

TCP Congestion Control and Network Services

Amin Vahdat
CSE 123b
April 25, 2006

Annoucements

- Midterm: May 9
- Second assignment available shortly

TCP Congestion Control

Advertised Window

- TCP uses a 16-bit advertised window field (flow control)
 - Specifies number of bytes that can be sent from sender to receiver
 - Recall “keeping pipe full” to obtain available bandwidth
 - 16-bit field translates to max 64KB advertised window
 - For 100 ms RTT T3 link (45 Mbps), delay-bandwidth product is 549 KB

Advertised window not large enough to keep pipe full

Poor bandwidth utilization

- Proposal: advertised window specifies chunks larger than byte granularity

Adaptive Retransmission for Reliable Delivery

- TCP retransmits packet if ACK not received within timeout period
 - Necessary for reliability on top of “best-effort” IP
- Round trip time varies with congestion, route changes, ...
 - If timeout too small, useless retransmits
 - If timeout too big, low utilization
- TCP: estimate RTT by timing ACKs
 - Exponential weighted moving average
 - Factor in RTT variability

Retransmission

- How long a timeout to set?
- Original TCP: Estimate round-trip time R
 - $R = \alpha R + (1 - \alpha)M$
 - α is a smoothing factor of 0.9
 - Places much more weight on historical result
 - Smooth out outlying measurements
 - M is measured round-trip time (ACK's to data)
- Timeout at $R\beta$, where β is a delay variance factor of 2.0
 - Conservative: do not retransmit until two RTT's have passed
 - Jacobson's TCP modifications allows for varying β

Jacobson's RTT estimator

- Problem:
 - Original TCP does not adapt to wide variance in RTT
 - Uses fixed β of 2.0
- Need to account for both estimated RTT *and* variance
- Jacobson:
 - Low variance $\rightarrow$ estimated RTT sufficient
 - High variance $\rightarrow$ estimated RTT could be far off
- Solution:
 - $\text{Timeout} = \mu * \text{EstimateRTT} + \phi * \text{Deviation}$
 - $\mu=1, \phi=4$

TCP Congestion Problems

- Original TCP sent full window of data
 - Based entirely on advertised window (flow control)
 - When links become loaded, queues fill up, leading to:
 - *Congestion collapse*: when round-trip time exceeds retransmit interval this can create a stable condition in which every packet is being retransmitted many times
 - Synchronized behavior: network oscillates between loaded and unloaded
- Feedback loop

TCP Congestion Control

- Adjust transmission rate to match network bandwidth
 - Additive increase/multiplicative decrease
Oscillate around bottleneck capacity
 - Slow start
Quickly identify bottleneck capacity
 - Fast retransmit
Bypass timeout for small number of losses (use dup ACKs)
 - Fast recovery
Use ACKs already in the pipeline to avoid slowstart

Jacobson Solution

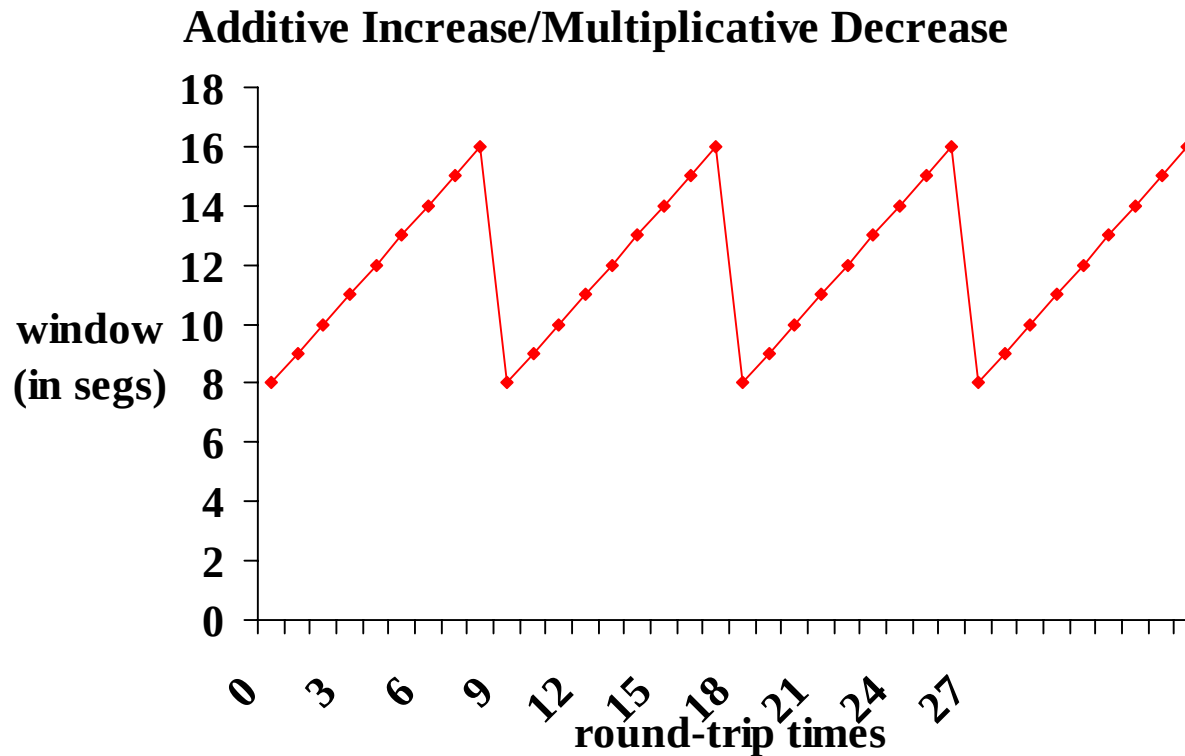
- Transport protocols should obey *conservation of packets*
 - Use ACKs to clock injection of new packets
- Modify retransmission timer to adapt to variations in delay
- Infer network bandwidth from packet loss
 - Drops → congestion → reduce rate
 - No drops → no congestion → increase rate
- Limit send rate based on minimum of *congestion window* and advertised window
 - Thus, interaction between flow control and congestion control

Tracking the Bottleneck Bandwidth

- Throughput = window size/RTT
- Multiplicative decrease
 - Timeout → dropped packet → cut window size in half
- Additive increase
 - ACK arrives → no drop → increase window size by one packet/window

TCP “Sawtooth”

- Oscillates around bottleneck bandwidth
 - Adjusts to changes in competing traffic

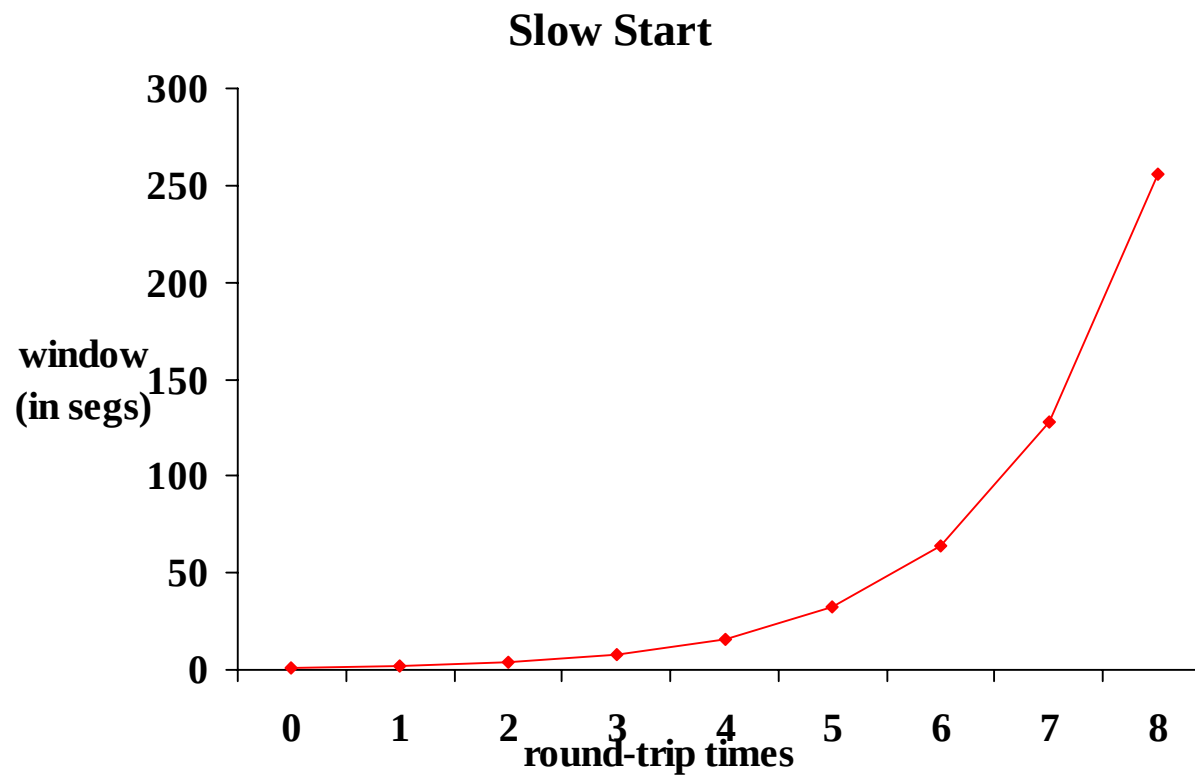


Slow Start

- How do we find bottleneck bandwidth?
- Cannot use ACKs to clock without reaching equilibrium
 - Start by sending a single packet
 - Start slow to avoid overwhelming network
 - Multiplicative increase until get packet loss
 - Quickly find bottleneck
 - Cut rate by half
 - Shift into linear increase/multiplicative decrease

Slow Start

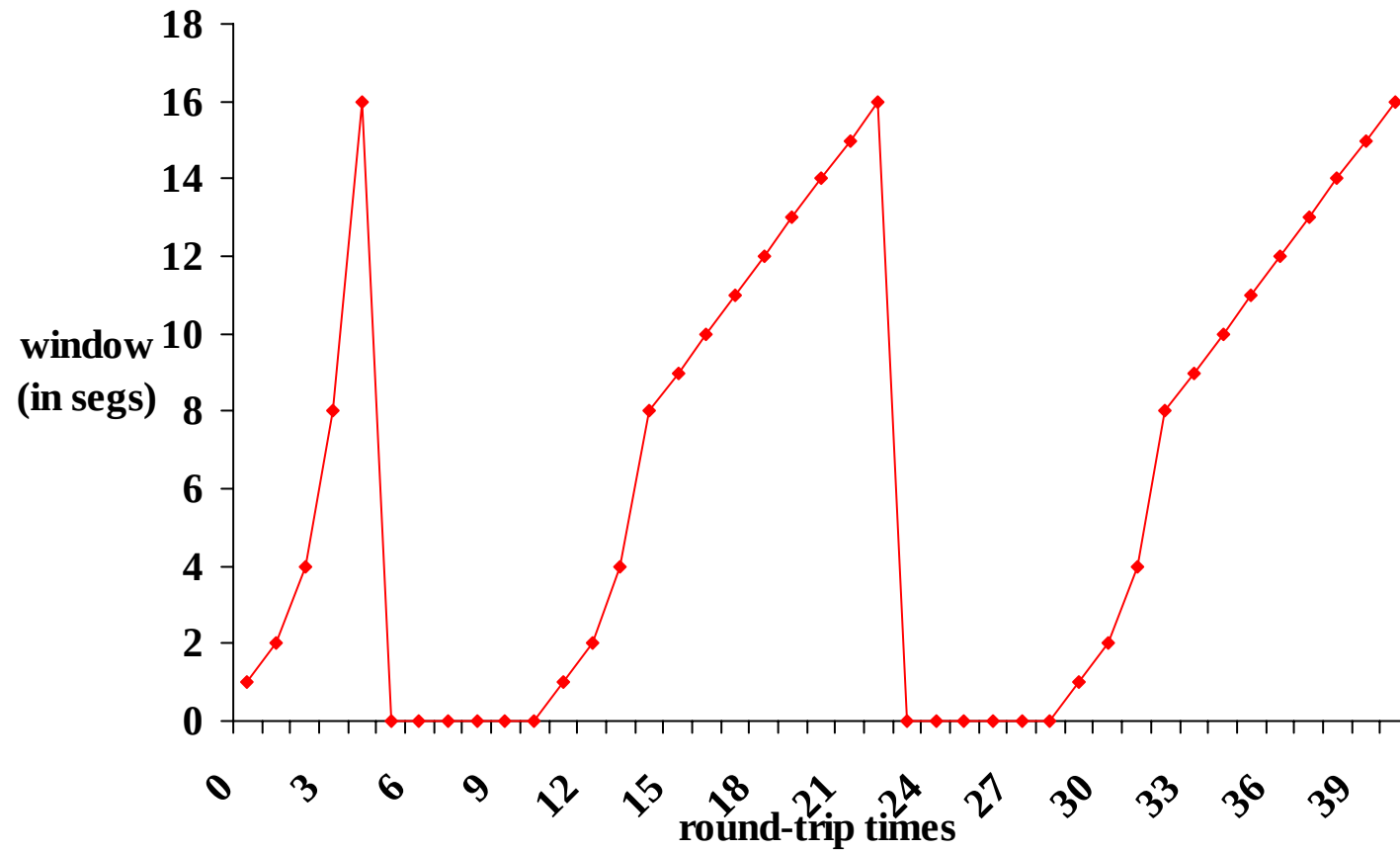
- Quickly find the bottleneck bandwidth



Slow Start Problems

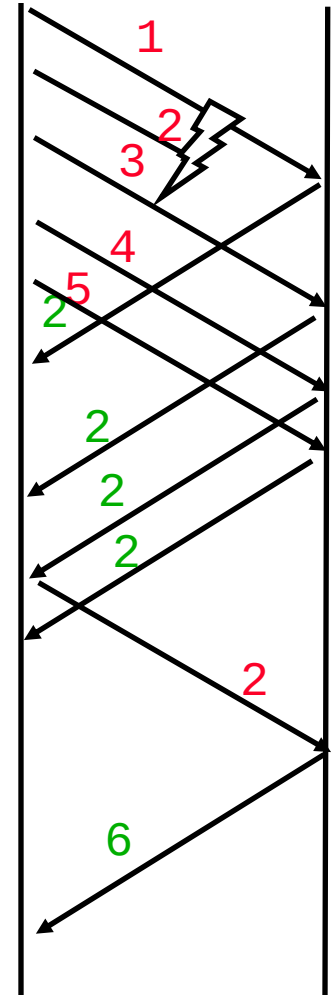
- Slow start usually overshoots bottleneck
 - Leading to many lost packets in window
 - Can lose up to half of window size
- Bursty traffic source
 - Will cause bursty losses for other flows
- Short flows
 - Can spend entire time in slow start
 - Especially for large bottleneck bandwidth
- Consider repeated connections to the same server
 - E.g., for web connections

Timeouts Dominate Performance



Fast Retransmit

- Can we detect packet loss without a timeout?
- Duplicate ACKs imply either
 - Packet reordering (route change)
 - Packet loss
- TCP Tahoe
 - Resend if see three dup ACKs
 - Eliminates timeout delay

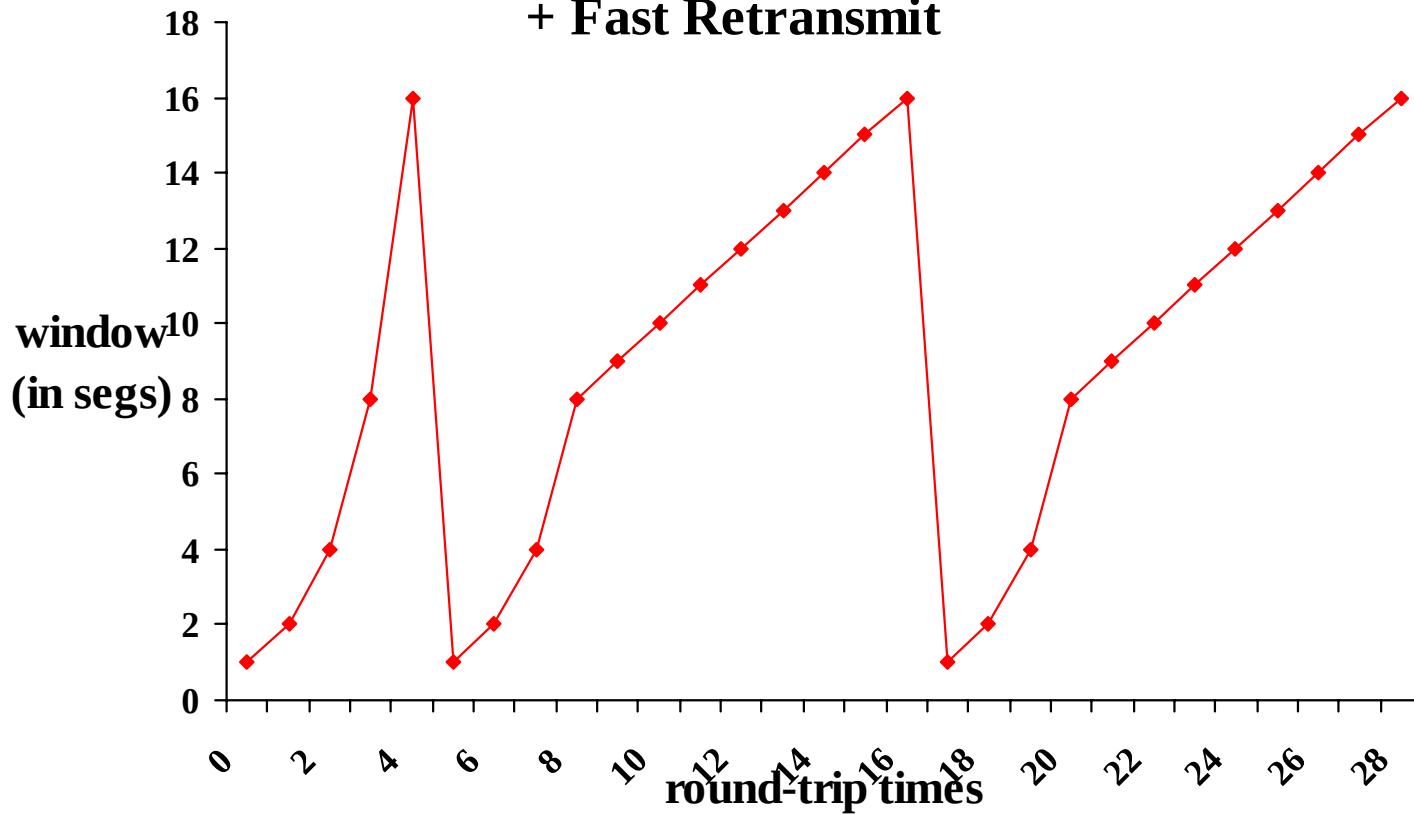


Fast Retransmit Caveats

- Requires in order packet delivery
 - Dynamically adjust number of dup ACKs needed for retransmit?
- Does not work with small windows
 - Why not?
 - E.g., modems
- Does not work if packets lost in burst
 - Why not?
 - E.g., at peak of slow start

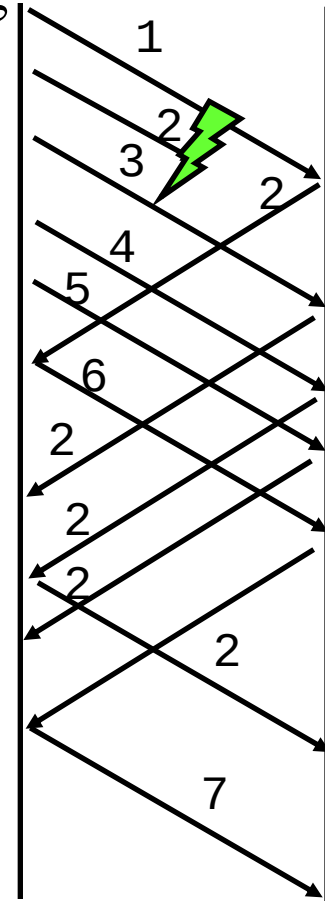
Fast Retransmit

**Slow Start + Congestion Avoidance
+ Fast Retransmit**



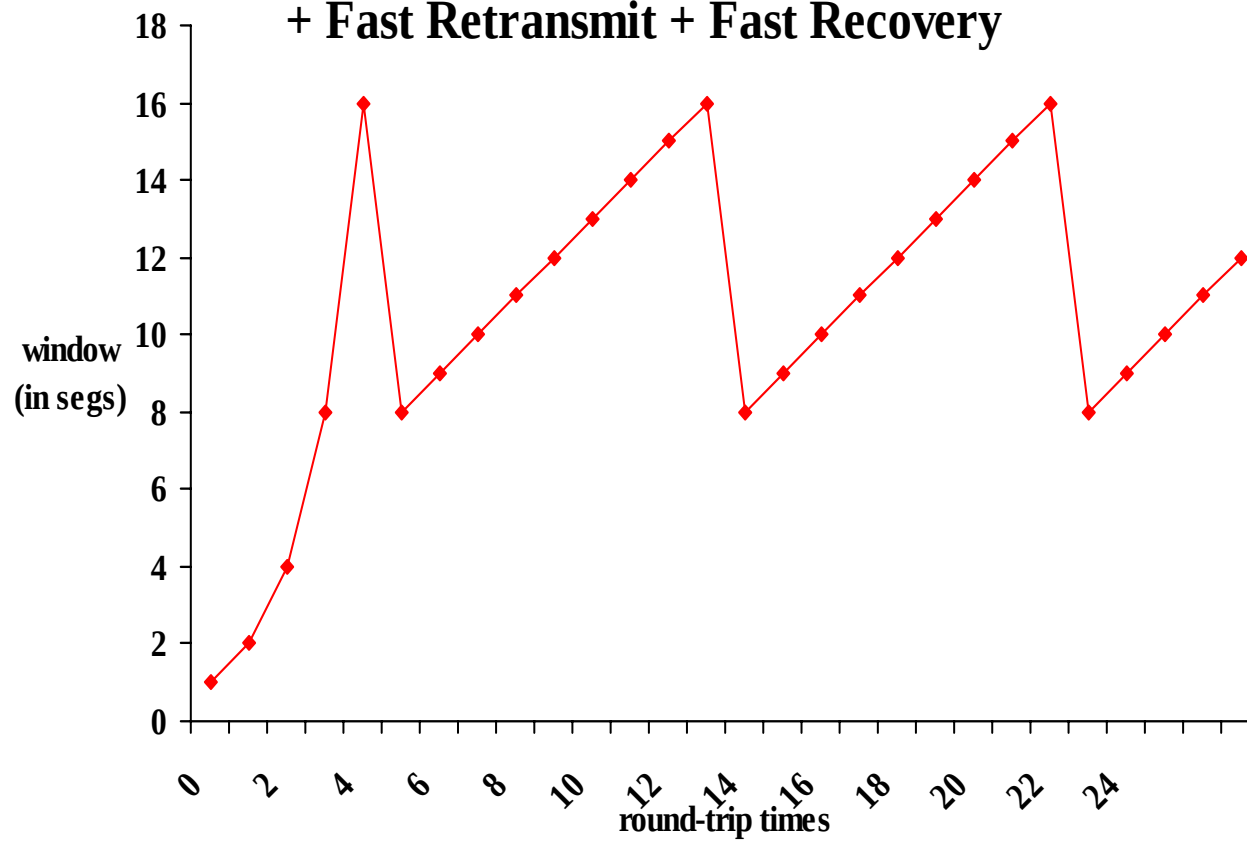
Fast Recovery

- Use duplicate ACKs to maintain ACK pacing
 - Dup ACK → packet left network
 - Every other ACK → send packet
- Fast recovery allows TCP to fall to half previous bottleneck bandwidth
 - Rather than all the way back to 1 packet/reinitiate slow start
 - Slow start only at beginning/on timeout



Fast Recovery

**Slow Start + Congestion Avoidance
+ Fast Retransmit + Fast Recovery**

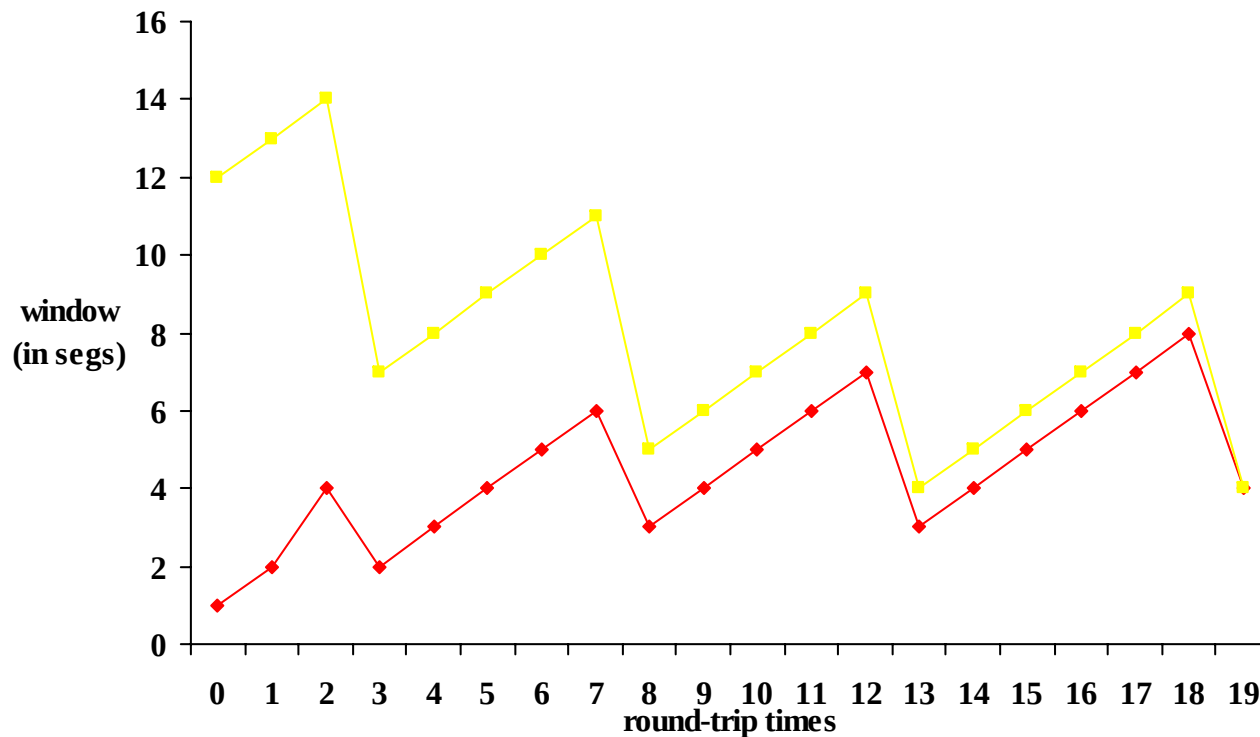


Delayed ACKs

- Problem:
 - In request/response programs, you send separate ACK and data packets for each transaction
- Goal: piggyback ACK with subsequent data packet
- Solution:
 - Do not ACK data immediately
 - Wait 200ms (must be less than 500ms)
 - Must ACK every other packet
 - Must not delay duplicate ACKs

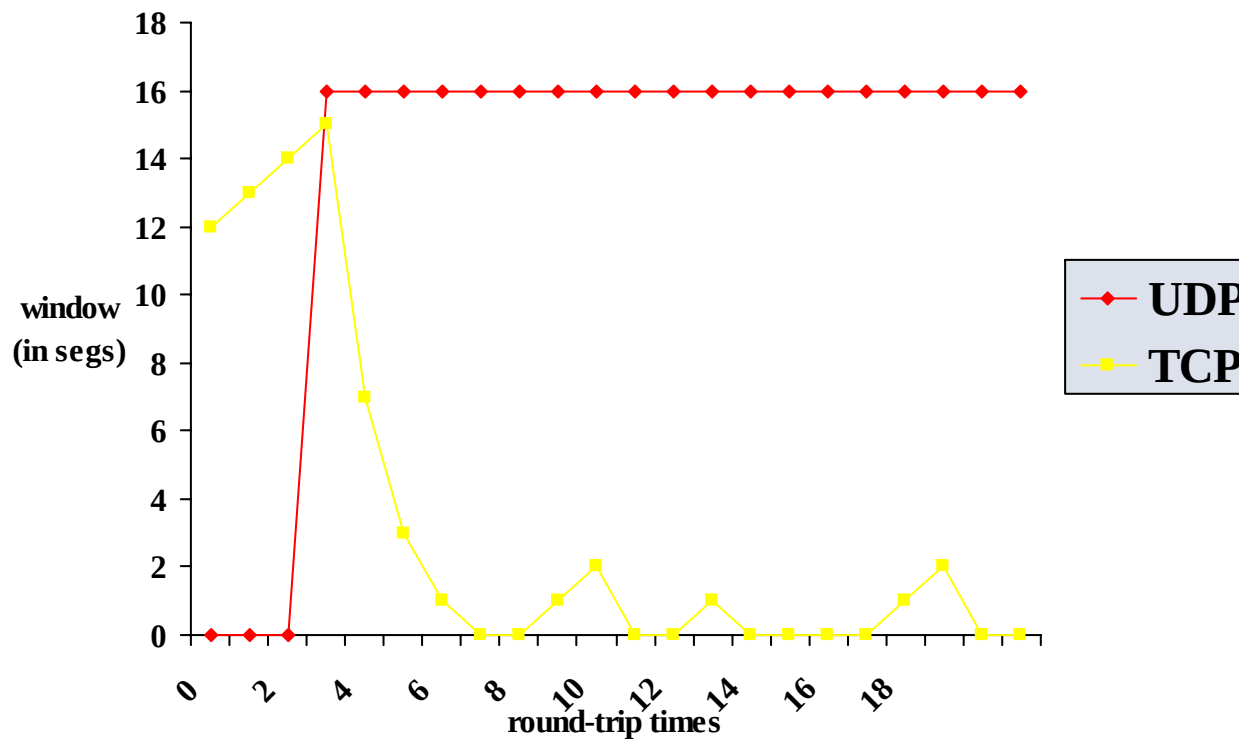
What if Two TCP Connections Share Link?

- Reach equilibrium independent of initial bandwidth (assuming equal RTTs)



What if TCP and UDP share link?

- Independent of initial rates, UDP will get priority! TCP will take what's left.



What if Two Different TCP Implementations Share Link?

- Problem: many different TCP implementations
- If cut back more slowly after drops → grab bigger share
- If add more quickly after ACKs → grab bigger share
- Incentive to cause congestion collapse
 - Many TCP “accelerators”
 - Easy to improve perf at expense of network
- Solutions?

What if Two Different TCP Implementations Share Link?

- Problem: many different TCP implementations
- If cut back more slowly after drops → grab bigger share
- If add more quickly after ACKs → grab bigger share
- Incentive to cause congestion collapse
 - Many TCP “accelerators”
 - Easy to improve perf at expense of network
- Solutions?
 - Per-flow fair queuing at router

TCP Congestion Control Summary

- Slow Start
- Adaptive retransmission
 - Account for average *and* variance
- Fast retransmission
 - Triple duplicate ACKs
- Fast restart
 - Use ACKs in pipeline to avoid shrinking congestion window to one

TCP Vegas

Vegas: New Retransmission Mechanism

- Reno uses coarse-grained timeouts and triple dup-ACKs
 - If bursty losses, or small window → no triple dup ACK
- Vegas reads system clock for every packet sent
 - On ACK arrival, Vegas calculates RTT on per-packet basis
- Vegas retransmits in two situations:
 - On duplicate ACK, check if elapsed time for “missing” packet exceeds RTT estimate
 - If so, retransmit without waiting for triple dup ACK
 - On first or second ACK after retransmission also check if any additional packets have exceeded RTT
- Why not just retransmit on single/double dup ACK?

Congestion Avoidance Mechanism

- Reno creates losses to determine available bandwidth
 - Each connection can create losses for other connections
 - No problem if advertised window < congestion window
- Use understanding of network behavior as it approaches congestion (not once it gets there)
 - Increased queue size → increased per-packet RTT
 - Decreased throughput → more congestion

TCP Vegas Congestion Avoidance

- Compare expected to actual throughput
 - Expected = window size / base RTT

How to measure base RTT?
 - Actual = ACKs / round trip time

Pick distinguished packet once every RTT for calculation
- If actual $\ll$ expected, queues increasing → decrease rate before packet drop
- If actual $\sim$ expected, queues decreasing → increase rate
- What if base RTT changes (route changes)? What if we have route flap?

TCP Vegas Congestion Avoidance

- Define two parameters α , β
- Let $\text{Diff} = \text{Expected} - \text{Actual}$
 - Always a positive value
- If $\text{Diff} < \alpha$, linearly increase congestion window
- If $\text{Diff} > \beta$, linearly decrease congestion window
- If $\alpha < \text{Diff} < \beta$, do nothing
- Why can we get away with linear decrease instead of multiplicative decrease?