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 \( \text{LAS} = \text{LAST ACK SENT} \)

• On receive:
  • Buffer segments \([\text{LAS}+1, \text{LAS}+W]\)
  • Send app in-order segments from \(\text{LAS}+1\), and update \(\text{LAS}\)
  • Send ACK for \(\text{LAS}\) 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 (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)
Sequence Time Plot (2)

Go-Back-N scenario
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
Benefit of ACK Clocking

• Consider what happens when sender injects a burst of segments into the network
Benefit of ACK Clocking (2)

- Segments are buffered and spread out on slow link
Benefit of ACK Clocking (3)

• **ACKs maintain the spread back to the original sender**
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?
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 (2)

• Suppose the next two segments arrive but app does not call recv()
Receiver Sliding Window (3)

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

```
.. Finished Acked .. Too high ..
```

```
W=5

Las  seq. number
```

CSE 461 University of Washington
Receiver Sliding Window (4)

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

.. Finished Aged Too high ..

W=5

Nothing Acceptable!

LAS seq. number
Receiver Sliding Window (5)

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

.. finished ..

W=5

Acceptable

.. Acked ..

LAS

seq. number
Flow Control

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

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

**Propagation (+transmission) delay ≈ 2D**

**BCN → SEA → BCN**
Example of RTTs (3)

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

![Graph showing RTT (ms) vs Seconds with SRTT and Svar lines]
Example of Adaptive Timeout (2)

Early timeout

Timeout (SRTT + 4*Svar)
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