CSE/EE 461 Lecture 10
Reliable Transport

Tom Anderson
tom@cs.washington.edu
Peterson, Chapter 2.5, 5.2

IP vs. TCP

- IP: routers can be arbitrarily bad
  - packets can be lost
  - packets can be reordered
  - packets can be duplicated
  - packets have limited size & can be fragmented
- TCP: applications need something better
  - reliable delivery
  - messages arrive in order
  - only one copy of each message is received
  - supports arbitrarily long messages
  - match speed of sender to speed of reciever
  - process to process communication

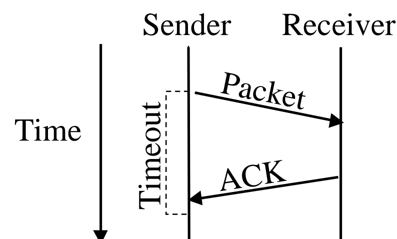
Reliable Transmission

How do we send packets reliably?

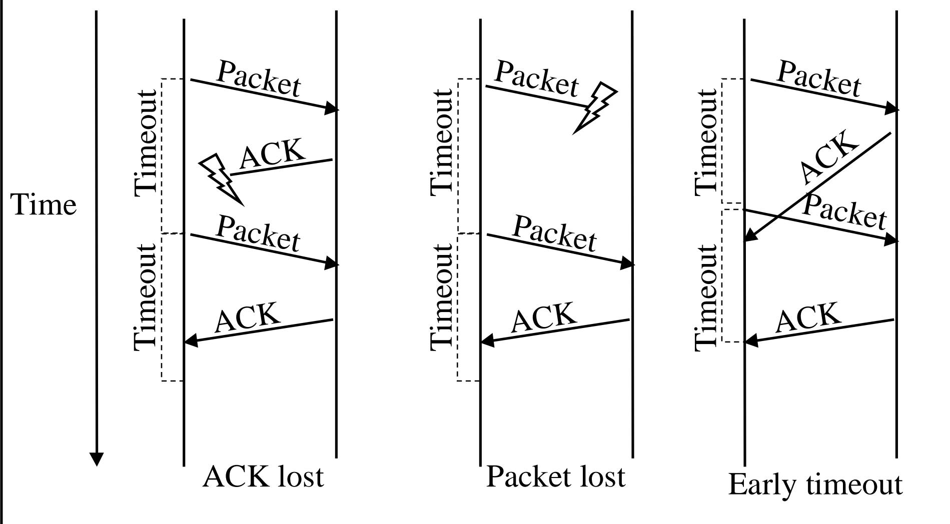
- Two mechanisms
  - Acknowledgements
  - Timeouts
- Simplest reliable protocol: Stop and Wait

Stop and Wait

- Send a packet, wait until ack arrives
  - retransmit if no ack within timeout
- Receiver acks each packet as it arrives

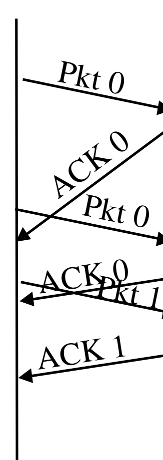


Recovering from error



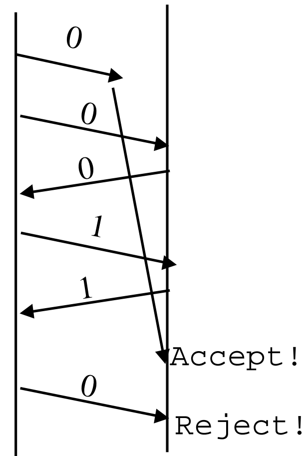
How can we recognize resends?

- Use unique ID for each pkt
  - for both packets and acks
- How many bits for the ID?
  - For stop and wait, a single bit!
  - assuming in-order delivery…



What if packets can be delayed?

- Solutions?
  - Never reuse a unique ID?
  - Change IP layer to eliminate packet reordering?
  - Prevent very late delivery?
    - IP routers keep hop count per pkt, discard if exceeded
    - ID's not reused within delay bound

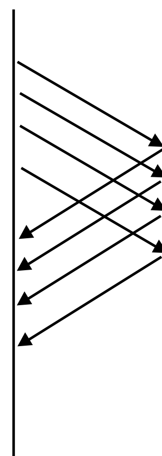


What happens on reboot?

- How do we distinguish packets sent before and after reboot?
  - Can't remember last sequence # used unless written to stable storage (disk or NVRAM)
- Solutions?
  - Restart sequence # at 0?
  - Assume boot takes max packet delay?
  - Store epoch number -- increment high order bits of sequence # on every boot?

How do we keep the pipe full?

- Unless the bandwidth*delay product is small, stop and wait can't fill pipe
- Solution: Send multiple packets without waiting for first to be acked
- Reliable, unordered delivery:
  - Send new packet after each ack
  - Sender keeps list of unack'ed packets; resends after timeout
  - Receiver same as stop&wait
- What if pkt 2 keeps being lost?



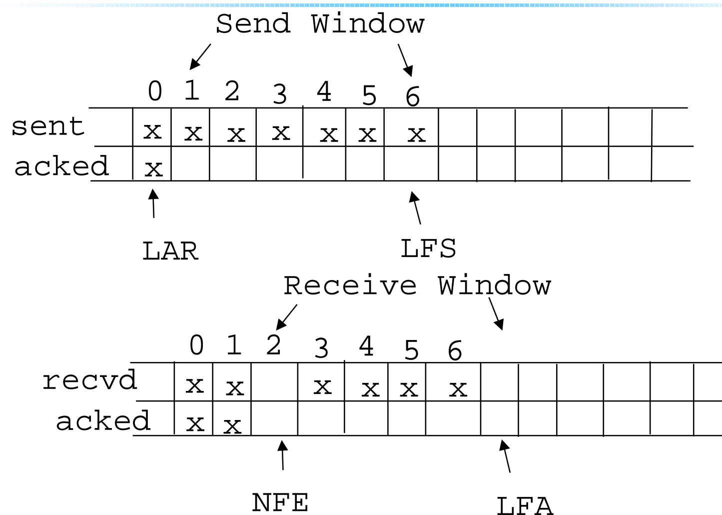
Sliding Window: Reliable, ordered delivery

- Two constraints:
  - Receiver can't deliver packet to application until all prior packets have arrived
  - Sender must prevent buffer overflow at receiver
- Solution: sliding window
  - circular buffer at sender and receiver
    - packets in transit <= buffer size
    - advance when sender and receiver agree packets at beginning have been received
  - How big should the window be?
    - bandwidth * round trip delay

Sender/Receiver State

- sender
  - packets sent and acked (LAR = last ack recvd)
  - packets sent but not yet acked
  - packets not yet sent (LFS = last frame sent)
- receiver
  - packets received and acked (NFE = next frame expected)
  - packets received out of order
  - packets not yet received (LFA = last frame ok)

Sliding Window



What if we lose a packet?

- Go back N
  - receiver acks “got up through k”
  - ok for receiver to buffer out of order packets
  - on timeout, sender restarts from k+1
- Selective retransmission
  - receiver sends ack for each pkt in window
  - on timeout, resend only missing packet

Sender Algorithm

- Send full window, set timeout
- On ack:
  - if it increases LAR (packets sent & acked)
  - send next packet(s)
- On timeout:
  - resend LAR+1

Receiver Algorithm

- On packet arrival:
  - if packet is the NFE (next frame expected)
  - send ack
  - increase NFE
  - hand packet(s) to application
  - else
  - send ack
  - discard if < NFE

Can we shortcut timeout?

- If packets usually arrive in order, out of order signals drop
  - Negative ack
    - receiver requests missing packet
  - Fast retransmit
    - sender detects missing ack

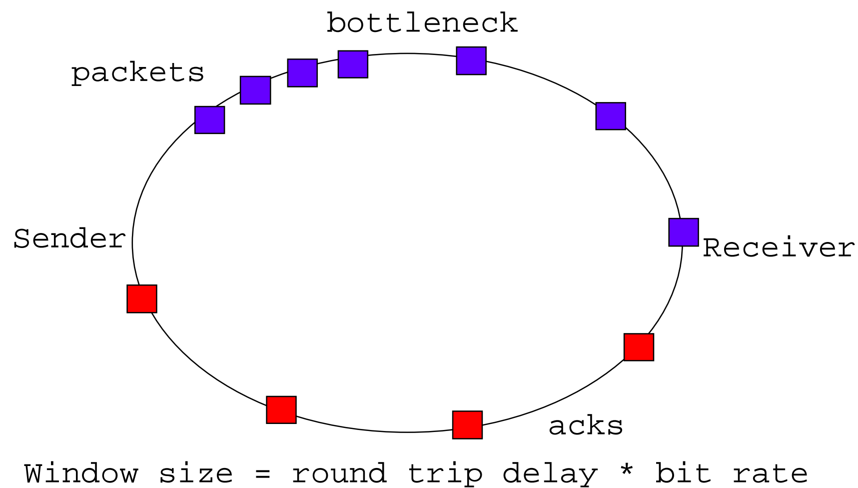
What does TCP do?

- Go back N + fast retransmit
  - receiver acks with NFE-1
  - if sender gets acks that don't advance NFE, resends missing packet
    - stop and wait for ack for missing packet?
    - Resend entire window?
- Proposal to add selective acks

What if link is very lossy?

- Wireless packet loss rates can be 10-30%
  - end to end retransmission will still work, even with go back N
  - will be inefficient
- Solution: hop by hop retransmission
  - performance optimization, not for correctness
- End to end principle
  - ok to do optimizations at lower layer
  - still need end to end retransmission

Avoiding burstiness: ack pacing



Window size = round trip delay * bit rate

How many sequence #'s?

- Window size + 1?
  - Suppose window size = 3
  - Sequence space: 0 1 2 3 0 1 2 3
  - send 0 1 2, all arrive
    - if acks are lost, resend 0 1 2
    - if acks arrive, send new 3 0 1
- Window <= (max seq # + 1) / 2

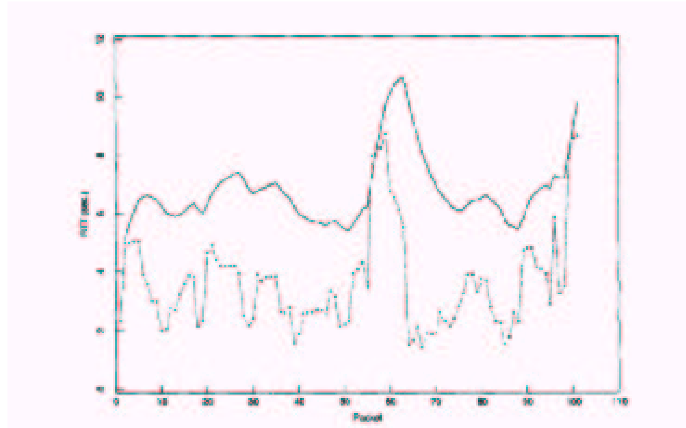
How do we determine timeouts?

- If timeout too small, useless retransmits
  - causes extra traffic, increasing congestion, increasing packet losses, …
- If timeout too big, inefficient
- Timeout should be based on round trip time (RTT)
  - varies with destination subnet, routing changes, congestion, …

Estimating RTTs

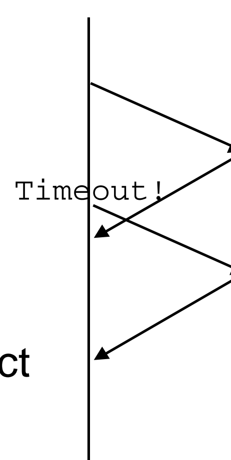
- Idea: Adapt based on recent past measurements
  - For each packet, note time sent and time ack received
  - Compute RTT samples and average recent samples for timeout
  - EstimatedRTT = α x EstimatedRTT + (1 - α) x SampleRTT
  - This is an exponentially-weighted moving average (low pass filter) that smoothes the samples. Typically, α = 0.8 to 0.9.
  - Set timeout to small multiple (2) of the estimate

Estimated Retransmit Timer



Retransmission ambiguity

- How do we distinguish first ack from retransmitted ack?
  - First send to first ack?
    - What if ack dropped?
  - Last send to last ack?
    - What if last ack dropped?
- Might never be able to correct too short timeout!



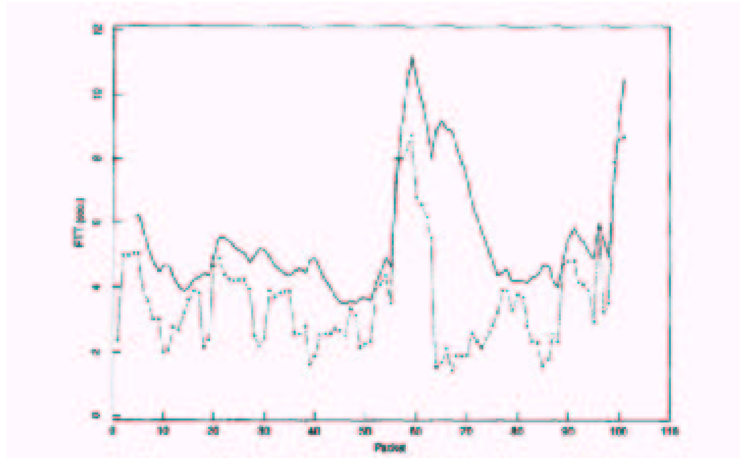
Retransmission ambiguity: Solutions?

- TCP: Karn-Partridge
  - ignore RTT estimates for retransmitted pkts
  - double timeout on every retransmission
- Add sequence #'s to retransmissions (retry #1, retry #2, …)
- TCP proposal: Add timestamp into packet header; ack returns timestamp

Jacobson/Karels Algorithm

- Problem:
  - Variance in RTTs gets large as network gets loaded
  - Average RTT isn't a good predictor when we need it most
- Solution: Track variance too.
  - Difference = SampleRTT – EstimatedRTT
  - EstimatedRTT = EstimatedRTT + (δ x Difference)
  - Deviation = Deviation + δ(|Difference|- Deviation)
  - Timeout = μ x EstimatedRTT + ϕ x Deviation
  - In practice, δ = 1/8, μ = 1 and ϕ = 4

Estimate with Mean + Variance



Summary

- Transport layer (TCP) implements reliable, in order packet delivery
  - retransmissions, sliding window, RTT est.
- Why not retransmit at link layer?
  - wireless: 802.11 implements ARQ
  - Ethernet, other link protocols implement CRC's
- End to end principle: put functionality at lowest layer it can be completely and correctly implemented
  - lower layers can implement as optimization