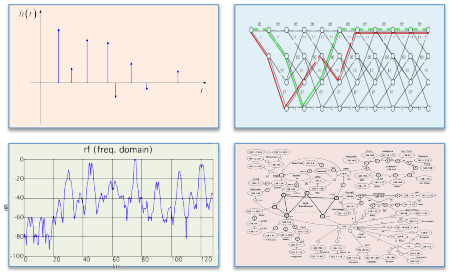


INTRODUCTION TO EECS II

DIGITAL COMMUNICATION SYSTEMS

6.02 Spring 2011 Lecture #21

HKN Eval!

- Redundancy via careful retransmission
- Sequence numbers & acks
- RTT estimation and timeouts
- Stop-and-wait protocol

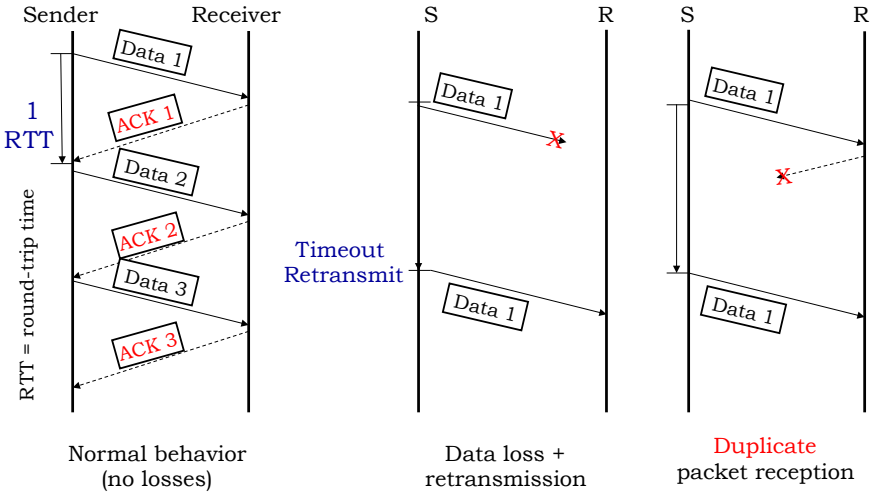
The Problem

- Given: Best-effort network in which
  - Packets may be lost arbitrarily
  - Packets may be reordered arbitrarily
  - Packet delays are variable (queueing)
  - Packets may even be duplicated
- Sender S and receiver R want to communicate reliably
  - Application at R wants *all* data bytes in exactly the same order that S sent them
  - Each byte must be delivered exactly once
- These functions are provided by a *reliable transport protocol*
  - Application "layered above" transport protocol

Proposed Plan

- Transmitter
  - Each packet includes a sequentially increasing sequence number
  - When transmitting, save (xmit time,packet) on un-ACKed list
  - When acknowledgement (ACK) is received from the destination for a particular sequence number, remove the corresponding entry from un-ACKed list
  - Periodically check un-ACKed list for packets sent awhile ago
    - Retransmit, update xmit time in case we have to do it again!
    - "awhile ago": xmit time < now – timeout
- Receiver
  - Send ACK for each received packet, reference sequence number
  - Deliver packet payload to application

Stop and Wait Protocol



Wanted "exactly once", got "at least once"

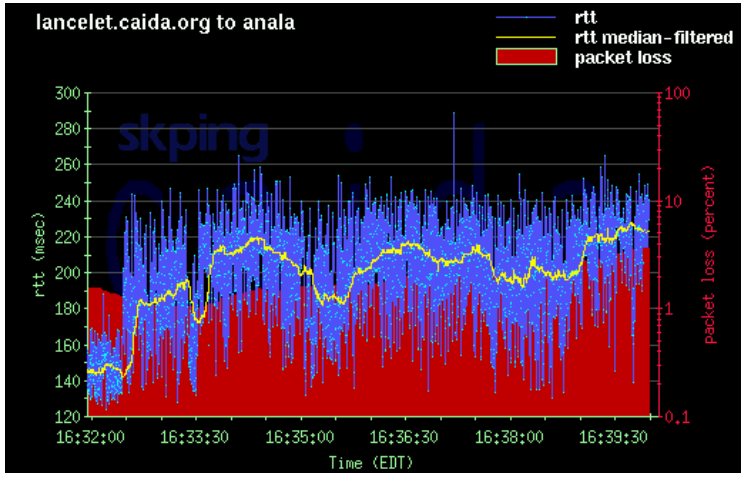
Revised Plan

- Transmitter
  - Each packet includes a sequentially increasing sequence number
  - When transmitting, save (xmit time,packet) on un-ACKed list
  - When acknowledgement (ACK) is received from the destination for a particular sequence number, remove the corresponding entry from un-ACKed list
  - Periodically check un-ACKed list for packets sent awhile ago
    - Retransmit, update xmit time in case we have to do it again!
    - "awhile ago": xmit time < now − timeout
- Receiver
  - Send ACK for each received packet, reference sequence number
  - Deliver packet payload to application in sequence number order
    - By keeping track of next sequence number to be delivered to app, it's easy to recognize duplicate packets and not deliver them a second time.

Issues

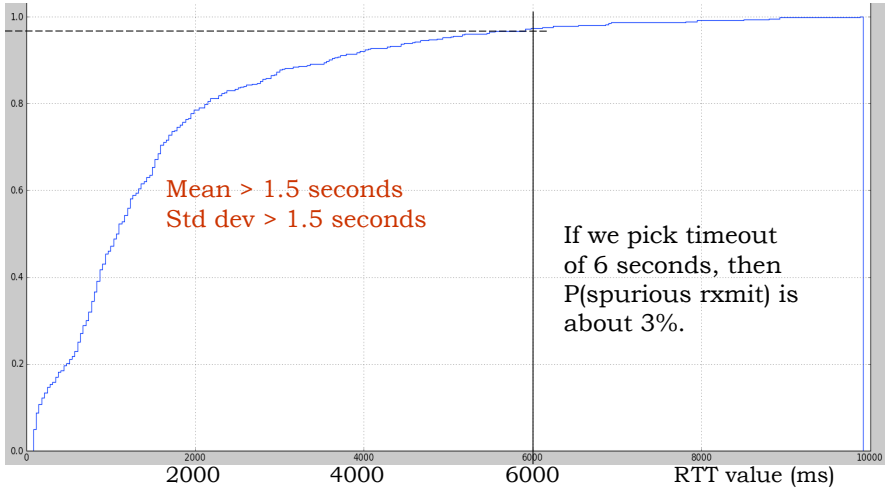
- Protocol must handle lost packets correctly
  - Lost data: retransmission will provide missing data
  - Lost ACK: retransmission will trigger another ACK from receiver
- Size of packet buffers
  - At transmitter
    - Buffer holds un-ACKed packets
    - Stop transmitting if buffer space an issue
  - At receiver
    - Buffer holds packets received out-of-order
    - Stop ACKing if buffer space an issue
- Choosing timeout value: related to RTT
  - Too small: unnecessary retransmissions
  - Too large: poor throughput
    - Delivery stalled while waiting for missing packets

RTT Measurements

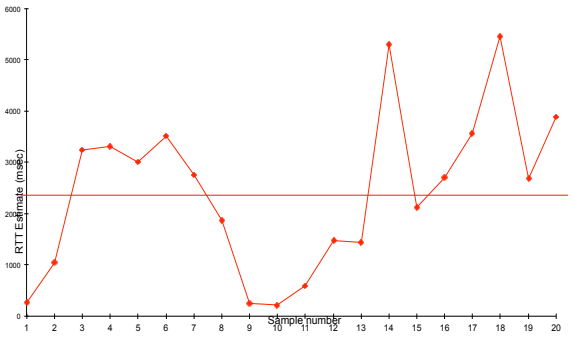


CDF of RTT over Verizon Wireless 3G Network

Cumulative probability (CDF)



RTT Can Be Highly Variable



Example from a TCP connection over a wide-area wireless link
Mean RTT = 2.4 seconds; Std deviation = 1.5 seconds!

Estimating RTT from Data

- Gather samples of RTT by comparing time when ACK arrives with time corresponding packet was transmitted
  - Sample of random variable with some unknown distribution (not Gaussian!)

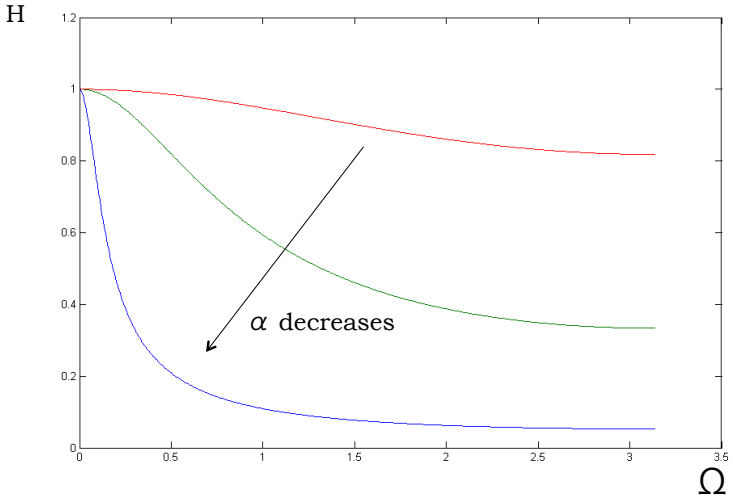
- Chebyshev's Inequatility tells us that for a random variable X with mean μ and finite variance σ^2:

$$prob(|X-\mu| \geq k\sigma) \leq \frac{1}{k^2}$$

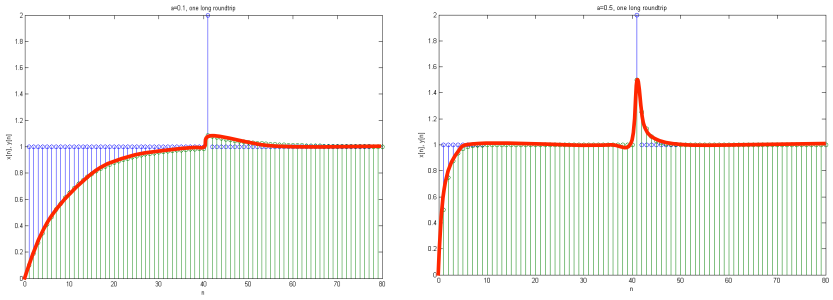
  - To minimize the chance of unnecessary retransmissions – packet wasn't lost, just the round trip time for packet/ACK was long – we want our timeout to be greater than most observed RTTs.
  - So choose a k that makes the chances small…
  - We need an estimate for μ and σ

Exponential Weighted Moving Average (EWMA) LPF Frequency Response

srtt ← α*rtt_sample + (1-α)*srtt



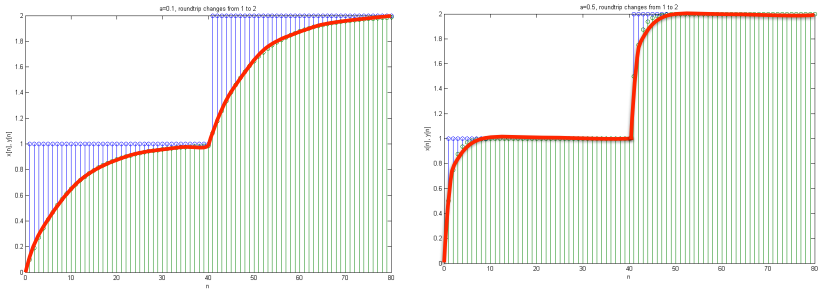
Response to One Long RTT Sample



α = 0.1 α = 0.5

Responds too quickly?

RTT changes from 1 to 2



$\alpha = 0.1$ $\alpha = 0.5$

Doesn't respond quickly enough?

Timeout Algorithm

- EWMA for smoothed RTT (srtt)
  - srtt ← α*rtt_sample + (1-α)*srtt
  - Typically 0.1 ≤ α ≤ 0.25 on networks prone to congestion. TCP uses α=0.125.
- Use another EWMA for smoothed RTT deviation (srttdev)
  - Mean linear deviation easy to compute (but could also do std deviation)
  - dev_sample = |rtt_sample – srtt|
  - srttdev ← β*dev_sample + (1-β)*srttdev,
- Retransmit Timeout
  - timeout = srtt + k·srttdev
  - k = 4 for TCP
  - Makes the "tail probability" of a spurious retransmission low

Throughput of Stop-and-Wait

- We want to calculate the time T between successful deliveries of packets. Throughput = 1/T.
- We can't just assume T = RTT since packets get lost
  - Suppose there are N links in the round trip between sender and receiver
  - If the per-link probability of losing a packet is p, then the probability it's delivered over the link is (1-p), and thus the probability it's delivered over N links is $(1-p)^N$.
  - So the probability a packet/ACK gets lost is L = 1 – $(1-p)^N$.
- Now we can write an equation for T:

$$T = (1-L)\cdot RTT + L\cdot(timeout + T)$$

$$= RTT + \frac{L}{1-L}timeout$$

Bottom Line

- Suppose RTT is the same for every packet, so timeout = RTT

$$T = RTT + \frac{L}{1-L}RTT = \frac{1}{1-L}RTT$$

$$\text{Throughput} = \frac{1-L}{RTT} = \frac{(1-p)^N}{RTT}$$

- If we can transmit 100 packets/sec and the RTT is 100 ms, then, using stop-and-wait, the maximum throughput is 10 packets/sec.
  - Urk! Only 10% of the capacity of the channel.
  - We need a better reliable transmission protocol… next time: sliding window protocol