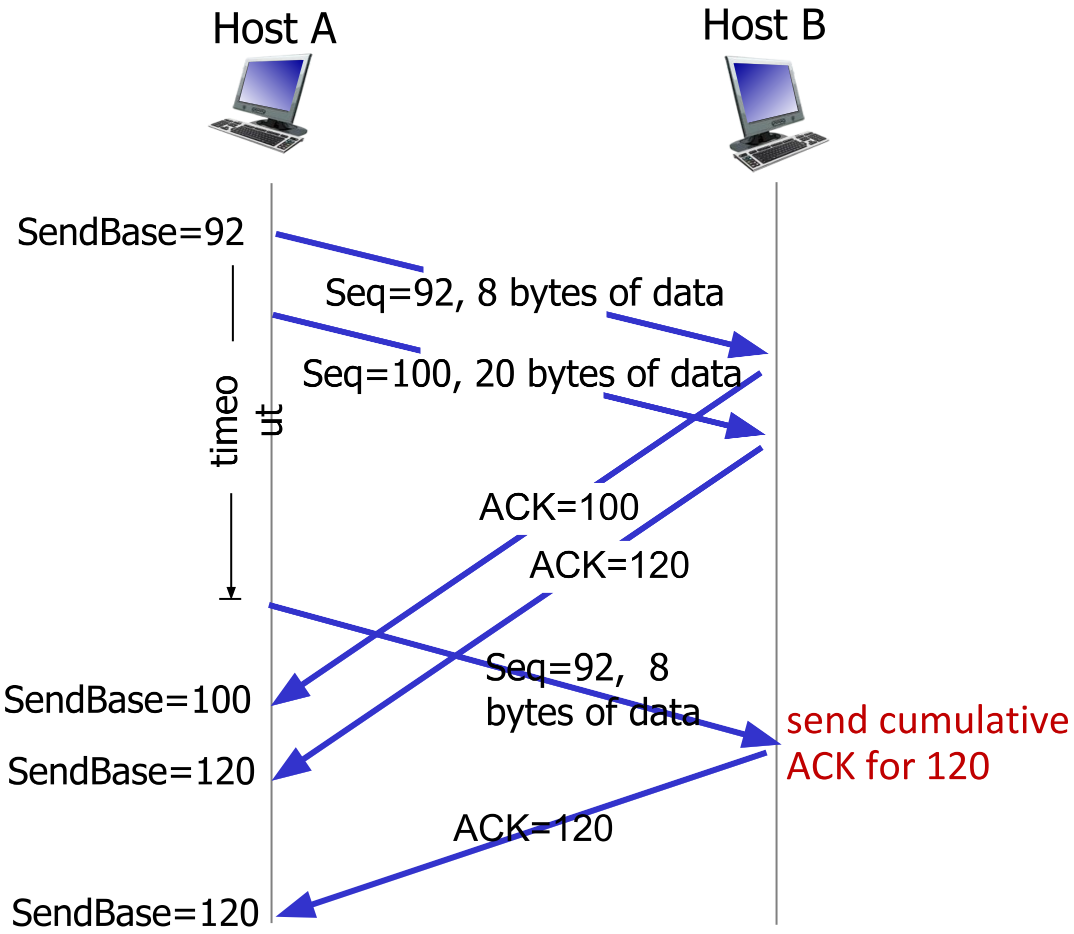


simple telnet scenario

TCP: retransmission scenarios

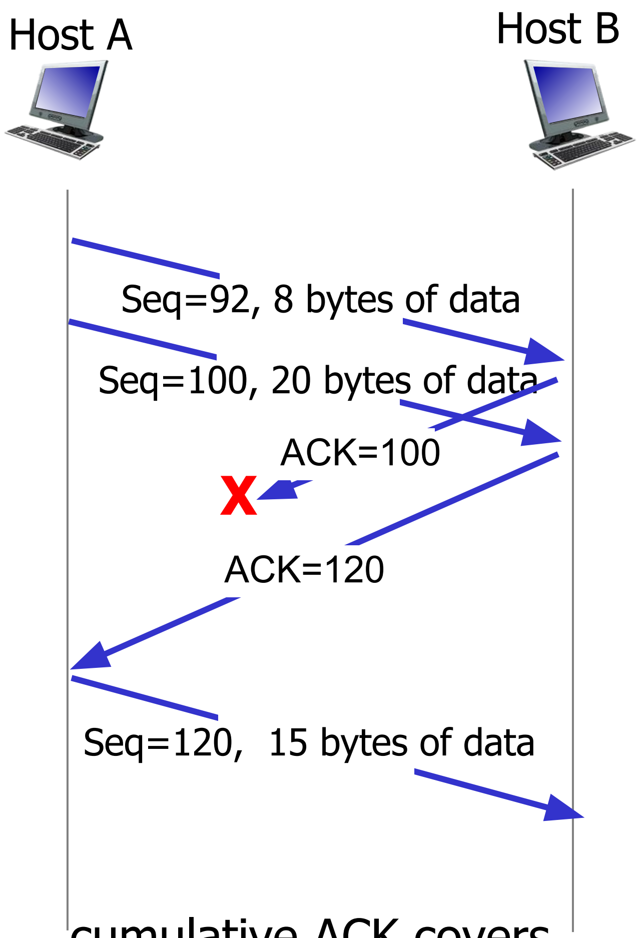


lost ACK scenario



premature timeout

TCP: retransmission scenarios



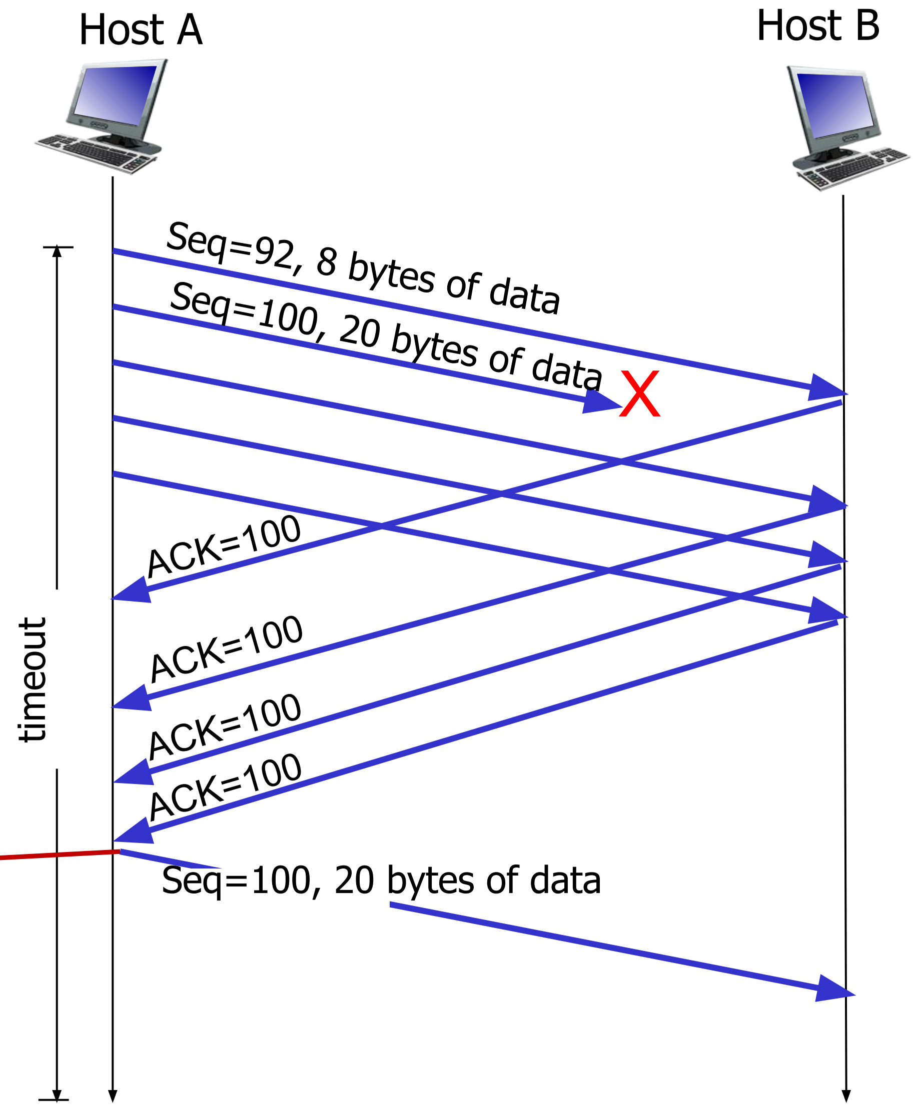
cumulative ACK covers
for earlier lost ACK

TCP fast retransmit

TCP fast retransmit

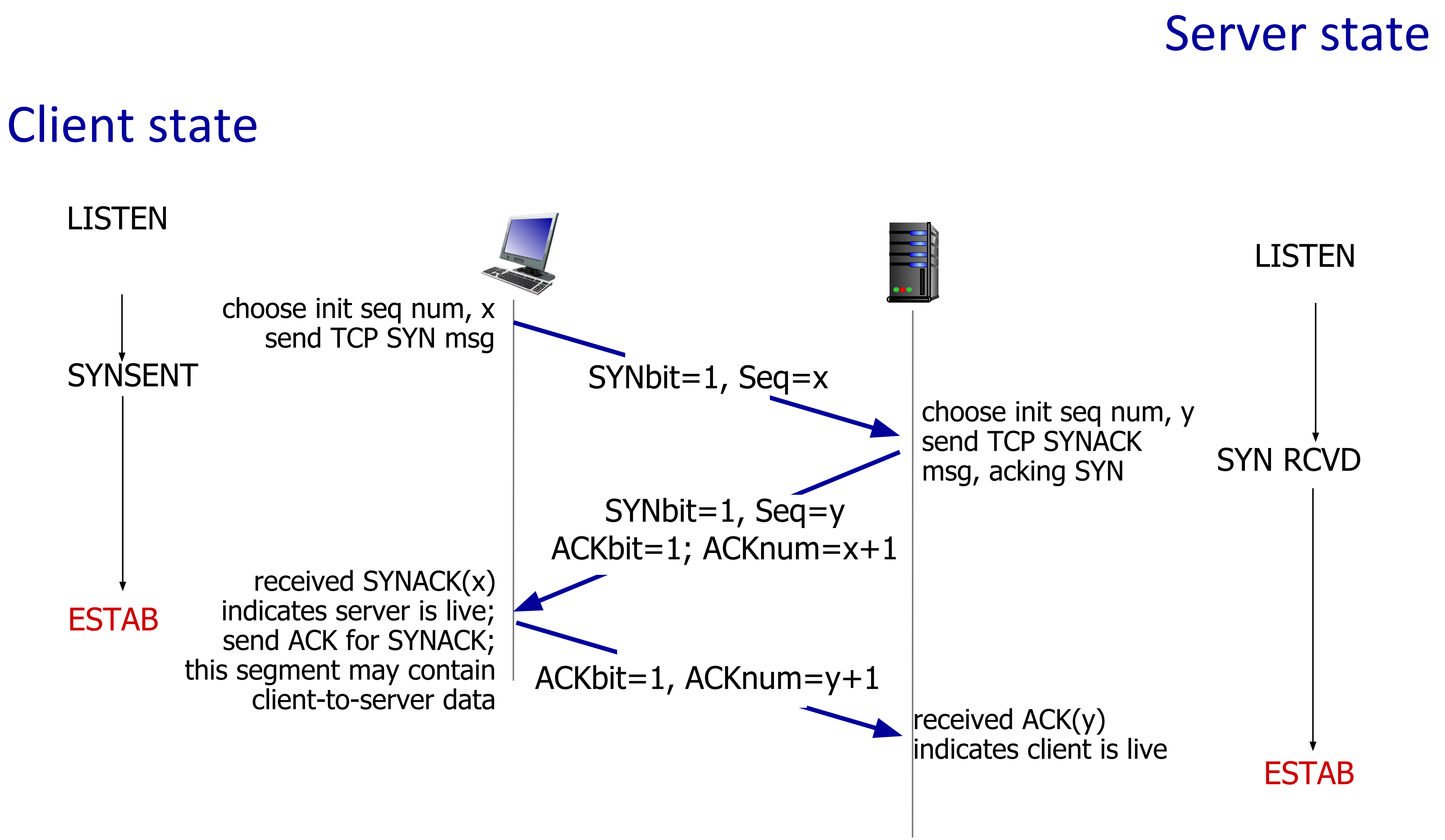
if sender receives 3 additional ACKs for same data ("triple duplicate ACKs"), resend unACKed segment with smallest seq #

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



Receipt of three duplicate ACKs indicates 3 segments received after a missing segment – lost segment is likely. So retransmit!

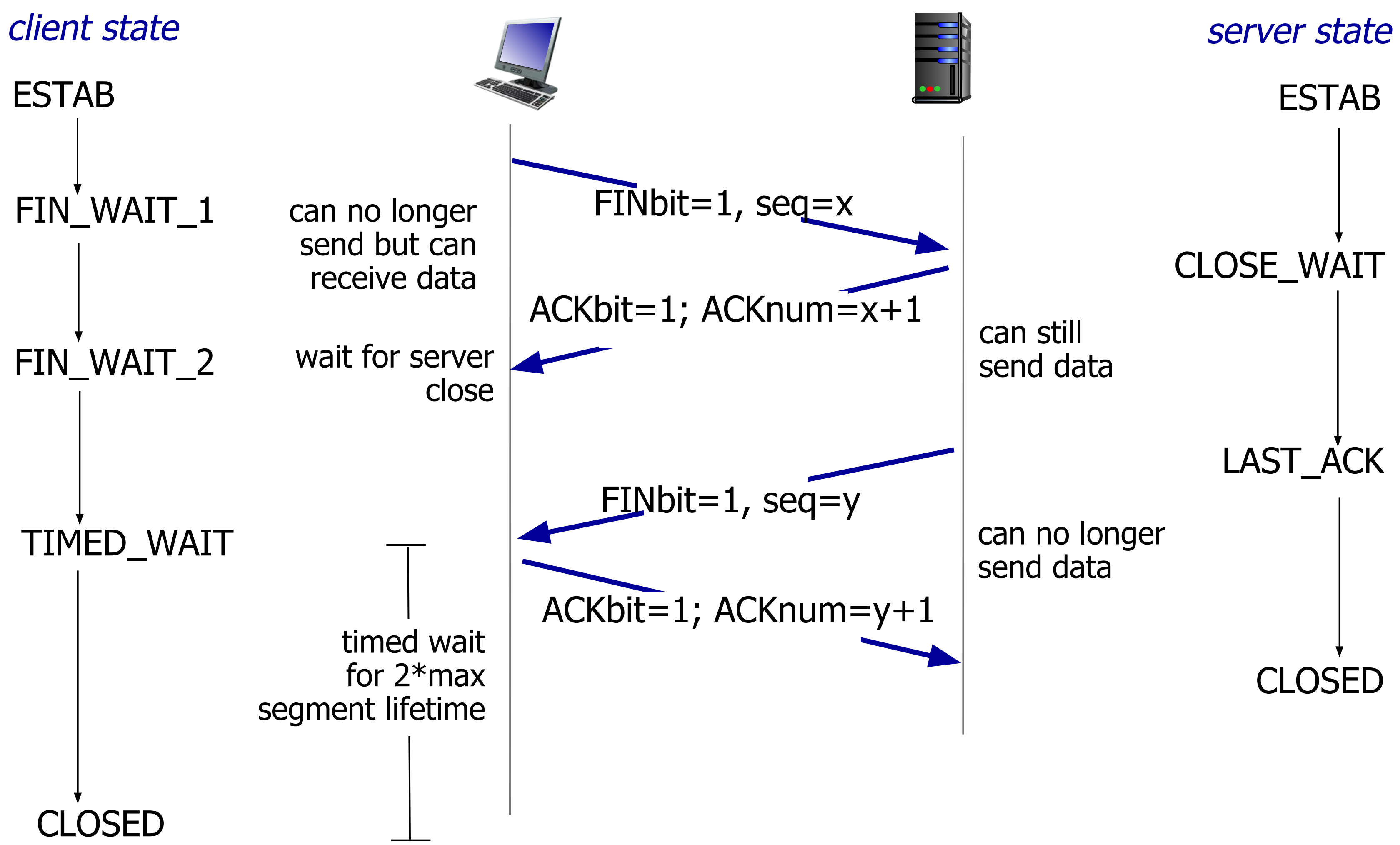
TCP 3-way handshake



Closing a TCP connection

- client, server each close their side of connection
  - send TCP segment with FIN bit = 1
- respond to received FIN with ACK
  - on receiving FIN, ACK can be combined with own FIN
- simultaneous FIN exchanges can be handled

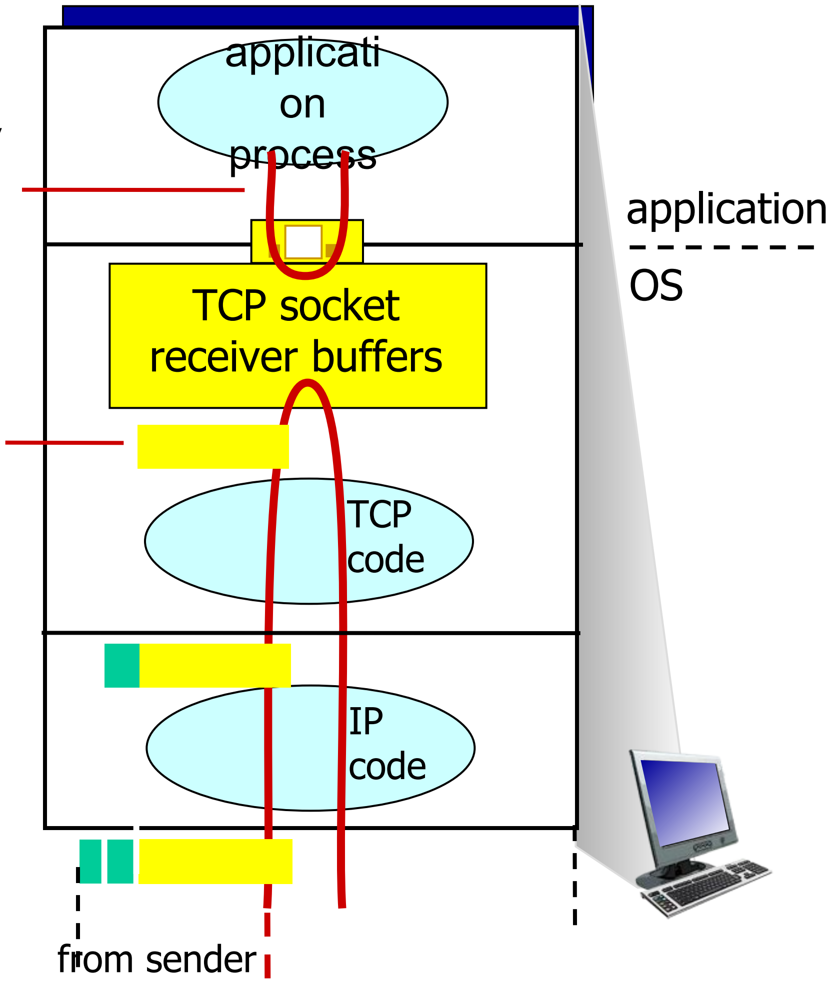
TCP: closing a connection



TCP flow control

application may remove data from TCP socket buffers ….

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



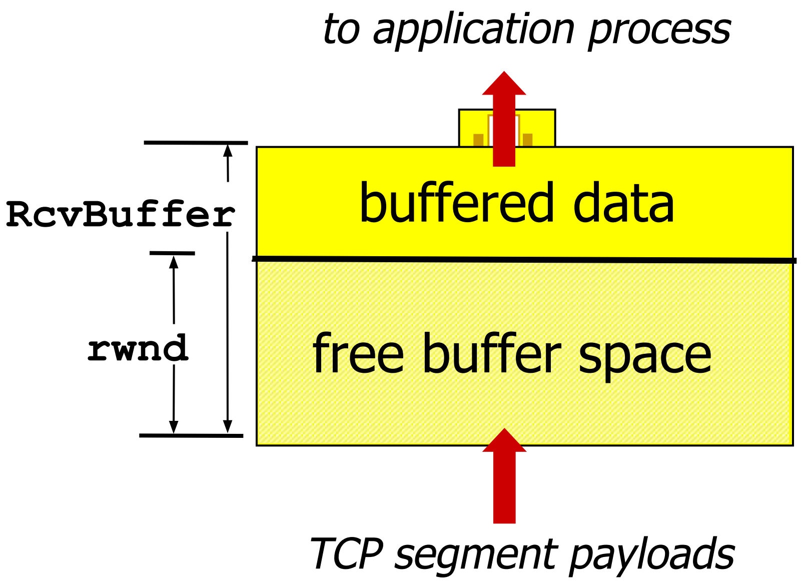
receiver protocol stack

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



receiver-side buffering

TCP congestion control: AIMD

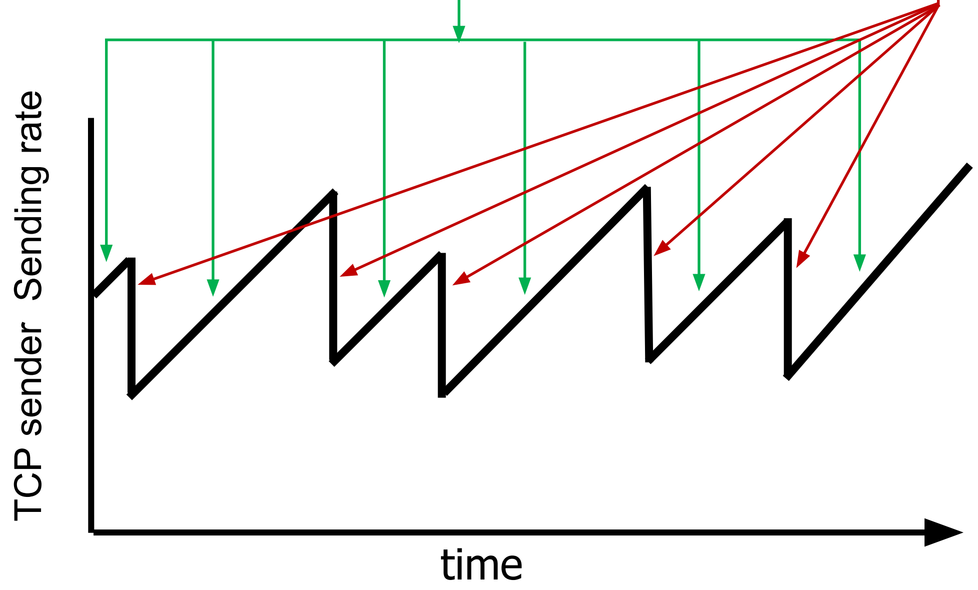
- *approach:* senders can increase sending rate until packet loss (congestion) occurs, then decrease sending rate on loss event

***A**dditive **I**ncrease*

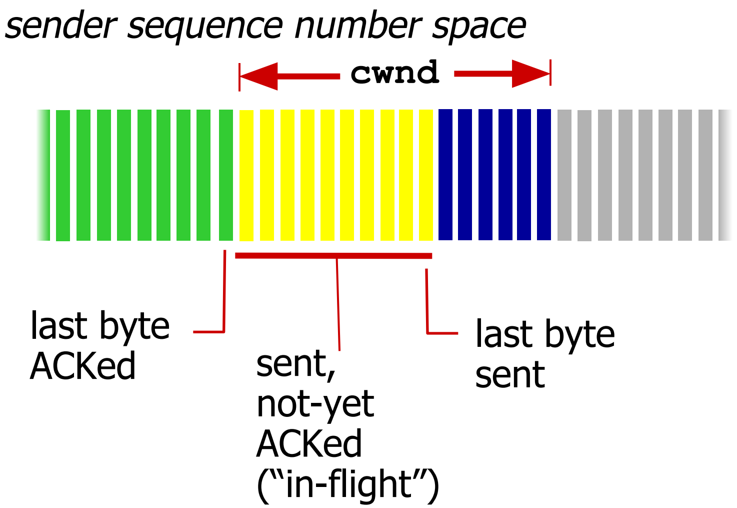
increase sending rate by 1 maximum segment size every RTT until loss detected

***M**ultiplicative **D**ecrease*

cut sending rate in half at each loss event



TCP Congestion Control: details



- sender limits transmission:

$$\texttt{LastByteSent-LastByteAcked} \leq \texttt{cwnd}$$

- **`cwnd`** is dynamic, function of perceived network congestion

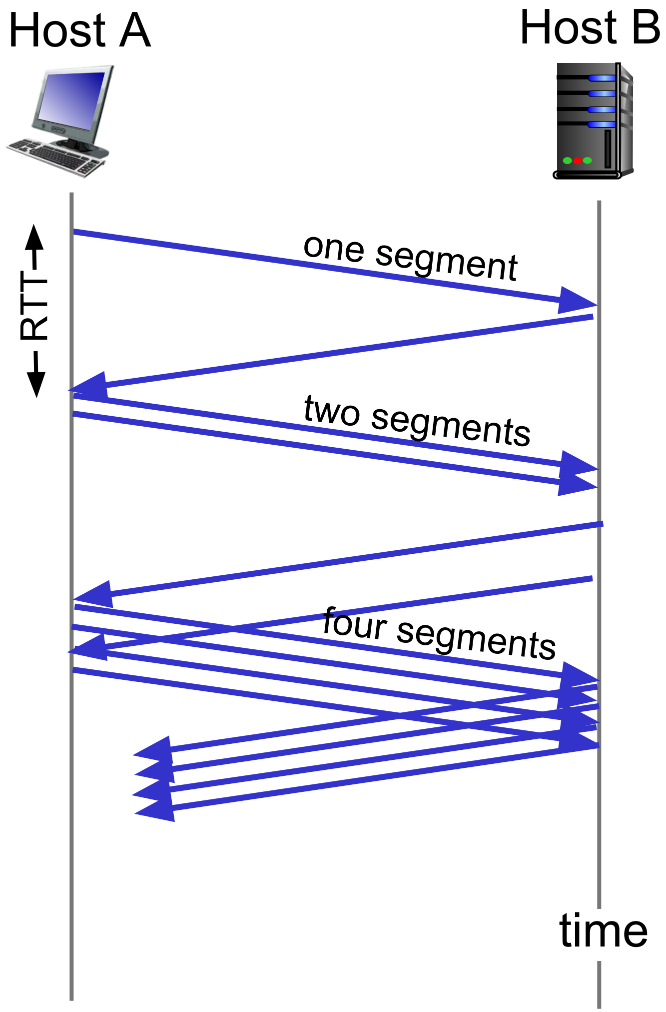
TCP sending rate:

- *roughly:* send cwnd bytes, wait RTT for ACKS, then send more bytes

$$\text{rate} \approx \frac{\text{cwnd}}{\text{RTT}} \text{ bytes/sec}$$

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 Slow Start

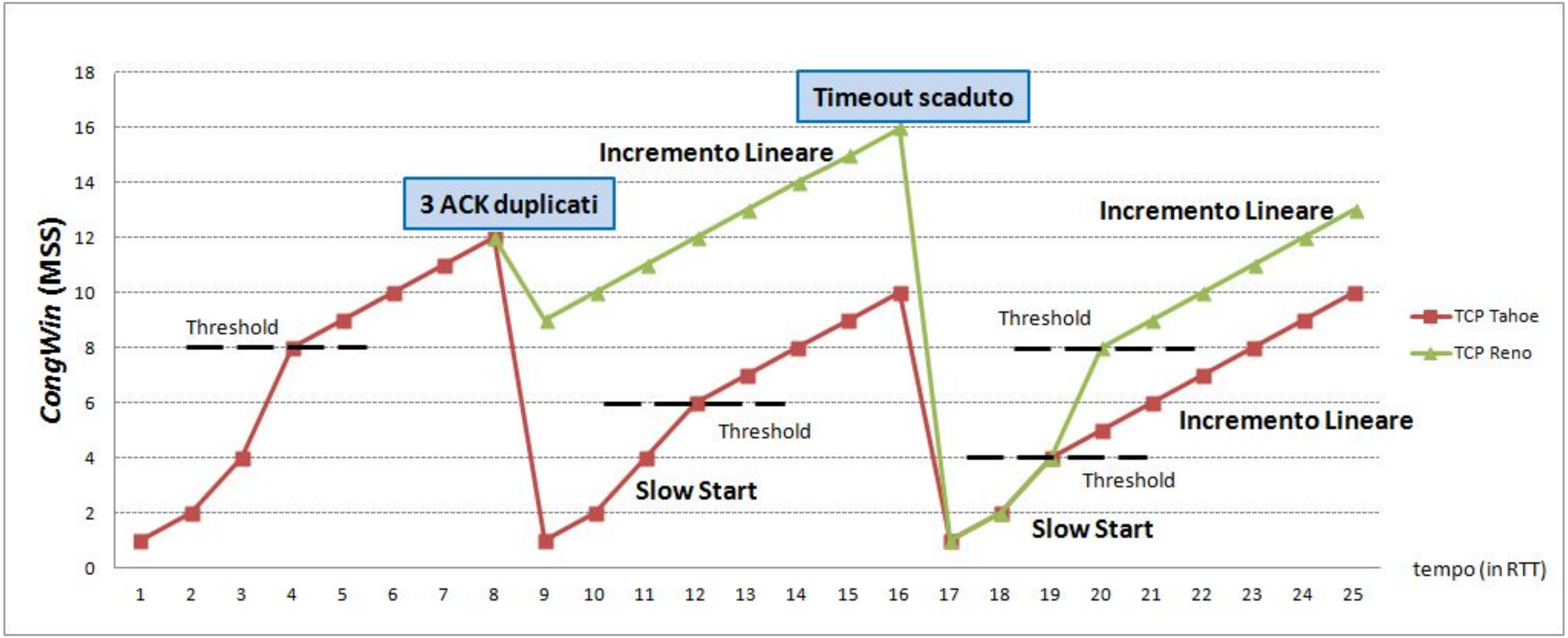
- loss indicated by timeout:
  - **`cwnd`** set to 1 MSS;
  - window then grows exponentially (as in slow start) to threshold, then grows linearly
  - Set the threshold value ssthresh is equal to cwnd/2
- When the value of cwnd >= ssthresh, Slow Start ends and Congestion Avoidance (CA) starts.
- loss indicated by 3 duplicate ACKs: TCP enters in the fast recovery mode.

TCP: Congestion Avoidance (CA)

- Rather than doubling the cwnd value, cwnd is increased by just a single MSS every RTT.
- **When the congestion avoidance ends?**
  - Depends on the timeout events and triple duplicates
  - dup ACKs indicate network capable of delivering some segments
- Fast Recovery: 3 dup ACKs
  - TCP Tahoe always sets **`cwnd`** to 1 then grows exponentially (timeout or 3 duplicate acks) [Earlier Style]

`TCP Reno cut the cwnd` in half window then grows linearly [New Version] **ssthresh= cwnd/2 and cwnd = ssthresh + 3**

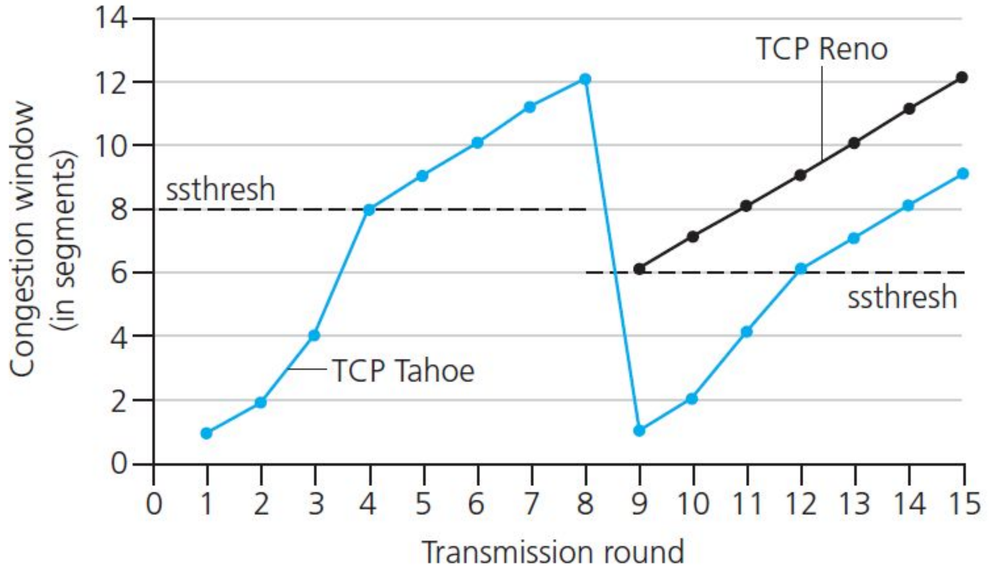
Congestion Control



TCP: switching from slow start to CA

Q: when should the exponential increase switch to linear?

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

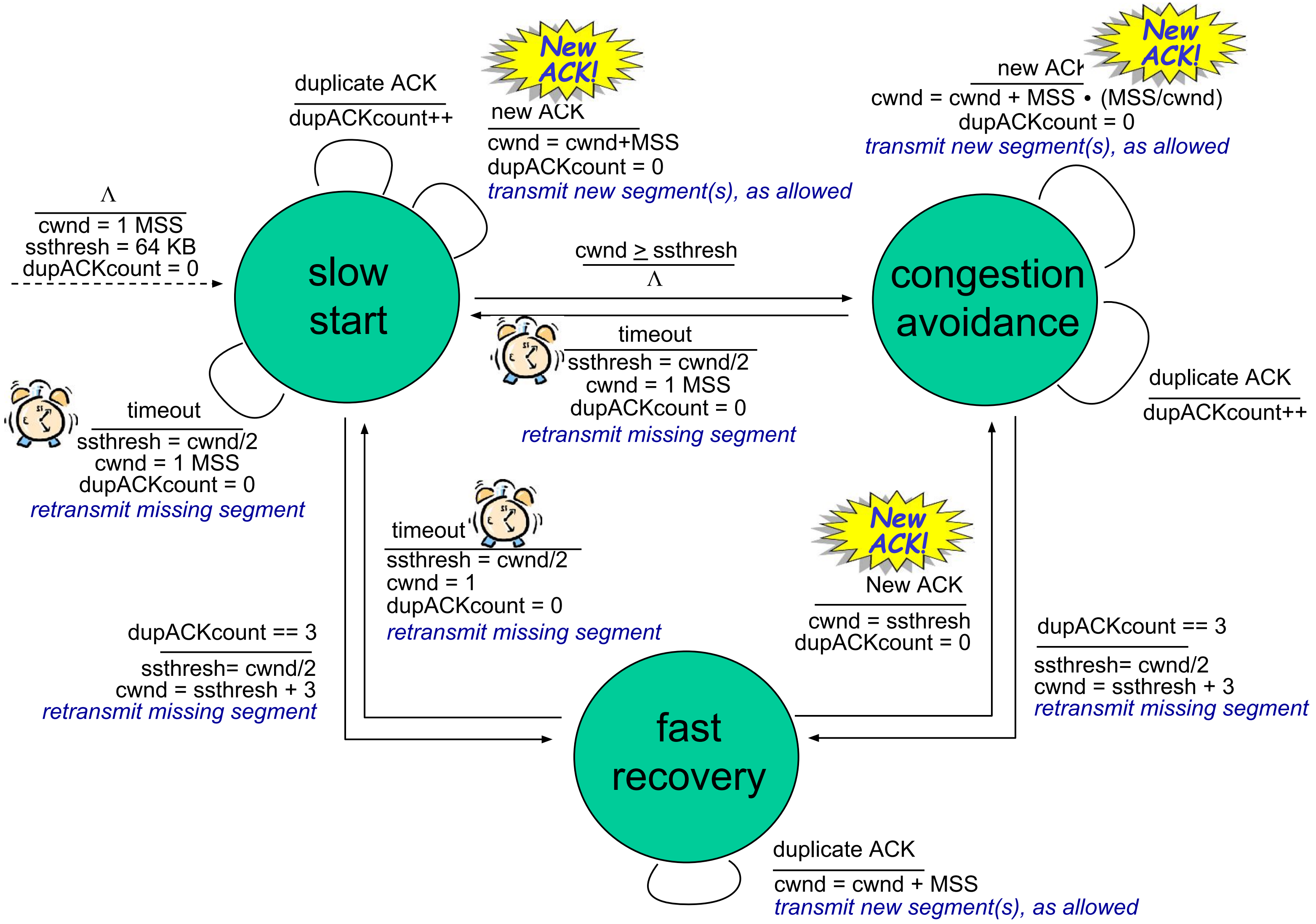


Implementation:

- variable **ssthresh**
- on loss event, **ssthresh** is set to 1/2 of **cwnd** just before loss event

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

Summary: TCP Congestion Control



Summary: TCP congestion control

- When CongWin is below Threshold, sender in **slow start** phase, window grows exponentially.
- When CongWin is above Threshold, sender is in **congestion avoidance** phase, window grows linearly.
- When a triple duplicate ACK occurs, Threshold set to CongWin/2 and CongWin set to Threshold + 3.
- When timeout occurs, Threshold set to CongWin/2 and CongWin is set to 1 MSS.