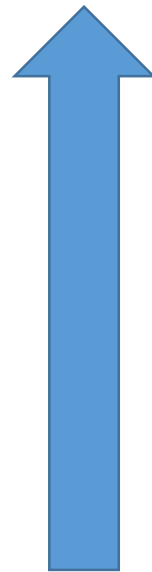
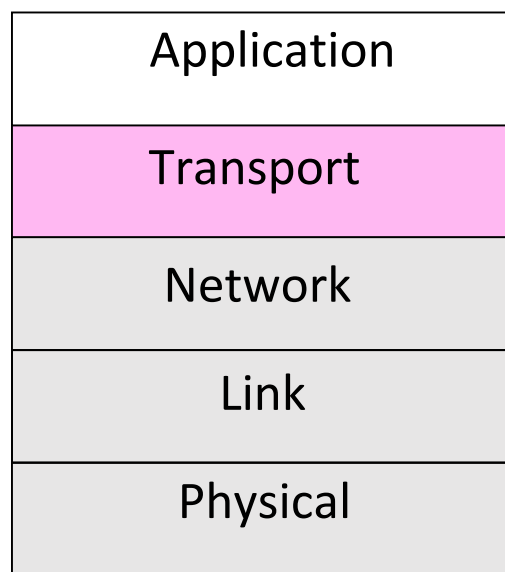


Transport Layer (TCP/UDP)

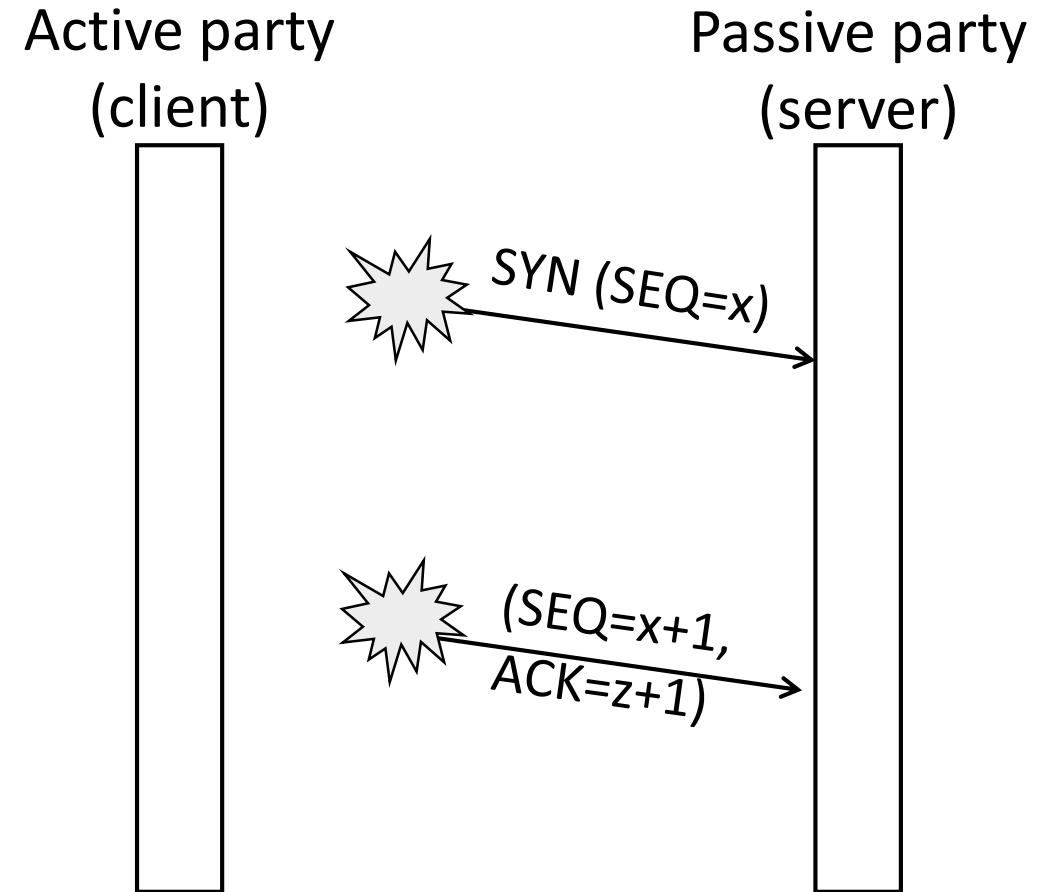
Where we are in the Course

- Moving on up to the Transport Layer!



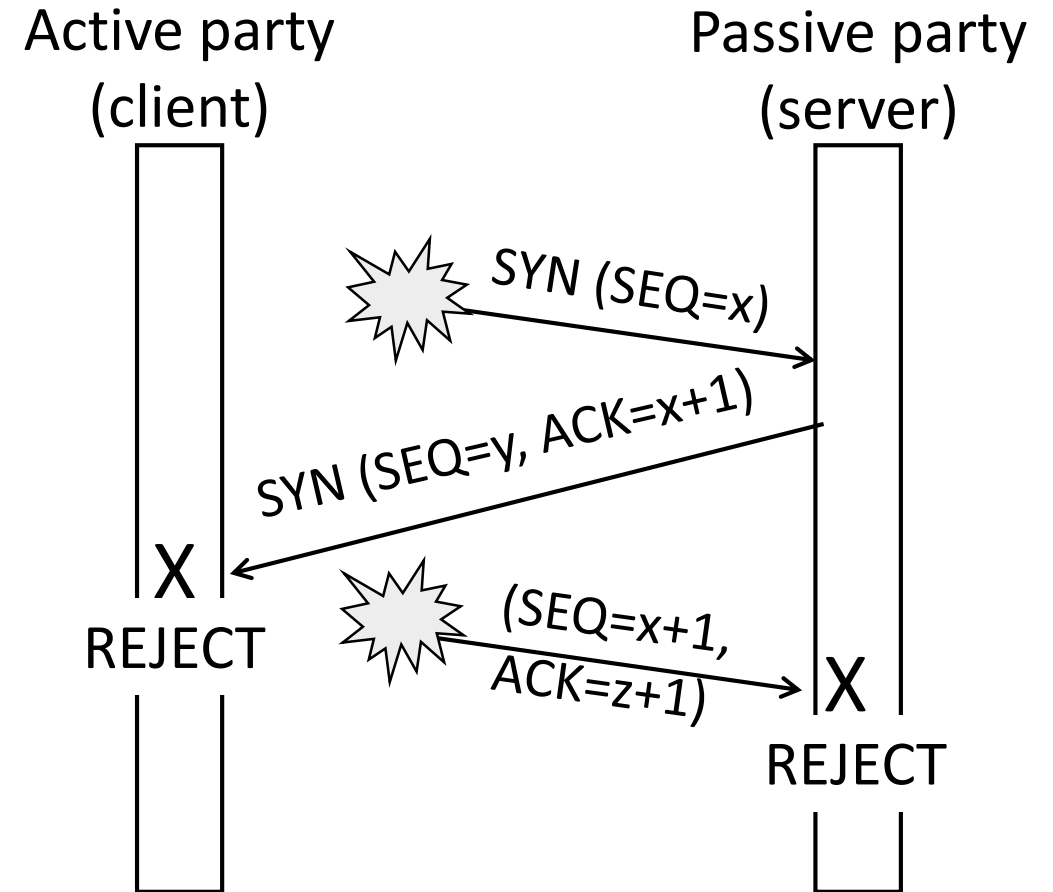
Three-Way Handshake (3)

- Suppose delayed, duplicate copies of the SYN and ACK arrive at the server!
 - Improbable, but anyhow ...



Three-Way Handshake (4)

- Suppose delayed, duplicate copies of the SYN and ACK arrive at the server!
 - Improbable, but anyhow ...
- Connection will be cleanly rejected on both sides 😊



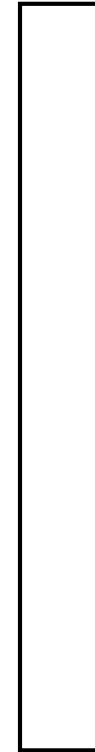
Connection Release

- Orderly release by both parties when done
 - Delivers all pending data and “hangs up”
 - Cleans up state in sender and receiver
- Key problem is to provide reliability while releasing
 - TCP uses a “symmetric” close in which both sides shutdown independently

TCP Connection Release

- **Two steps:**
 - Active sends FIN(x), passive ACKs
 - Passive sends FIN(y), active ACKs
 - FINs are retransmitted if lost
- Each FIN/ACK closes one direction of data transfer

Active party

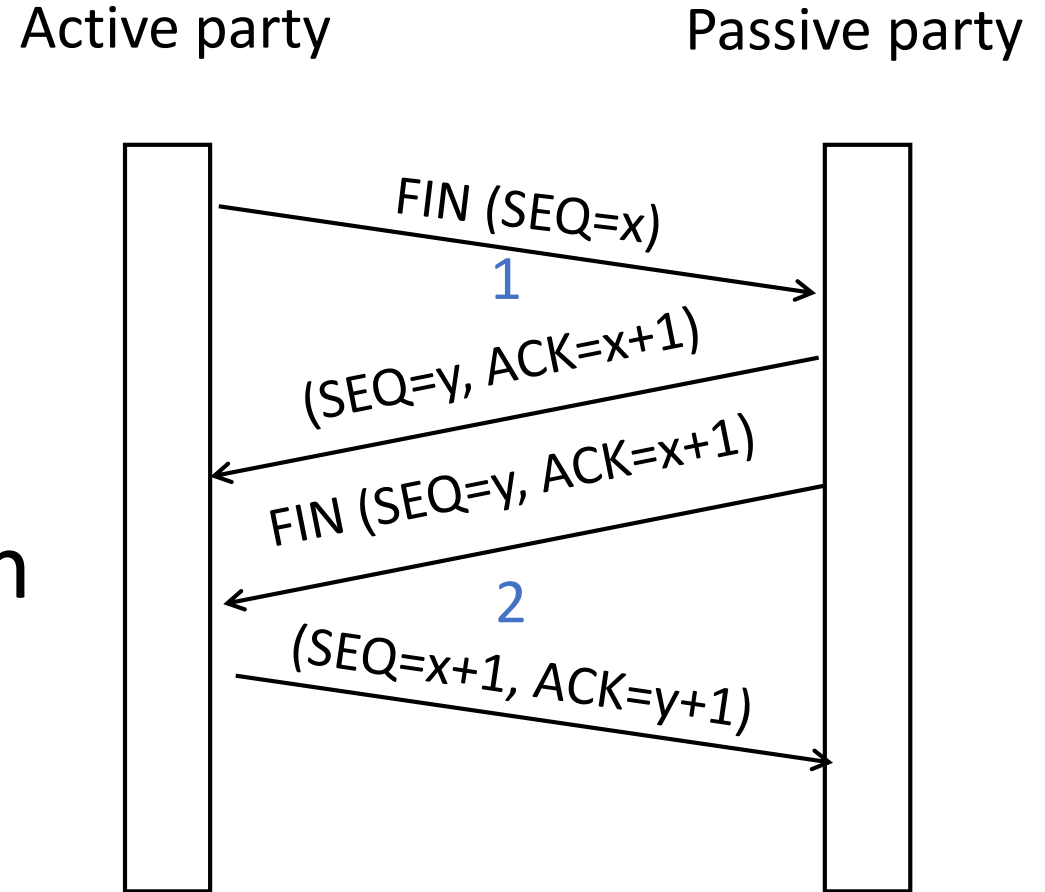


Passive party



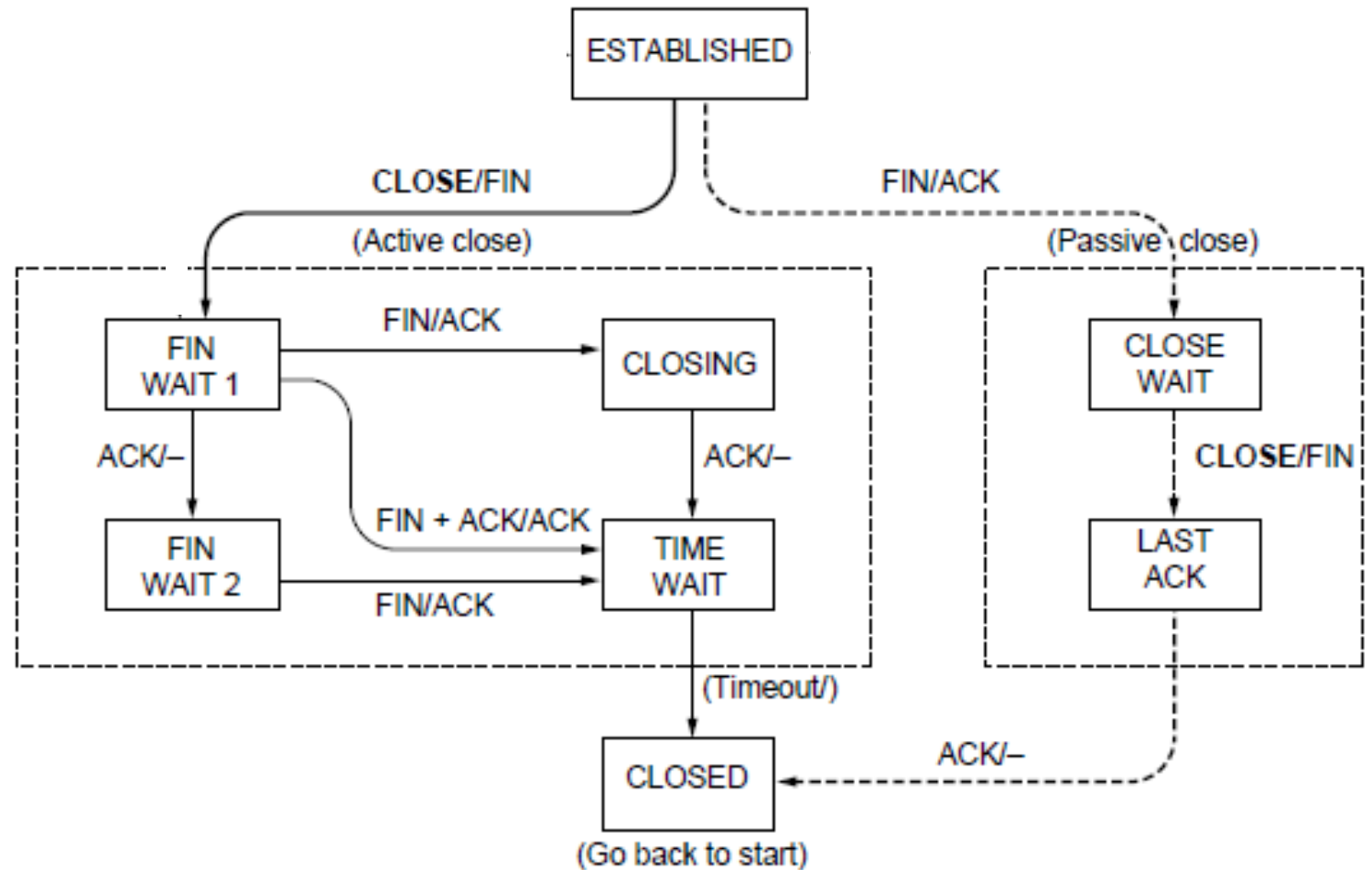
TCP Connection Release (2)

- Two steps:
 - Active sends FIN(x), passive ACKs
 - Passive sends FIN(y), active ACKs
 - FINs are retransmitted if lost
- Each FIN/ACK closes one direction of data transfer



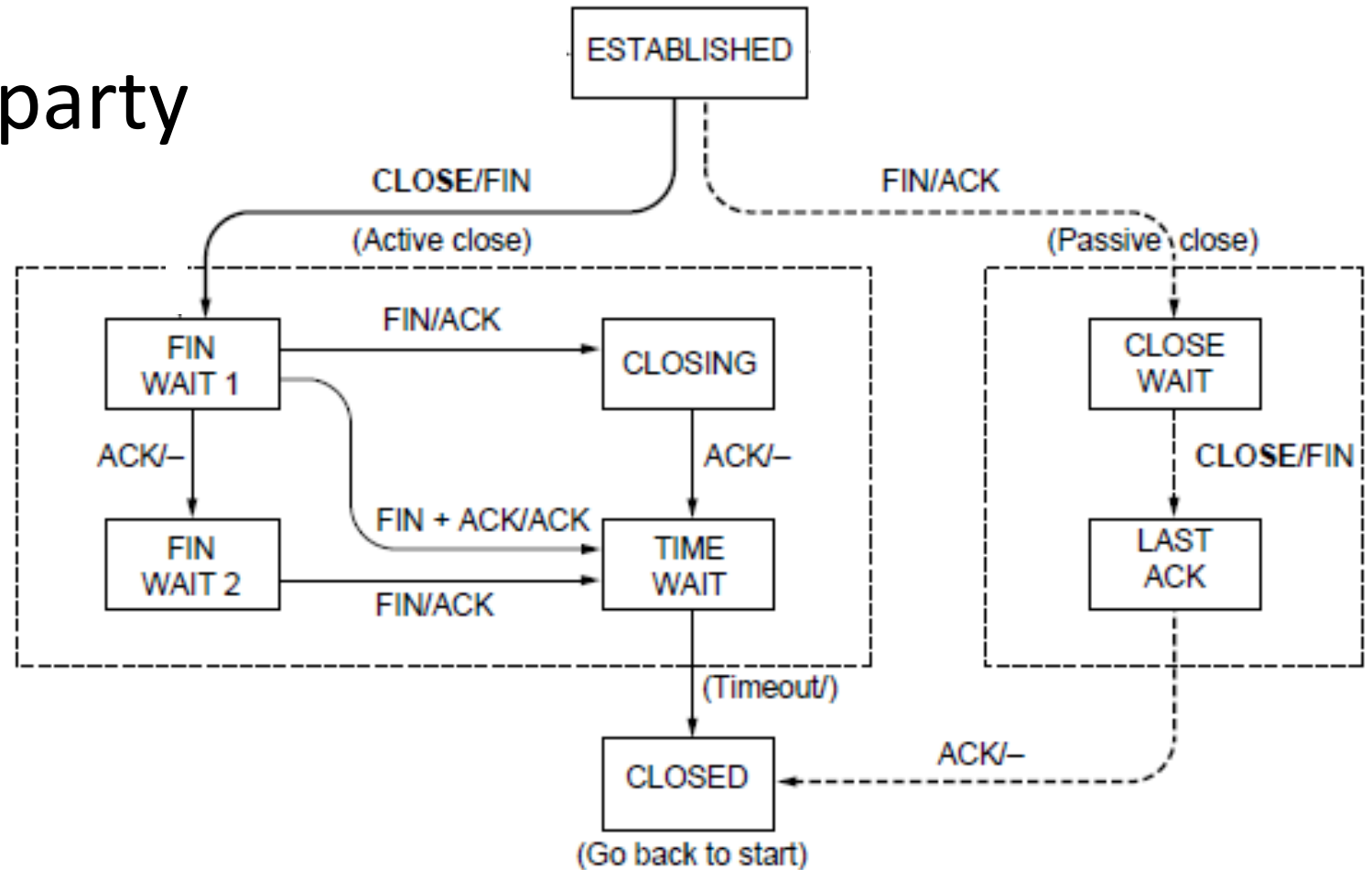
TCP Connection State Machine

Both parties run instances of this state machine



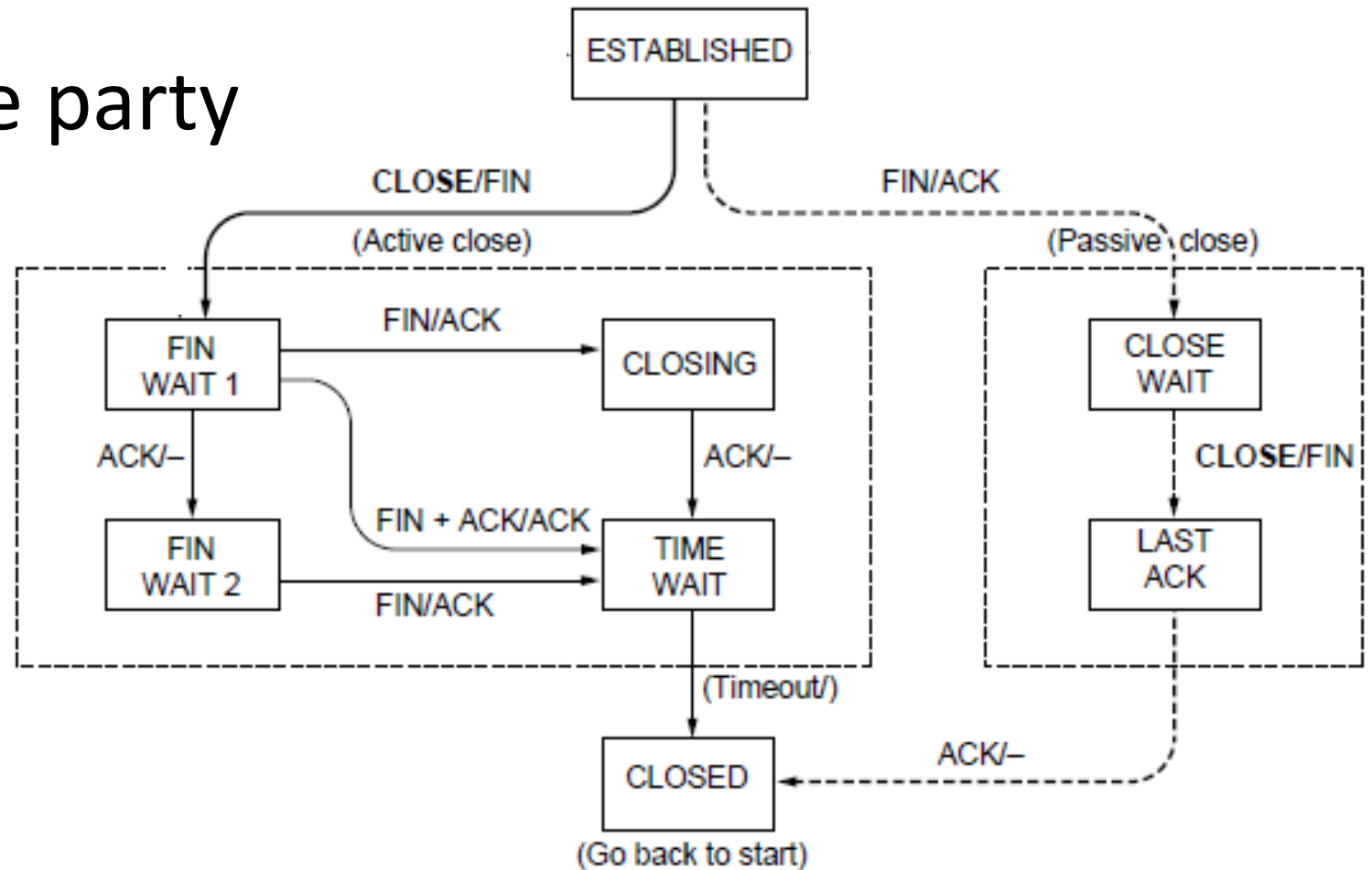
TCP Release

- Follow the active party



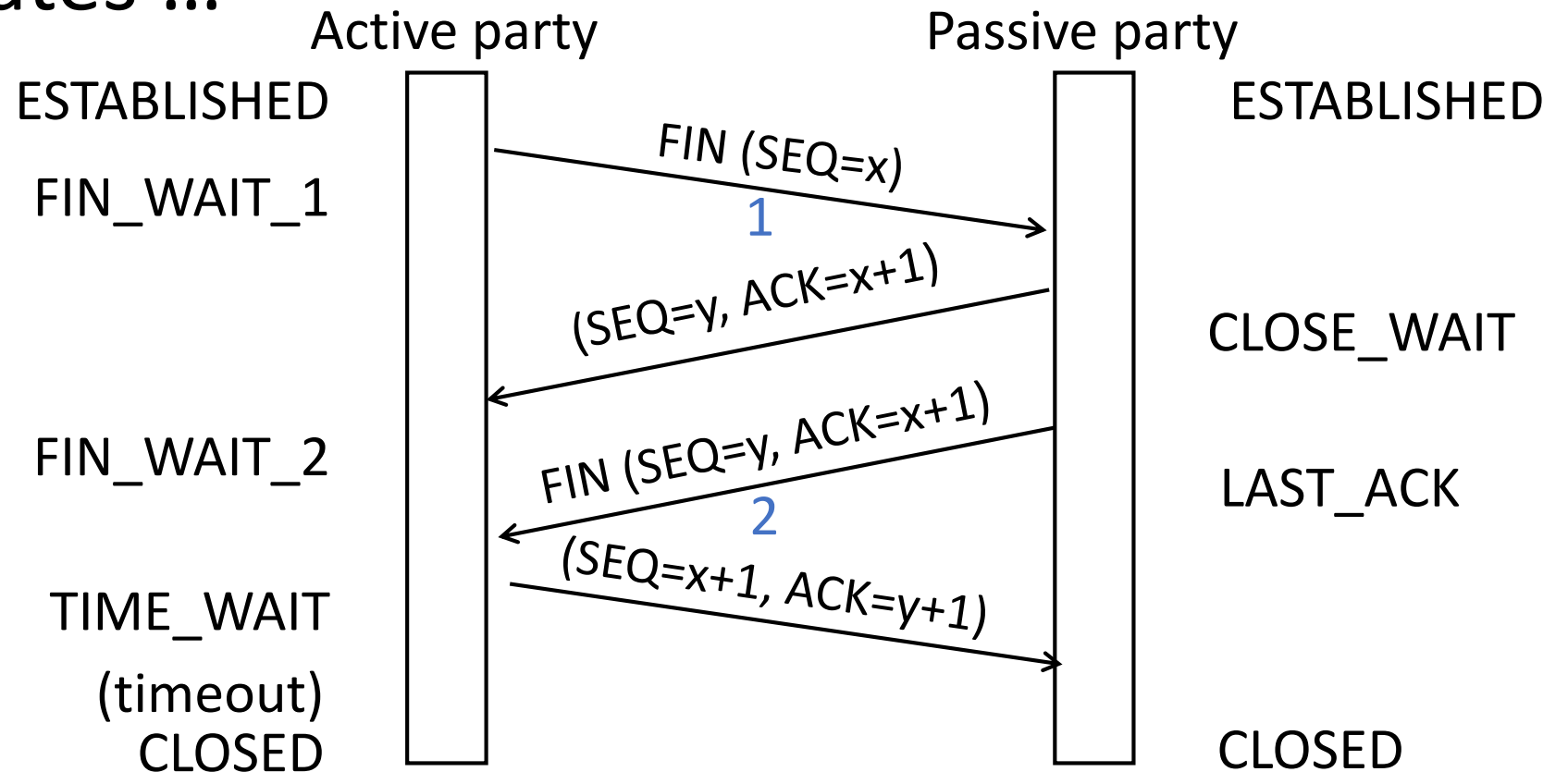
TCP Release (2)

- Follow the passive party



TCP Release (3)

- Again, with states ...



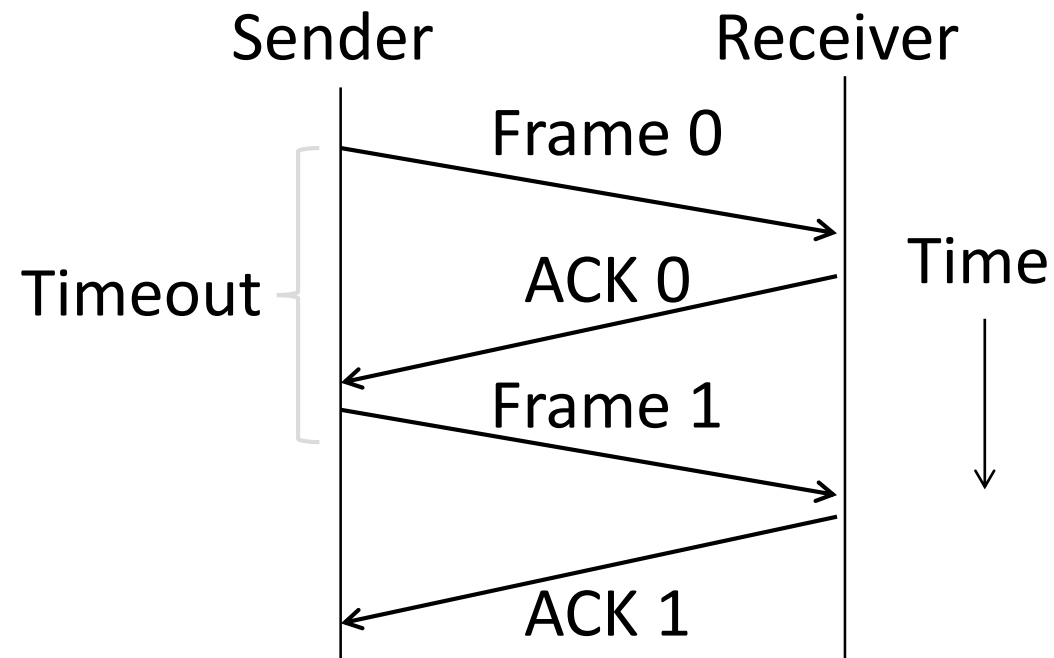
TIME_WAIT State

- Wait a long time after sending all segments and before completing the close
 - Two times the maximum segment lifetime of 60 seconds
- Why?
 - ACK might have been lost, in which case FIN will be resent for an orderly close
 - Could otherwise interfere with a subsequent connection

Flow Control

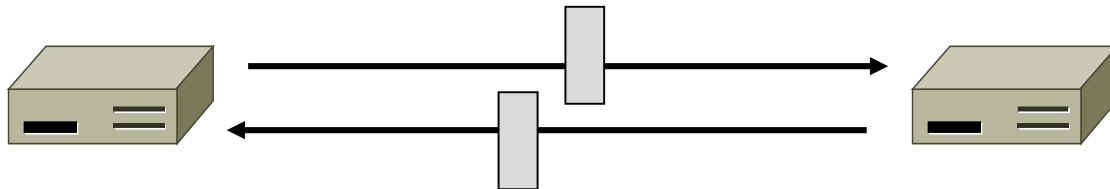
Recall

- ARQ with one message at a time is Stop-and-Wait (normal case below)



Limitation of Stop-and-Wait

- It allows only a single message to be outstanding from the sender:
 - Fine for LAN (only one frame fit)
 - Not efficient for network paths with $BD \gg 1$ packet



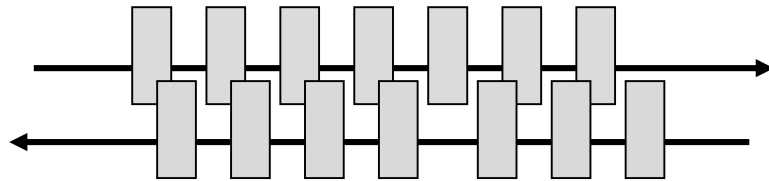
Limitation of Stop-and-Wait (2)

- Example: $R=1$ Mbps, $D = 50$ ms
 - RTT (Round Trip Time) = $2D = 100$ ms
 - How many packets/sec?

- What if $R=10$ Mbps?

Sliding Window

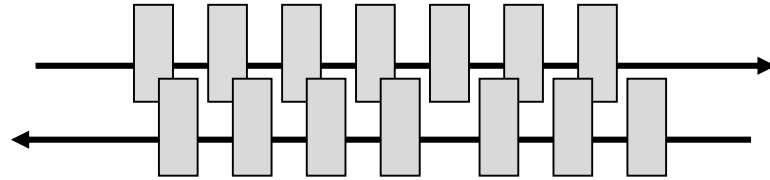
- Generalization of stop-and-wait
 - Allows W packets to be outstanding
 - Can send W packets per RTT ($=2D$)



- Pipelining improves performance
- Need $W=2BD$ to fill network path

Sliding Window (3)

- Ex: $R=1$ Mbps, $D = 50$ ms
 - $2BD = 10^6$ b/sec $\times 100 \cdot 10^{-3}$ sec = 100 kbit
 - $W = 2BD = 10$ packets of 1250 bytes



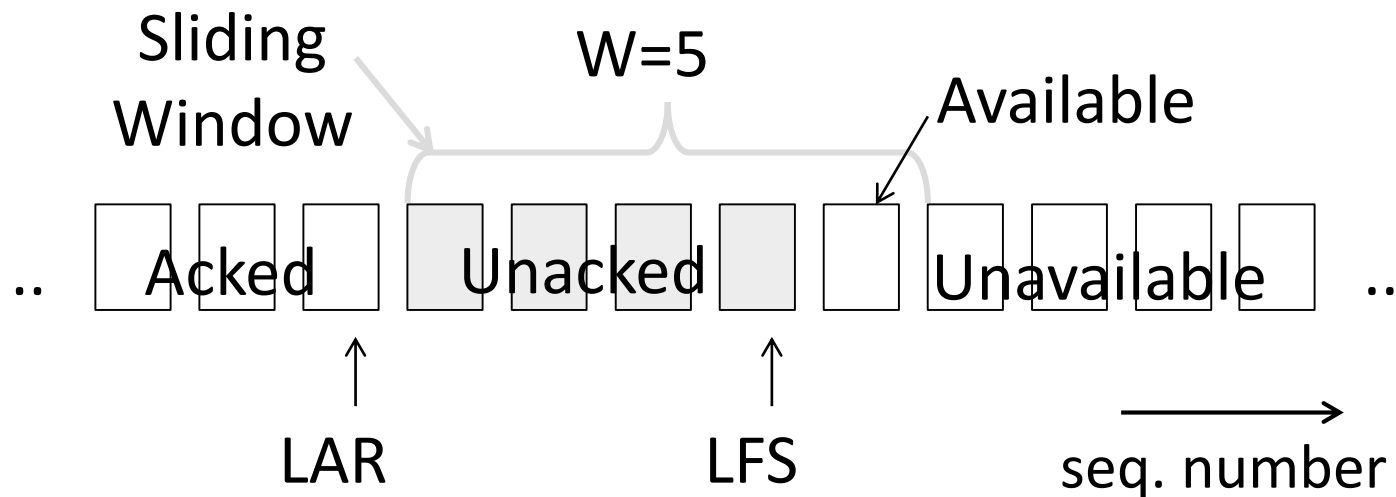
- Ex: What if $R=10$ Mbps?
 - $2BD = 1000$ kbit
 - $W = 2BD = 100$ packets of 1250 bytes

Sliding Window Protocol

- Many variations, depending on how buffers, acknowledgements, and retransmissions are handled
- Go-Back-N
 - Simplest version, can be inefficient
- Selective Repeat
 - More complex, better performance

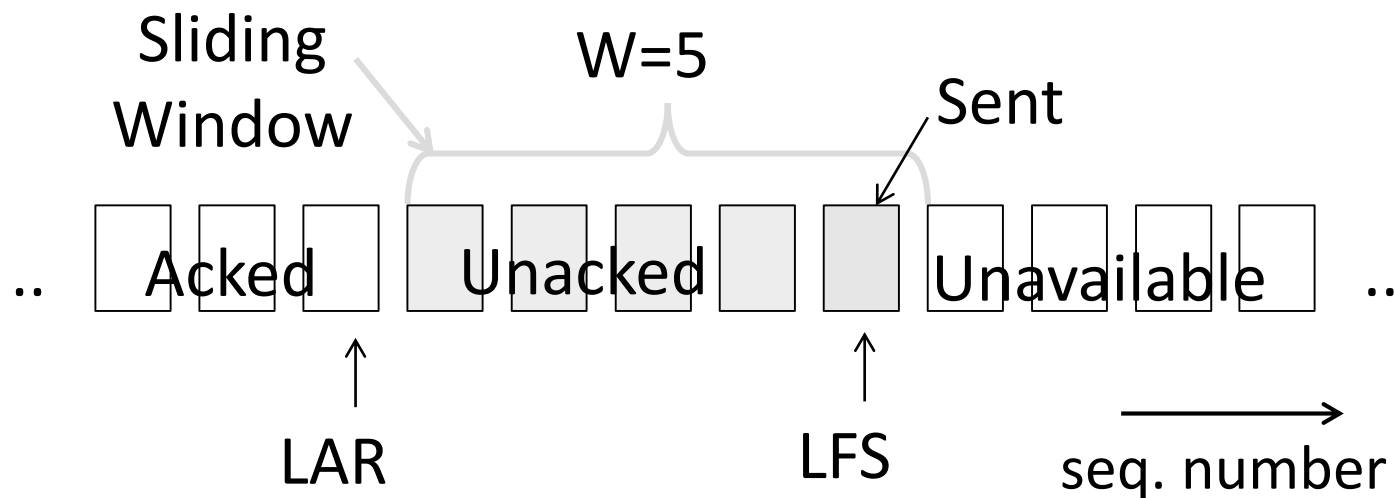
Sliding Window – Sender

- Sender buffers up to W segments until they are acknowledged
 - LFS=LAST FRAME SENT, LAR=LAST ACK REC'D
 - Sends while $LFS - LAR \leq W$



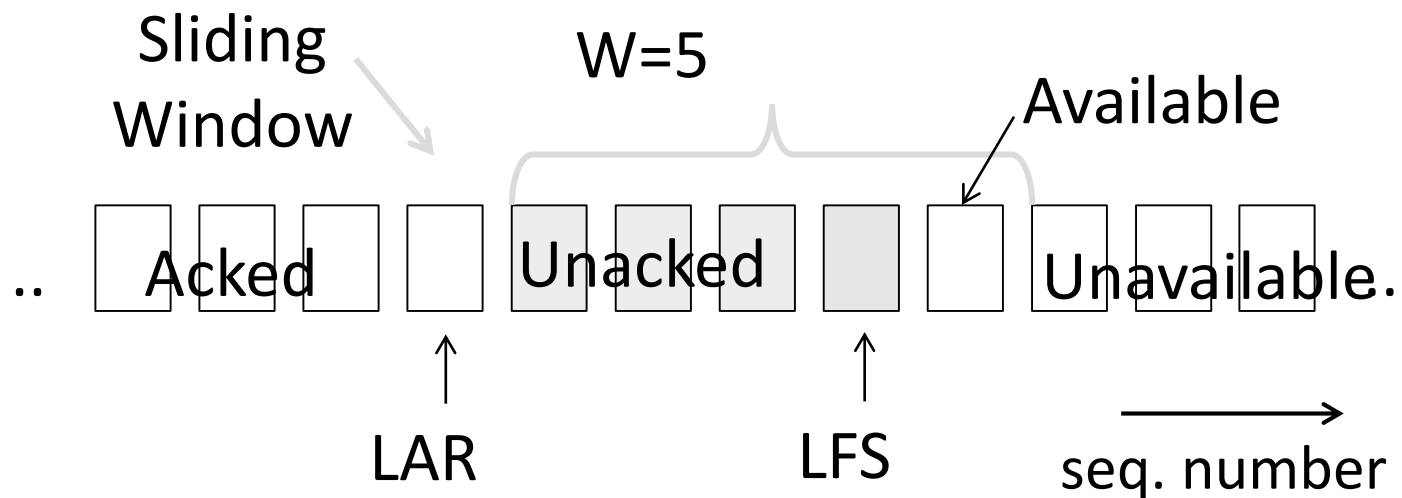
Sliding Window – Sender (2)

- Transport accepts another segment of data from the Application ...
 - Transport sends it (as LFS–LAR \rightarrow 5)



Sliding Window – Sender (3)

- Next higher ACK arrives from peer...
 - Window advances, buffer is freed
 - LFS–LAR \rightarrow 4 (can send one more)



Sliding Window – Go-Back-N

- Receiver keeps only a single packet buffer for the next segment
 - State variable, $LAS = \text{LAST ACK SENT}$
- On receive:
 - If seq. number is $LAS+1$, accept and pass it to app, update LAS , send ACK
 - Otherwise discard (as out of order)

Sliding Window – Selective Repeat

- Receiver passes data to app in order, and buffers out-of-order segments to reduce retransmissions
- ACK conveys highest in-order segment, plus hints about out-of-order segments
- TCP uses a selective repeat design; we'll see the details later

Sliding Window – Selective Repeat (2)

- Buffers W segments, keeps state variable $LAS = \text{LAST ACK SENT}$
- On receive:
 - Buffer segments $[LAS+1, LAS+W]$
 - Send app in-order segments from $LAS+1$, and update LAS
 - Send ACK for LAS regardless

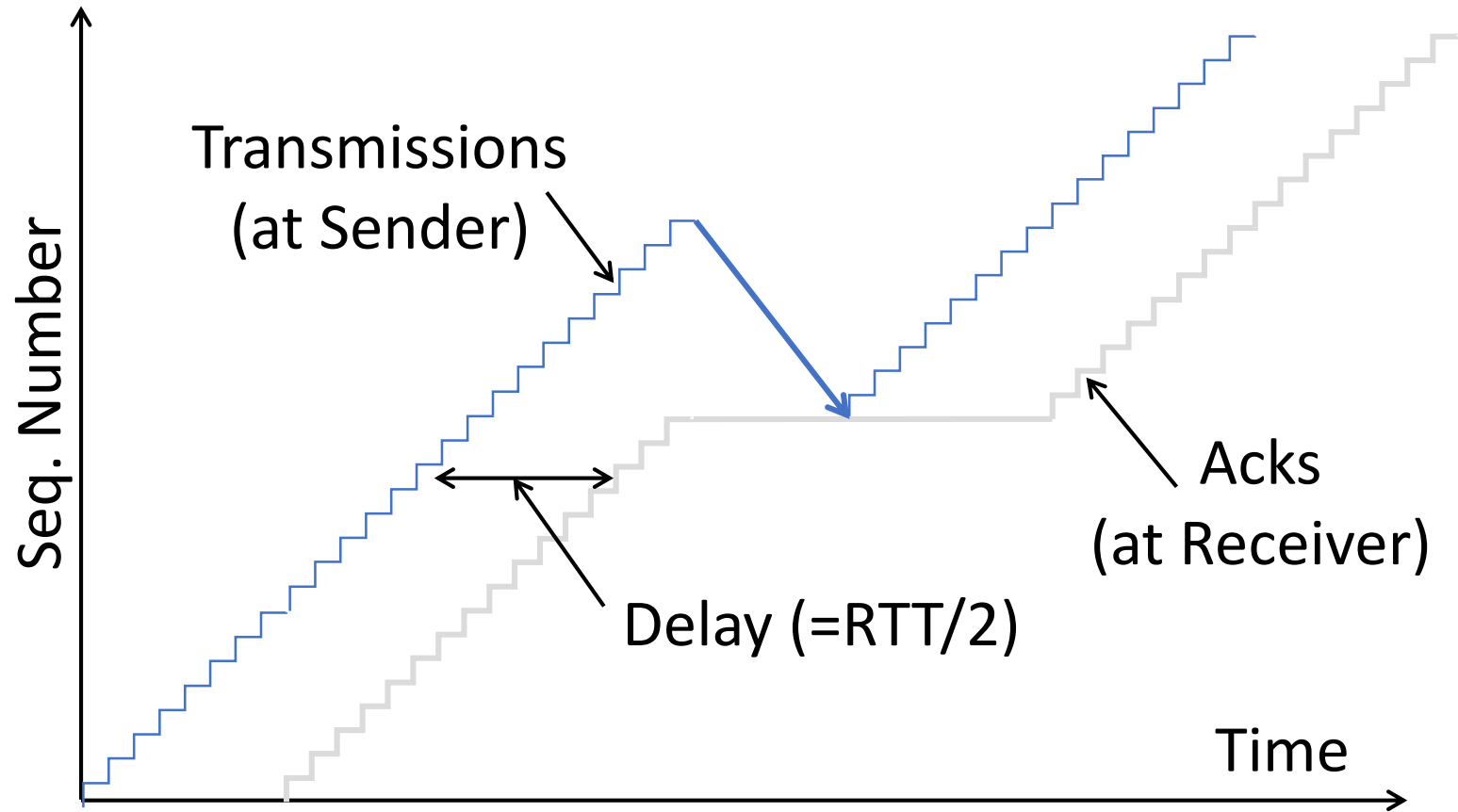
Sliding Window – Retransmissions

- Go-Back-N uses a single timer to detect losses
 - On timeout, resends buffered packets starting at LAR+1
- Selective Repeat uses a timer per unacked segment to detect losses
 - On timeout for segment, resend it
 - Hope to resend fewer segments

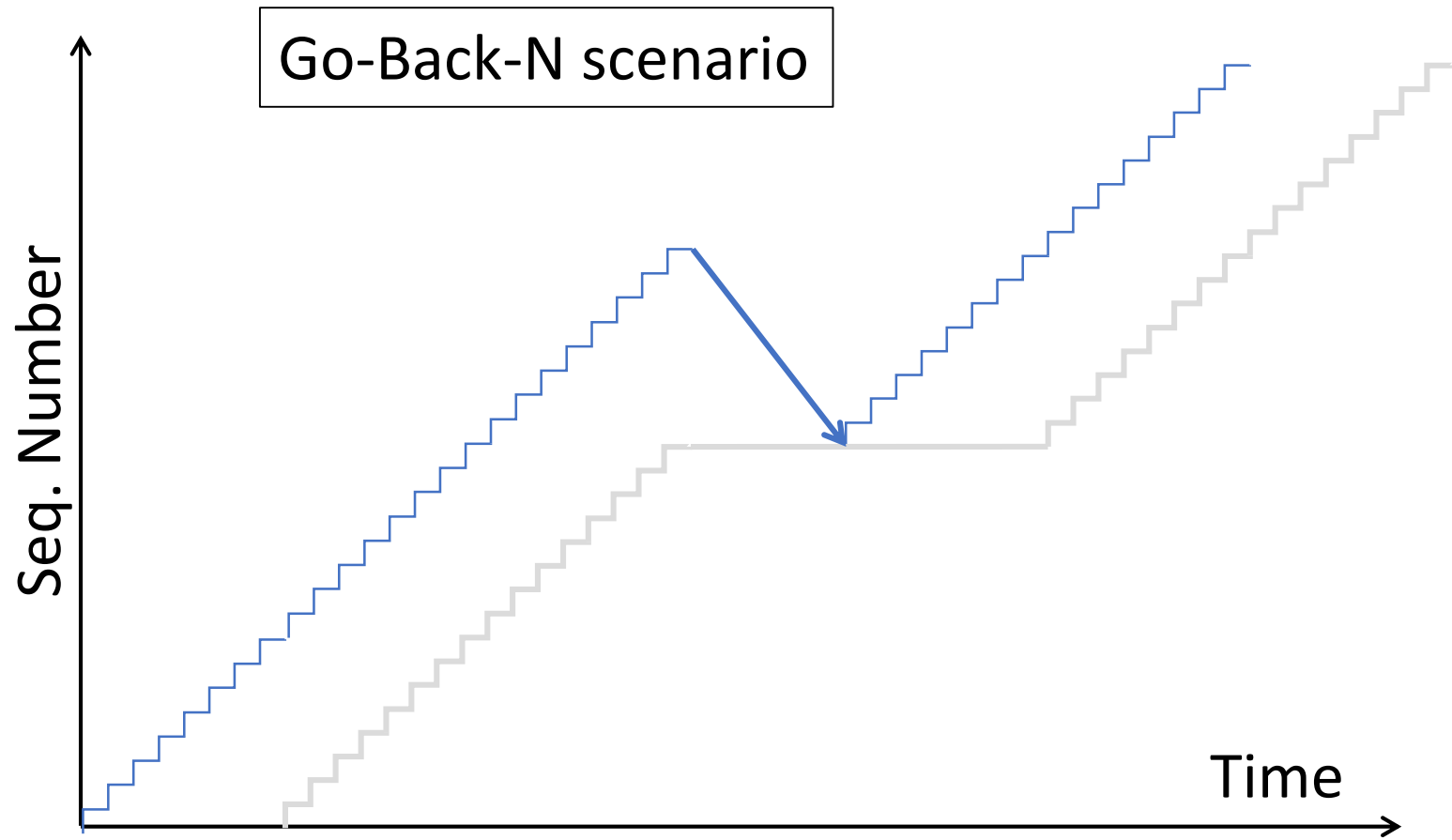
Sequence Numbers

- Need more than 0/1 for Stop-and-Wait ...
 - But how many?
- For Selective Repeat, need W numbers for packets, plus W for acks of earlier packets
 - $2W$ seq. numbers
 - Fewer for Go-Back- N ($W+1$)
- Typically implement seq. number with an N -bit counter that wraps around at $2^N - 1$
 - E.g., $N=8$: ..., 253, 254, 255, 0, 1, 2, 3, ...

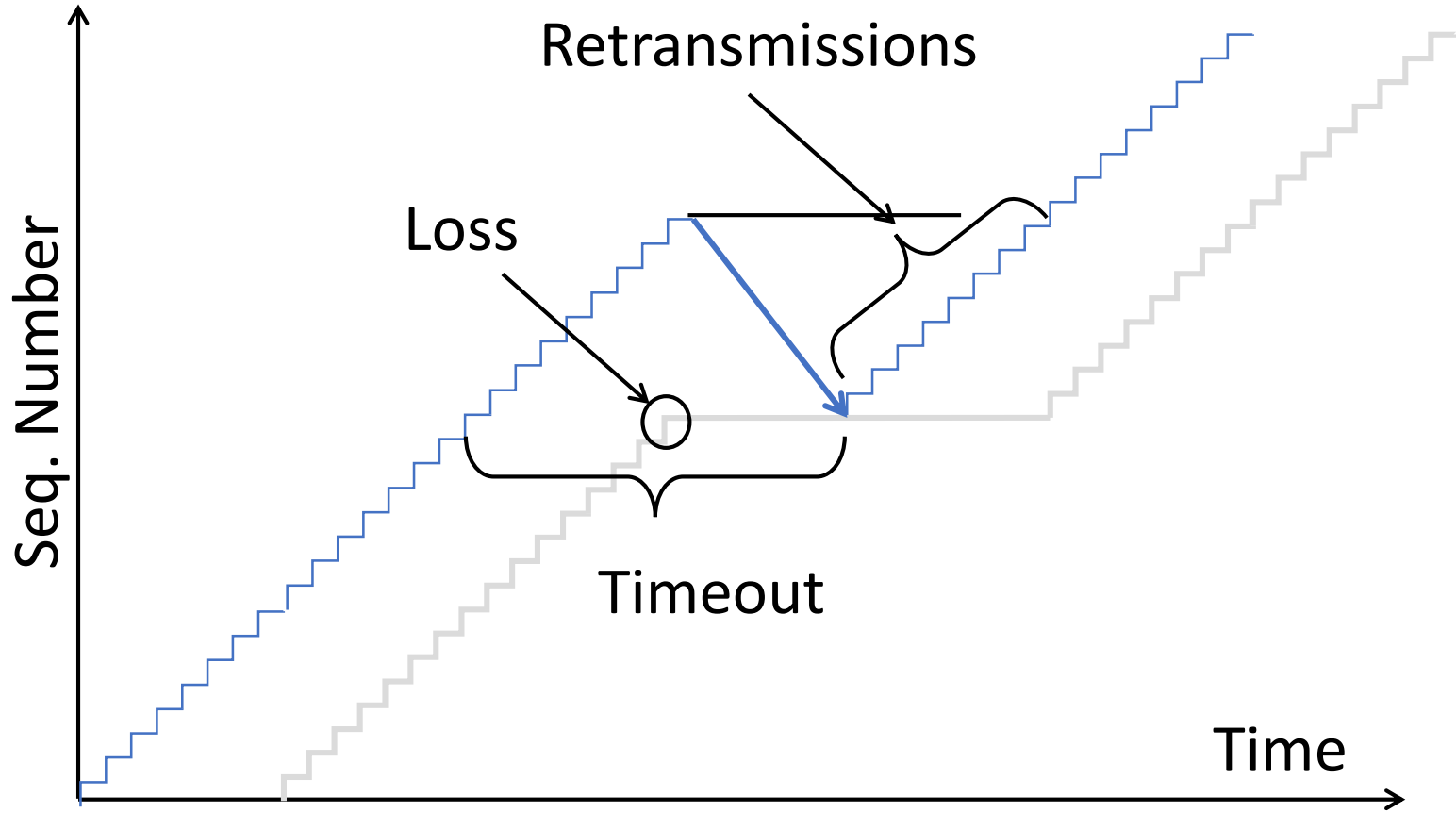
Sequence Time Plot



Sequence Time Plot (2)

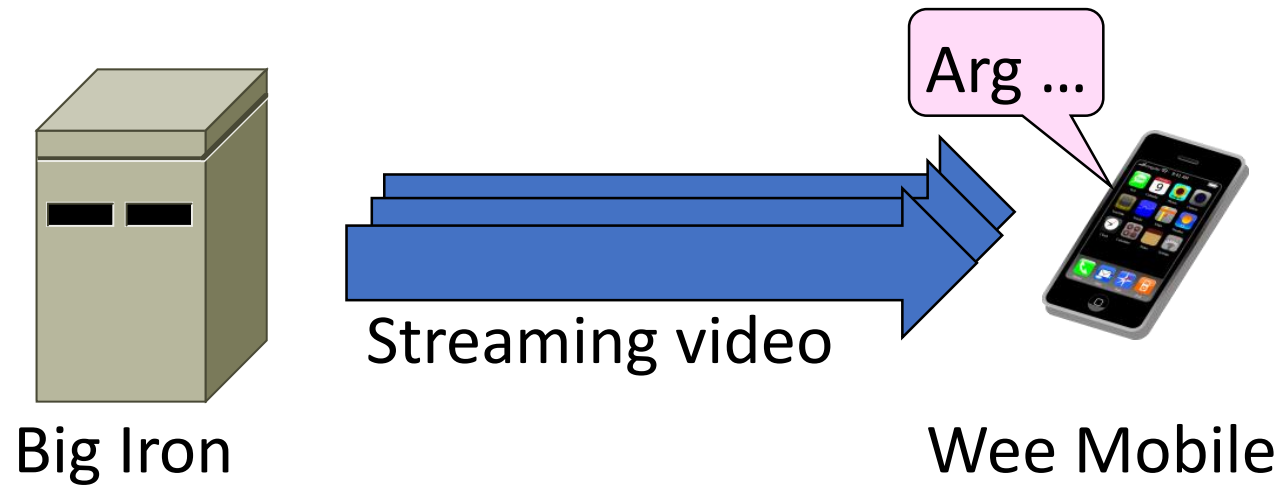


Sequence Time Plot (3)



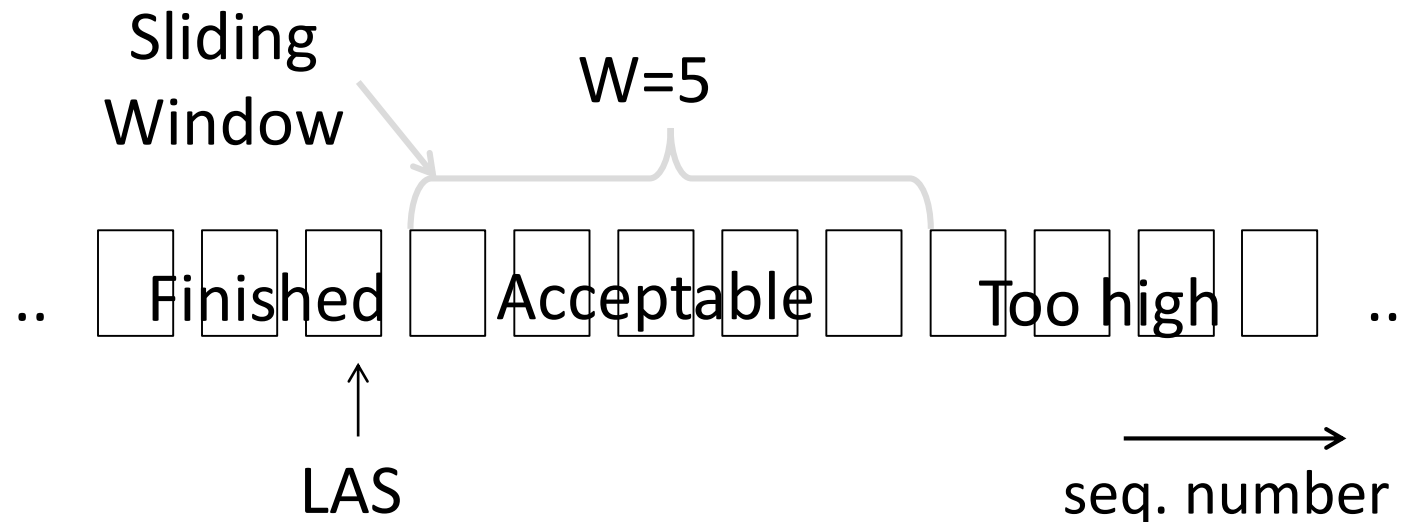
Problem

- Sliding window has pipelining to keep network busy
 - What if the receiver is overloaded?



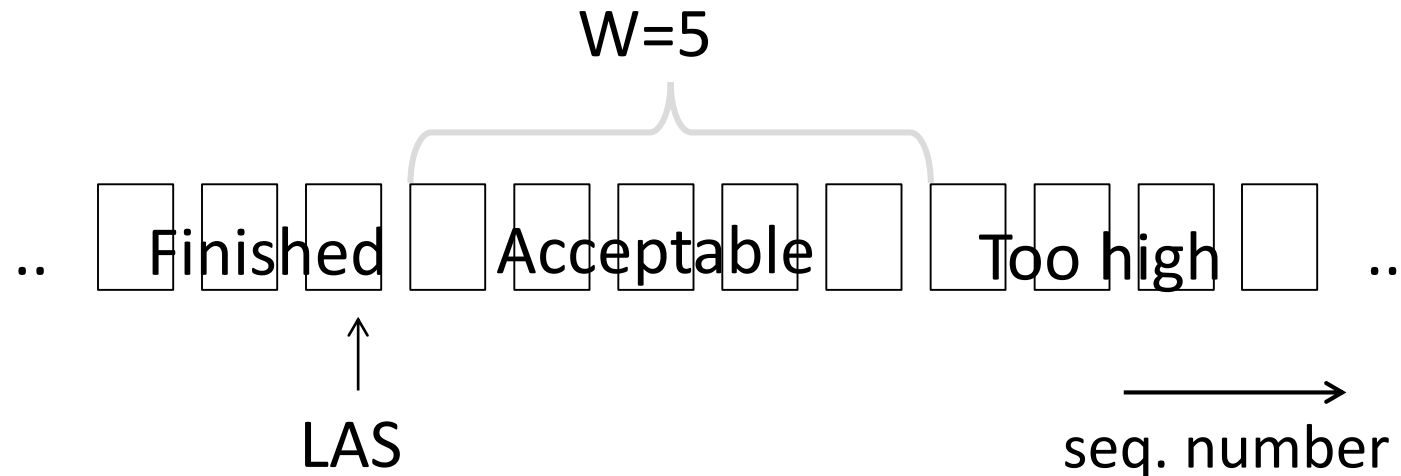
Sliding Window – Receiver

- Consider receiver with W buffers
 - LAS=LAST ACK SENT, app pulls in-order data from buffer with `recv()` call



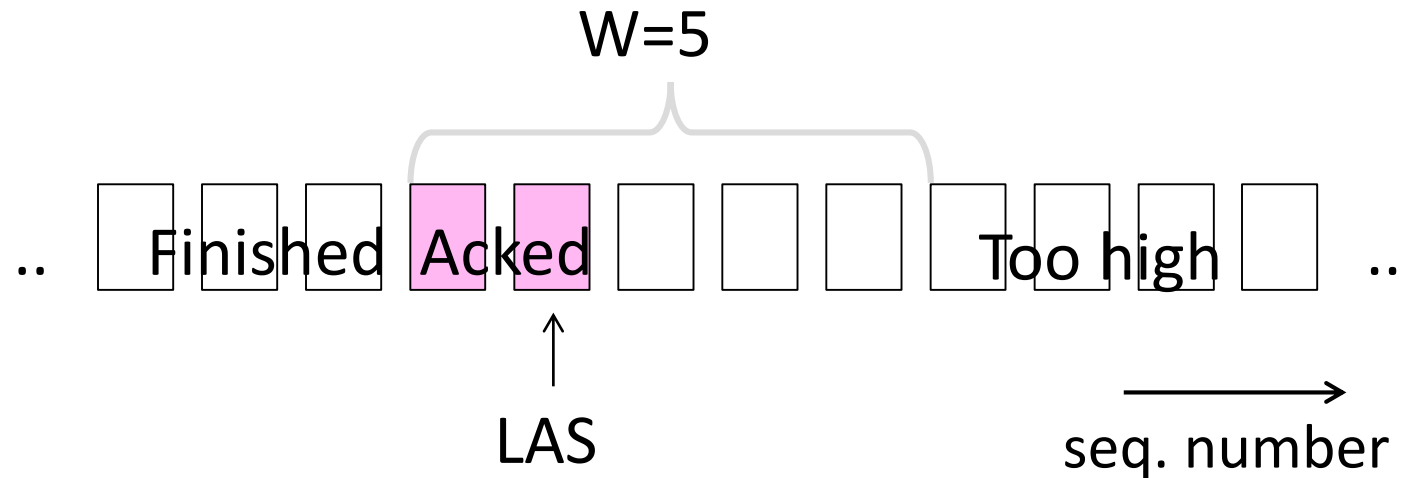
Sliding Window – Receiver (2)

- Suppose the next two segments arrive but app does not call `recv()`



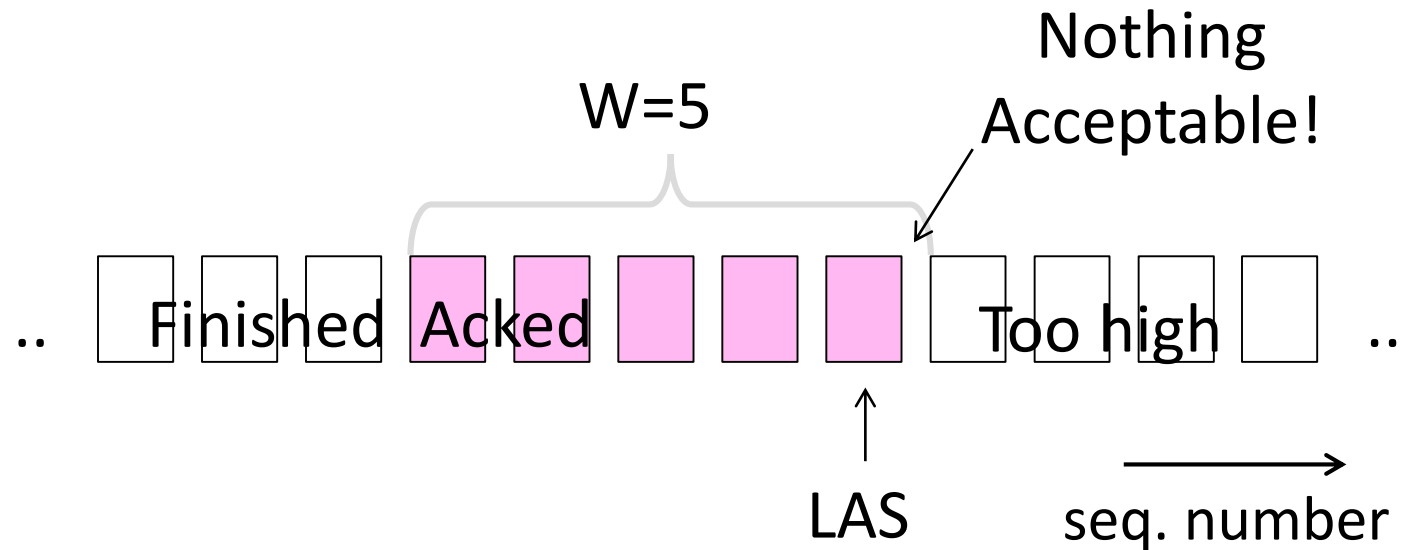
Sliding Window – Receiver (3)

- Suppose the next two segments arrive but app does not call `recv()`
 - LAS rises, but we can't slide window!



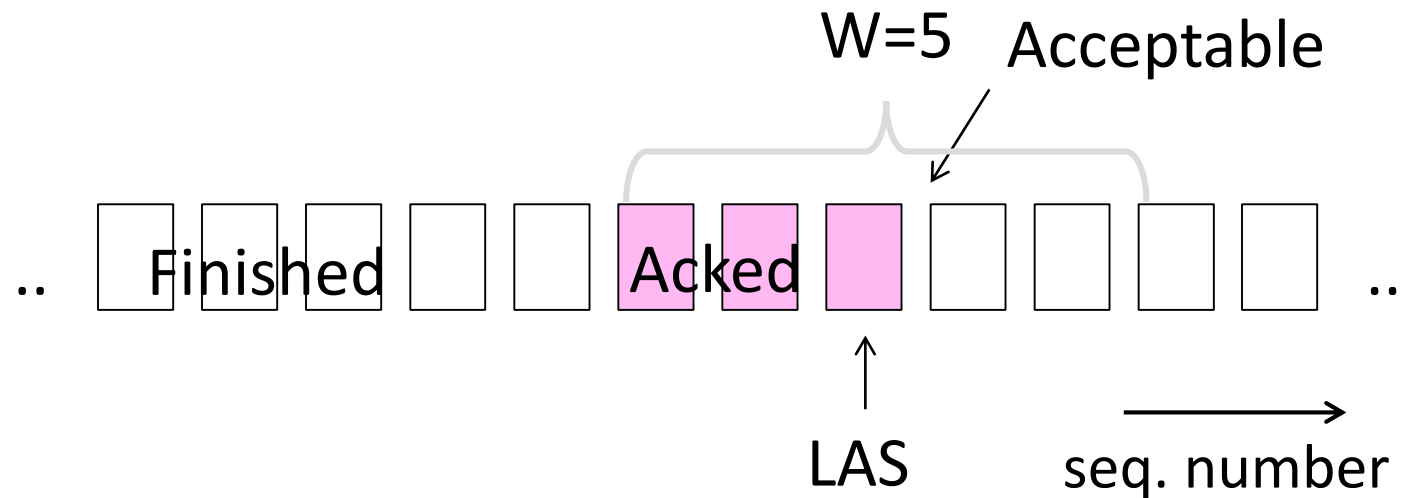
Sliding Window – Receiver (4)

- Further segments arrive (in order) we fill buffer
 - Must drop segments until app recvs!



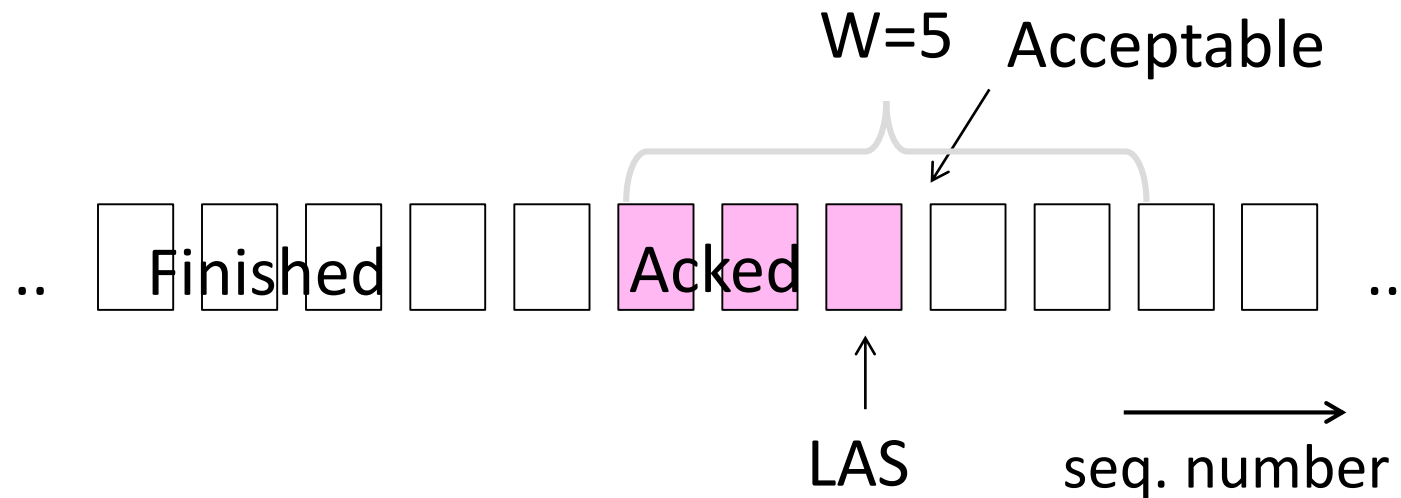
Sliding Window – Receiver (5)

- App recv() takes two segments
 - Window slides (phew)



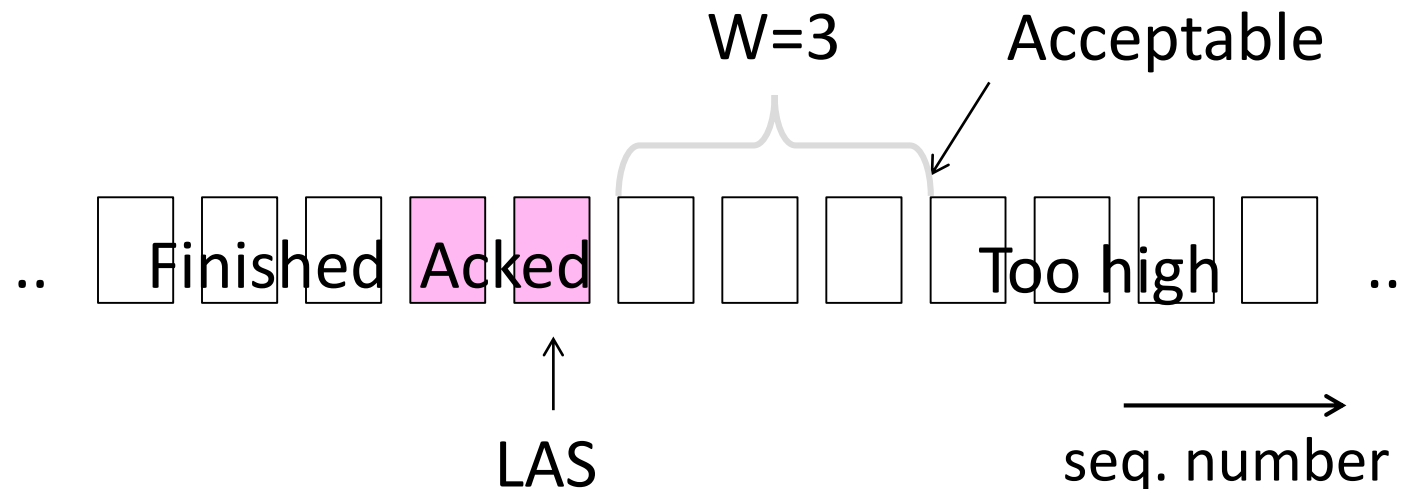
Flow Control

- Avoid loss at receiver by telling sender the available buffer space
 - $WIN = \# \text{Acceptable}$, not W (from LAS)



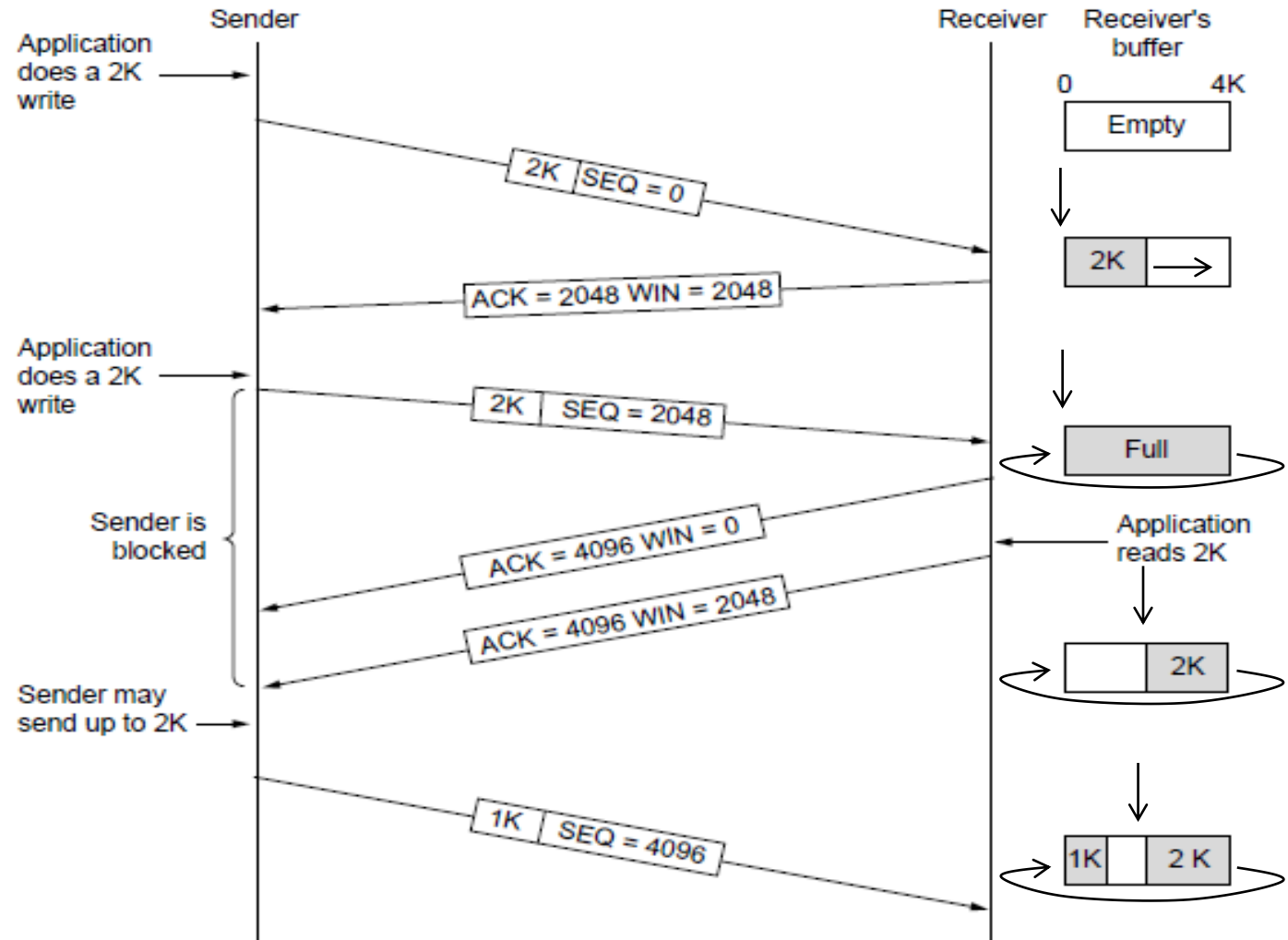
Flow Control (2)

- Sender uses lower of the sliding window and flow control window (WIN) as the effective window size



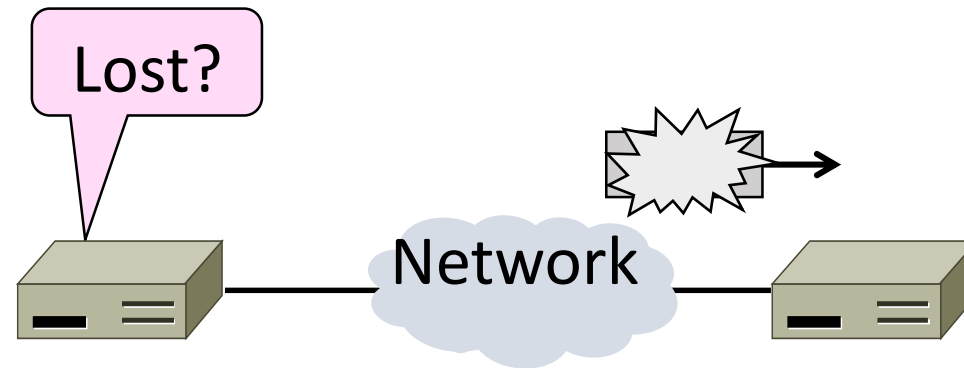
Flow Control (3)

- TCP-style example
 - SEQ/ACK sliding window
 - Flow control with WIN
 - $SEQ + length < ACK + WIN$
 - 4KB buffer at receiver
 - Circular buffer of bytes



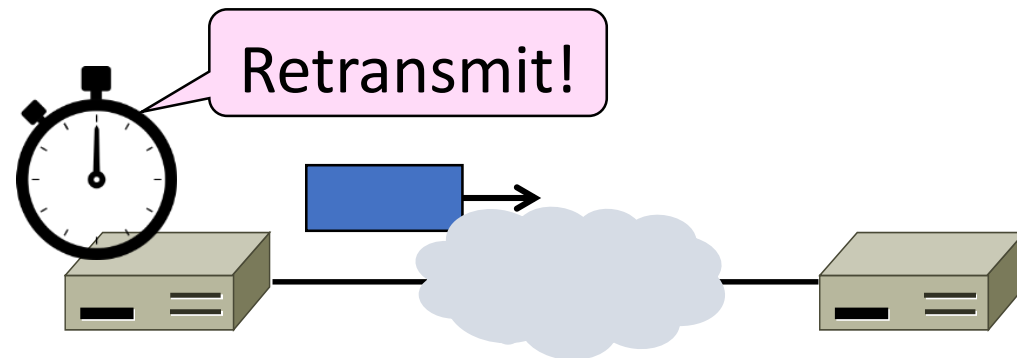
Topic

- How to set the timeout for sending a retransmission
 - Adapting to the network path



Retransmissions

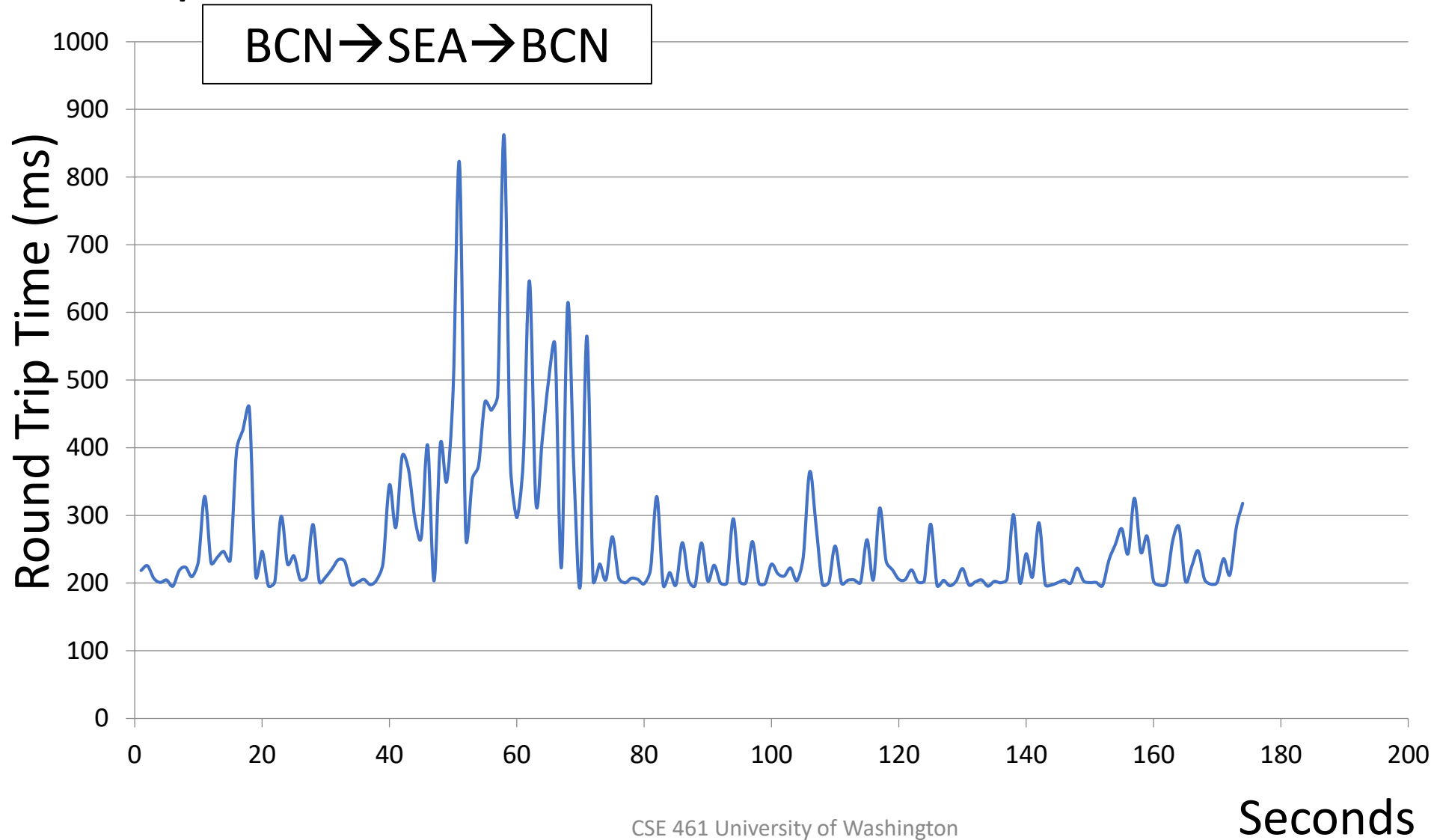
- With sliding window, detecting loss with timeout
 - Set timer when a segment is sent
 - Cancel timer when ack is received
 - If timer fires, retransmit data as lost



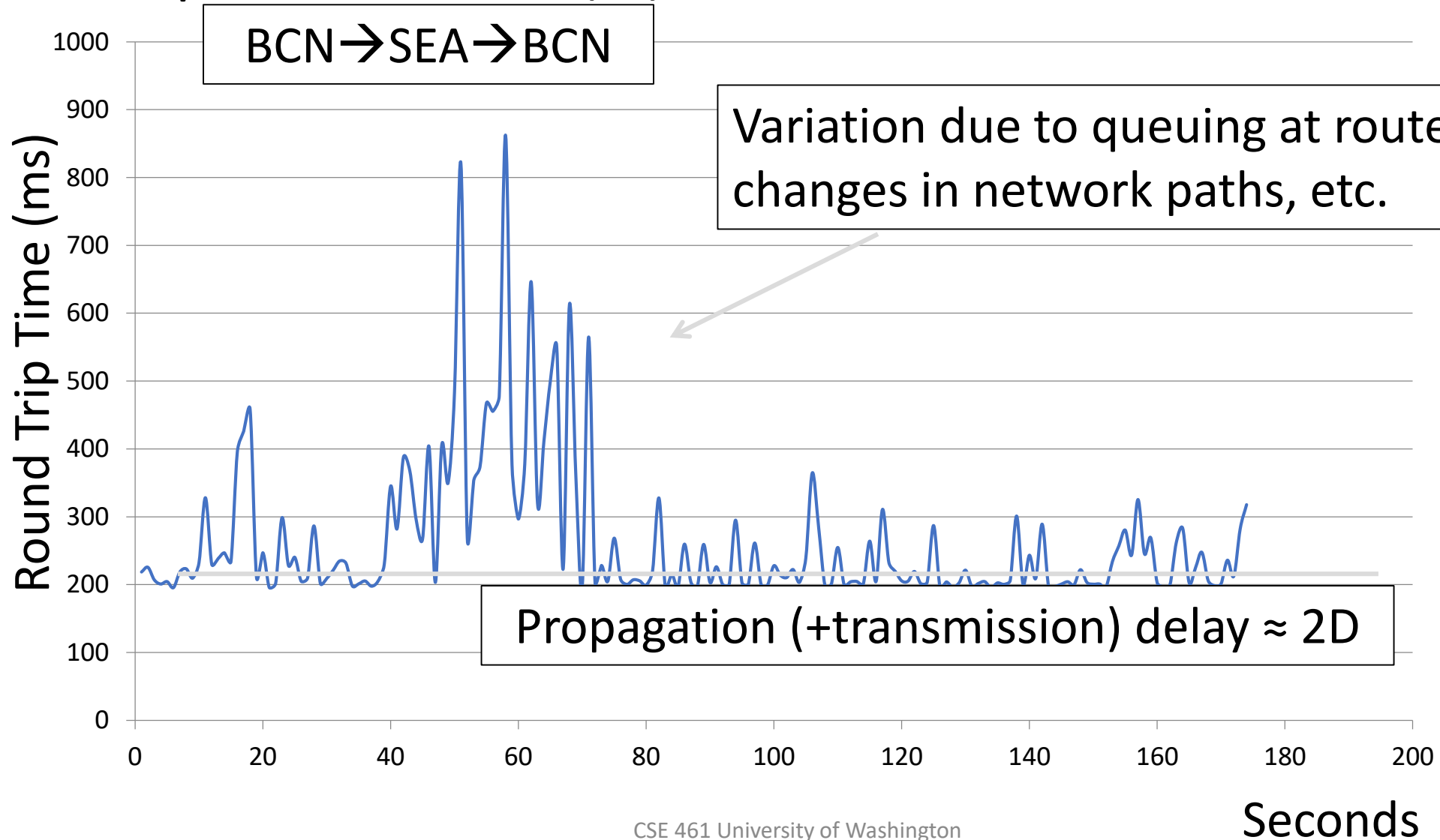
Timeout Problem

- Timeout should be “just right”
 - Too long wastes network capacity
 - Too short leads to spurious resends
 - But what is “just right”?
- Easy to set on a LAN (Link)
 - Short, fixed, predictable RTT
- Hard on the Internet (Transport)
 - Wide range, variable RTT

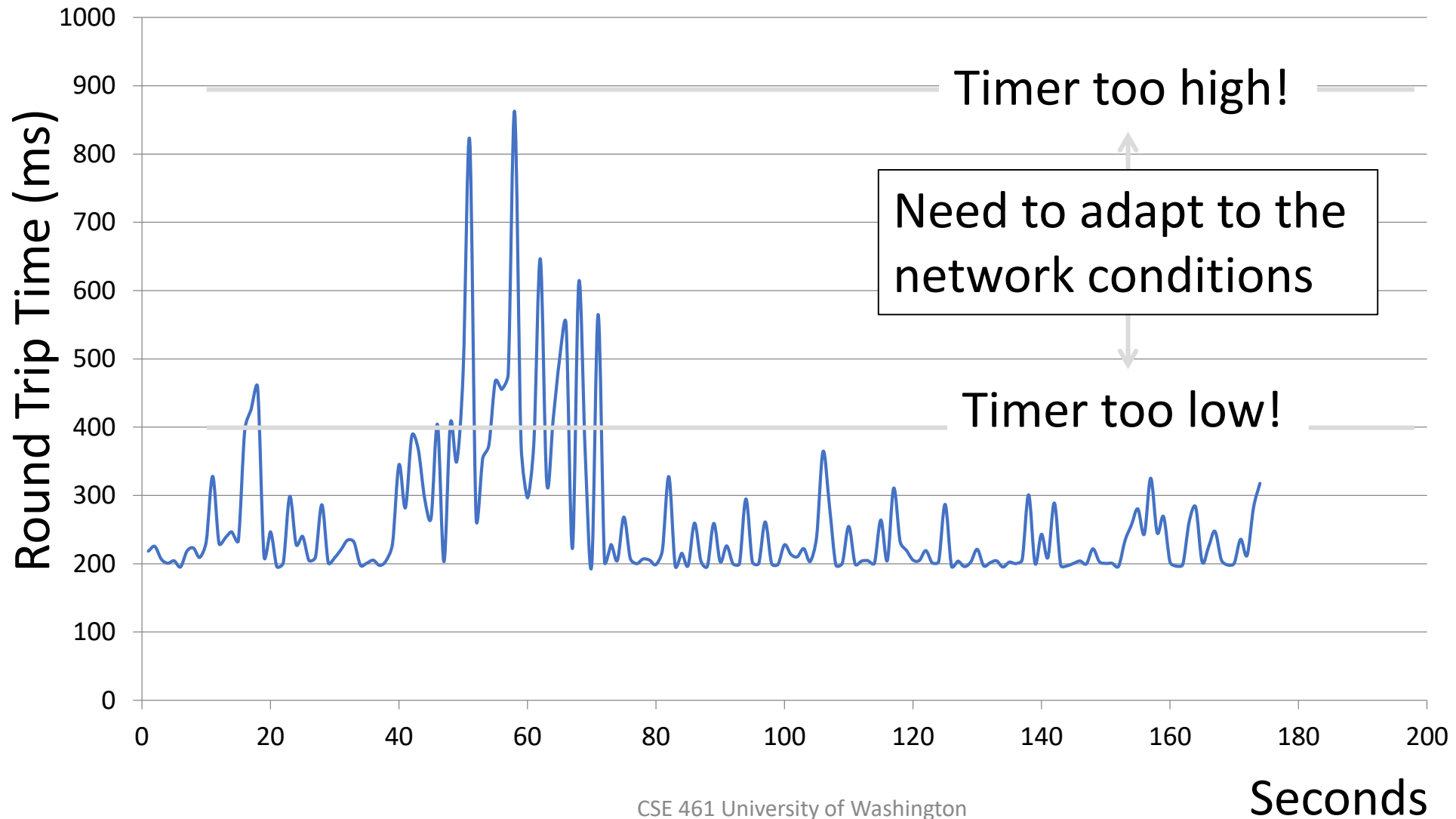
Example of RTTs



Example of RTTs (2)



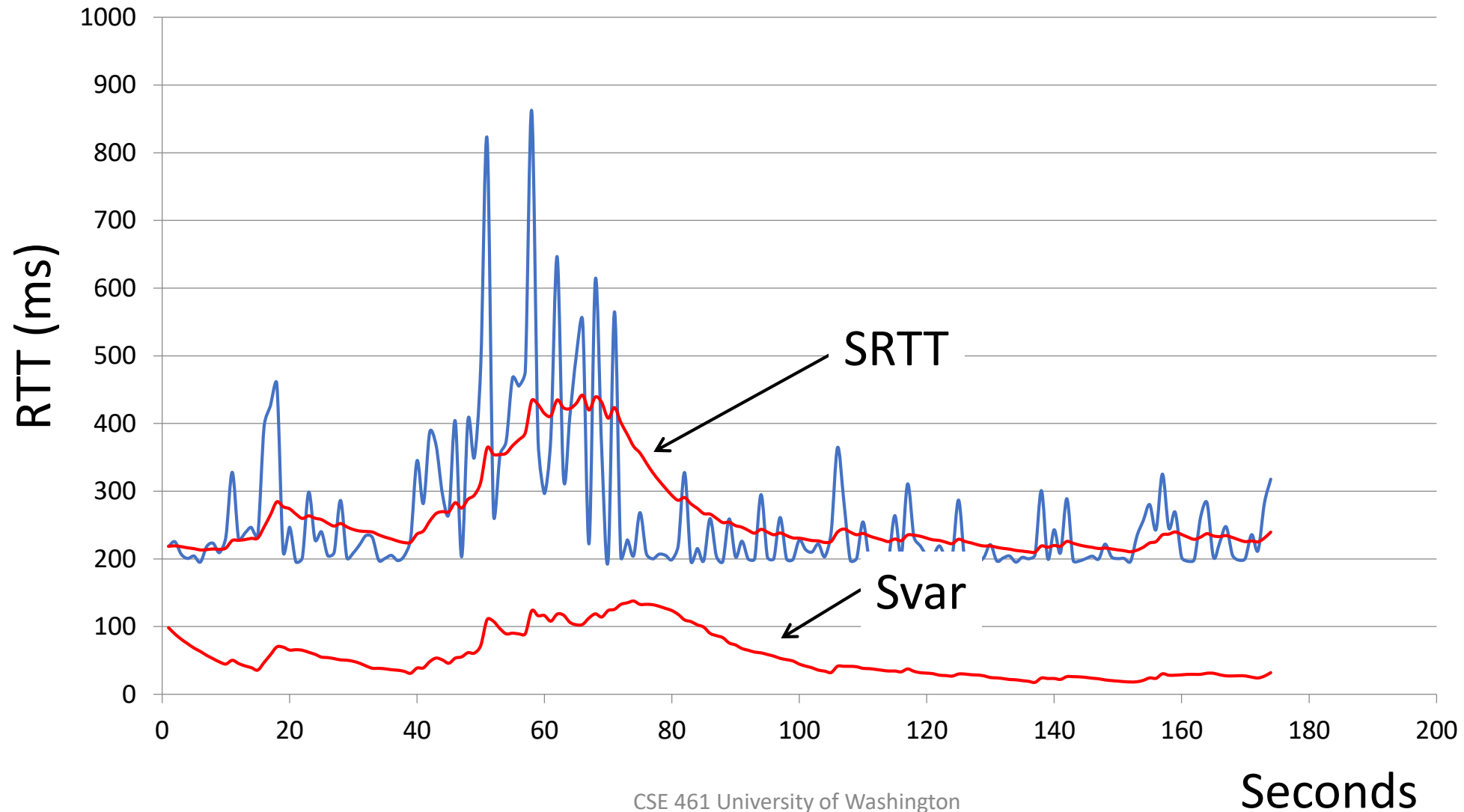
Example of RTTs (3)



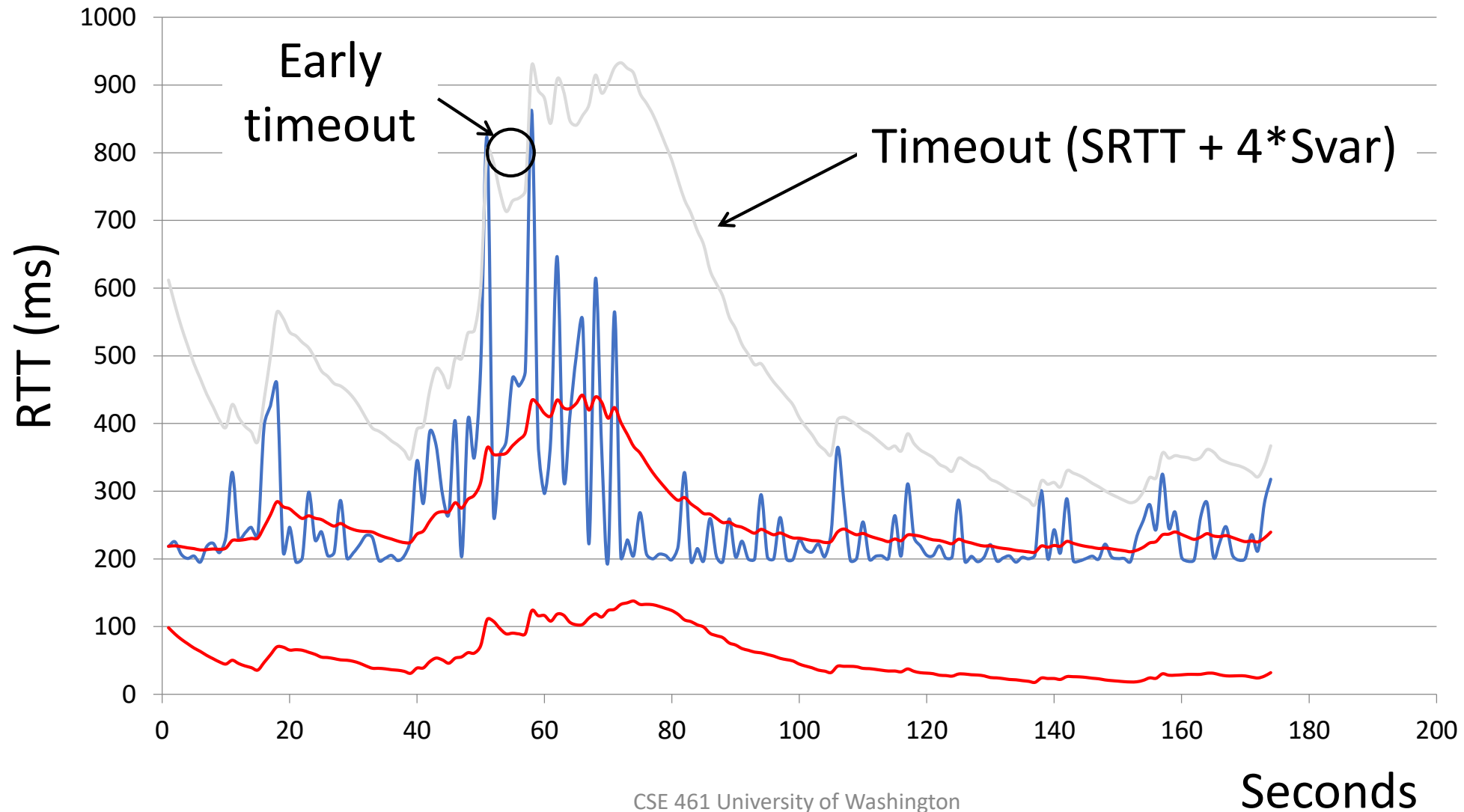
Adaptive Timeout

- Smoothed estimates of the RTT (1) and variance in RTT (2)
 - Update estimates with a moving average
 1. $SRTT_{N+1} = 0.9 * SRTT_N + 0.1 * RTT_{N+1}$
 2. $Svar_{N+1} = 0.9 * Svar_N + 0.1 * |RTT_{N+1} - SRTT_{N+1}|$
- Set timeout to a multiple of estimates
 - To estimate the upper RTT in practice
 - $TCP\ Timeout_N = SRTT_N + 4 * Svar_N$

Example of Adaptive Timeout



Example of Adaptive Timeout (2)



Adaptive Timeout (2)

- Simple to compute, does a good job of tracking actual RTT
 - Little “headroom” to lower
 - Yet very few early timeouts
- Turns out to be important for good performance and robustness