Network Programming
TCP Flow Control

Minchen Yu
SDS@CUHK-SZ
Spring 2026

香港中文大學(深圳)
The Chinese University of Hong Kong, Shenzhen
SCHOOL OF DATA SCIENCE
數據科學學院

TCP

Consists of 3 primary phases:

- Connection Establishment (Setup)
- Sliding Windows/Flow Control
- Connection Release (Teardown)

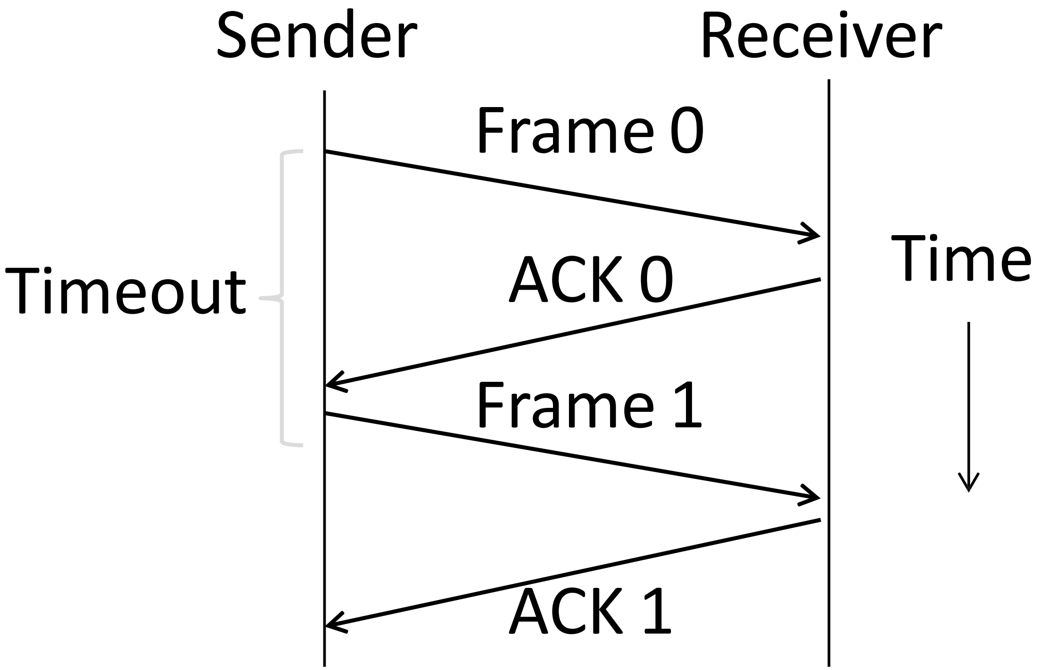
Flow control goal

Match transmission speed to reception capacity

- Otherwise data will be lost

ARQ: Automatic repeat query

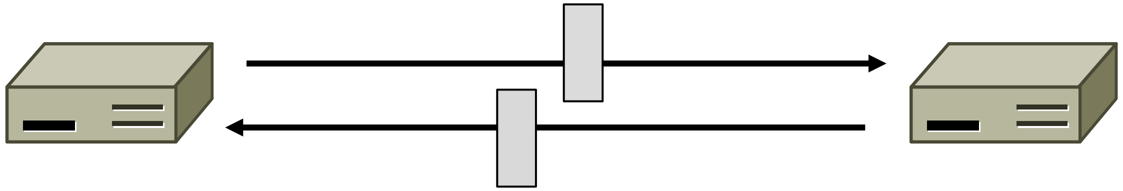
ARQ with one message at a time is Stop-and-Wait



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

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?

  $(10{,}000 \times 10) / (1 \times 10^6) = 10\%$

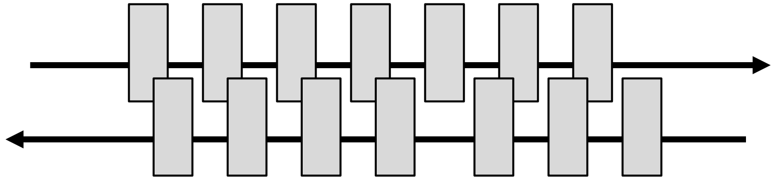
- What is the efficiency if B=10 Mbps?

  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 2BD to fill network path

Sliding window

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

- Ex: B=1 Mbps, D = 50 ms

  $2BD = 2 \times 10^6 \times 50/1000 = 100$ Kb

  W = 100 kb/10 = 10 packets

- Ex: What if B=10 Mbps?

  W = 100 packets

Sliding window protocol

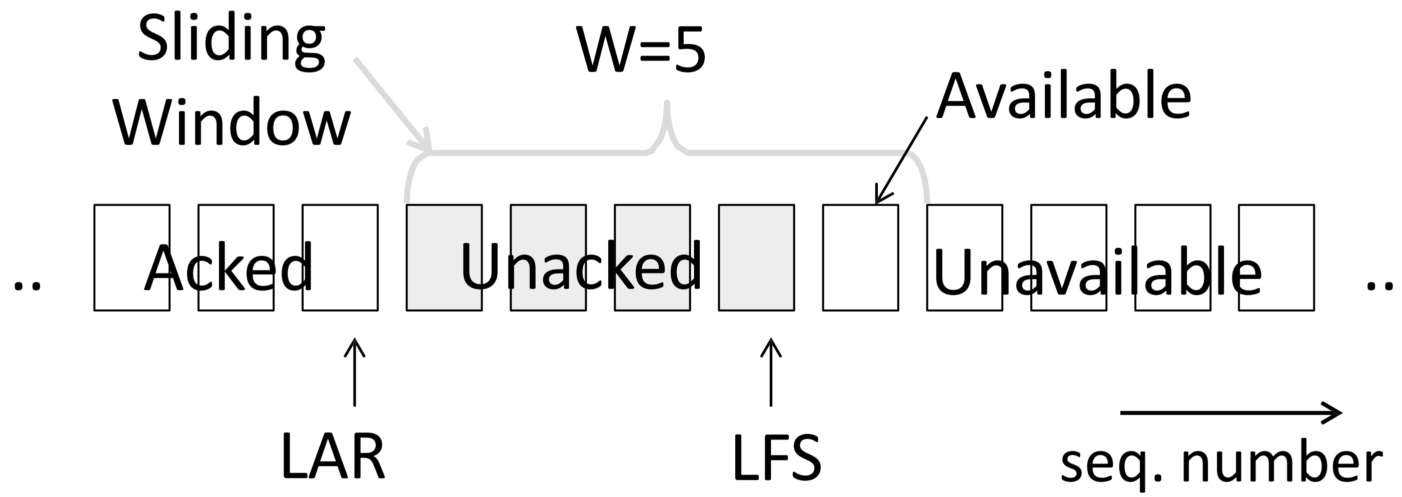
Many variations, depending on how buffers, acknowledgements, and retransmissions are handled

- Go-Back-N
  - Simplest version, can be inefficient
- Selective Repeat
  - More complex, better performance

Sender sliding window

Sender buffers up to W segments until they are acknowledged

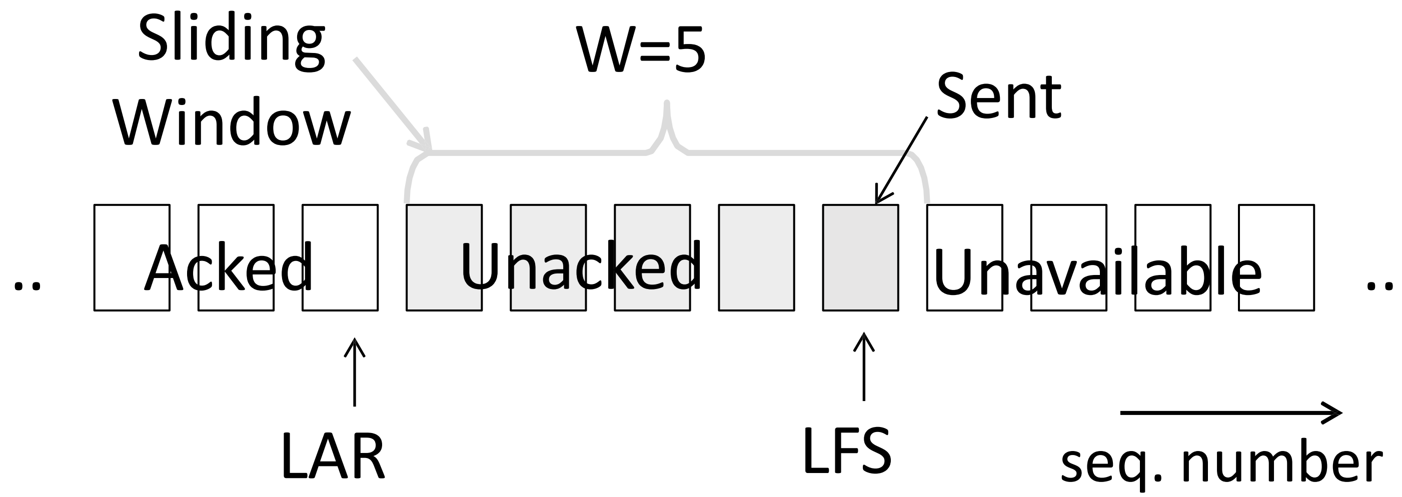
- LFS=last frame sent, LAR=last ack rec'd
- Sends while ensuring LFS – LAR ≤ W



Sender sliding window

Transport accepts another segment of data from the Application…

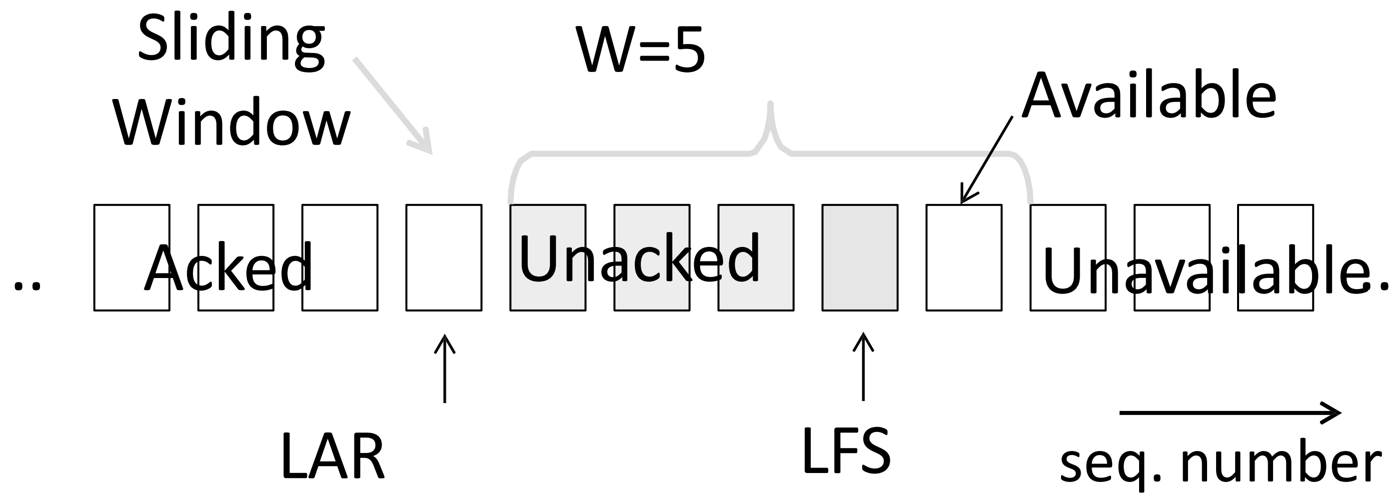
- Transport sends it (LFS–LAR -> 5)



Sender sliding window

Next higher ACK arrives from peer…

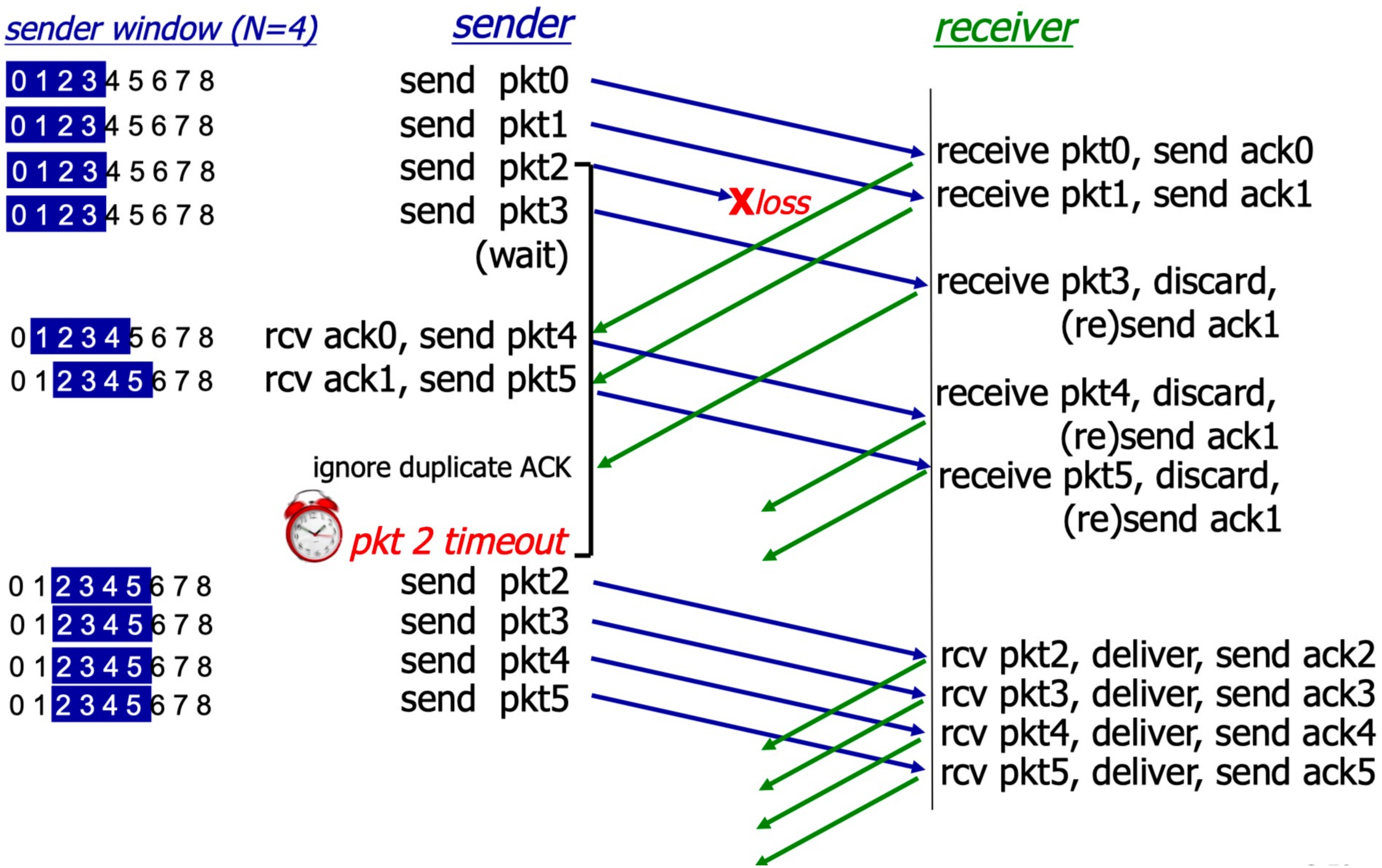
- Window advances, buffer is freed
- LFS–LAR -> 4 (can send one more)



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)

Go-Back-N in action



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

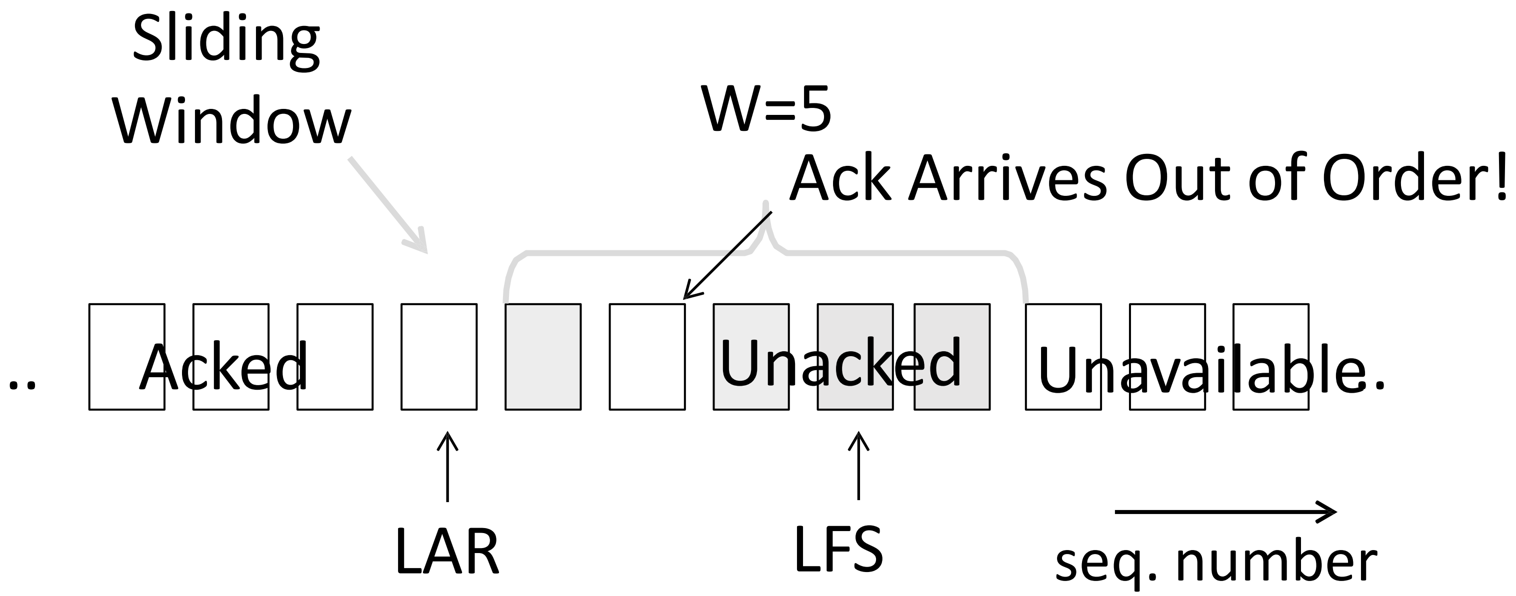
Receiver sliding window – Selective Repeat

- 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

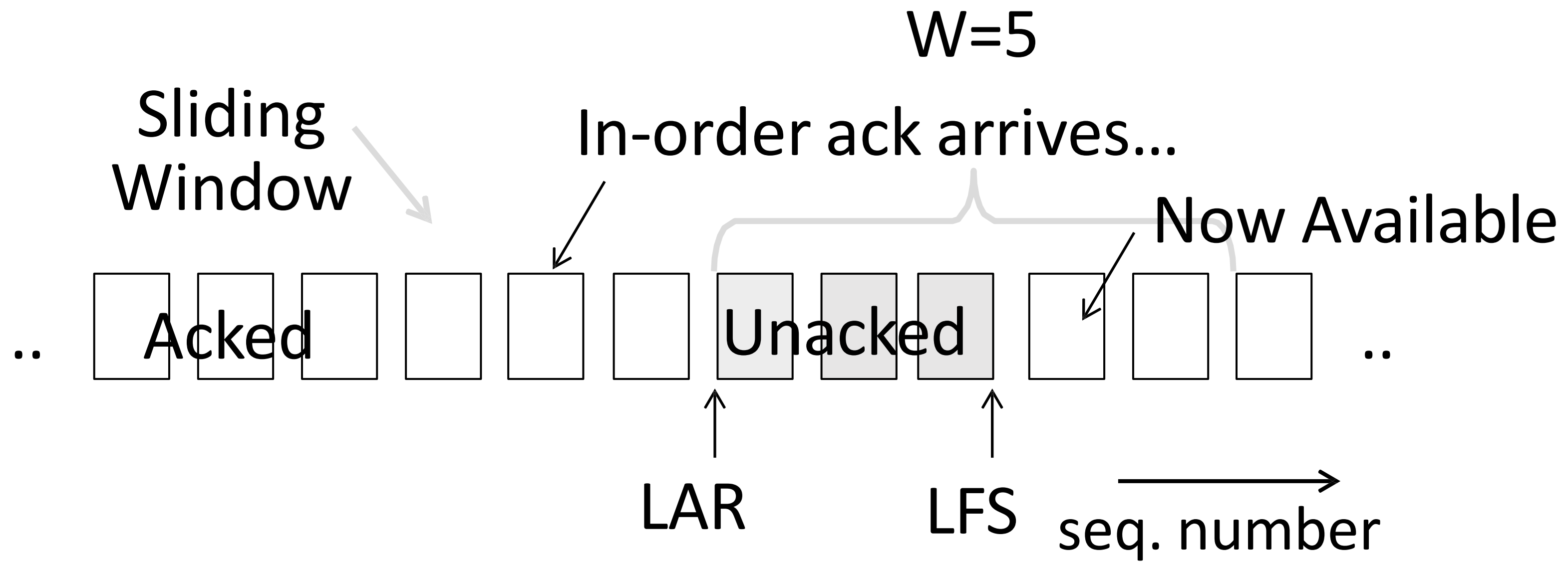
Sender sliding window – Selective Repeat

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

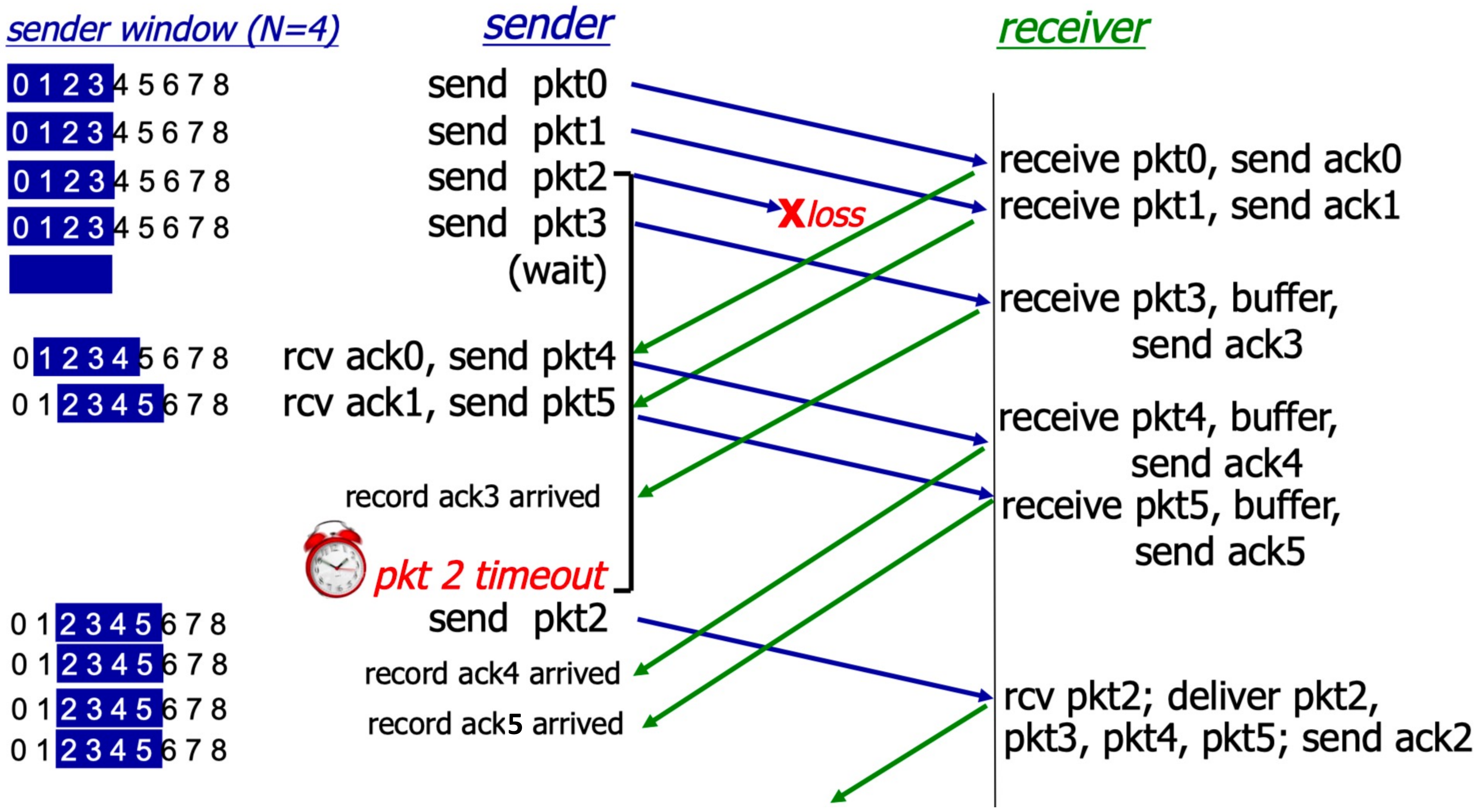


Sender sliding window – Selective Repeat

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



Selective Repeat in action



Sliding window – Retransmissions

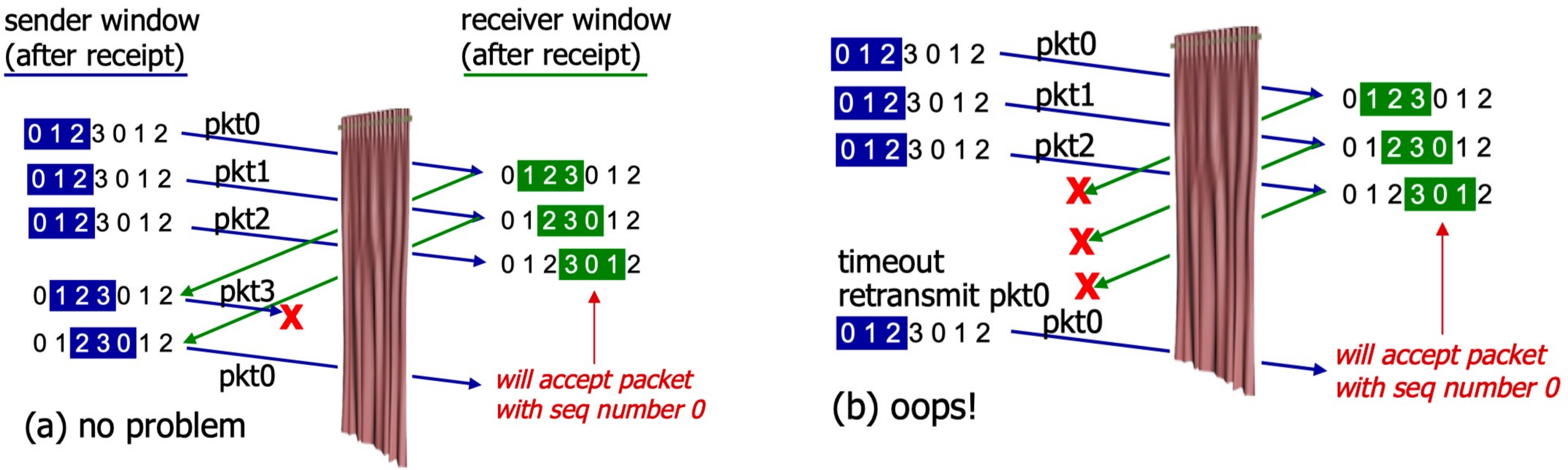
- Go-Back-N uses a single timer to detect losses
  - On timeout, resends buffered packets starting at LAR+1
- Selective Repeat uses a timer per unacked segment to detect losses
  - On timeout for segment, resend it
  - Hope to resend fewer segments

Sequence numbers

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 numbers



receiver can't see sender side.
receiver behavior identical in both cases!
something's (very) wrong!

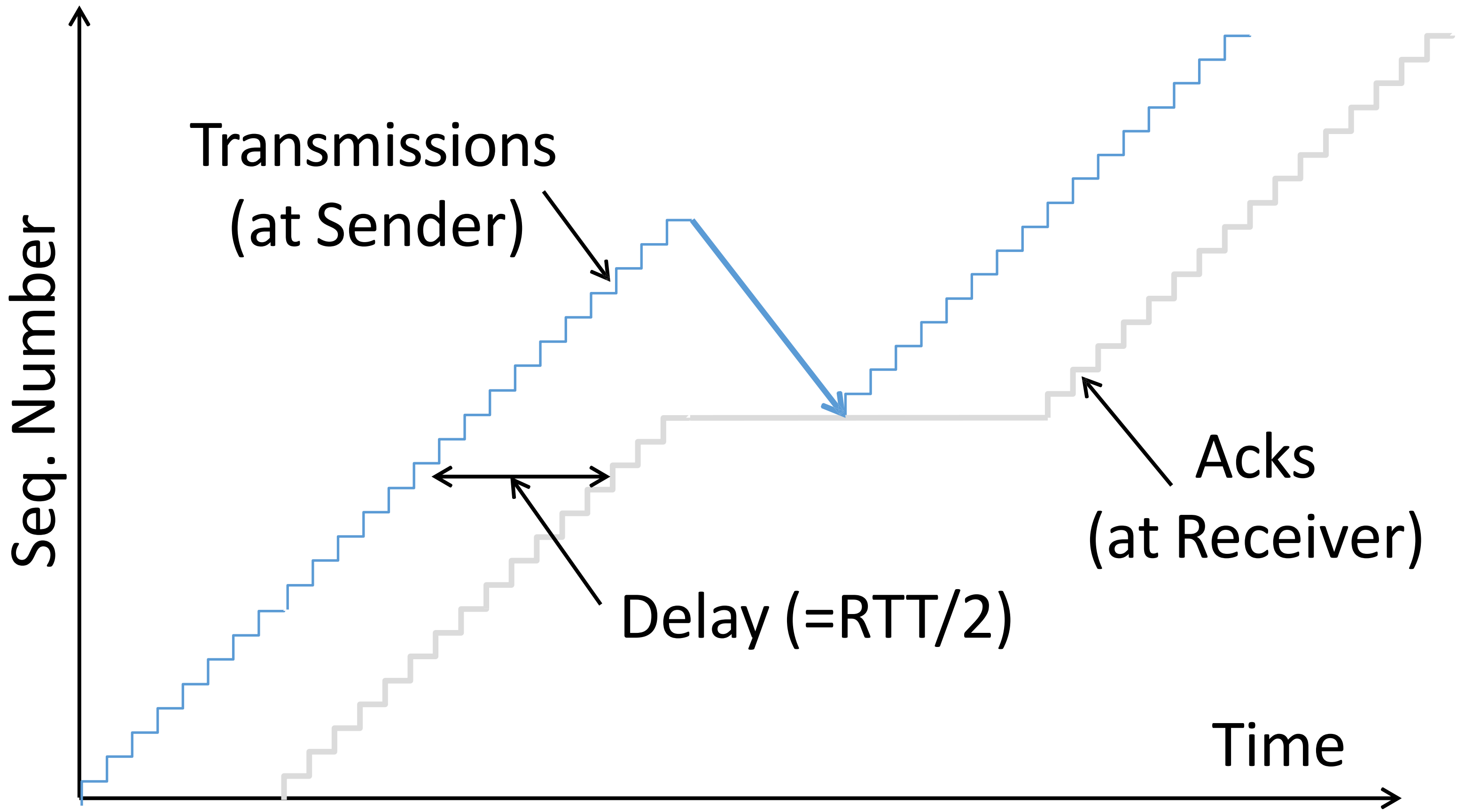
How many sequence numbers?

For Selective Repeat: 2W seq numbers

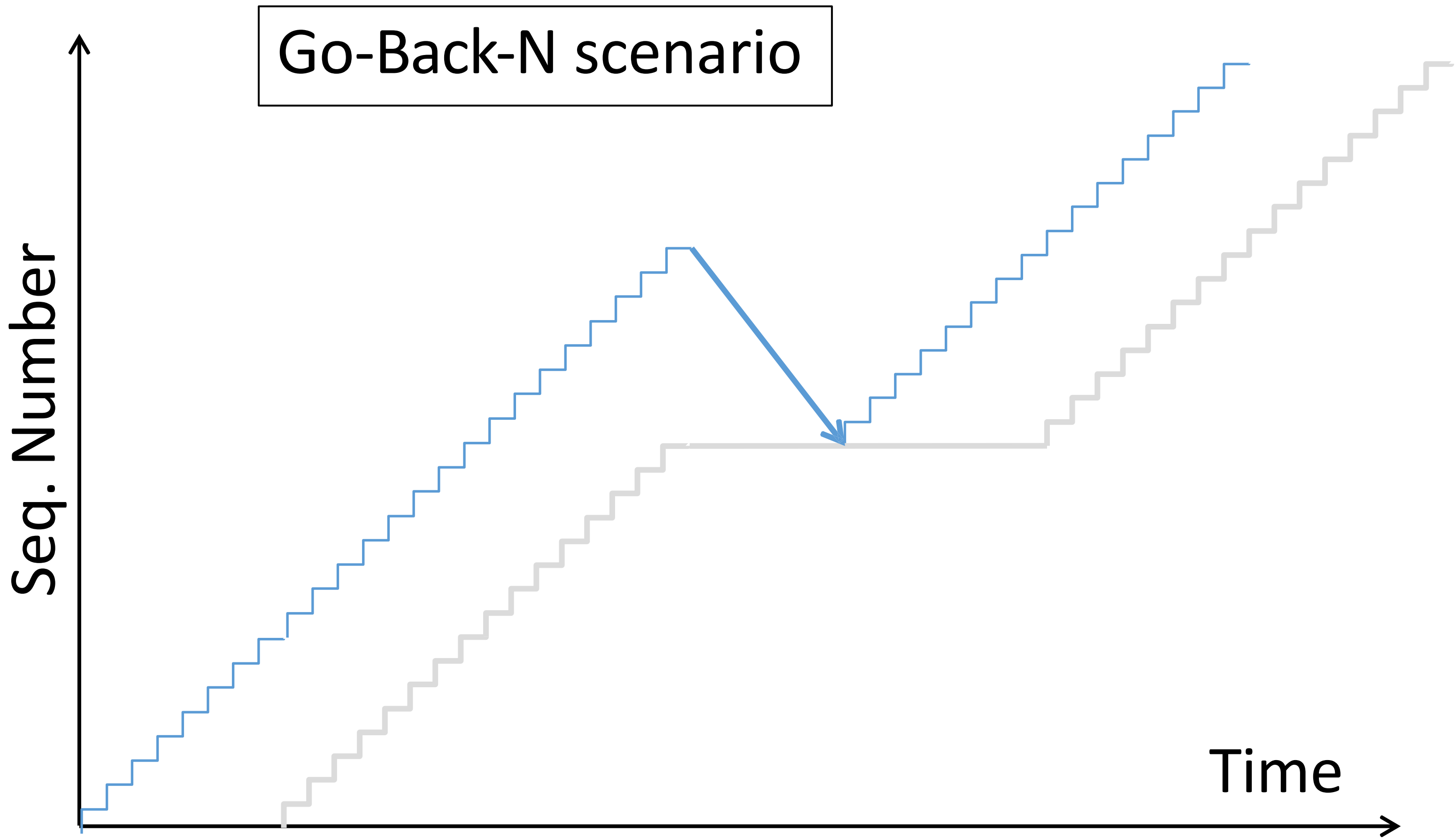
- W for packets, plus W for earlier acks

For Go-Back-N: W+1 sequence numbers

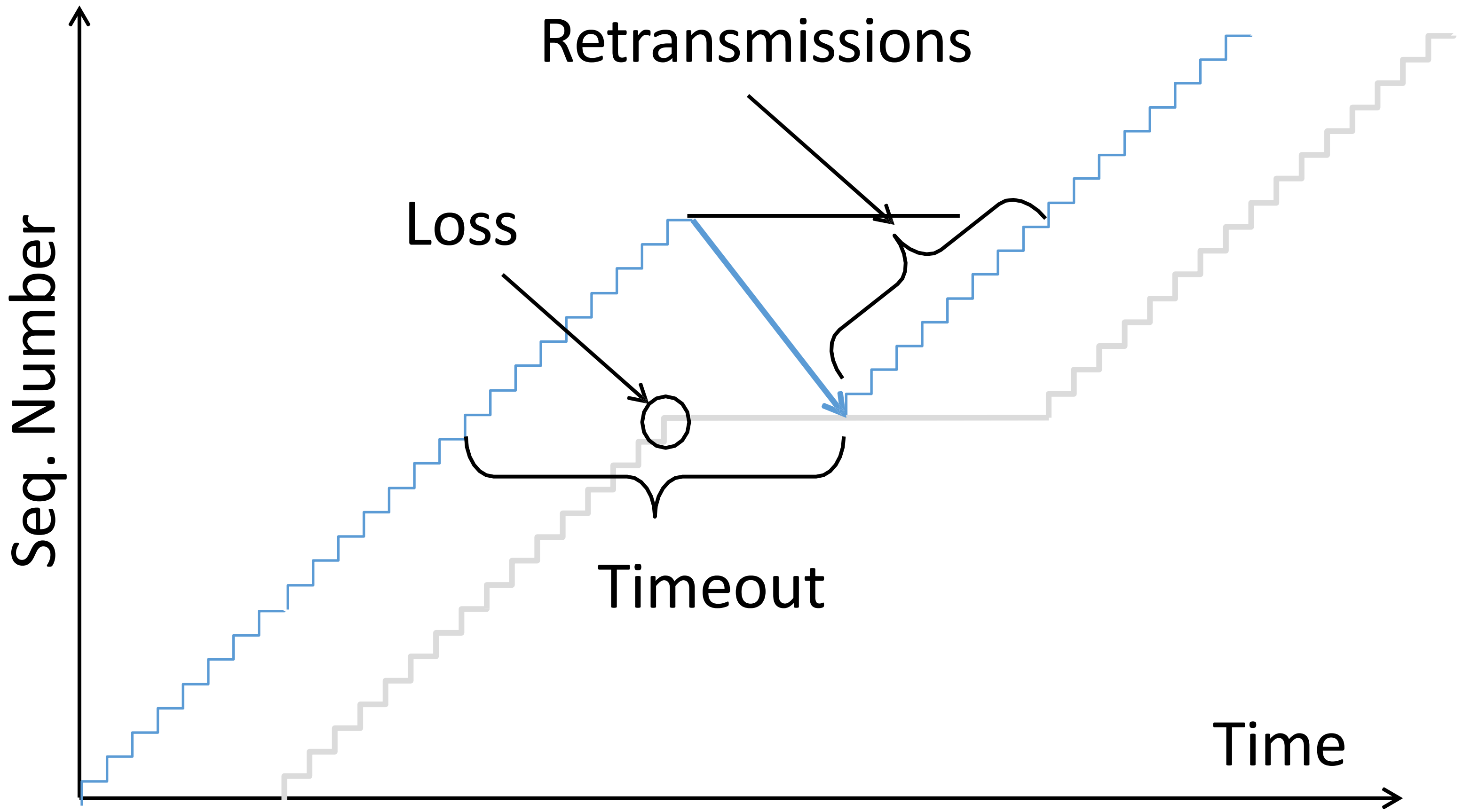
Sequence time plot



Sequence time plot

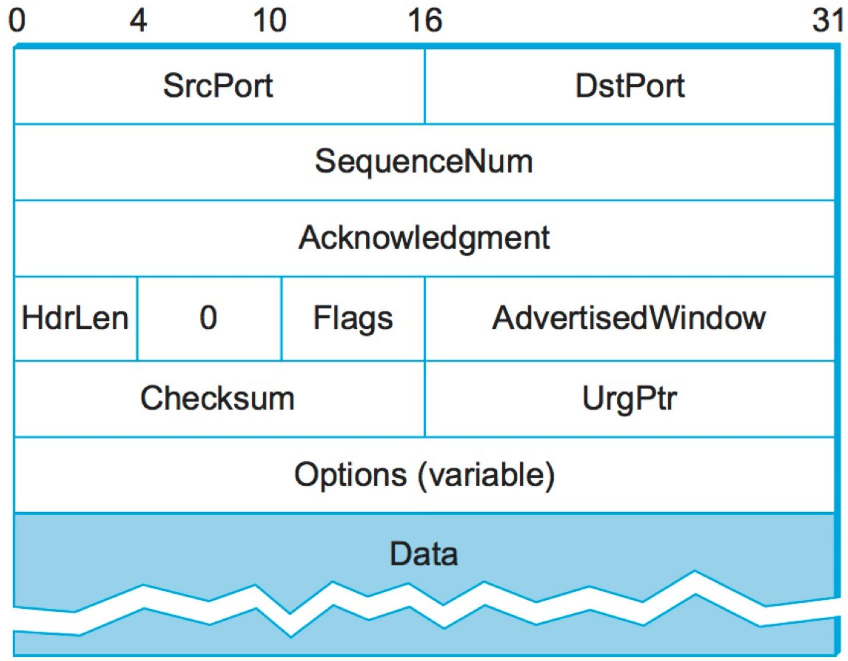


Sequence time plot



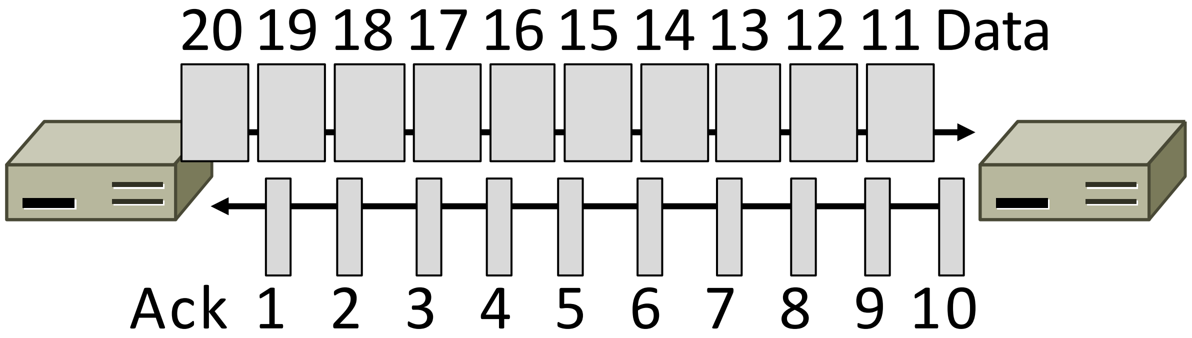
TCP header

- Uses ports to identify sending and receiving sockets
- Seq. and ACK numbers counted by bytes of data (not packets)
- Advertised window size: # of bytes in receiver's available buffer
- Checksum (as in UDP)



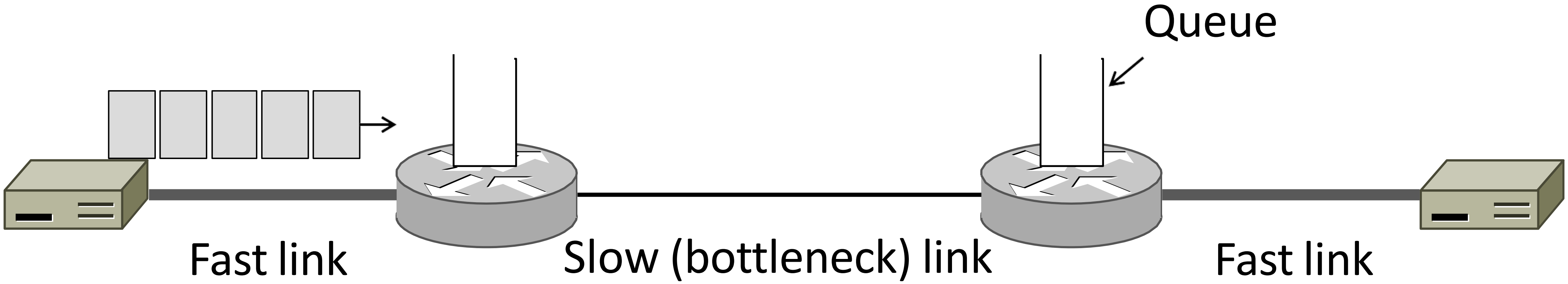
Sliding window ACK clock

- 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



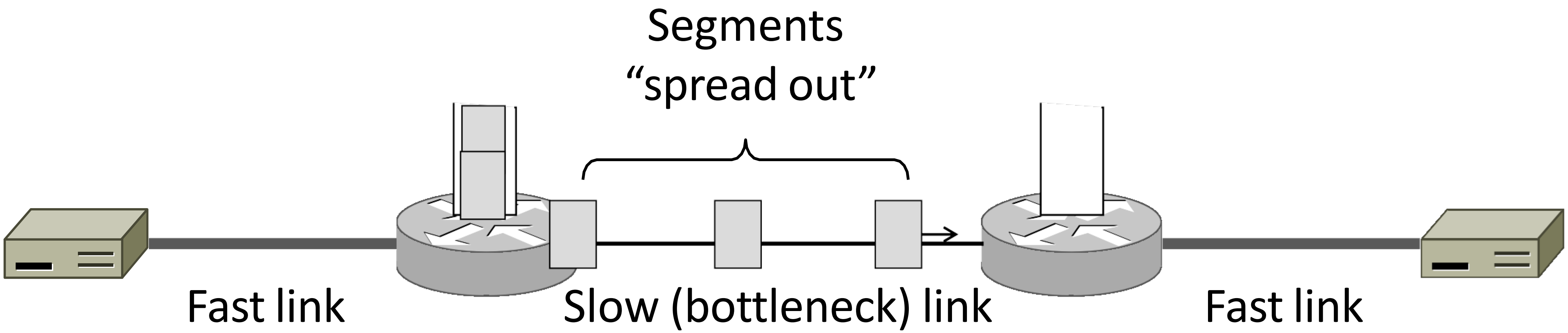
Benefit of ACK clocking

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



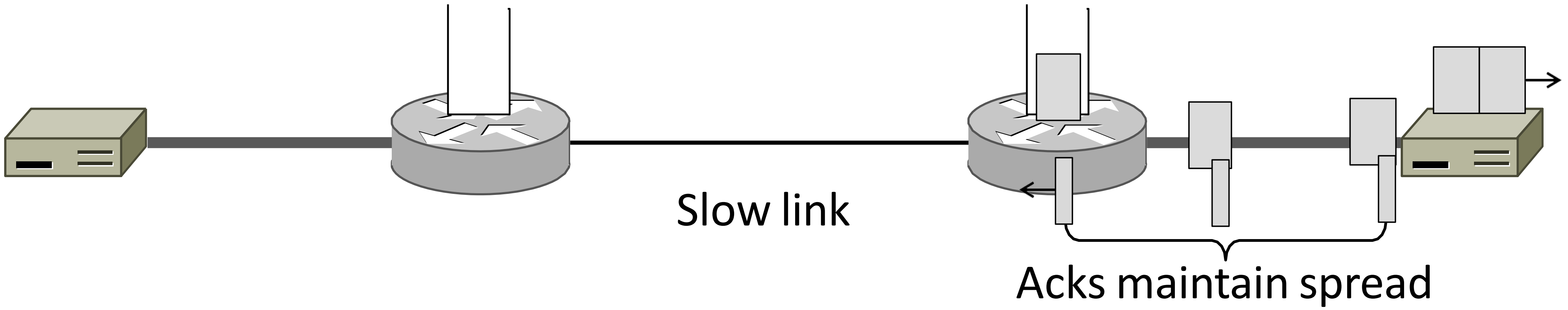
Benefit of ACK clocking

Segments are buffered and spread out on slow link



Benefit of ACK clocking

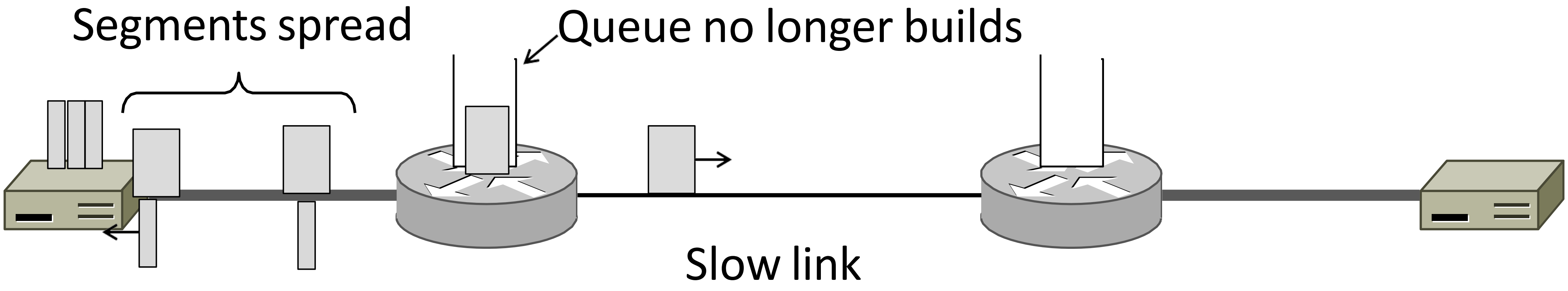
ACKs maintain the spread back to the original sender



Benefit of ACK clocking

Sender clocks new segments with the spread

- Now sending at the bottleneck link without queuing!



Benefit of ACK clocking

- 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

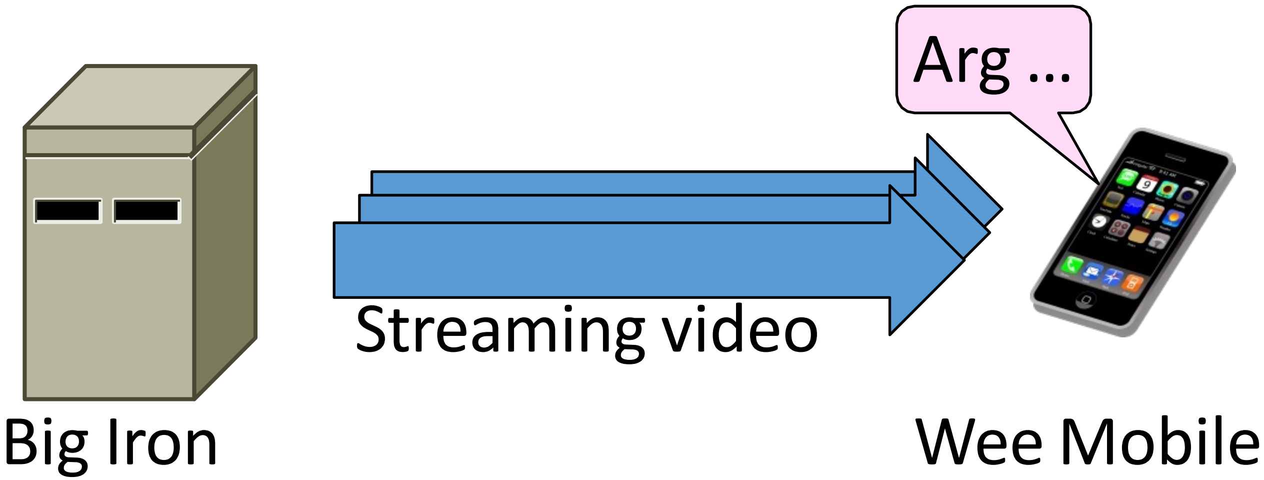
TCP uses ACK clocking

- 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

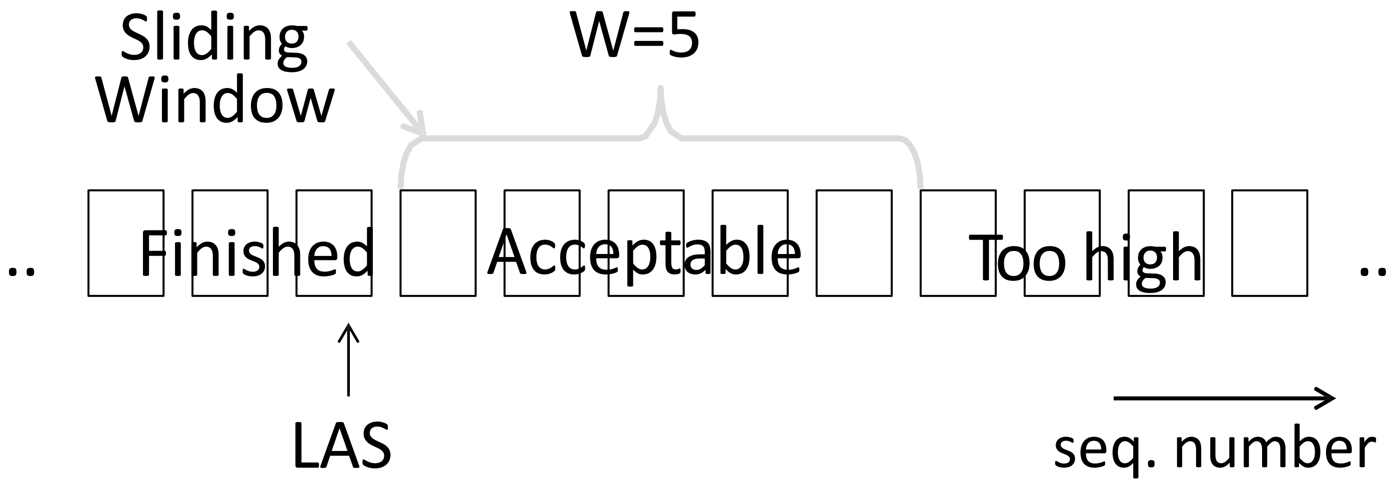
Sliding window has pipelining to keep network busy

What if the receiver is overloaded?



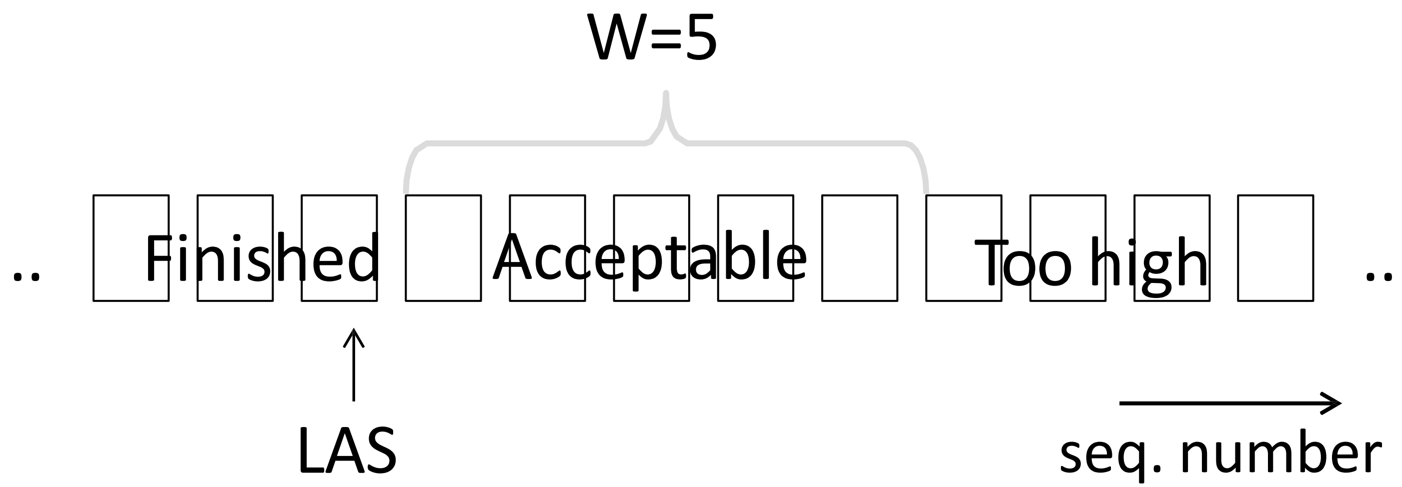
Receiver sliding window

- Consider receiver with W buffers
  - LAS=last ack sent
  - app pulls in-order data from buffer with recv() call



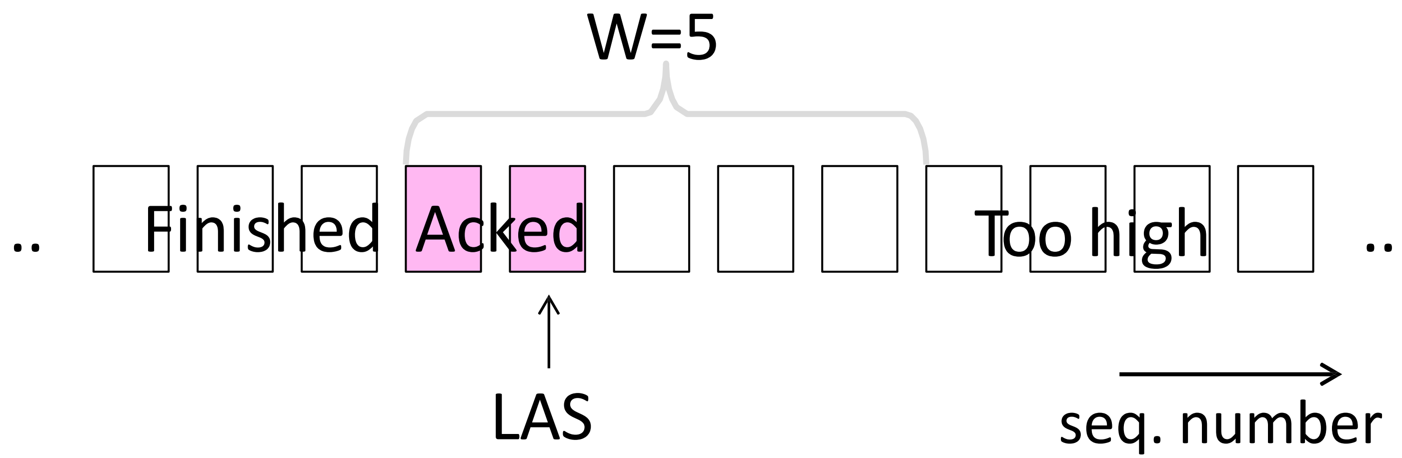
Receiver sliding window

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



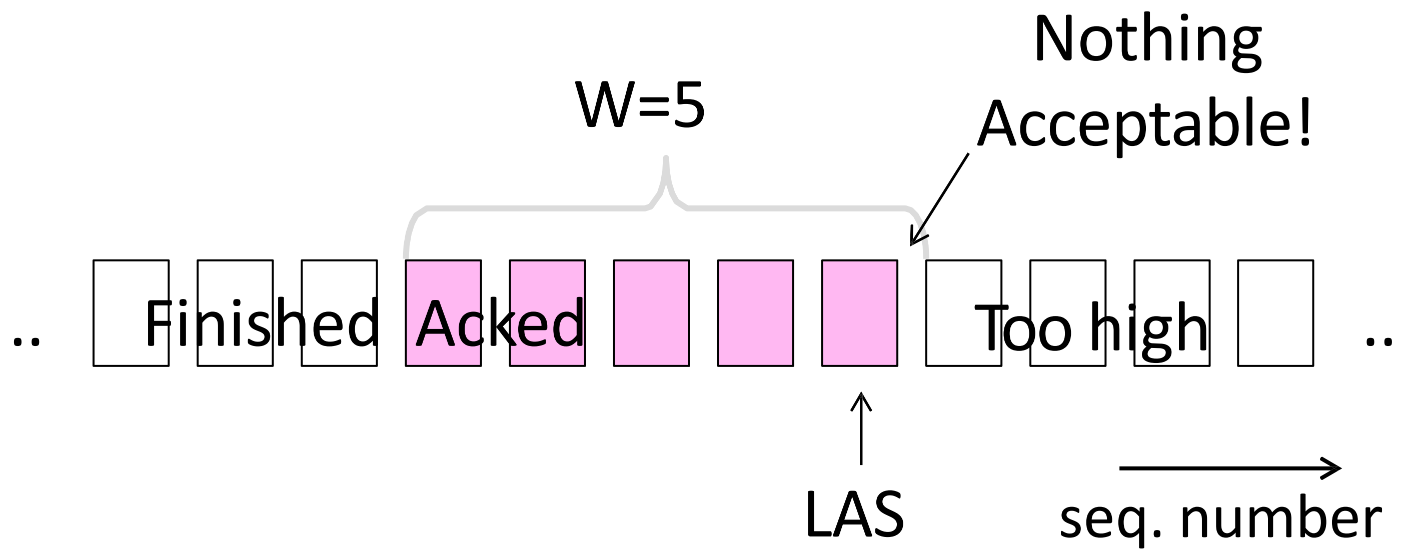
Receiver sliding window

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



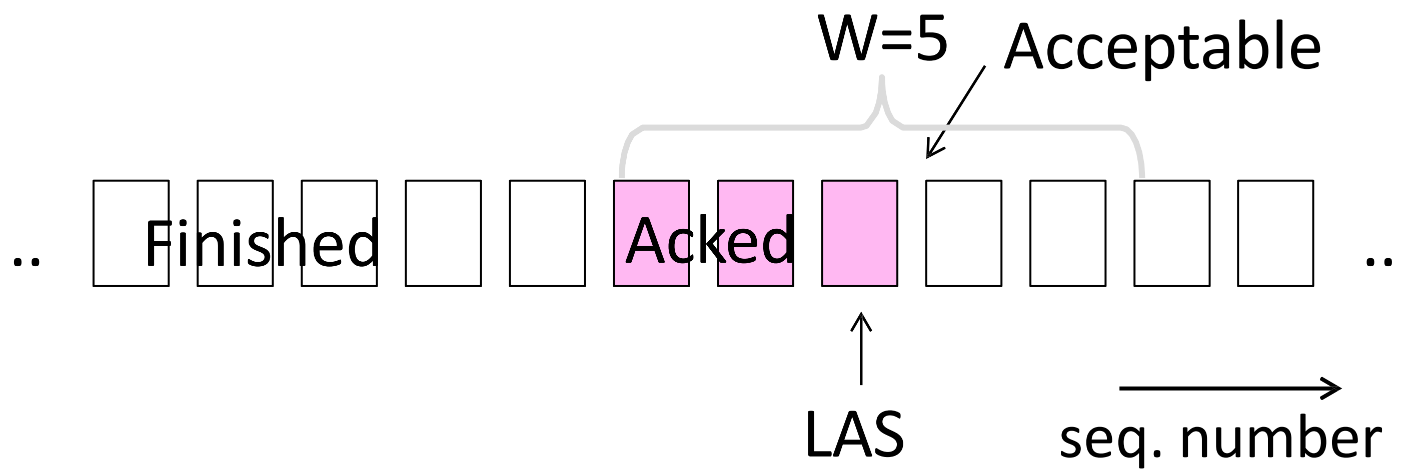
Receiver sliding window

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



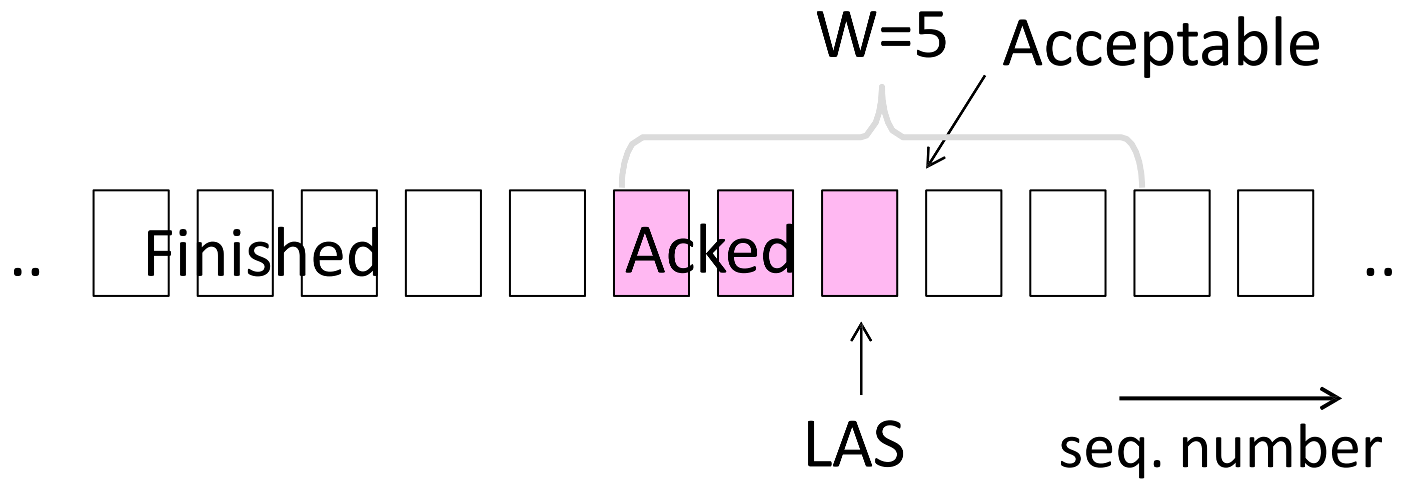
Receiver sliding window

- App recv() takes two segments
  - Window slides



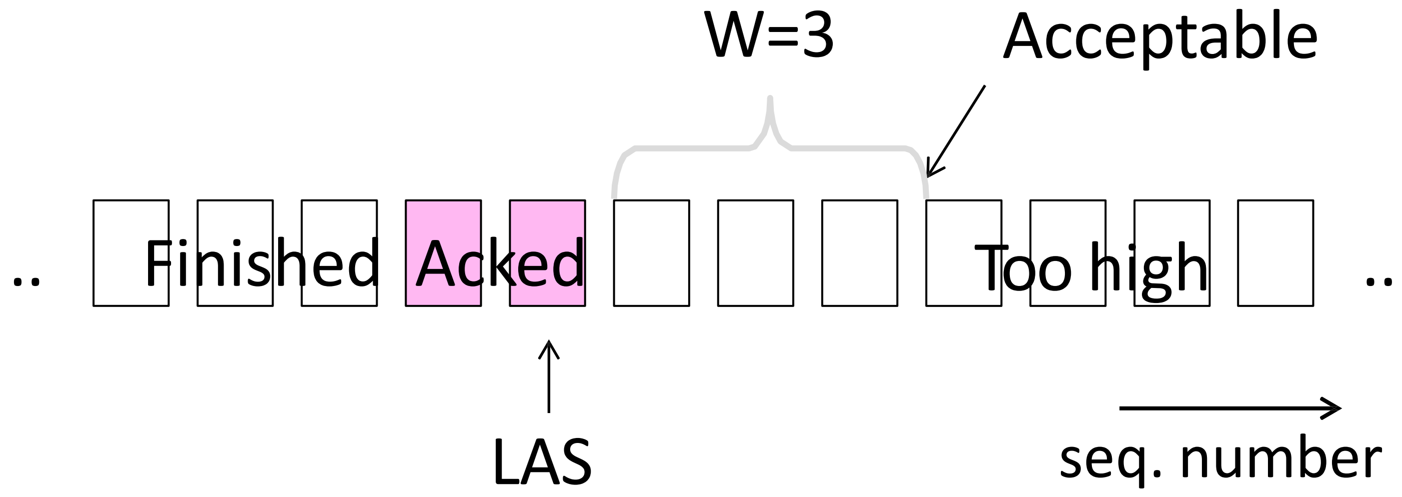
Flow control

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



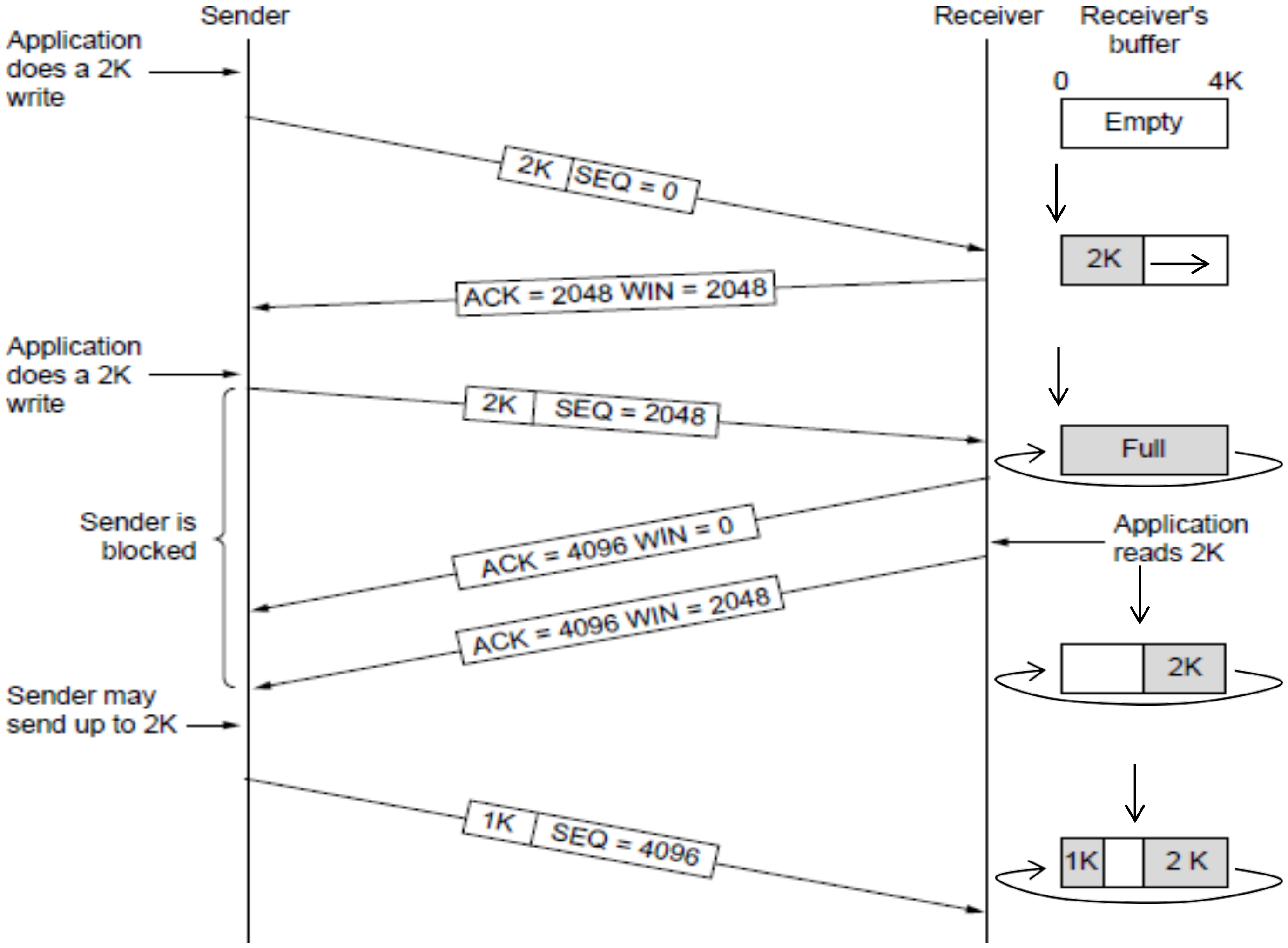
Flow control

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



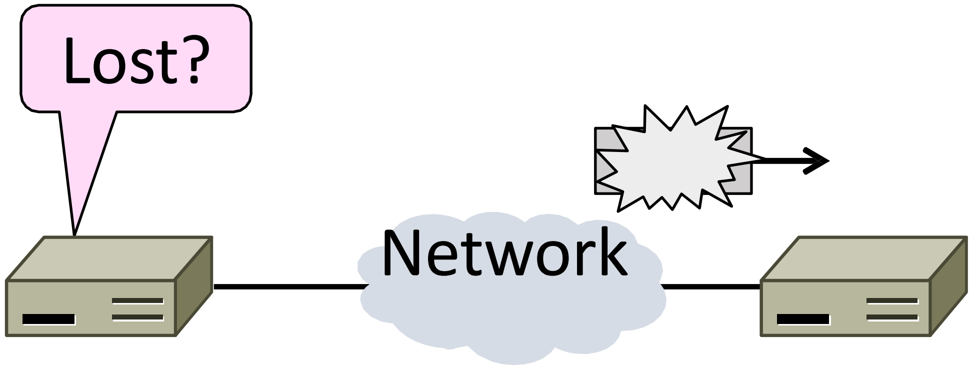
Flow control

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



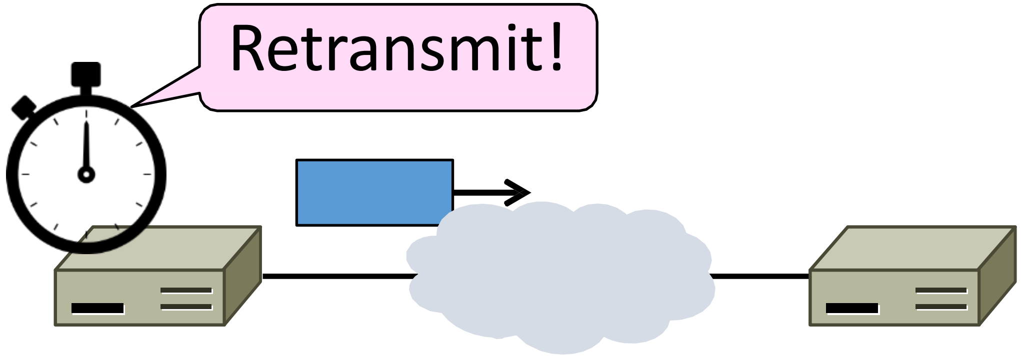
Topic

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



Retransmissions

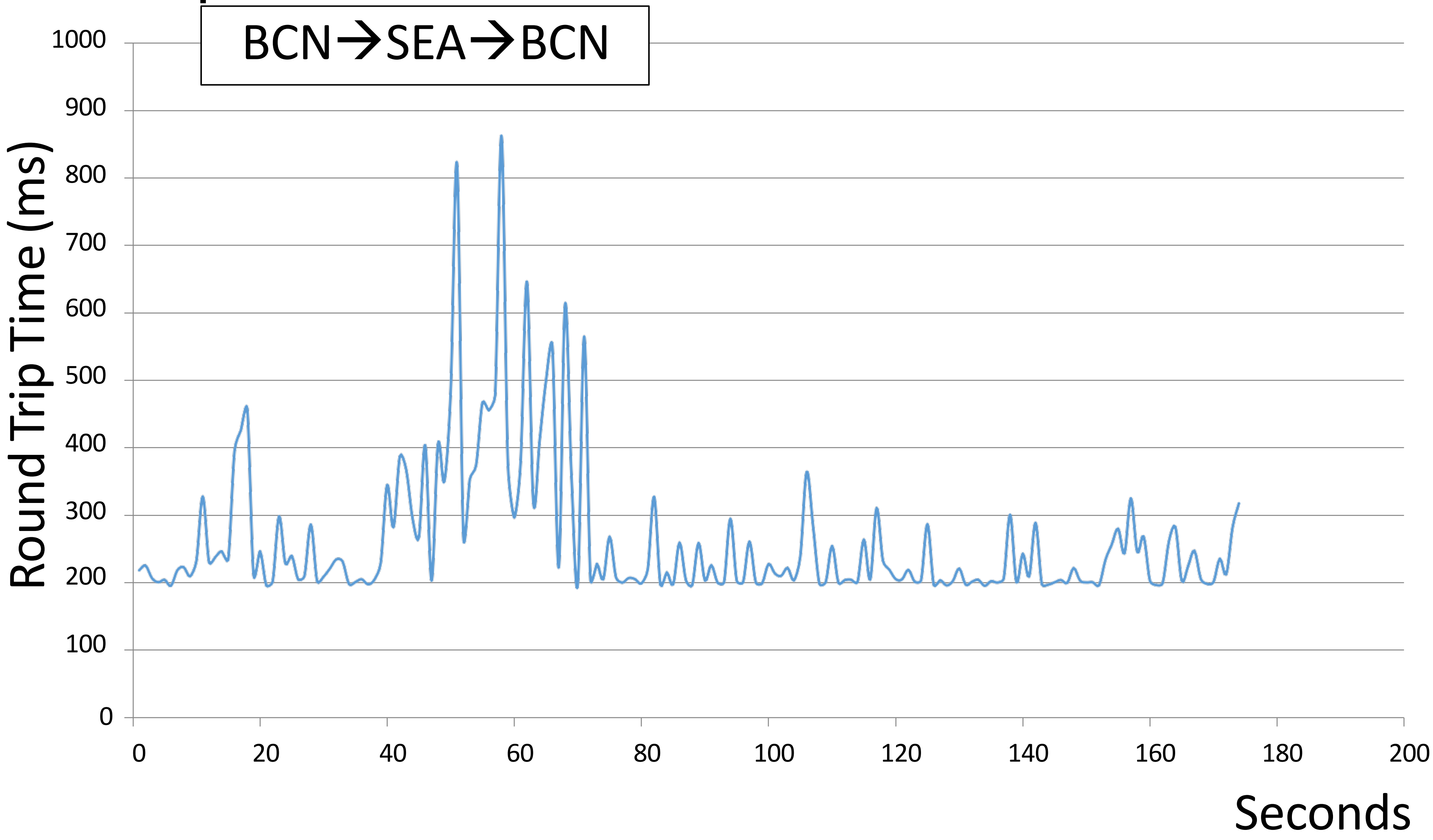
- With sliding window, detecting loss with timeout
  - Set timer when a segment is sent
  - Cancel timer when ack is received
  - If timer fires, retransmit data as lost



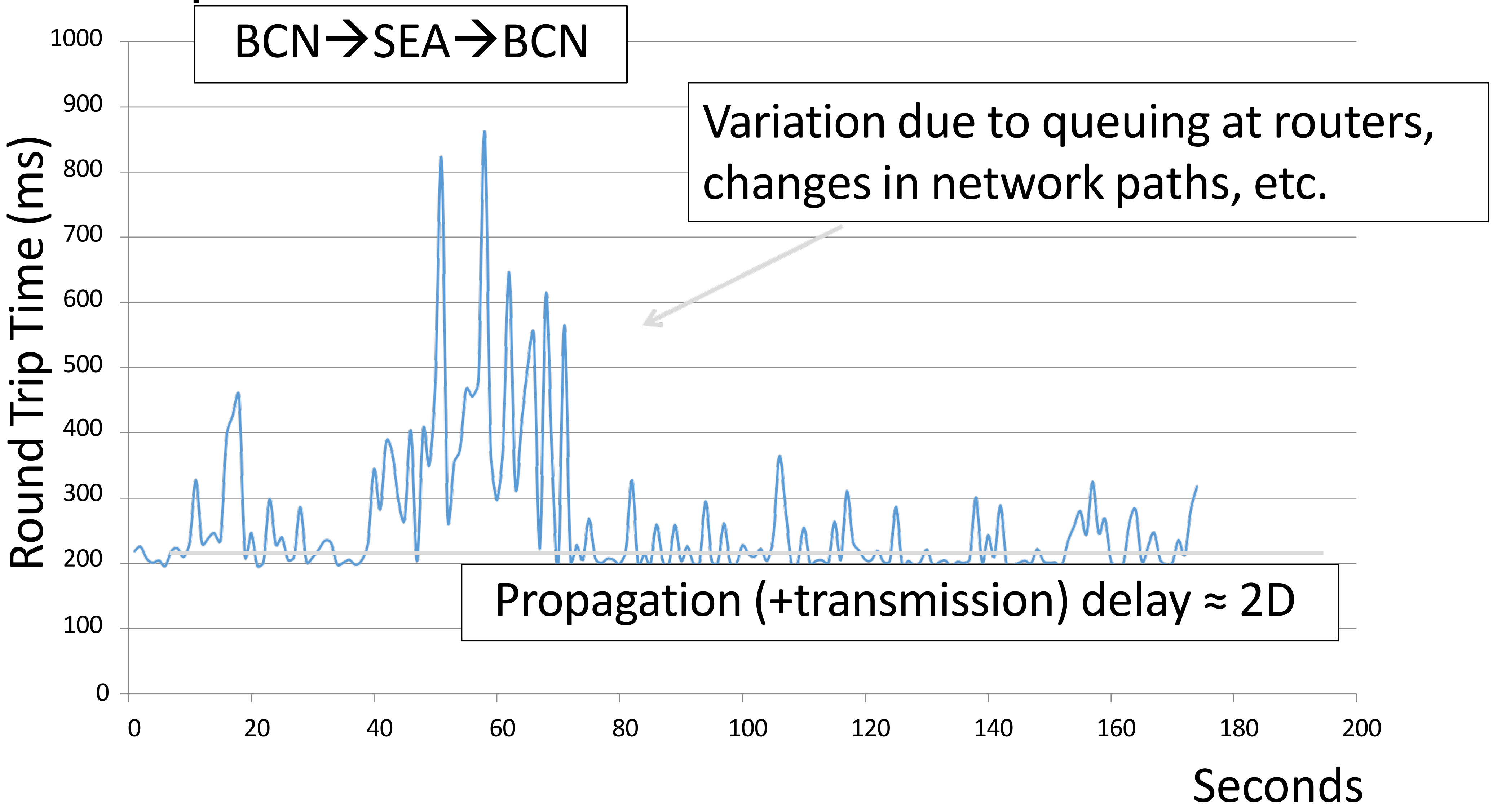
Timeout problem

- Timeout should be “just right”
  - Too long -> 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

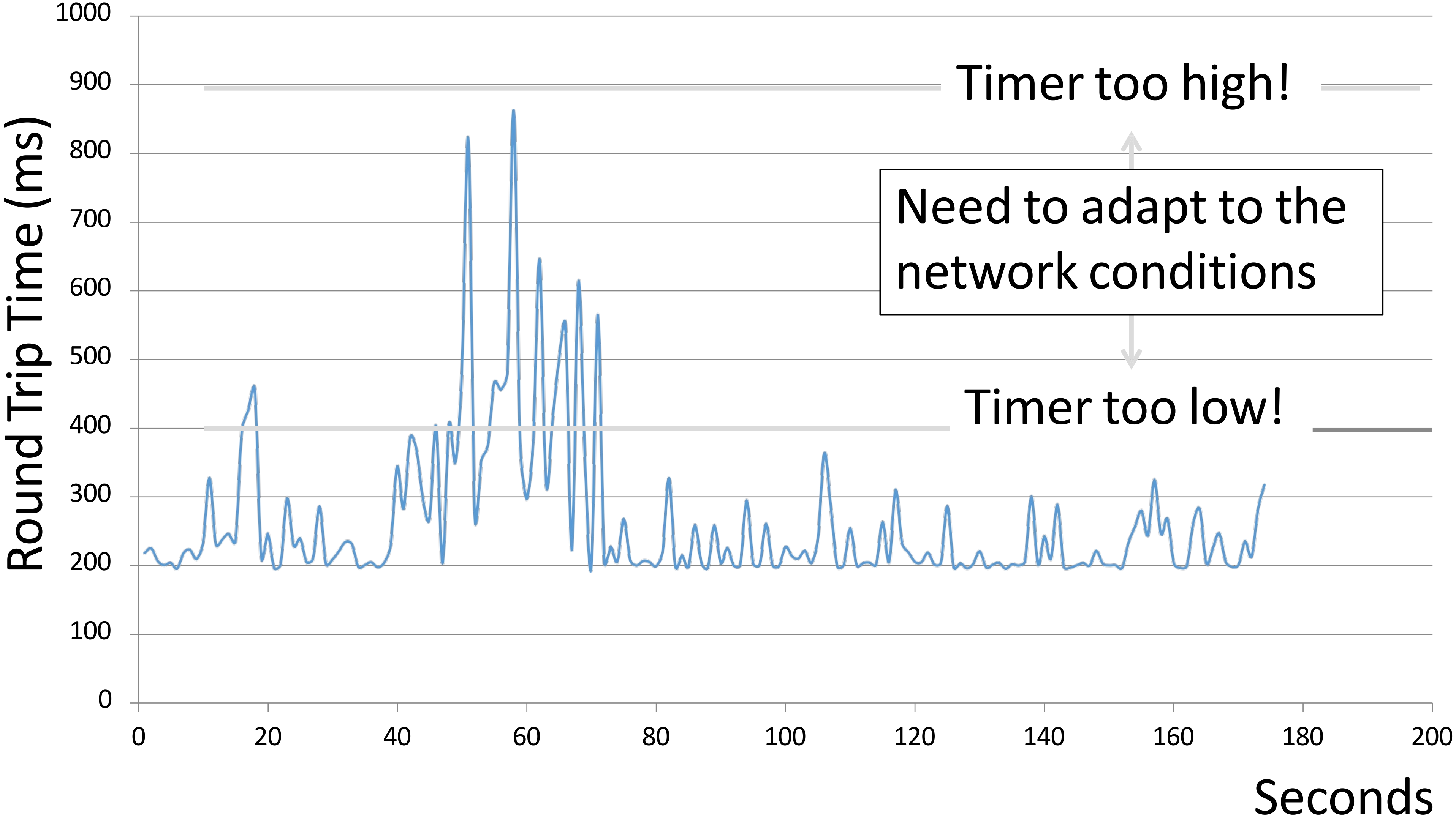
Example of RTTs



Example of RTTs



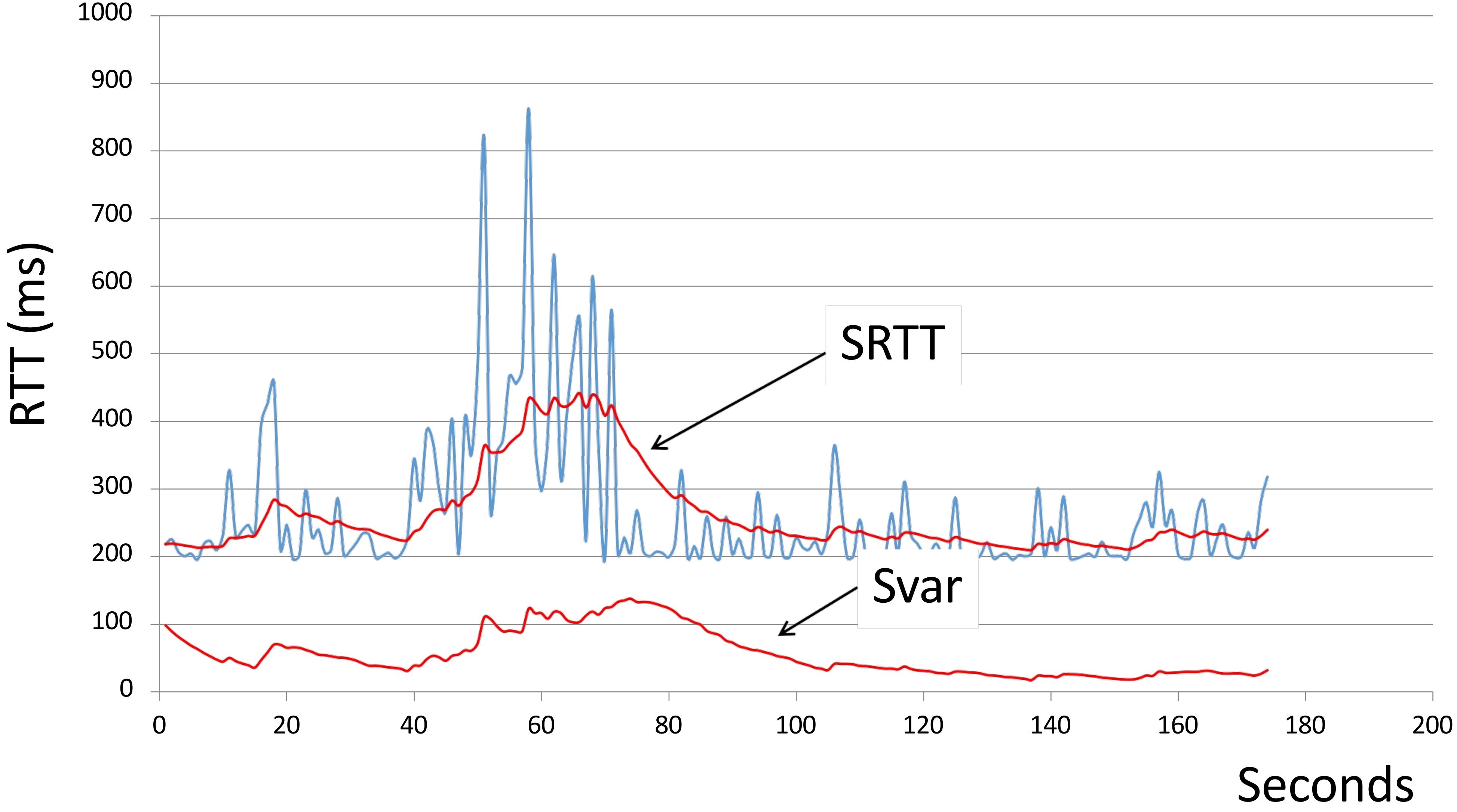
Example of RTTs



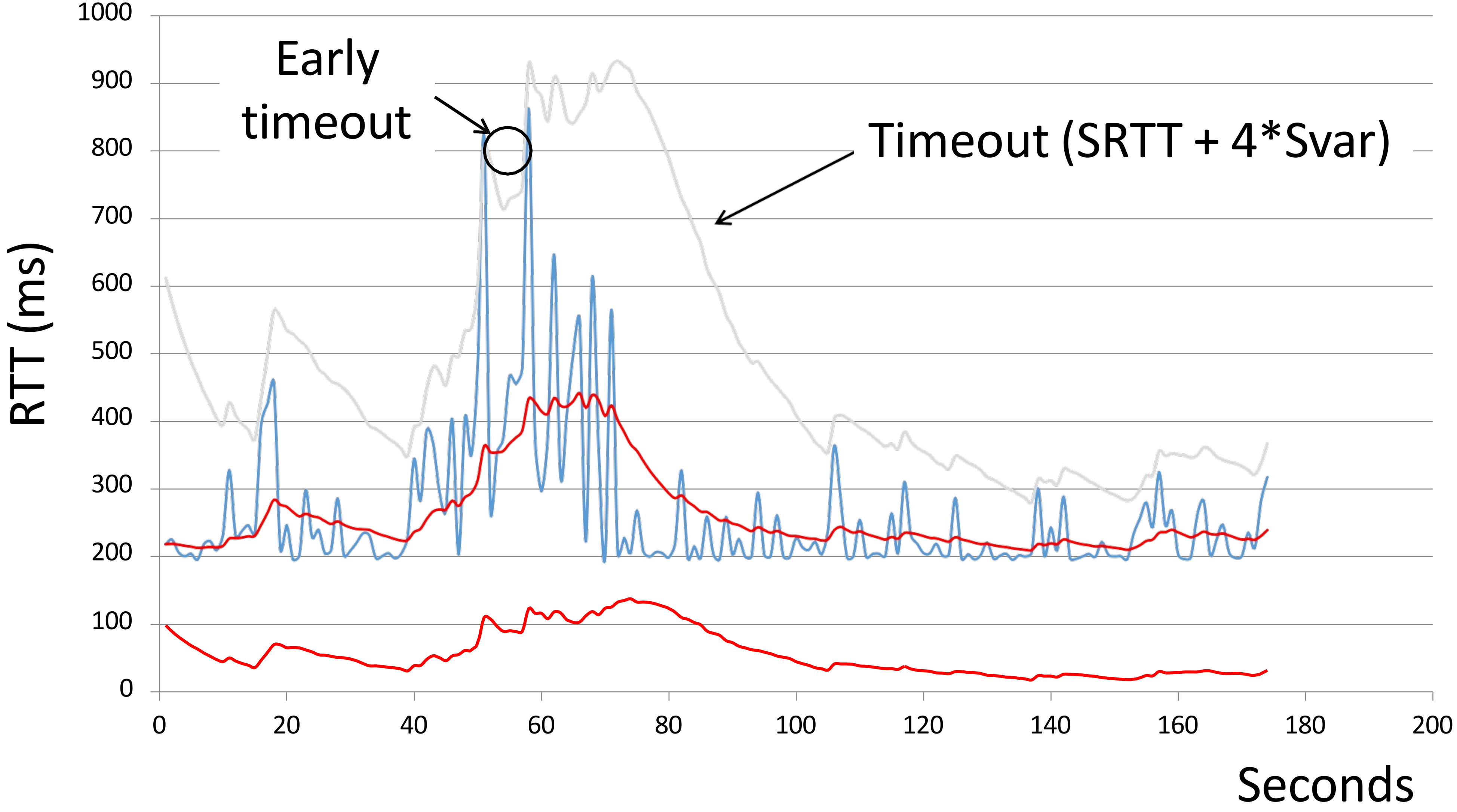
Adaptive timeout

- Smoothed estimates of the RTT (SRTT) and variance in RTT (Svar) with sampled RTT (RTT)
  - Update estimates with a moving average
  - $SRTT_{N+1} = (1 - \alpha)*SRTT_N + \alpha*RTT_{N+1}$ e.g., $\alpha = 0.125$
  - $Svar_{N+1} = (1 - \beta)*Svar_N + \beta*|RTT_{N+1} – SRTT_{N+1}|$ e.g., $\beta = 0.25$
- 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



Adaptive timeout

- Simple to compute, does a good job of tracking actual RTT
- Turns out to be important for good performance and robustness

Credits

- Some slides are adapted from course slides of CSE 461 in UW