Congestion

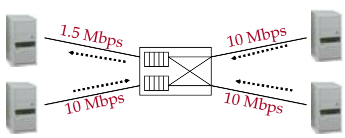
Adolfo Rodriguez
CPS 214
February 19, 2004

DUKE Computer Science

Congestion Control

DUKE Computer Science

Router Congestion



- What if rate of packets arriving > rate of packets departing

DUKE Computer Science

Congestion Control Overview

- Challenge: how do we efficiently share network resources among billions of hosts?
  - Today: TCP
    - Hosts adjust rate based on packet losses
  - Alternative solutions
    - Fair queuing, RED (router support)
    - Vegas, packet pair (add functionality to TCP)
    - Rate control, credits

DUKE Computer Science

Congestion Control Taxonomy

- Router-centric versus host-centric
- Reservation-based versus feedback-based
- Window-based versus Rate-based

DUKE Computer Science

Queuing Disciplines

- How to distribute buffers among users/flows
  - When buffer overflows, which packet to drop?
- Simple solution: FIFO
  - First in, first out
  - If packet comes along with no available buffer space, drop it

DUKE Computer Science

Fair Queuing

- Goals:
  - Allocate resources equally among all users/flows
  - Low delay for interactive users
  - Protection against misbehaving users
- Approach: simulate general processor sharing (from OS world)
  - Bitwise round robin
  - Need to compute number of competing flows at each instant

Scheduling Background

- How do you minimize avg. response time?
  - By being *unfair*: shortest job first
- Example: equal size jobs, start at t=0
  - Round robin ➔ all finish at same time
  - FIFO ➔ minimizes avg. response time
- Unequal size jobs
  - Round robin ➔ bad if lots of jobs
    - Analogy: OS thrashing, spending all its time context switching
  - FIFO ➔ small jobs delayed behind big ones

TCP Congestion Problems

- Original TCP sent full window of data
- 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
  - Fast recovery

Jacobson Solution

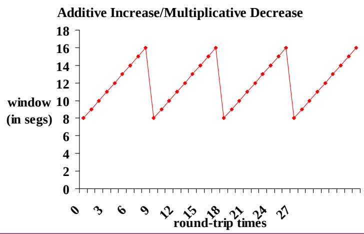
- Transport protocols should obey *conservation of packets*
  - Use ACKs to clock injection of new packets
- Modify retransmission timer to adapt to variations in delay
- Infer network bandwidth from packet loss
  - Drops ➔ congestion ➔ reduce rate
  - No drops ➔ no congestion ➔ increase rate
- Limit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

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

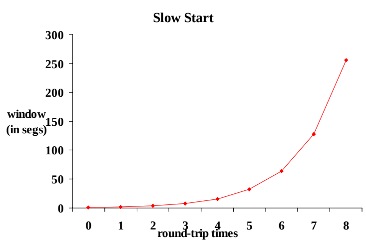


Slow Start

- 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

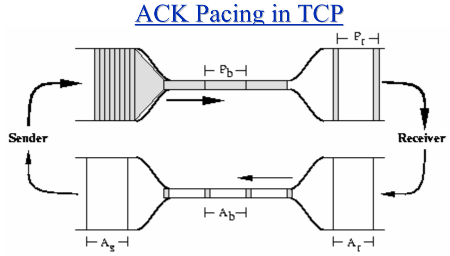
- Quickly find the bottleneck bandwidth



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

ACK Pacing in TCP



- ACKs open up slots in the congestion/advertised window
  - Bottleneck link determines rate to send
  - ACK indicates one packet has left the network

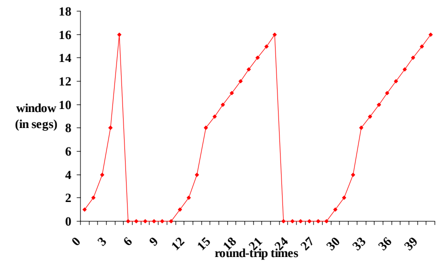
Problems with ACK Pacing

- ACK compression
  - Variations in queuing delays on return path changes spacing between ACKs
  - Example: ACK waits for single long packet
  - Worse with bursty cross-traffic
- What happens after a timeout?

Problems with ACK Pacing

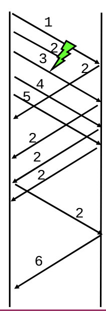
- ACK compression
  - Variations in queuing delays on return path changes spacing between ACKs
  - Example: ACK waits for single long packet
  - Worse with bursty cross-traffic
- What happens after a timeout?
  - Potentially, no ACKs to time packet transmissions
- Congestion avoidance
  - Slow start back to last successful rate
  - Back to linear increase/multiplicative increase at this point

Timeouts Dominate Performance



Fast Retransmit

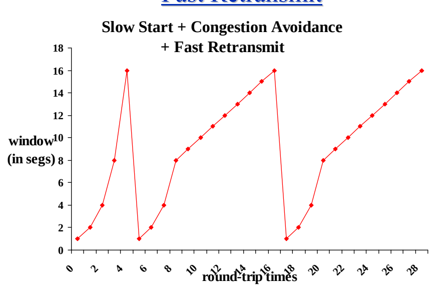
- Can we detect packet loss without a timeout?
- Duplicate ACKs imply either
  - Packet reordering (route change)
  - Packet loss
- TCP Tahoe
  - Resend if see three dup ACKs
  - Eliminates timeout delay



Fast Retransmit Caveats

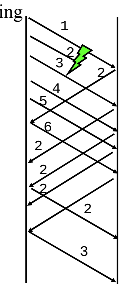
- Requires in order packet delivery
  - Dynamically adjust number of dup ACKs needed for retransmit?
- Does not work with small windows
  - Why not?
  - E.g., modems
- Does not work if packets lost in burst
  - Why not?
  - E.g., at peak of slow start

Fast Retransmit

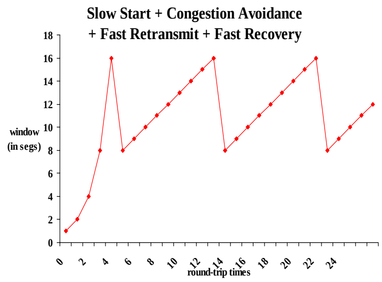


Fast Recovery

- 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

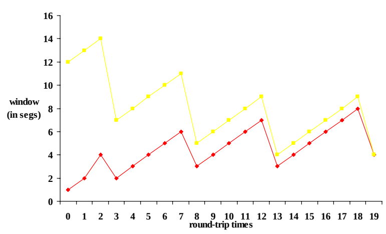


Delayed ACKs

- 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

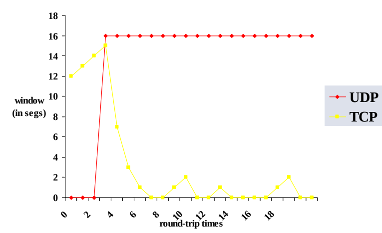
What if Two TCP Connections Share Link?

- 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 recovery
  - Use ACKs in pipeline to avoid shrinking congestion window to one
  - Cuts out going back to slow start when detecting congestion with fast retransmission

DUKE Computer Science

TCP Vegas

DUKE Computer Science

Overview/Goals

- Goals:
  - Increase useful throughput of TCP
    - Vegas increases throughput by 37-71%
  - Decrease retransmissions
    - Vegas retransmits 1/5 to ½ the data of Reno
- Note: easy to increase throughput at the expense of other connections
- TCP Reno controls congestion by *causing* it
  - Vegas aims to avoid congestion using only host-based measurements

DUKE Computer Science

Implementation

- Retrofitted x-kernel with BSD implementations of TCP Reno and Vegas
  - Ran both simulations and real wide-area experiments
- Simulated cross traffic (e.g., FTP/NNTP/Telnet) using *tcplib*

DUKE Computer Science

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?

DUKE Computer Science

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

DUKE Computer Science

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 << expected, queues increasing ➔ decrease rate before packet drop
- If actual ~= expected, queues decreasing ➔ increase rate
- What if base RTT changes (route changes)?

DUKE Computer Science

TCP Vegas Congestion Avoidance

- Define two parameters $\alpha < \beta$
- Let Diff = Expected – Actual
  - Always a positive value
- If Diff < α, linearly increase congestion window
- If Diff > β, linearly decrease congestion window
- If α < Diff < β, do nothing
- Why can we get away with linear decrease instead of multiplicative decrease?
  - We are avoiding congestion, not reacting to it

DUKE Computer Science

TCP Vegas Congestion Avoidance

- α and β are measured in terms of throughput (e.g., kb/s) however, they really represent extra buffers in the network
- Intuitively, want each connection to occupy one extra buffer in the network
  - If extra capacity becomes available, Vegas flows will capture them (since they sit in one buffer in steady state)
  - In times of congestion, Vegas flows occupy too many buffers, so Vegas backs off
- Typical values for $\alpha = 1$ and $\beta = 3$
  - Goal: have Vegas flows occupy between 1 and 3 router buffers

DUKE Computer Science

TCP Vegas Slow Start

- Reno doubles congestion window every RTT in slow start
  - Can overshoot capacity, cause many losses
- Vegas doubles congestion window *every other* RTT
  - Only double window if actual rate is within equivalent of 1 router buffer of expected rate
    Note 1 KB buffers with 100 ms RTT equals 10 KB/s
- Vegas* uses packet-pair mechanism to estimate available bandwidth
  - Slow start to avail. bandwidth, then back to linear increase
  - Why not go straight to bottleneck bandwidth?
  - Vegas* did not result in significant perf/loss improvements

DUKE Computer Science

Vegas Discussion

- Does not involve a modification to the TCP spec
  - Can be deployed incrementally
- Does not steal bandwidth from other implementations
- Uses additional information available at hosts to better estimate congestion
  - Congestion *avoidance* vs. *control*
- Additional processor overhead
  - Increases throughput/reduces wasted transmissions
- Should congestion control be in hosts/routers/both?

DUKE Computer Science