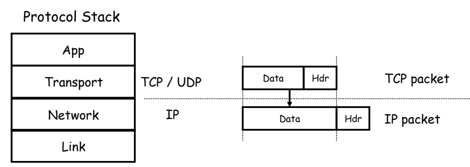
Reliability & Congestion Control

6.033 Lecture 13
Dina Katabi & Frans Kaashoek

The Internet Stack



Internet: Best Effort

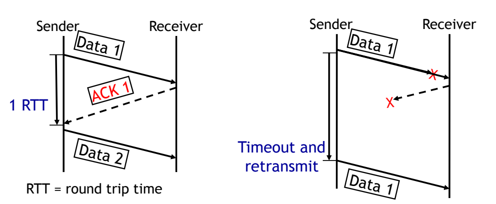
No Guarantees:

- Variable Delay (jitter)
- Variable rate
- Packet loss
- Duplicates
- Reordering
- Maximum length

E2E Transport

- Reliability: "At Least Once Delivery"
  - Lock-step
  - Sliding Window
- Congestion Control
  - Flow Control
  - Additive Increase Multiplicative Decrease

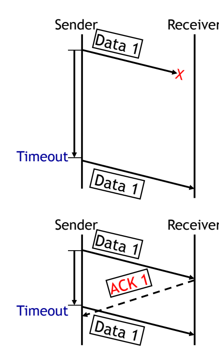
"At Least Once" (Take 1): Lock-Step



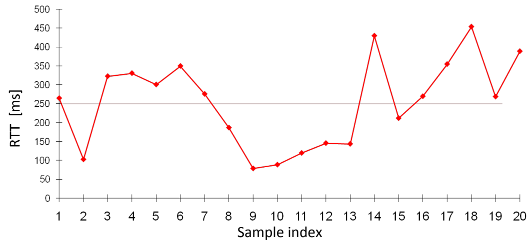
- Each data packet has a *sequence number* set by sender
- Receiver: upon receipt of packet *k*, sends acknowledgment (ack) for *k* ("I got *k*")
- Sender: Upon ack *k*, sends *k*+1. If no ack within *timeout*, then retransmit *k* (until acked)

How Long to Set Timeout?

- Fixed timeouts don't work well
  - Too big → delay too long
  - Too small → 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)

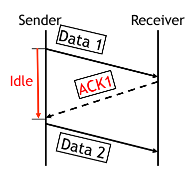
Exponentially Weighted Moving Average (EWMA)

- 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 */

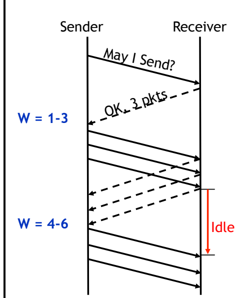
- **Procedure calc_timeout(rtt_avg, rtt_dev)**
  Timeout ← rtt_avg + 4*rtt_dev

Improving Performance



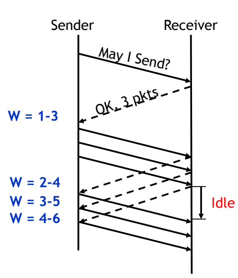
- Lock-step protocol is too slow: send, wait for ack, send, wait for ack, …
- Throughput is just one packet per RTT
- Solution: Use a *window*
  - Keep multiple packets in the network at once
  - overlap data with acks

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

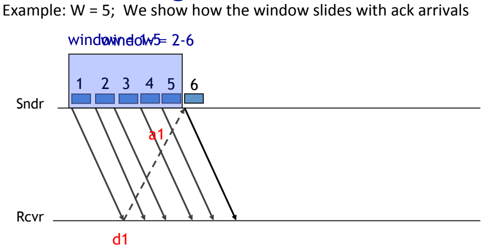
At Least Once (Take 3): *Sliding* Window



- 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

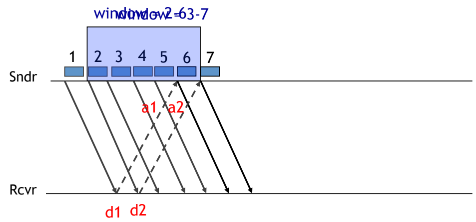
Sliding Window in Action

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



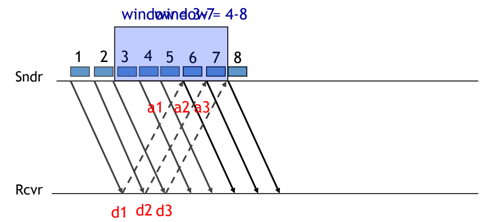
Sliding Window in Action

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

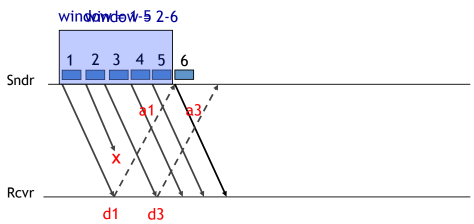


Sliding Window in Action

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



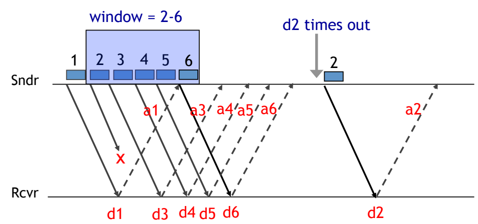
Handling Packet Loss



Sender advances the window on arrivals of in-sequence acks

→ Can't advance on a3's arrival

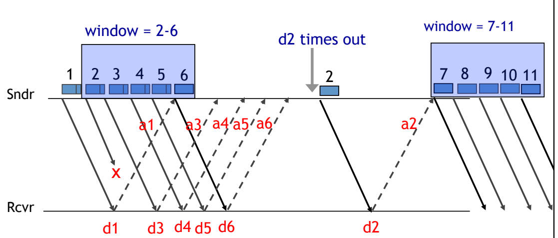
Handling Packet Loss



Sender advances the window on arrivals of in-sequence acks

→ Can't advance on a3's arrival

Handling Packet Loss

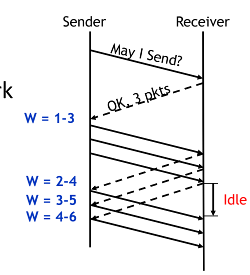


Sender advances the window on arrivals of in-sequence acks

→ 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



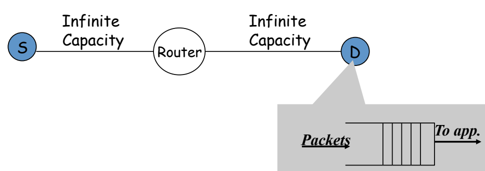
Case study: TCP

- TCP: reliable pipe to send bytes
- Uses acknowledgements to adopt to:
  - link capacity
  - rate at which server processes
  - congestion in the network
  - lost packets
- Explicit setup and tear-down

E2E Transport

- Reliability: "At Least Once Delivery"
  - Lock-step
  - Sliding Window
- Congestion Control
  - Flow Control
  - Additive Increase Multiplicative Decrease

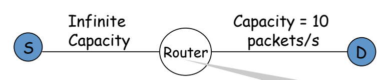
Setting Window Size: **Flow Control**

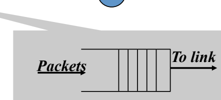


Window ≤ Receiver Buffer

- Otherwise receiver drops packets

Setting Window Size: **Congestion**

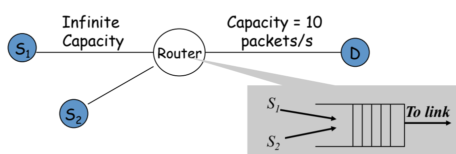




- Sender transmits faster than bottleneck capacity
  - → Queue builds up
  - → Router drops packets
- Tx Rate ≤ Bottleneck Capacity
- Tx Rate = Window / RTT

Window ≤ min(Receiver Buffer, Bottleneck_Cap * RTT)

Setting Window Size: **Congestion**



Bottleneck may be shared

Window ≤ min(Receiver Buffer, cwnd)

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

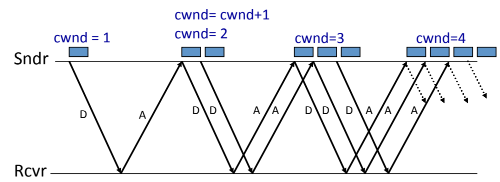
Congestion Control

- Basic Idea:
  - Increase cwnd slowly; if no drops → no congestion yet
  - If a drop occurs → 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

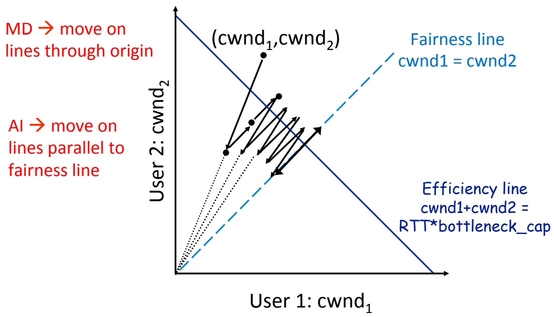
- Every RTT:
  - No drop: cwnd = cwnd + 1
  - A drop: cwnd = cwnd/2

Additive Increase



AIMD Leads to Efficiency and Fairness

Consider two users who have the same RTT



Summary of E2E Transport

- Reliability Using Sliding Window
  - Tx Rate = W/ RTT
- Congestion Control
  - W = min(Receiver_buffer, 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