Flow Control

CSE 461

Ratul Mahajan
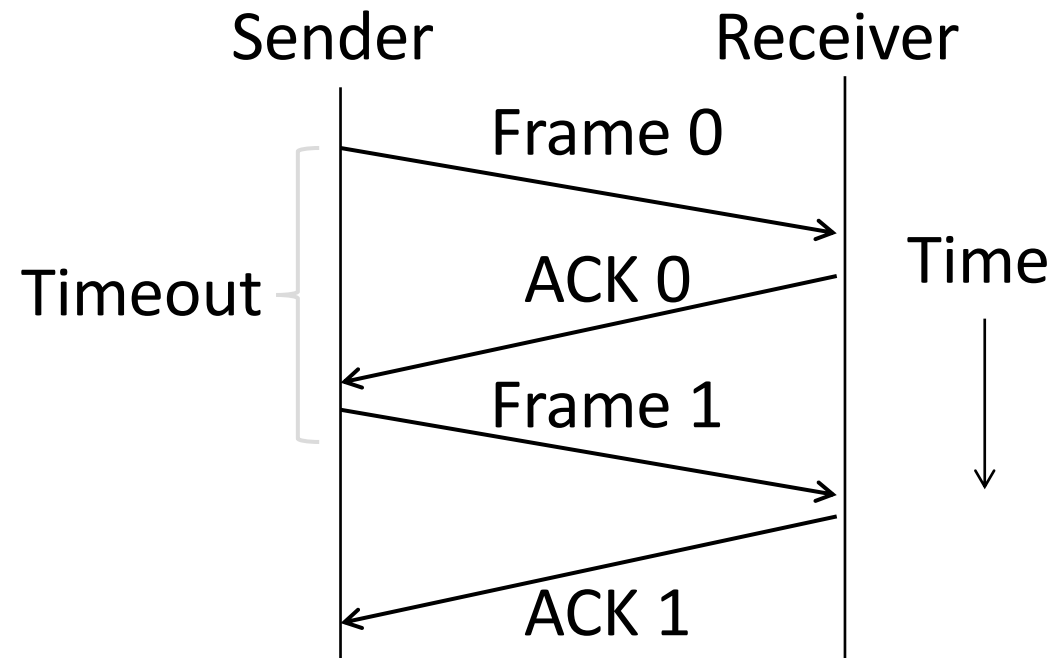
Flow control goal

Match transmission speed to reception capacity
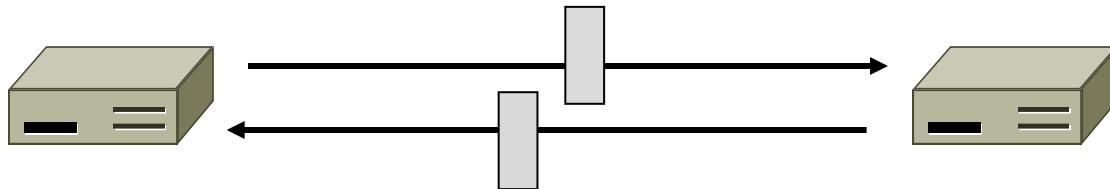
- Otherwise data will be lost

ARQ: Automatic repeat query

- ARQ with one message at a time is Stop-and-Wait

Limitation of Stop-and-Wait

- It allows only a single message to be outstanding from the sender:
  - Fine for LAN (only one frame fits in network anyhow)
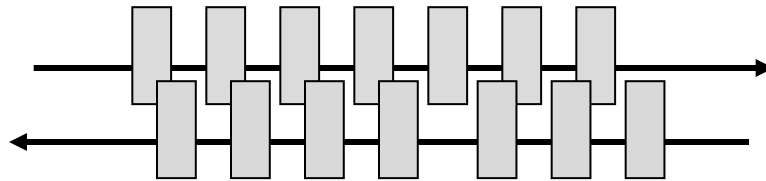 - Not efficient for network paths with longer delays

Limitation of Stop-and-Wait (2)

- Example: B=1 Mbps, D = 50 ms
  - RTT (Round Trip Time) = 2D = 100 ms
  - How many packets/sec?
    - 10
  - Usage efficiency if packets are 10kb?
    - $(10,000 \times 10) / (1 \times 10^6) = 10\%$

  - What is the efficiency if B=10 Mbps?
    - 1%

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 (2)

What W will use the network capacity with 10kb packets?

- Ex: B=1 Mbps, D = 50 ms
 - 2BD = 2 x 10^6 x 50/1000 = 100 Kb
  - W = 100 kb/10 = 10 packets

- Ex: What if B=10 Mbps?
  - W = 100 packets

Sliding Window Protocol

- Many variations, depending on how buffers, acknowledgements, and retransmissions are handled

- <u>Go-Back-N</u>
  - Simplest version, can be inefficient

- <u>Selective Repeat</u>
 - More complex, better performance

Sender Sliding Window

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

Sliding Window   W=5   Available

.. Acked   Unacked   Unavailable ..

LAR   LFS   seq. number
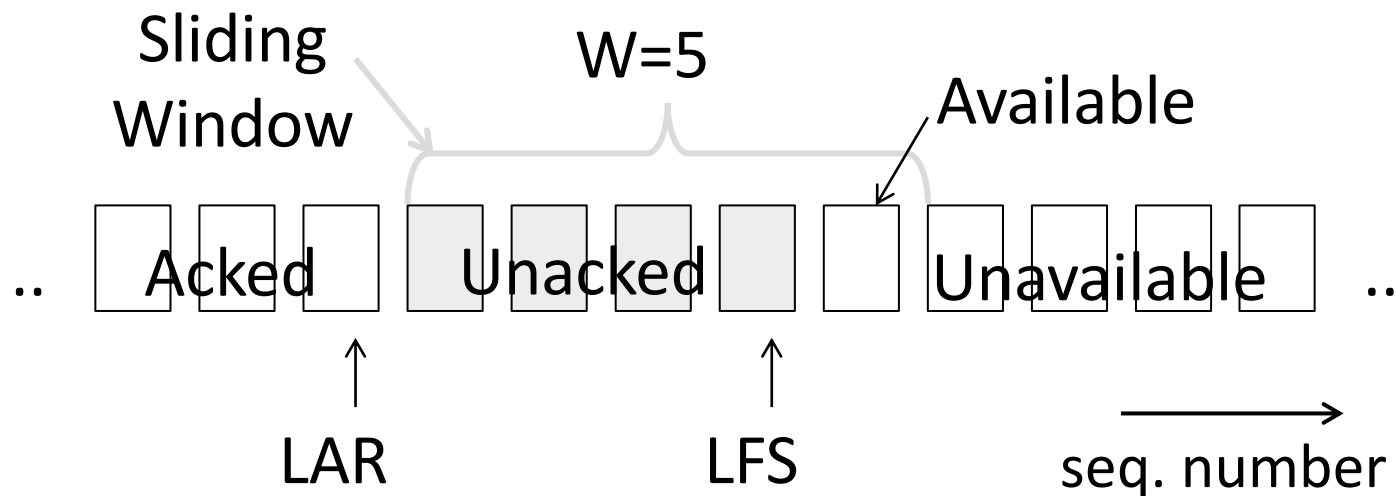
Sender Sliding Window (2)

- Transport accepts another segment of data from the Application ...
  - Transport sends it (LFS–LAR → 5)
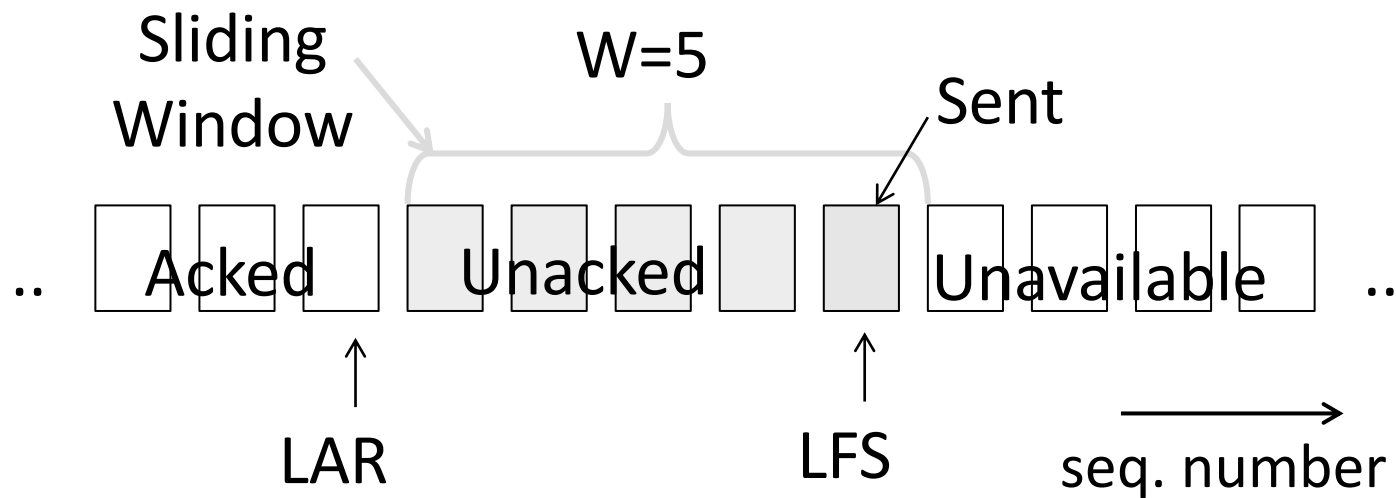
Sender Sliding Window (3)

- Next higher ACK arrives from peer…
  - Window advances, buffer is freed
  - LFS–LAR → 4 (can send one more)
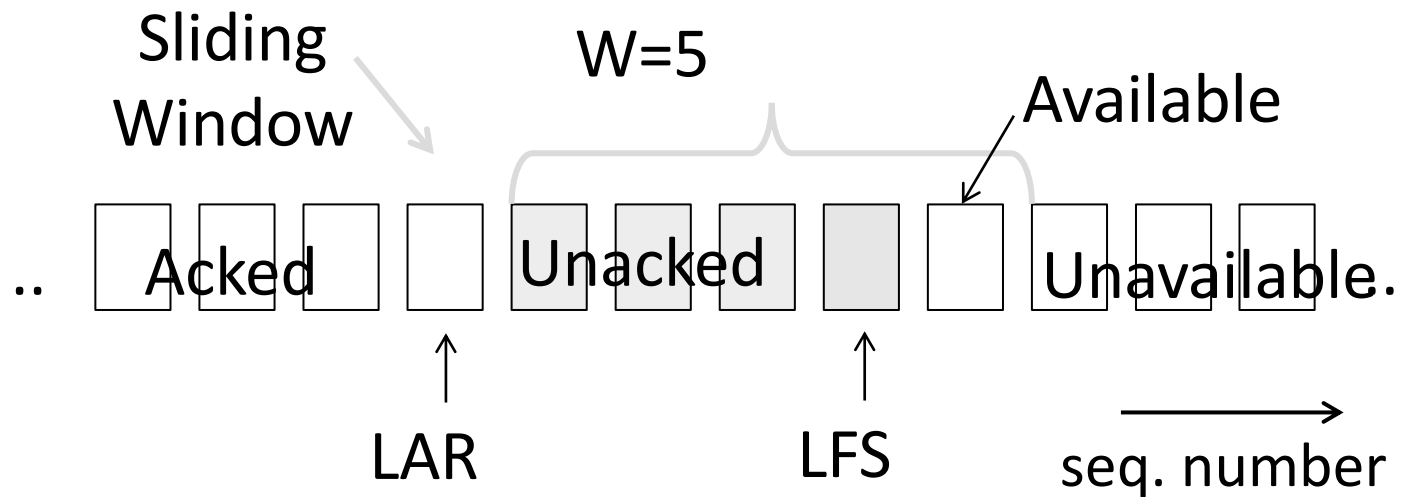
Receiver Sliding Window – Go-Back-N

- Receiver keeps only a single packet buffer for the next segment
  - State variable, LAS = 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)

Flow control recap

Goal: Match sending speed to receiver's capacity

3 increasingly complex and increasingly efficient solutions
- Stop and wait
- Sliding window: go back N
- Sliding window: selective repeat

Go back N

Sender sent packets 42, 43, 44, 45, …

If 43 is lost, all of 43, 44, 45 must be resent

Receiver does not buffer out of order packets (simple)

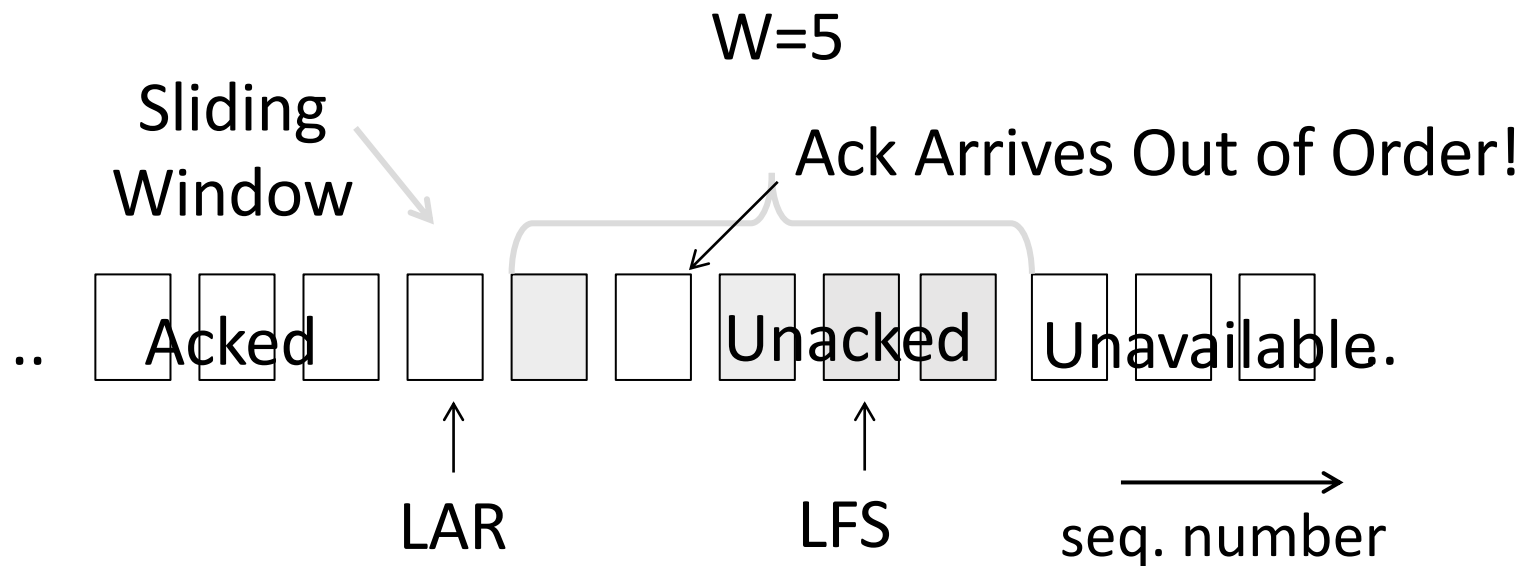# Receiver Sliding Window – Selective Repeat

- 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
  - Ex: I got everything up to 42 (LAS), and got 44, 45

- TCP uses a selective repeat design; we'll see the details later

Receiver Sliding Window – Selective Repeat (2)

- Buffers W segments, keeps state variable LAS = LAST ACK SENT

- On receive:
  - Buffer segments [LAS+1, LAS+W]
  - Send app in-order segments from LAS+1, and update LAS
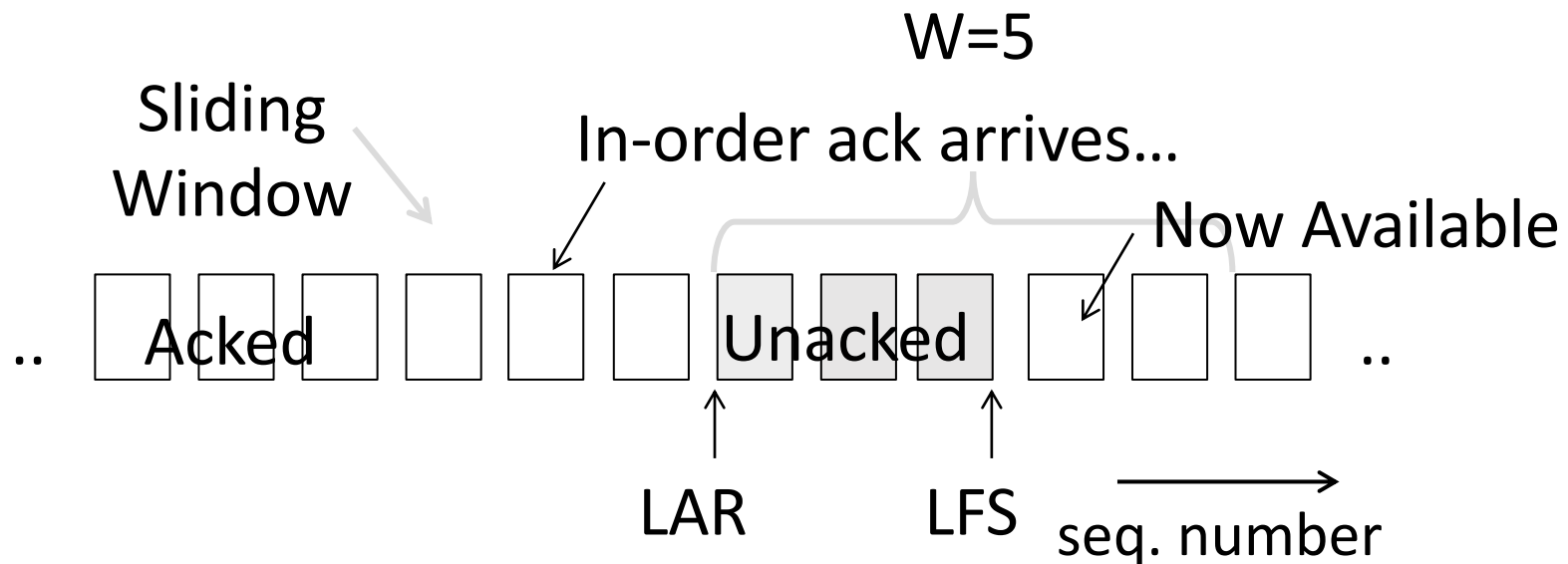 - Send ACK for LAS regardless

Sender Sliding Window – Selective Repeat

- Keep normal sliding window
- If out-of-order ACK arrives
  - Send LAR+1 again!

W=5

Sliding Window

Ack Arrives Out of Order!

.. | Acked | | | | Unacked | Unavailable.

↑ LAR

↑ LFS

→ seq. number

Sender Sliding Window – Selective Repeat (2)

- Keep normal sliding window
- If in-order ACK arrives
  - Move window and LAR, send more messages

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
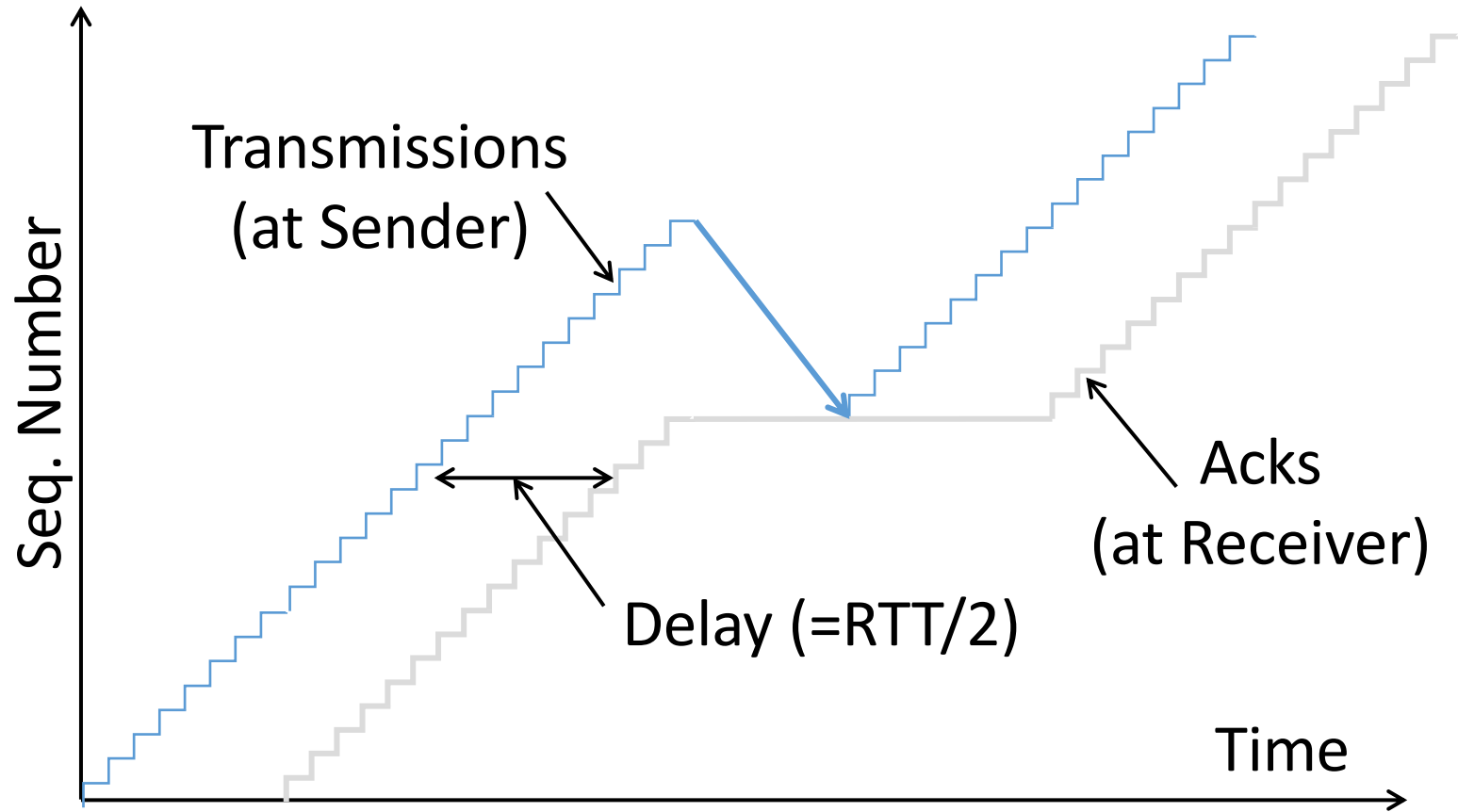 - Hope to resend fewer segments

Sequence Numbers

Need more than 0/1 for Stop-and-Wait … but how many?
- For Selective Repeat: 2W seq numbers
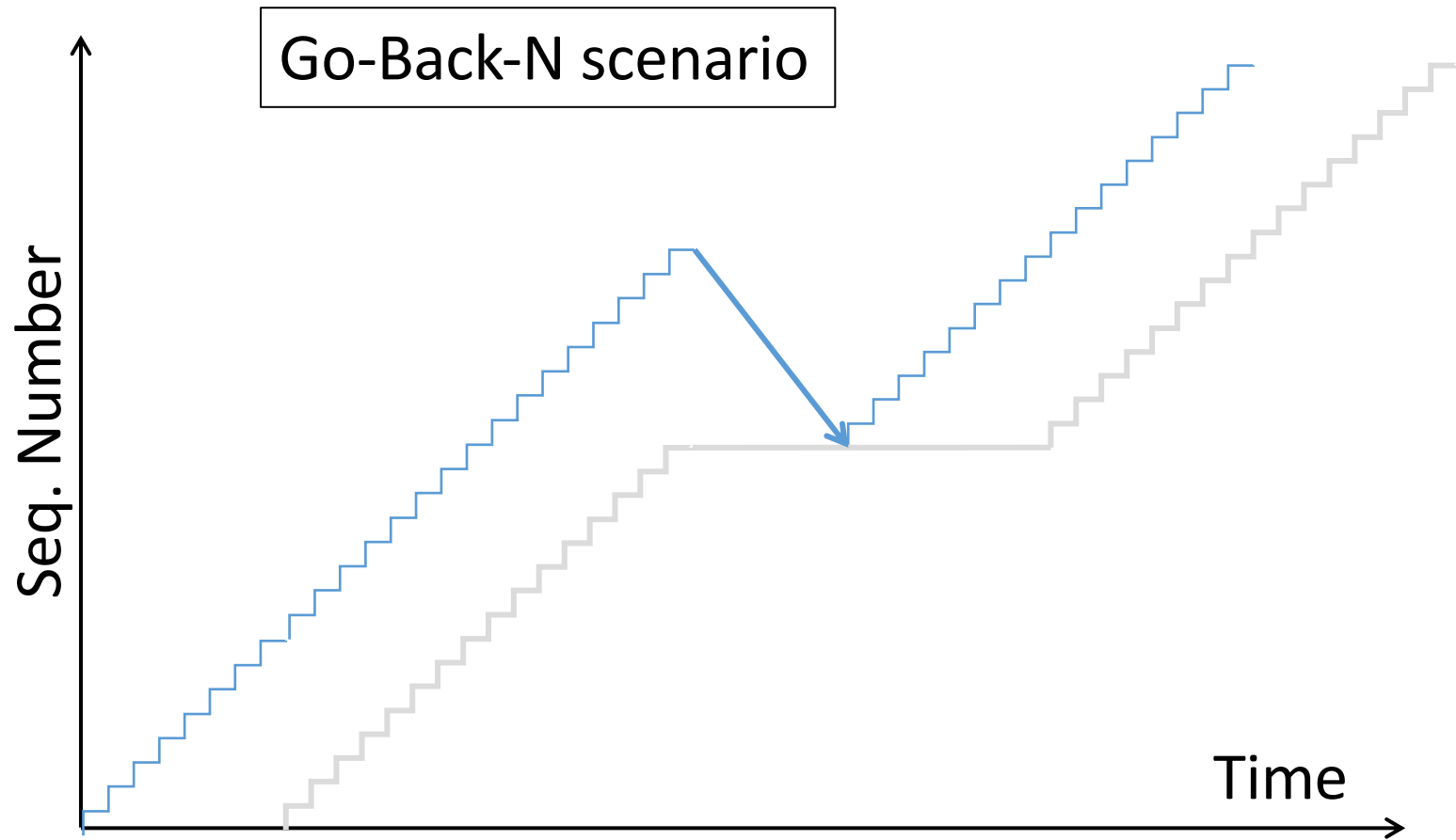 - W for packets, plus W for earlier acks
- For Go-Back-N: W+1 sequence numbers

Typically implement seq. number with an N-bit counter that wraps around at 2^N—1
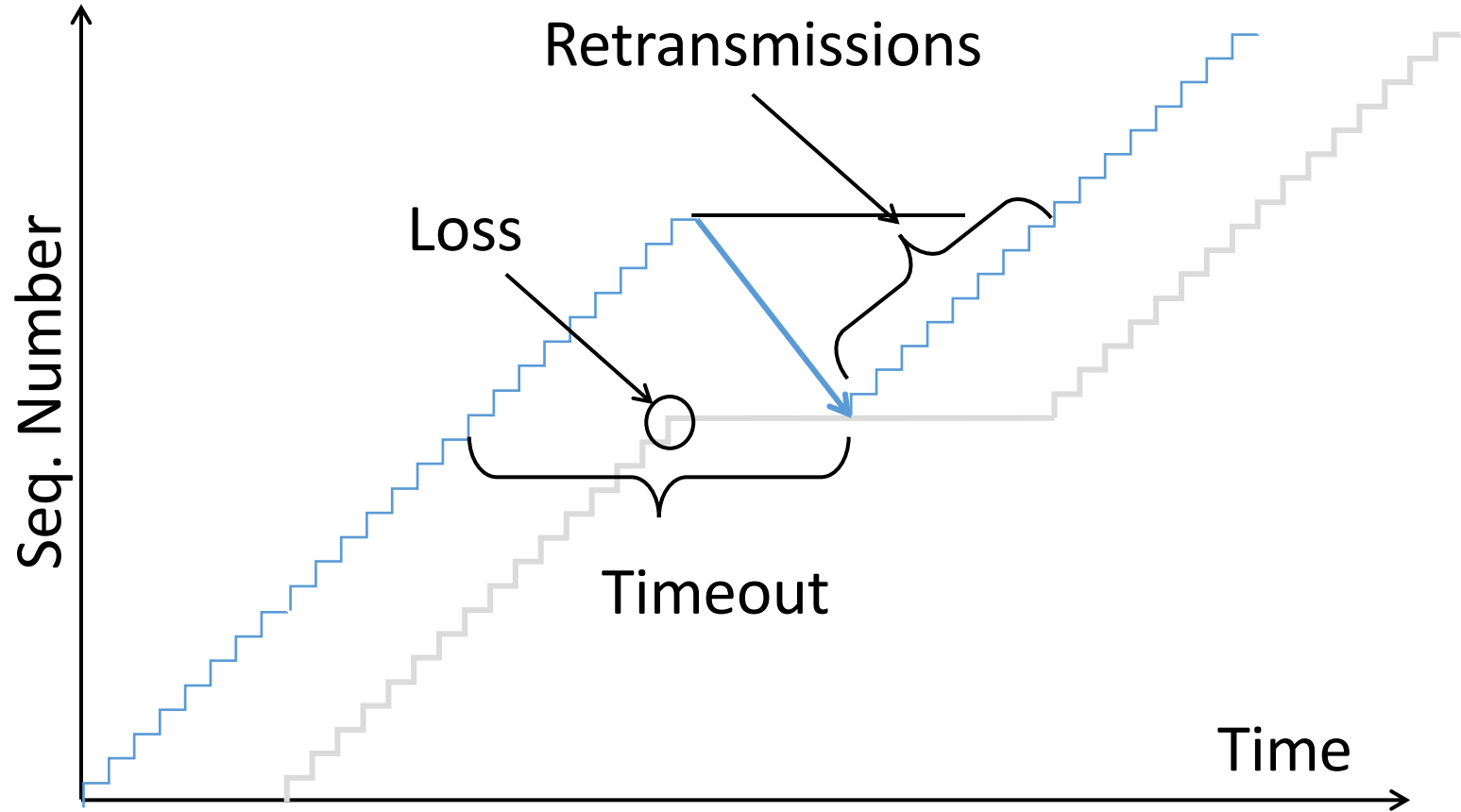- E.g., N=8:   …, 253, 254, 255, 0, 1, 2, 3, …

Sequence Time Plot

Transmissions (at Sender)

Acks (at Receiver)

Delay (=RTT/2)

Seq. Number

Time

Sequence Time Plot (2)

Go-Back-N scenario

Seq. Number

Time

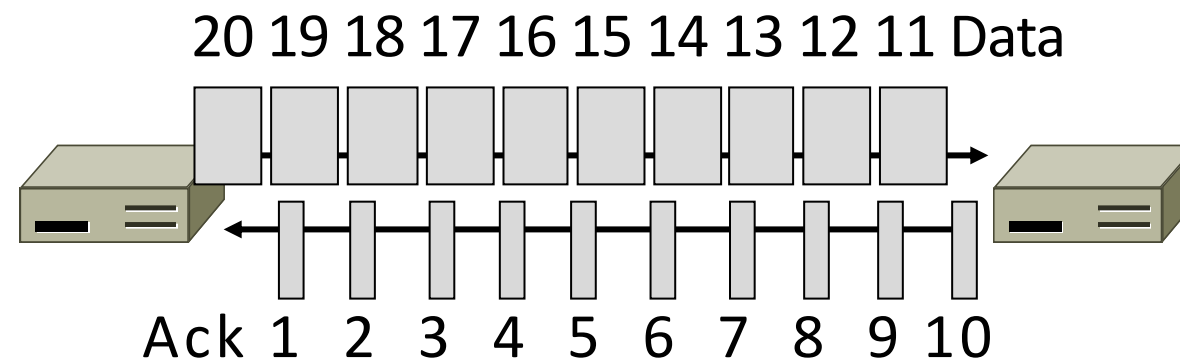# Sequence Time Plot (3)

Retransmissions

Loss

Timeout

Time

Seq. Number

ACK Clocking

Sliding Window ACK Clock

- Typically, the sender does not know B or D
- Each new ACK advances the sliding window and lets a new segment enter the network
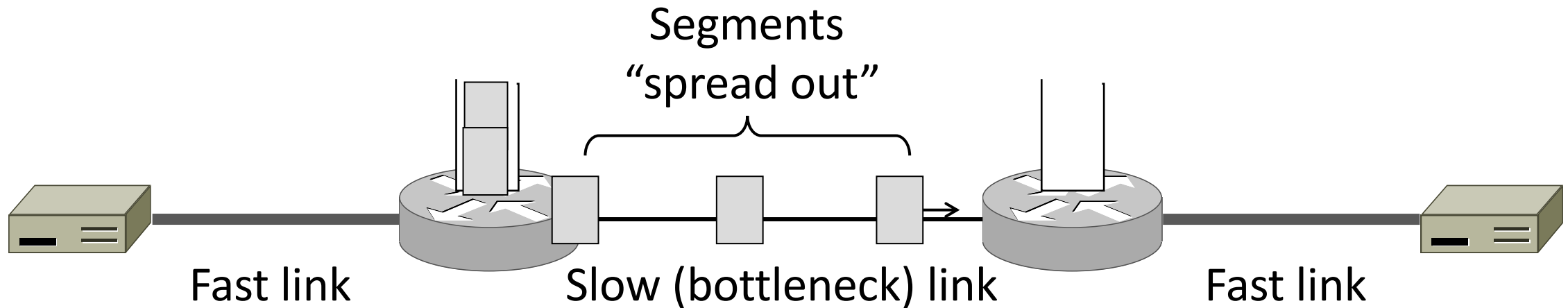 - ACKs "clock" data segments

20 19 18 17 16 15 14 13 12 11 Data

Ack 1 2 3 4 5 6 7 8 9 10

Benefit of ACK Clocking

- Consider what happens when sender injects a burst of segments into the network

Queue

Fast link Slow (bottleneck) link Fast link
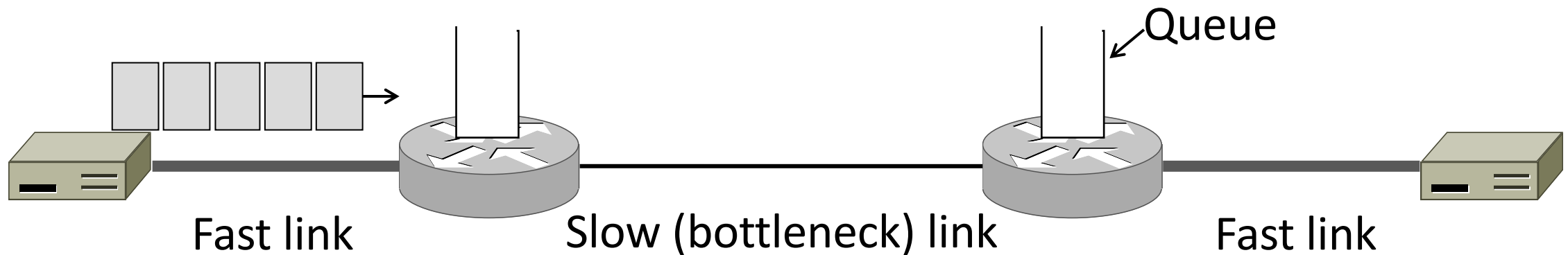
Benefit of ACK Clocking (2)

• Segments are buffered and spread out on slow link

Segments "spread out"

Fast link   Slow (bottleneck) link   Fast link

Benefit of ACK Clocking (3)

- ACKs maintain the spread back to the original sender

Slow link

Acks maintain spread
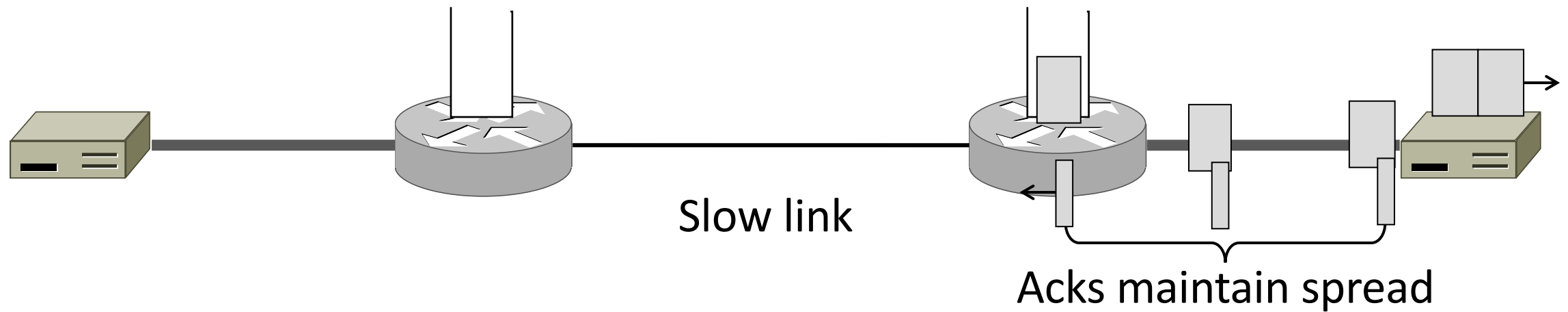
Benefit of ACK Clocking (4)

- Sender clocks new segments with the spread
  - Now sending at the bottleneck link without queuing!

Segments spread

Queue no longer builds

Slow link
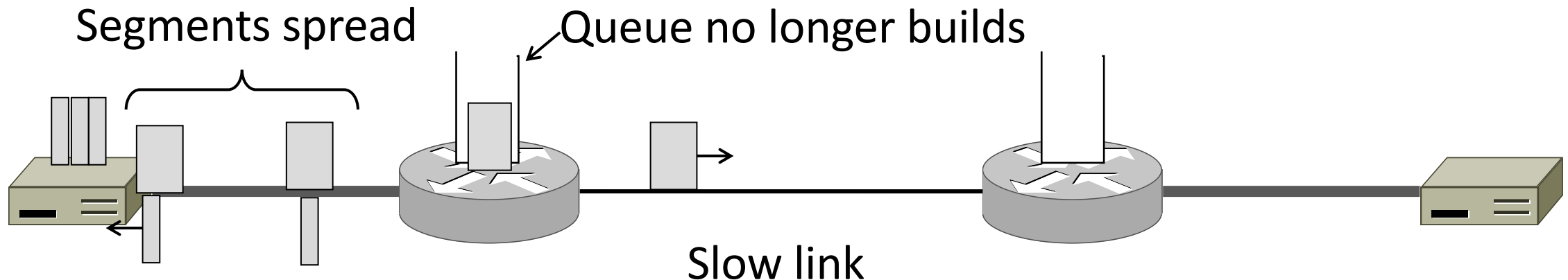
Benefit of ACK Clocking (4)

- Helps run with low levels of loss and delay!
- The network smooths out the burst of data segments
- ACK clock transfers this smooth timing back to sender
- Subsequent data segments are not sent in bursts so do not queue up in the network

TCP Uses ACK Clocking

- TCP uses a sliding window because of the value of ACK clocking

- Sliding window controls how many segments are inside the network

- TCP only sends small bursts of segments to let the network keep the traffic smooth

Problem

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

Streaming video

Arg ...

Big Iron

Wee Mobile
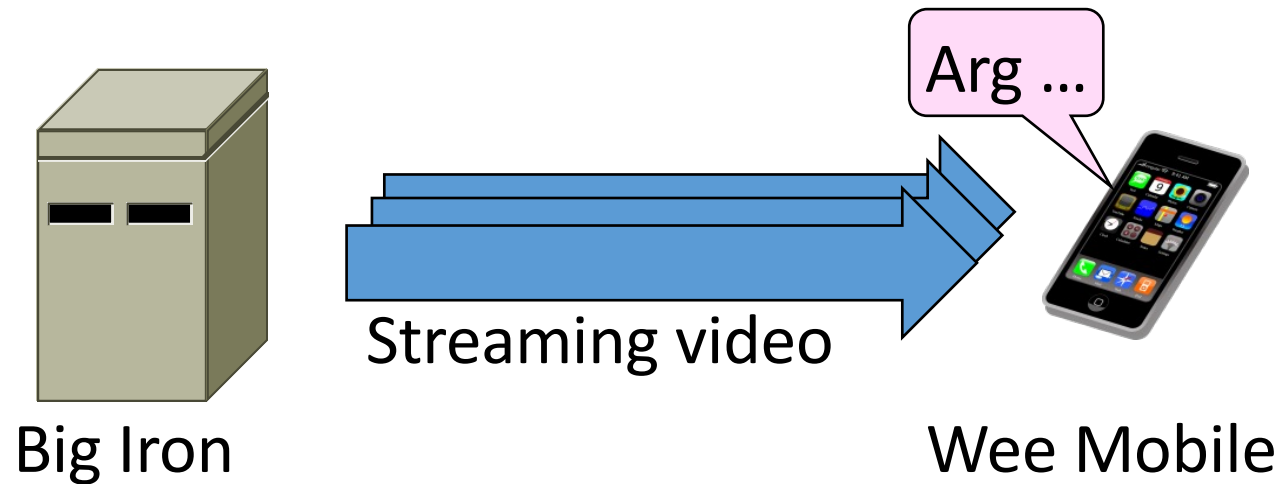
Receiver Sliding Window

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

Sliding Window

W=5

.. Finished Acceptable Too high ..

↑
LAS

→
seq. number
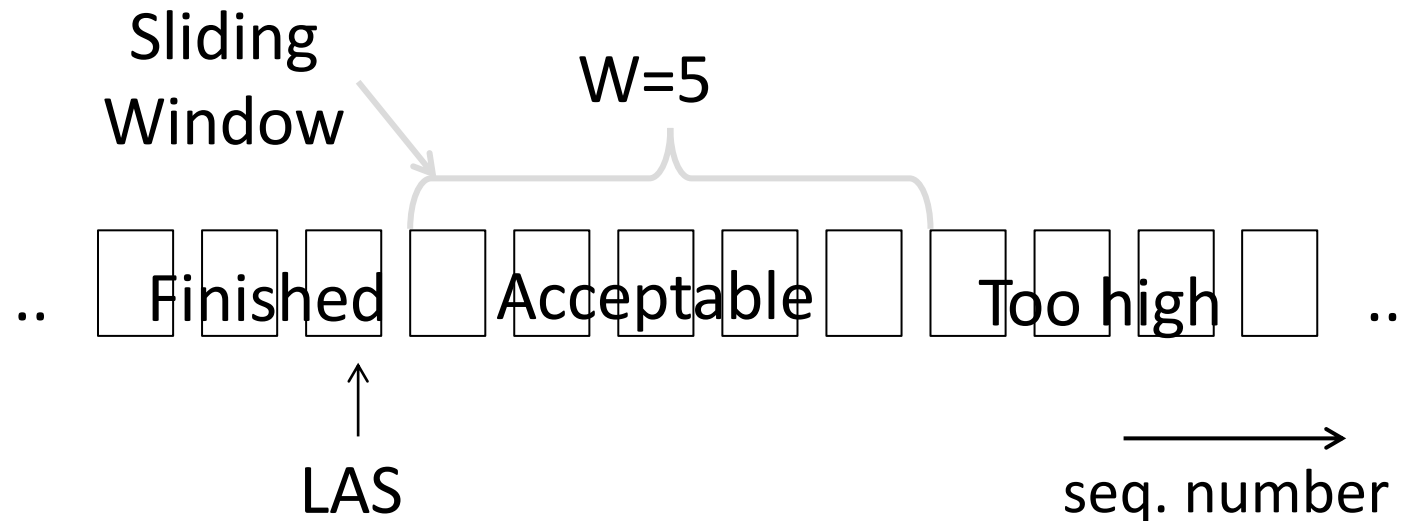
Receiver Sliding Window (2)

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

W=5

.. Finished Acceptable Too high ..

↑
LAS

→
seq. number
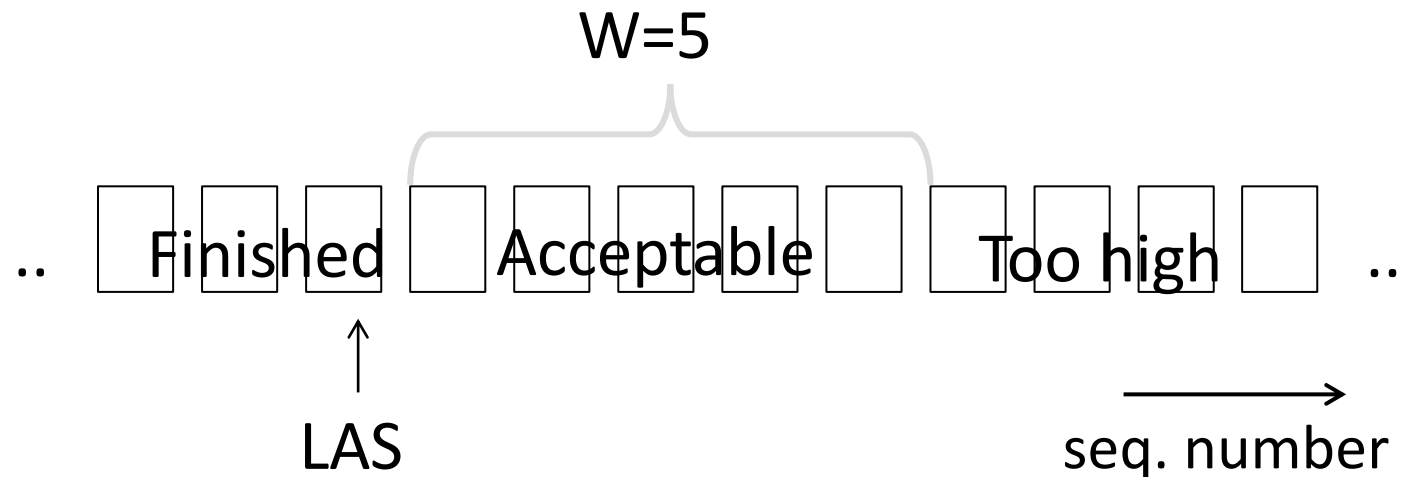
Receiver Sliding Window (3)

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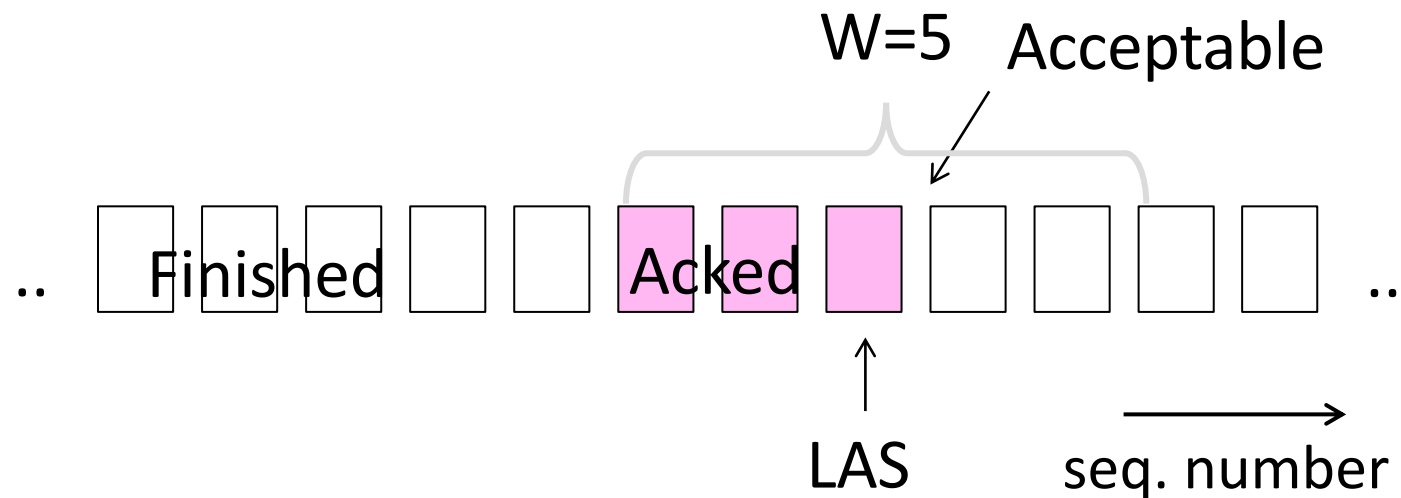 - LAS rises, but we can't slide window!

W=5

.. Finished Acked Too high ..

↑
LAS

→
seq. number
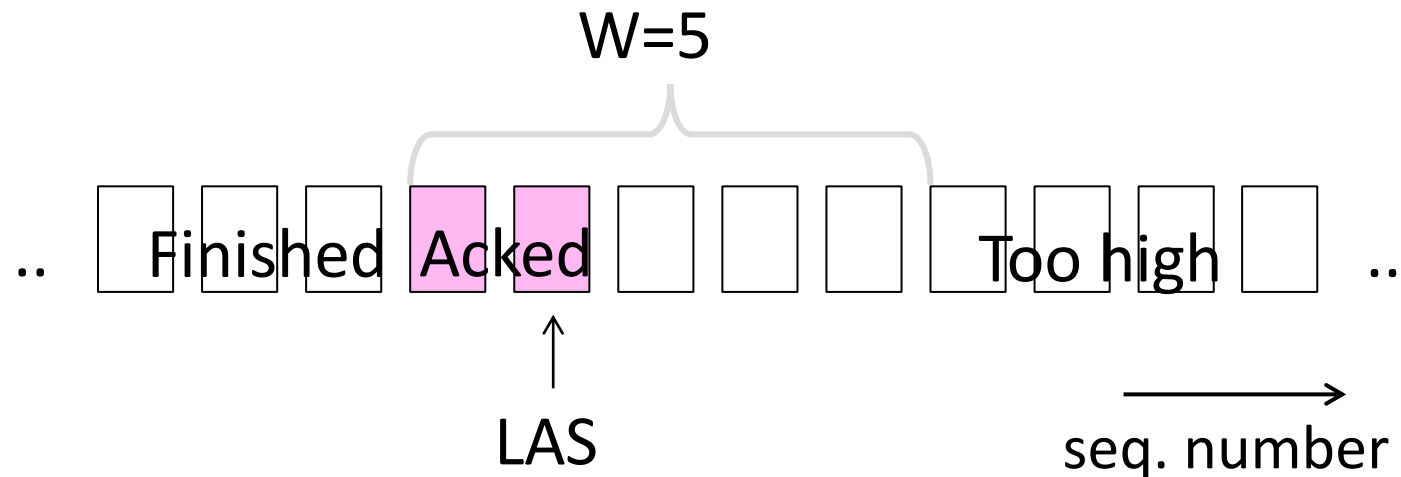
Receiver Sliding Window (4)

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

W=5

Nothing Acceptable!

.. Finished Acked Too high ..
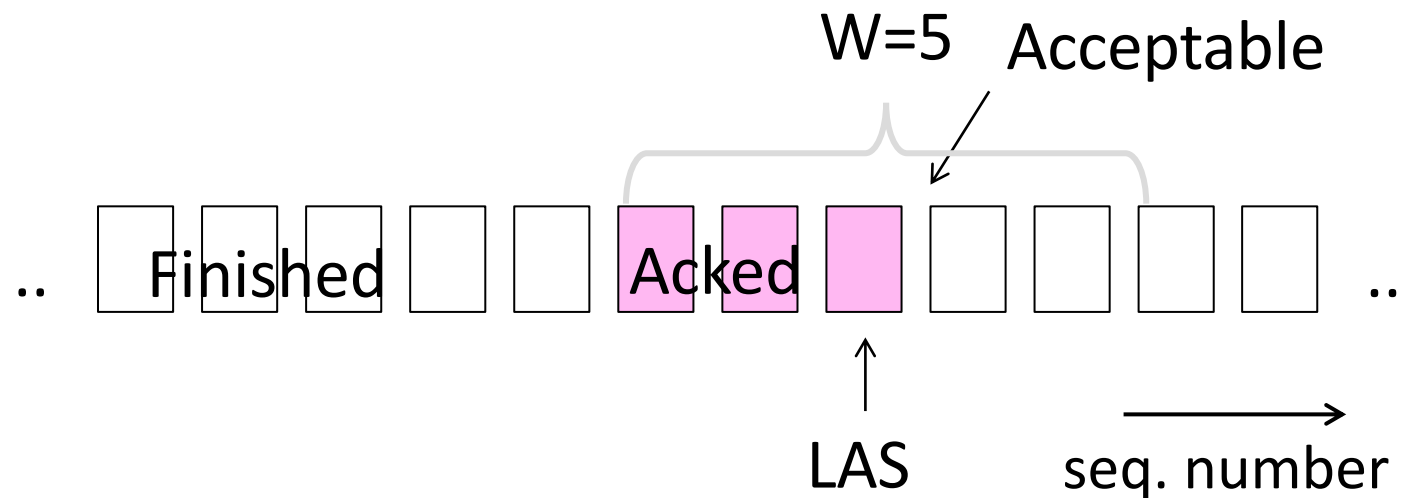
LAS

seq. number
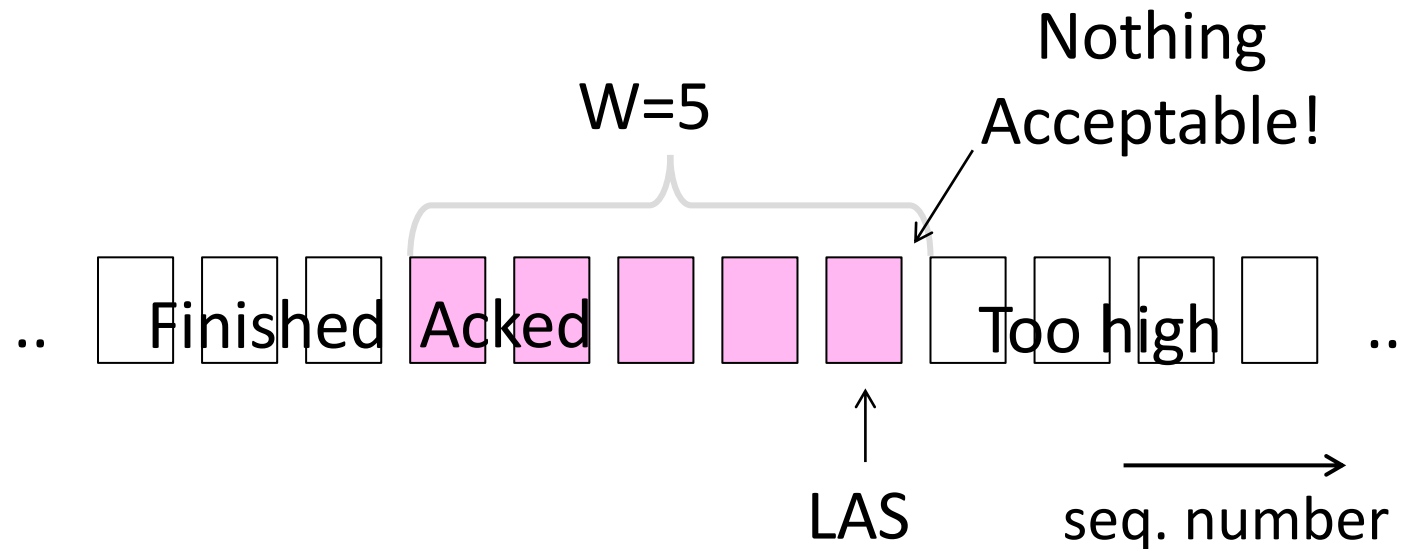
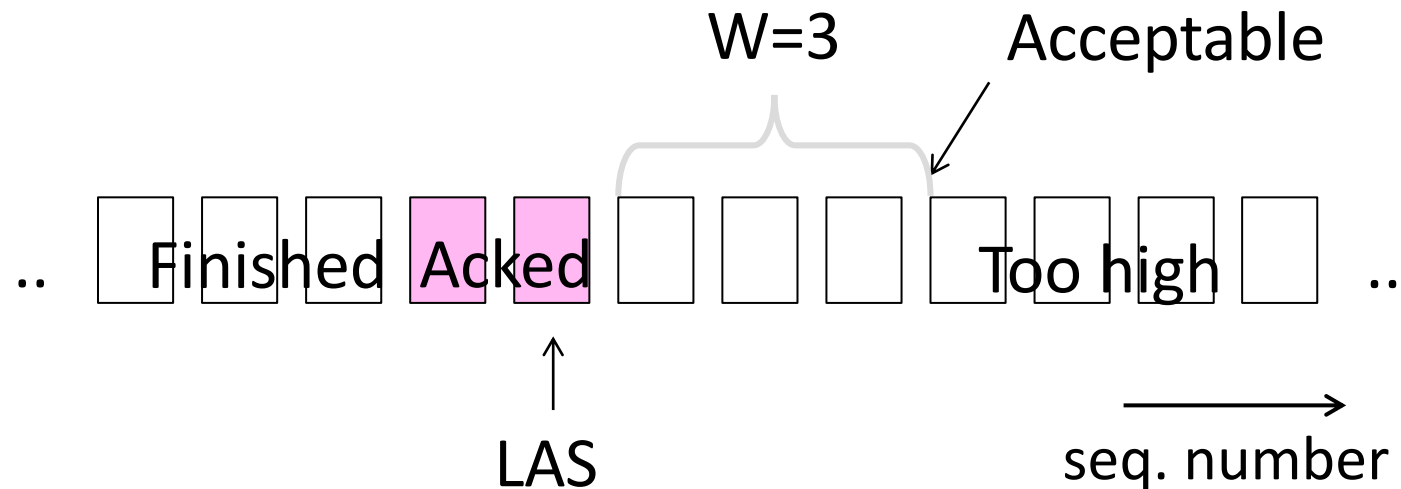# Receiver Sliding Window (5)

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

Flow Control

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

W=5   Acceptable

.. Finished   Acked   ..
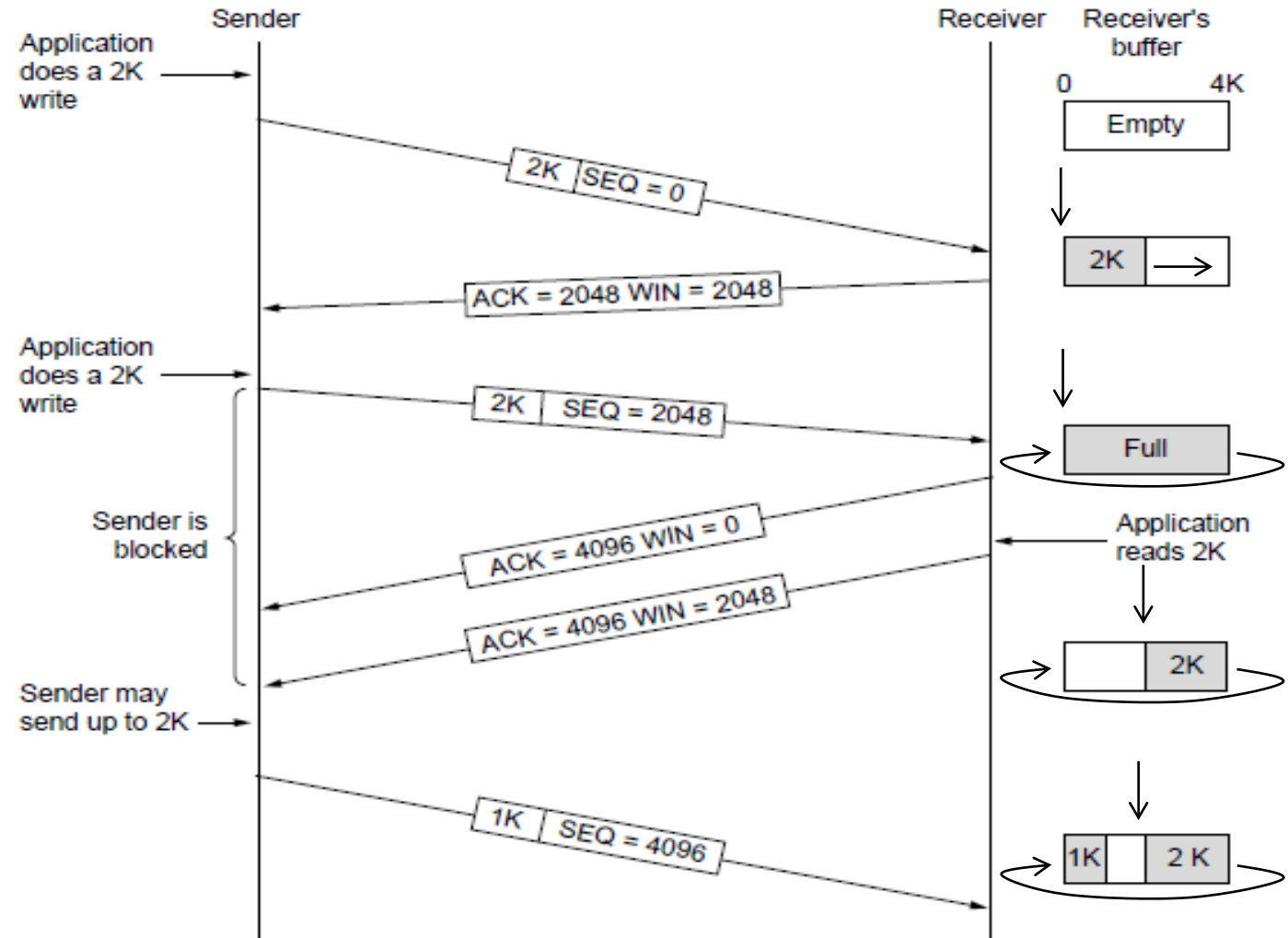
LAS   seq. number

Flow Control (2)

- Sender uses lower of the sliding window and <u>flow control window</u> (WIN) as the effective window size
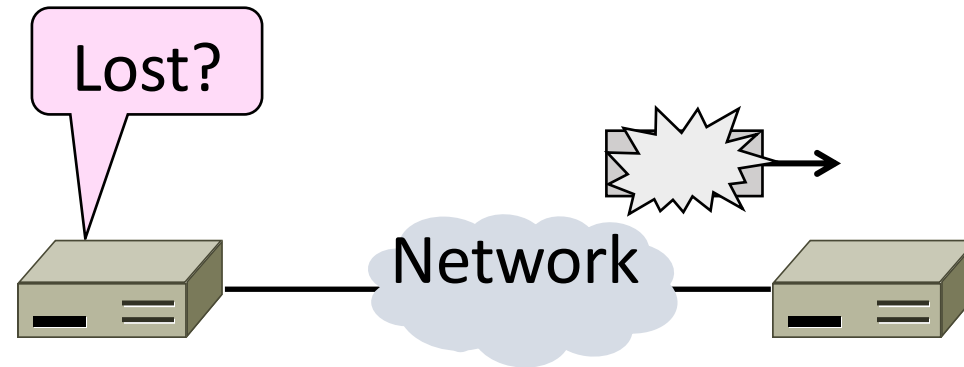
Flow Control (3)

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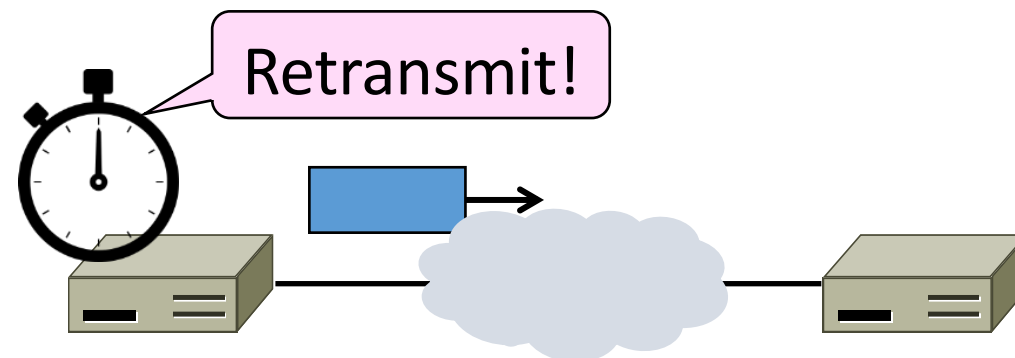 - Circular buffer of bytes

Topic

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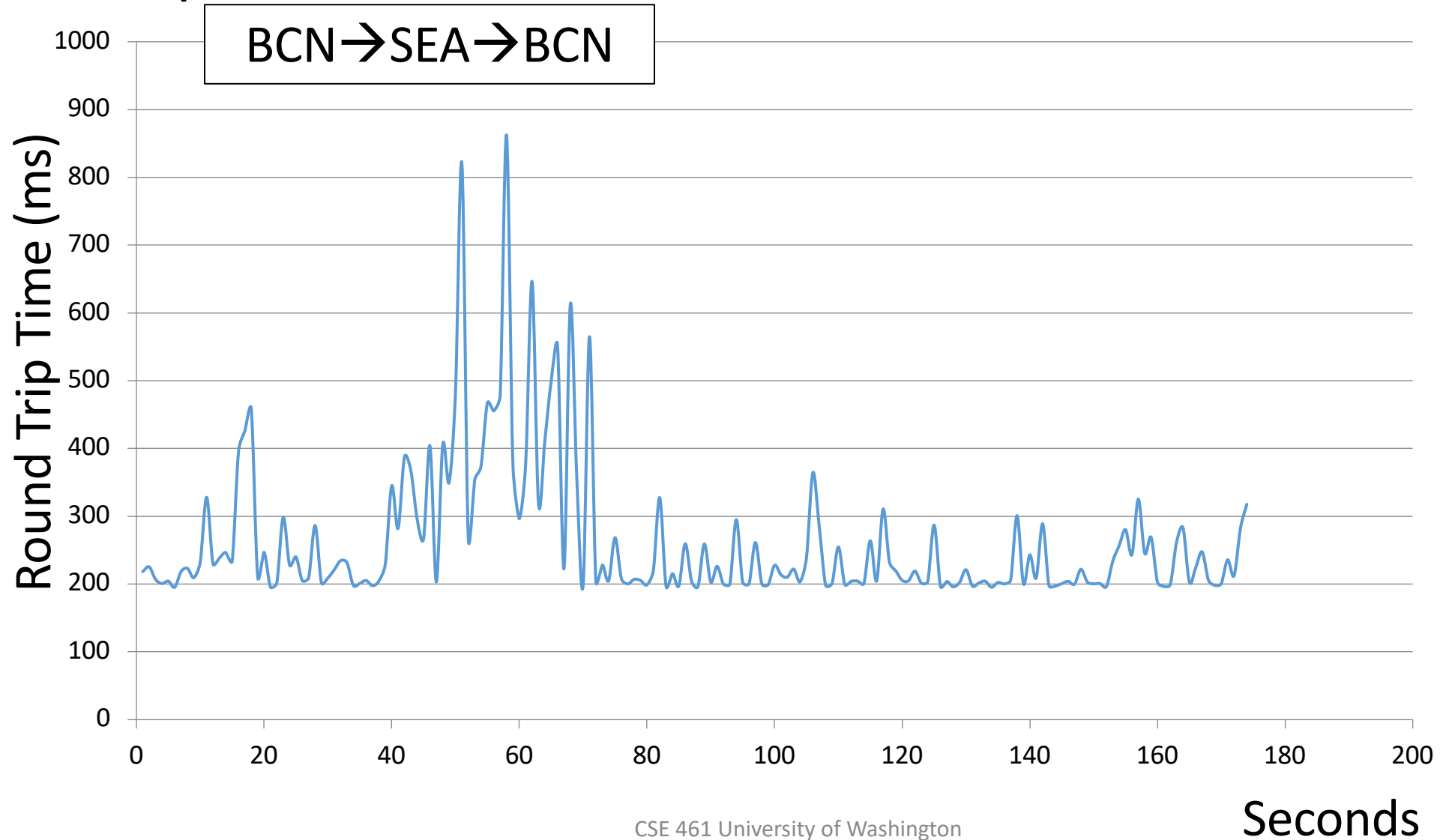
Retransmissions

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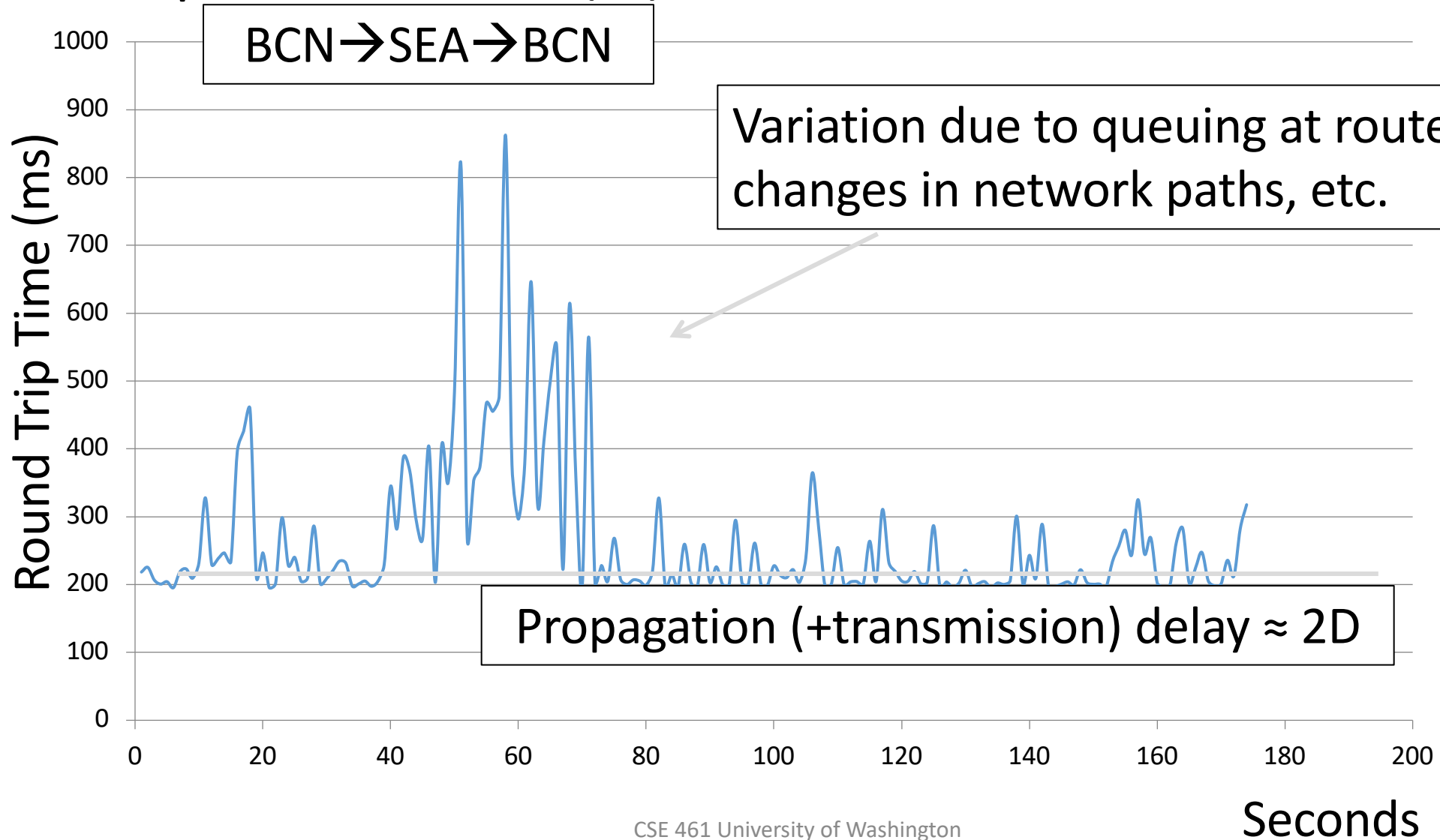 - If timer fires, <u>retransmit</u> data as lost

Retransmit!

Timeout Problem

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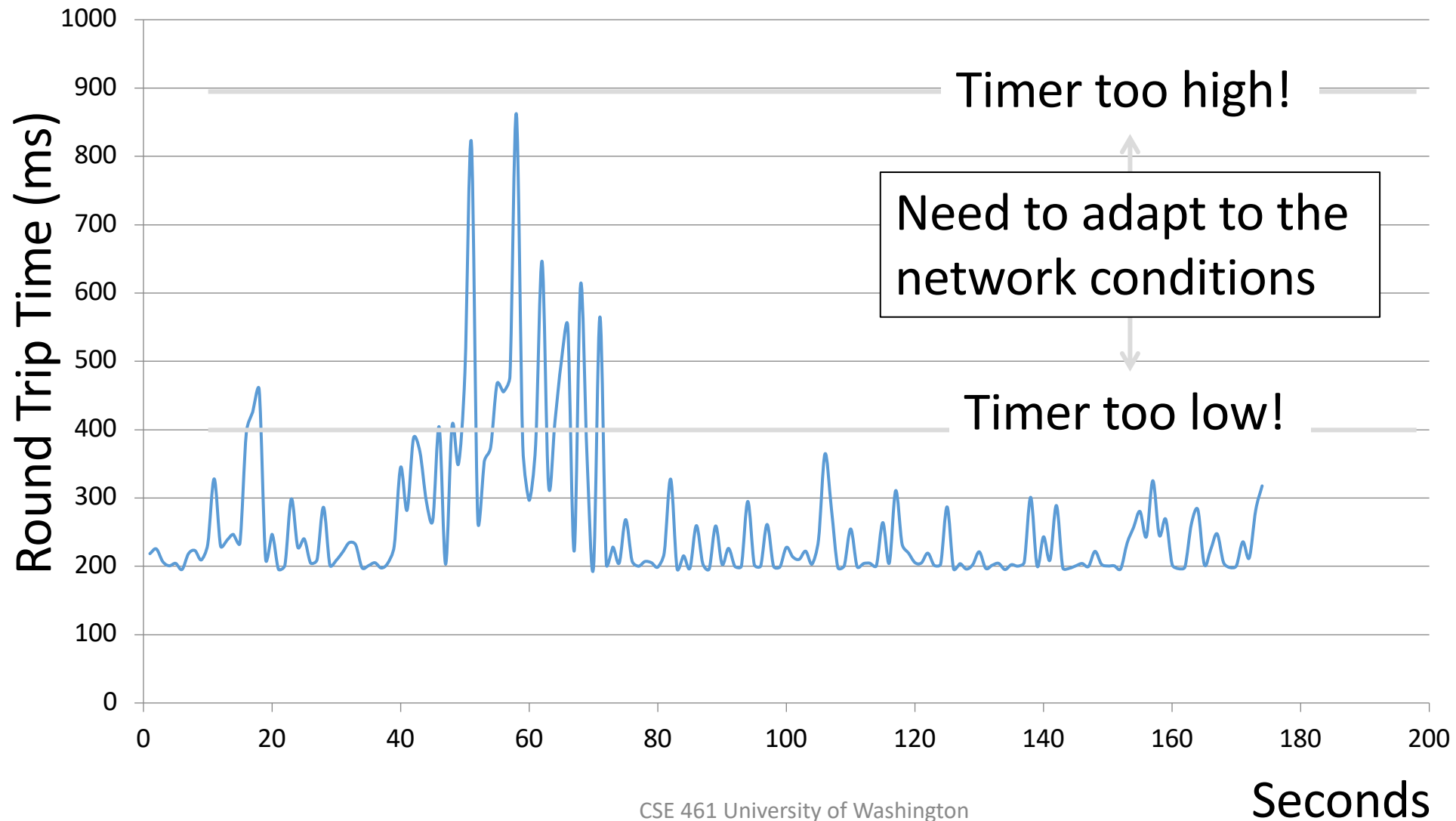   - Wide range, variable RTT

Example of RTTs

BCN→SEA→BCN

Round Trip Time (ms)

Seconds

Example of RTTs (2)

BCN→SEA→BCN

Variation due to queuing at routers, changes in network paths, etc.

Propagation (+transmission) delay ≈ 2D

Round Trip Time (ms)

Seconds

Example of RTTs (3)

Round Trip Time (ms) vs Seconds

- Timer too high!
- Need to adapt to the network conditions
- Timer too low!

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)

Early timeout

Timeout (SRTT + 4*Svar)

RTT (ms)

Seconds
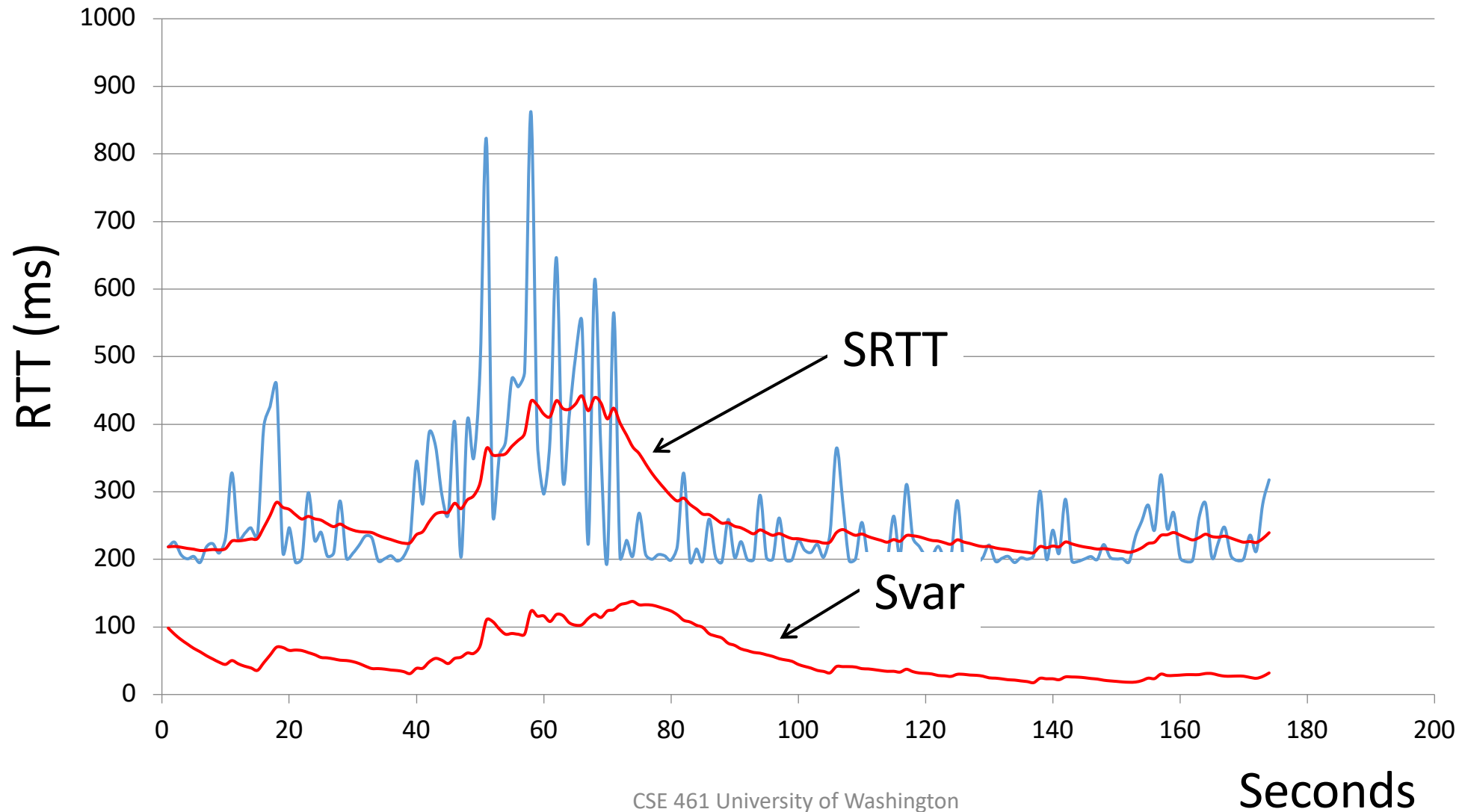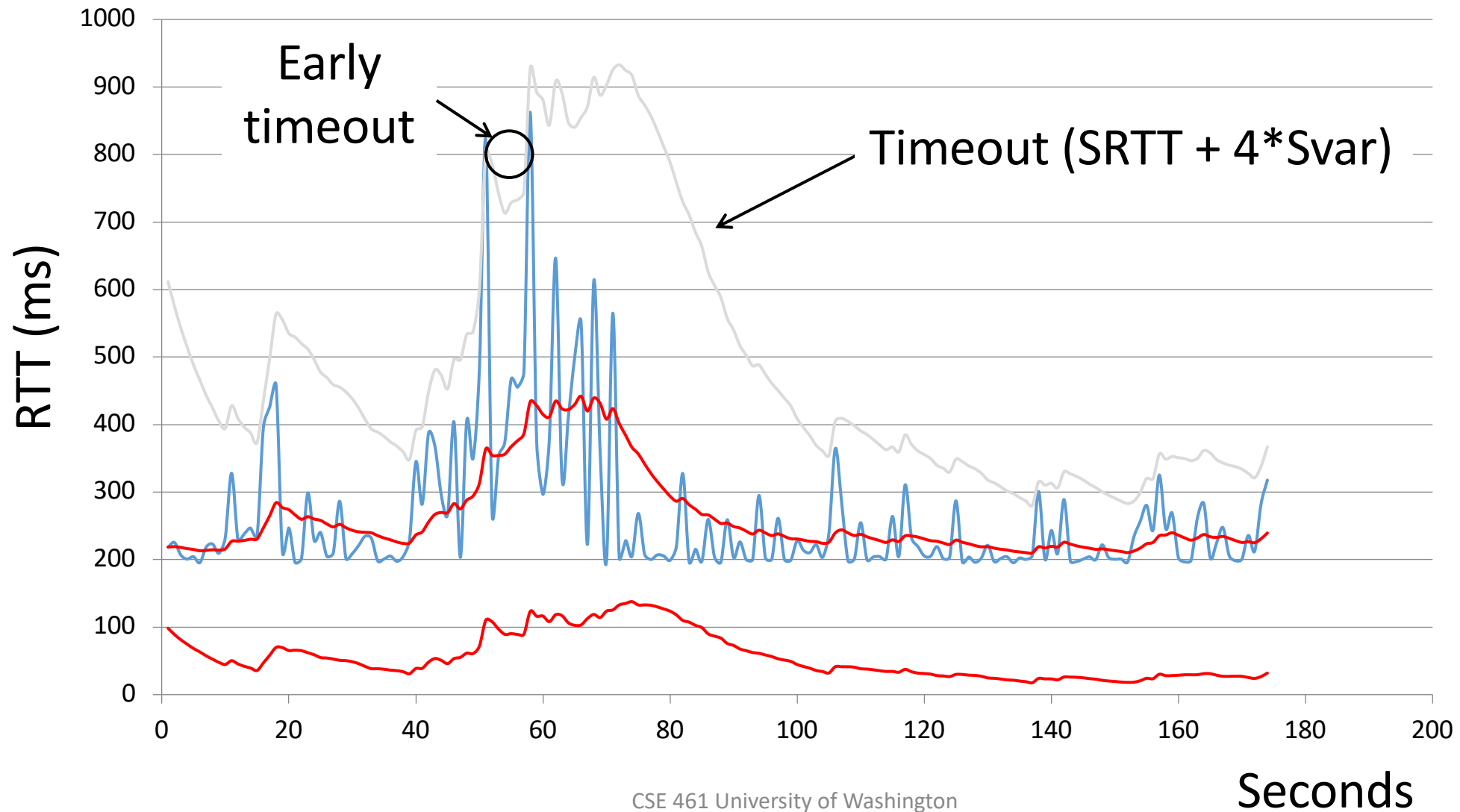
Adaptive Timeout (2)

- 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