6.033
Reliability & Congestion Control

March, 2011
E2E Transport

- Reliability: “At Least Once Delivery”
  - Lock-step
  - Sliding Window

- Congestion Control
  - Flow Control
  - Additive Increase Multiplicative Decrease
“At Least Once” (Take 1): Lock-Step

- Each data packet has a *sequence number* set by sender
- Receiver: upon receipt of packet \( k \), sends acknowledgment (ack) for \( k \) ("I got \( k \")
- Sender: Upon ack \( k \), sends \( k+1 \). If no ack within *timeout*, then retransmit \( k \) (until acked)
How Long to Set Timeout?

• Fixed timeouts don’t work well
  – Too big $\rightarrow$ delay too long
  – Too small $\rightarrow$ unnecessary retransmission

• Solution
  – Timeout should depend on RTT
  – Sender measures the time between transmitting a packet and receiving its ack, which gives one sample of the RTT
But RTT Could Be Highly Variable

Example from a TCP connection over a wide-area wireless link
Mean RTT = 0.25 seconds; Std deviation = 0.11 seconds!

Can’t set timeout to an RTT sample; need to consider variations
Calculating RTT and Timeout: (as in TCP)

Exponentially Weighted Moving Average (EWMA)

• Estimate both the average `rtt_avg` and the deviation `rtt_dev`

• **Procedure calc_rtt(rtt_sample)**
  
  
  `rtt_avg ← a*rtt_sample + (1-a)*rtt_avg; /* a = 1/8 */`
  
  `dev ← absolute(rtt_sample – rtt_avg);`
  
  `rtt_dev ← b*dev + (1-b)*rtt_dev; /* b = 1/4 */`

• **Procedure calc_timeout(rtt_avg, rtt_dev)**
  
  Timeout ← rtt_avg + 4*rtt_dev
Improving Performance

• Lock-step protocol is too slow: send, wait for ack, send, wait for ack, ...

• Throughput is just one packet per RTT

• Solution: Use a window
  – Keep multiple packets in the network at once
  – overlap data with acks
At Least Once (Take 2): Fixed Window

- Receiver tells the sender a window size
- Sender sends window
- Receiver acks each packet as before
- Window advances when all pkts in previous window are acked
  - E.g., packets 4-6 sent, after 1-3 ack’d
- If a packet times out → rxmit pkt
- Still much idle time
At Least Once (Take 3): *Sliding Window*

- **Sender** advances the window by 1 for each in-sequence ack it receives
  - Reduces idle periods
  - Pipelining idea!

- **But what’s the correct value for the window?**
  - We’ll revisit this question
  - First, we need to understand windows
Sliding Window in Action

Example: $W = 5$; We show how the window slides with ack arrivals.

Example: $W = 5$; We show how the window slides with ack arrivals.
Sliding Window in Action

Example: \( W = 5 \); We show how the window slides with ack arrivals

\[ \text{window} = 2-6 \]

\[ \text{window} = 3-7 \]

\( \text{Sndr} \)

\( 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \)

\( \text{Rcvr} \)

\( \text{d1} \quad \text{d2} \)

\( \text{a1} \quad \text{a2} \)
Sliding Window in Action

Example: \( W = 5; \) We show how the window slides with ack arrivals
Handling Packet Loss

Sender advances the window on arrivals of in-sequence acks

→ Can’t advance on a3’s arrival
Handling Packet Loss

window = 2-6

d2 times out

Sender advances the window on arrivals of in-sequence acks
→ Can’t advance on a3’s arrival
Handling Packet Loss

Sender advances the window on arrivals of in-sequence acks

→ Can’t advance on a3’s arrival
What is the Right Window Size?

- Window is too small
  - long Idle time
  - Underutilized Network

- Window too large
  - Congestion

Sender

Receiver

May I Send?

OK, 3 pkts

W = 1-3

W = 2-4

W = 3-5

W = 4-6

Idle
E2E Transport

• Reliability: “At Least Once Delivery”
  – Lock-step
  – Sliding Window

• Congestion Control
  – Flow Control
  – Additive Increase Multiplicative Decrease
Setting Window Size: Flow Control

Window $\leq$ Receiver Buffer

– Otherwise receiver drops packets
Setting Window Size: Congestion

- Sender transmits faster than bottleneck capacity
  → Queue builds up
  → Router drops packets
- Tx Rate \( \leq \) Bottleneck Capacity
- Tx Rate = Window / RTT

Window \( \leq \) \text{min}(Receiver Buffer, Bottleleneck\_Cap \times RTT)
Setting Window Size: Congestion

Bottleneck may be shared

Window \leq \min(\text{Receiver Buffer, } cwnd)

Congestion Control Protocol adapts the congestion window (cwnd) to ensure efficiency and fairness
Congestion Control

• Basic Idea:
  – Increase cwnd slowly; if no drops $\rightarrow$ no congestion yet
  – If a drop occurs $\rightarrow$ decrease cwnd quickly

• Use the idea in a distributed protocol that achieves
  – Efficiency, i.e., uses the bottleneck capacity efficiently
  – Fairness, i.e., senders sharing a bottleneck get equal throughput (if they have demands)
Additive Increase Multiplicative Decrease

• Every RTT:
  
  No drop: \( cwnd = cwnd + 1 \)
  
  A drop: \( cwnd = cwnd/2 \)
Additive Increase

\[ \text{cwnd} = 1 \]

\[ \text{cwnd} = \text{cwnd} + 1 \]

\[ \text{cwnd} = 2 \]

\[ \text{cwnd} = 3 \]

\[ \text{cwnd} = 4 \]
AIMD Leads to Efficiency and Fairness

Consider two users who have the same RTT

MD → move on lines through origin

AI → move on lines parallel to fairness line

Efficiency line: \( cwnd_1 + cwnd_2 = \text{RTT} \times \text{bottleneck}_\text{cap} \)

Fairness line: \( cwnd_1 = cwnd_2 \)
Summary of E2E Transport

• Reliability Using Sliding Window
  – Tx Rate = W/ RTT

• Congestion Control
  – W = \min(\text{Receiver_buffer}, \text{cwnd})
  – cwnd is adapted by the congestion control protocol to ensure efficiency and fairness
  – TCP congestion control uses AIMD which provides fairness and efficiency in a distributed way