

INTRODUCTION TO EECS II
**DIGITAL
COMMUNICATION
SYSTEMS**

**6.02 Spring 2012
Lecture #22: Reliable Data Transport**

- Redundancy via careful retransmission
- Sequence numbers & acks
- Timeouts and round-trip time (RTT) estimation
- Two protocols: stop-and-wait & sliding window

6.02 Spring 2012 Lecture 22, Slide #1

The Problem

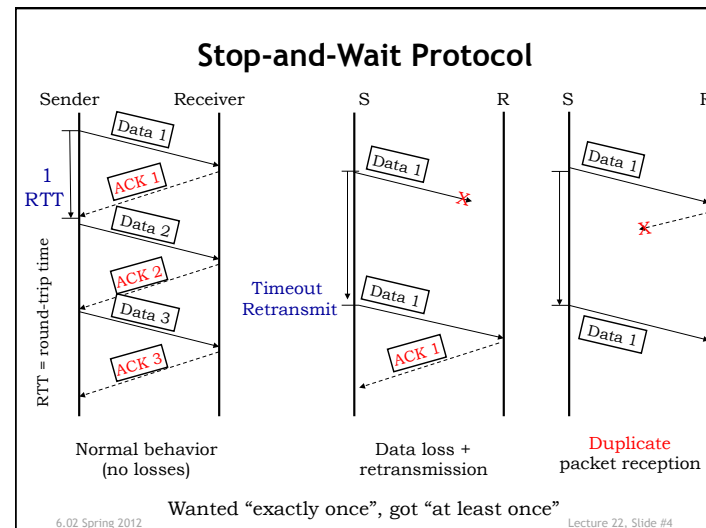
- Given: Best-effort network in which
 - Packets may be lost arbitrarily
 - Packets may be reordered arbitrarily
 - Packet delays are variable (queueing)
 - Packets may even be duplicated
- Sender S and receiver R want to communicate reliably
 - Application at R wants *all* data bytes in exactly the 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

6.02 Spring 2012 Lecture 22, Slide #2

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

6.02 Spring 2012 Lecture 22, Slide #3



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.

6.02 Spring 2012

Lecture 22, Slide #5

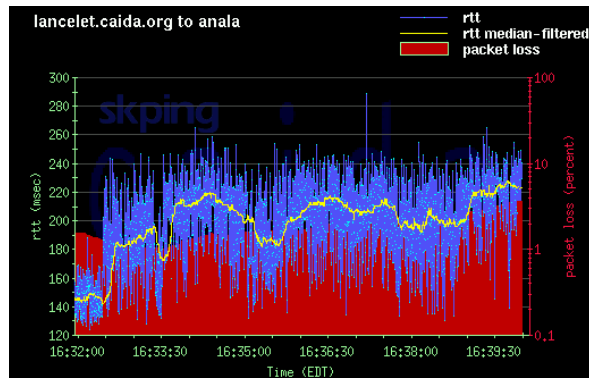
Issues

- 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

6.02 Spring 2012

Lecture 22, Slide #6

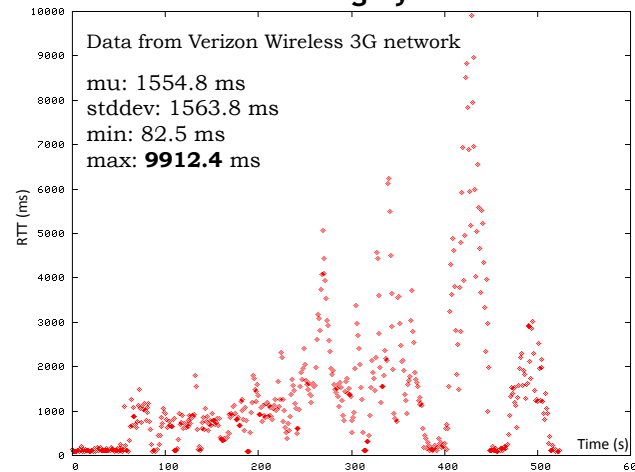
RTT Measurements



6.02 Spring 2012

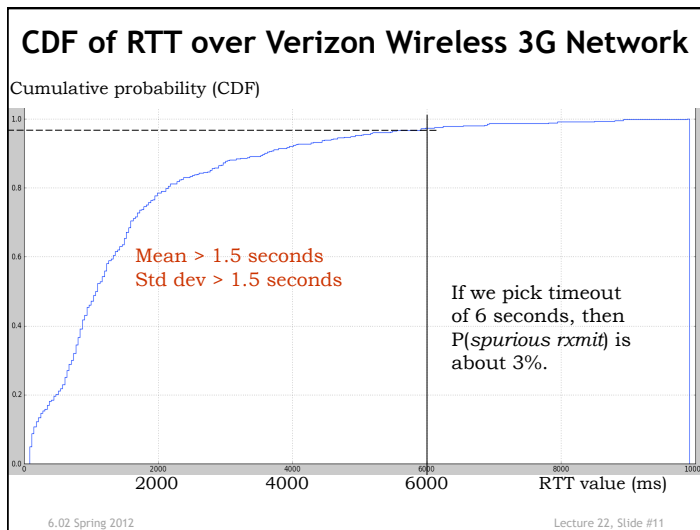
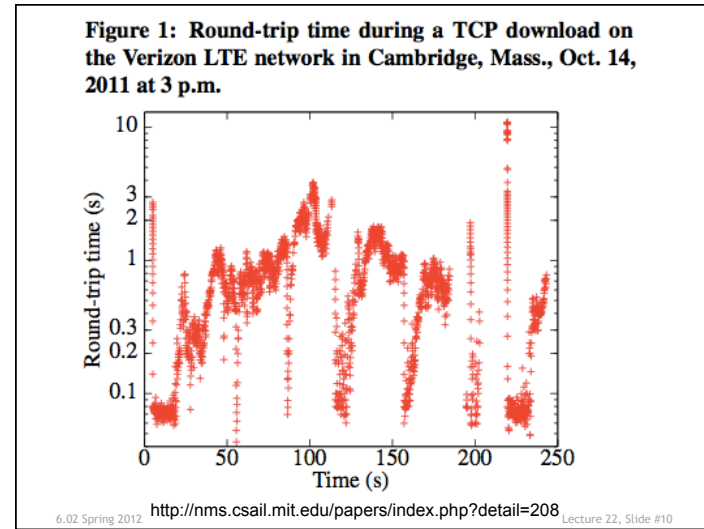
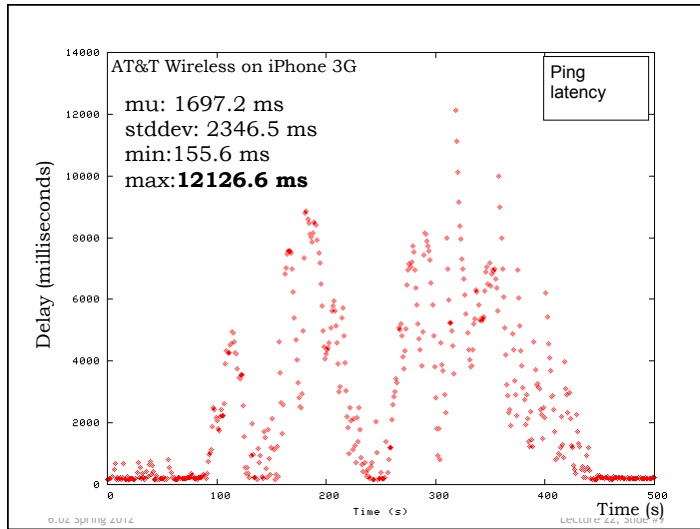
Lecture 22, Slide #7

RTTs can be highly variable



6.02 Spring 2012

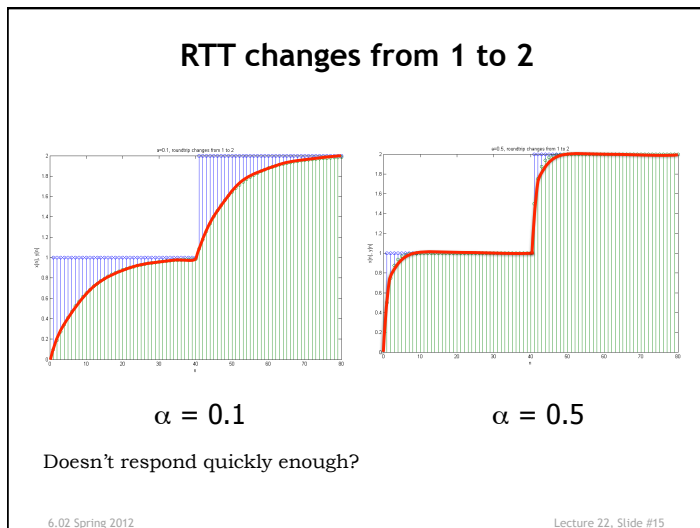
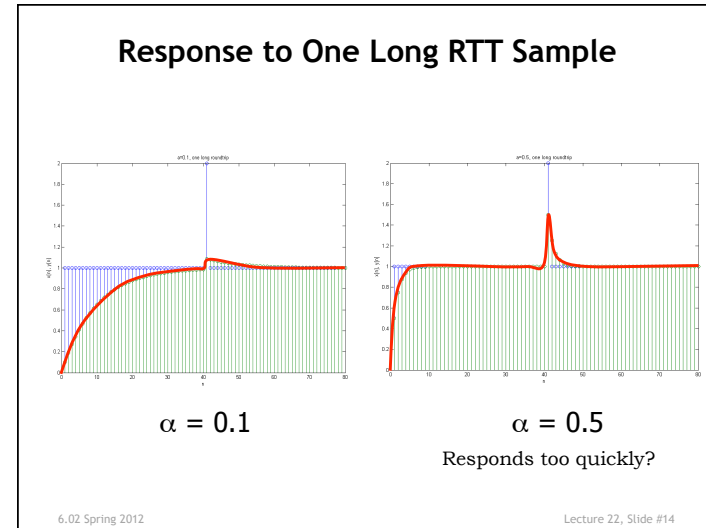
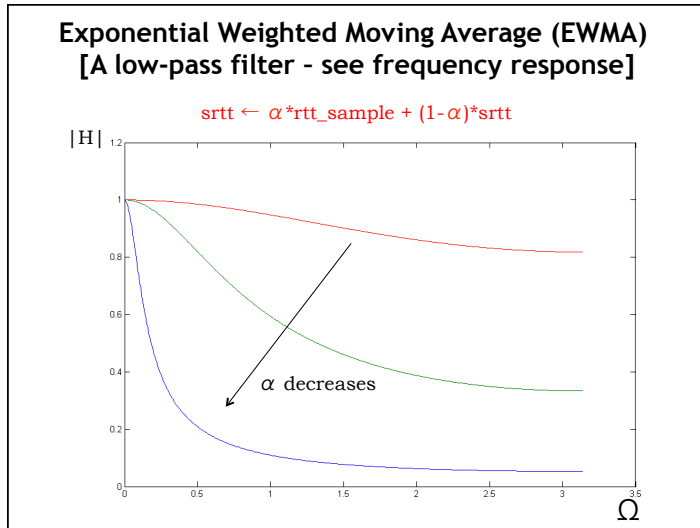
Lecture 22, Slide #8



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 necessarily Gaussian!)
- Chebyshev's inequality tells us that for a random variable X with mean μ and finite variance σ^2 :

$$P(|X - \mu| \geq k\sigma) \leq \frac{1}{k^2}$$
 - To reduce the chance of a *spurious* (i.e., unnecessary) retransmission – 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



- ### Timeout Algorithm
- EWMA for smoothed RTT (srtt)
 - $srtt \leftarrow \alpha * rtt_sample + (1-\alpha) * srtt$
 - Typically $0.1 \leq \alpha \leq 0.25$ on networks prone to congestion. TCP uses $\alpha = 0.125$.
 - Use another EWMA for smoothed RTT deviation (srttdev)
 - Mean linear deviation easy to compute (but could also do std deviation)
 - $dev_sample = |rtt_sample - srtt|$
 - $srttdev \leftarrow \beta * dev_sample + (1-\beta) * srttdev$
 - TCP uses $\beta = 0.25$
 - Retransmit Timeout, RTO
 - $RTO = srtt + k * srttdev$
 - $k = 4$ for TCP
 - Makes the "tail probability" of a spurious retransmission low
 - On successive *retransmission* failures, double RTO (exponential backoff)
- 6.02 Spring 2012 Lecture 22, Slide #16

Throughput of Stop-and-Wait

- We want to calculate the expected time, T between successful deliveries of packets. Throughput = 1/T.
- We can't just assume T = RTT because 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 in terms of RTT and the timeout, RTO:

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

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

6.02 Spring 2012

Lecture 22, Slide #17

The Best Case

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

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

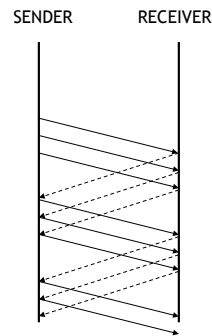
$$\text{Throughput} = \frac{(1-L)}{RTT}$$

- If bottleneck link can support 100 packets/sec and the RTT is 100 ms, then, using stop-and-wait, the maximum throughput is *at most only* 10 packets/sec.
 - Urk! Only 10% utilization
 - We need a better reliable transport protocol...

6.02 Spring 2012

Lecture 22, Slide #18

Idea: Sliding Window Protocol

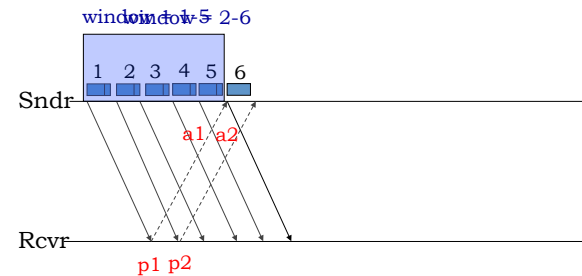


- Use a *window*
 - Allow W packets outstanding (i.e., unack'd) in the network at once (W is called the window size).
 - Overlap transmissions with ACKs
- Sender advances the window by 1 for each in-sequence ack it receives
 - I.e., window *slides*
 - So, idle period reduces
 - **Pipelining**
- Assume that the window size, W, is fixed and known
 - Later, we will discuss how one might set it
 - W = 3 in the example on the left

6.02 Spring 2012

Lecture 22, Slide #19

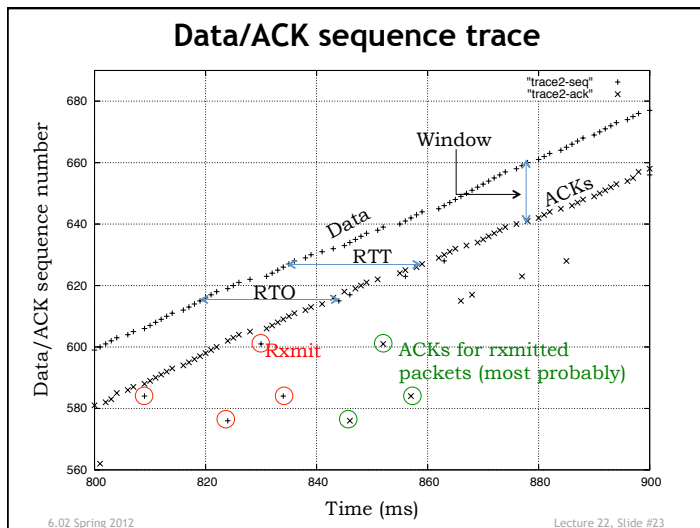
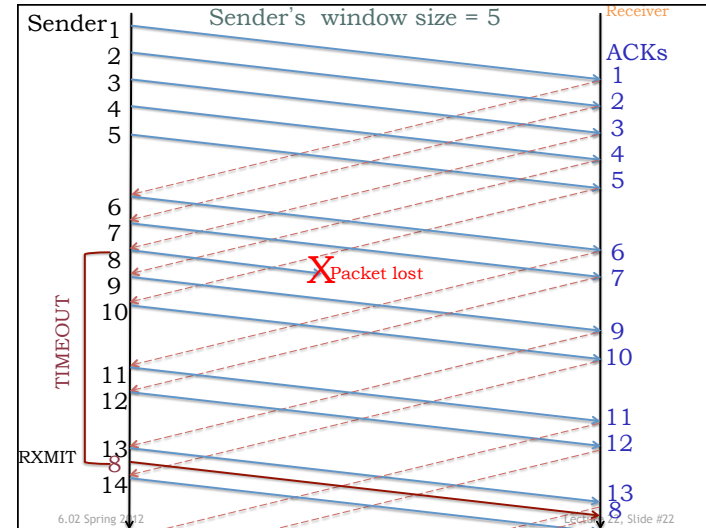
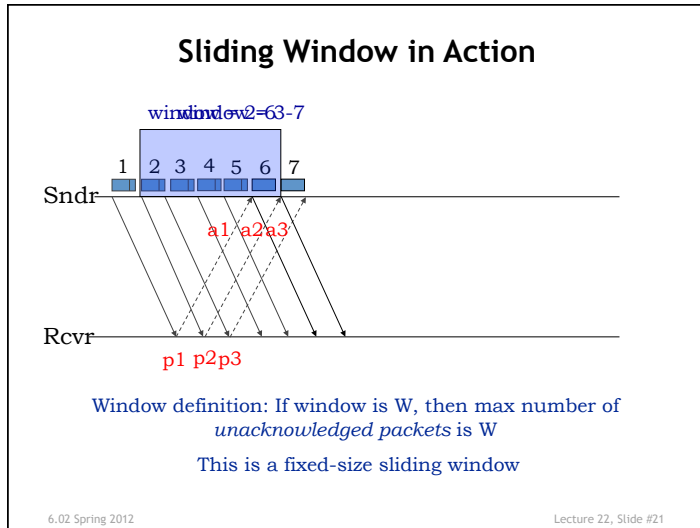
Sliding Window in Action



W = 5 in this example

6.02 Spring 2012

Lecture 22, Slide #20



- ### Sliding Window Implementation
- Transmitter
 - Each packet includes a sequentially increasing sequence number
 - When transmitting, save (xmit time, packet) on un-ACKed list
 - Transmit packets if $\text{len}(\text{un-ACKed list}) \leq \text{window size } W$
 - 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": $\text{xmit time} < \text{now} - \text{timeout}$
 - Receiver
 - Send ACK for each received packet, reference sequence number
 - Deliver packet payload to application in sequence number order
 - Save delivered packets in sequence number order in local buffer (remove duplicates). Discard incoming packets which have already been delivered (caused by retransmission due to lost ACK).
 - Keep track of next packet application expects. After each reception, deliver as many in-order packets as possible.
- 6.02 Spring 2012 Lecture 22, Slide #24